THE UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

UNIVERSITY OF
NEWCASTLE UPON TYNE

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

List of Figures

List of Tables

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,

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

subject to

$$\begin{aligned} \sum_{j=1}^{n} a_{ij}x_j \;&\leq\; b_i \quad (i \;=\; 1, 2, \ldots, m) \\ x_j \geq 0, \quad (j \;&=\; 1, 2, \ldots, n). \end{aligned}$$

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. logarithmic, 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 details 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. Häandler [13] and Hwang & Briggs [23] stress the availability of pipelining and the number of pipeline stages. Another 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

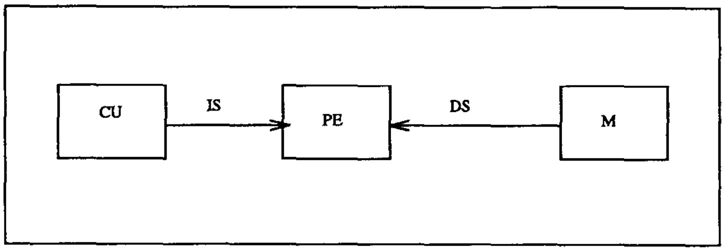

Figure 2.1: General Structure of SISD Architecture

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.

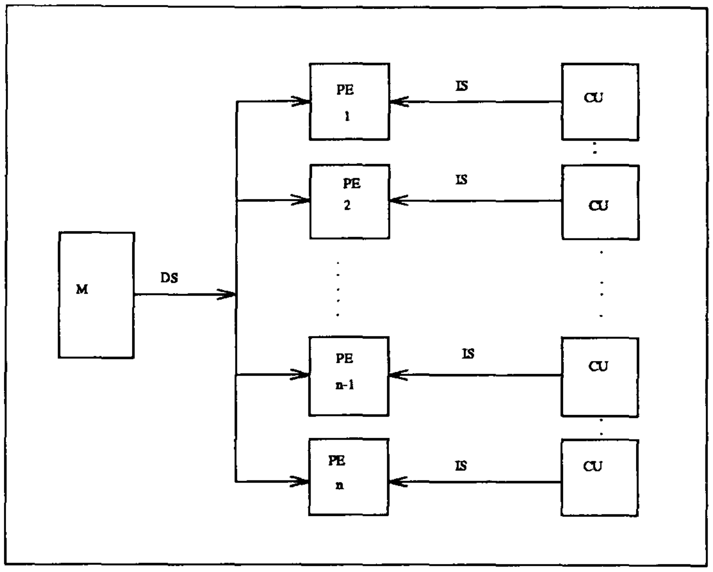

Figure 2.2: General Structure of MISD Architecture

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.

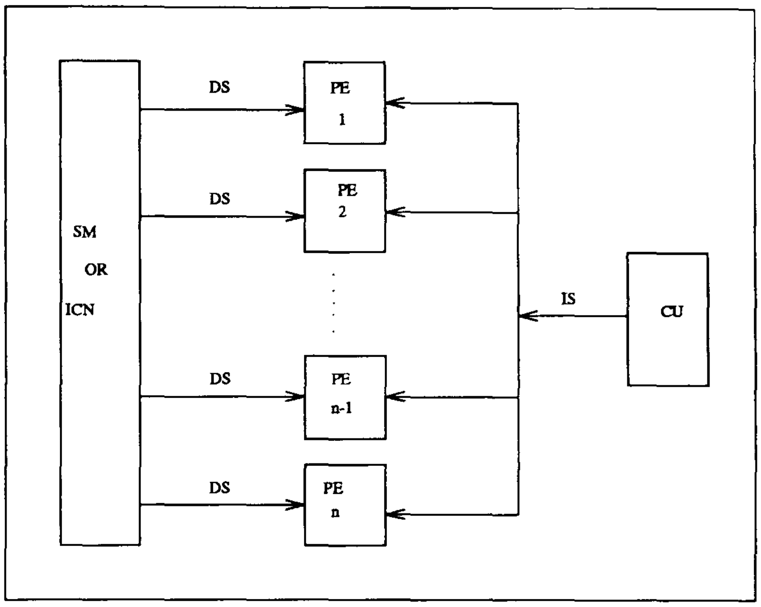

Figure 2.3: General Structure of SIMD Architecture

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 is 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].

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

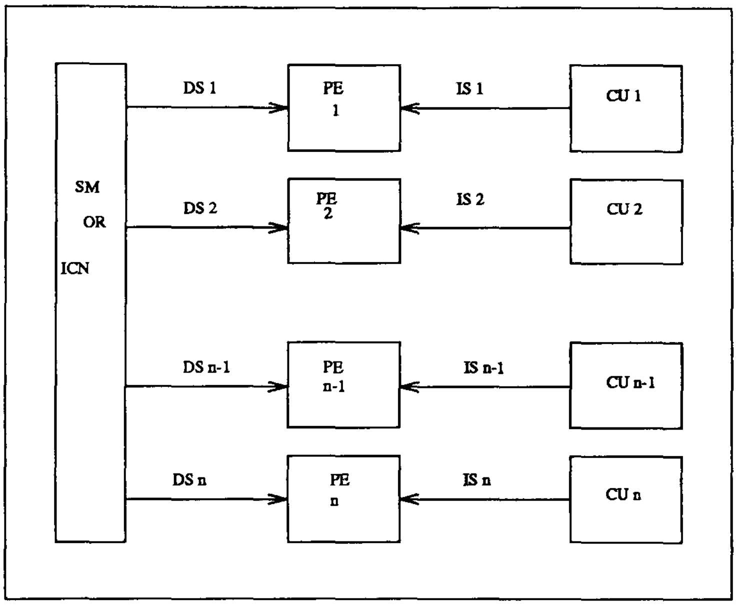

Figure 2.4: General Structure of MIMD Architecture

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

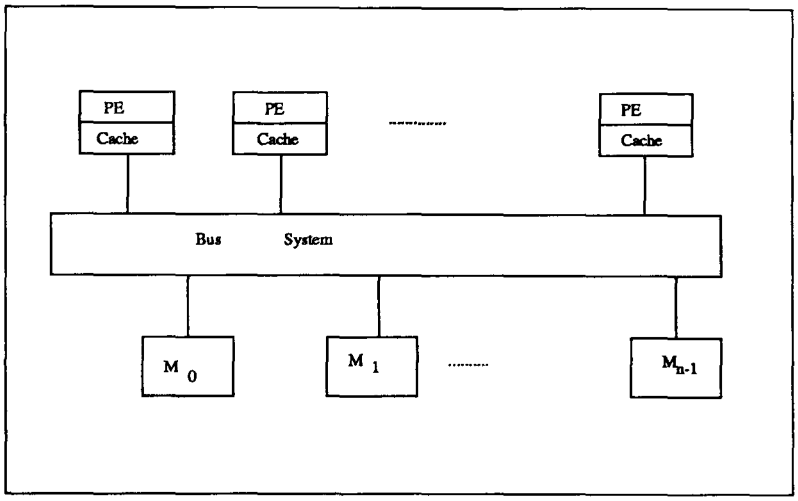

Figure 2.5: Bus Interconnection

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 x N MIN connects N processors to N memories by deploying multiple stages or banks of switches in

Table 2.1: Properties Of Multiprocessors

Property	Bus	Crossbar	Multistage
Speed	low	high	high
Cost	low	high	moderate
Reliability	low	high	high
Configurability	high	low	moderate
Complexity	low	high	moderate

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.

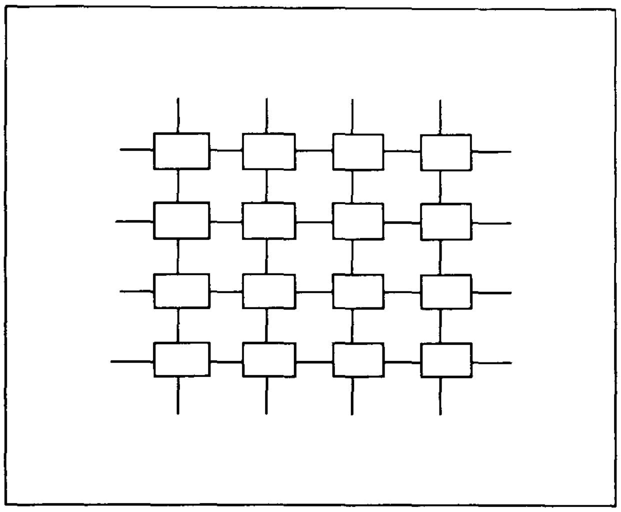

Figure 2.6: Transputer Network

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

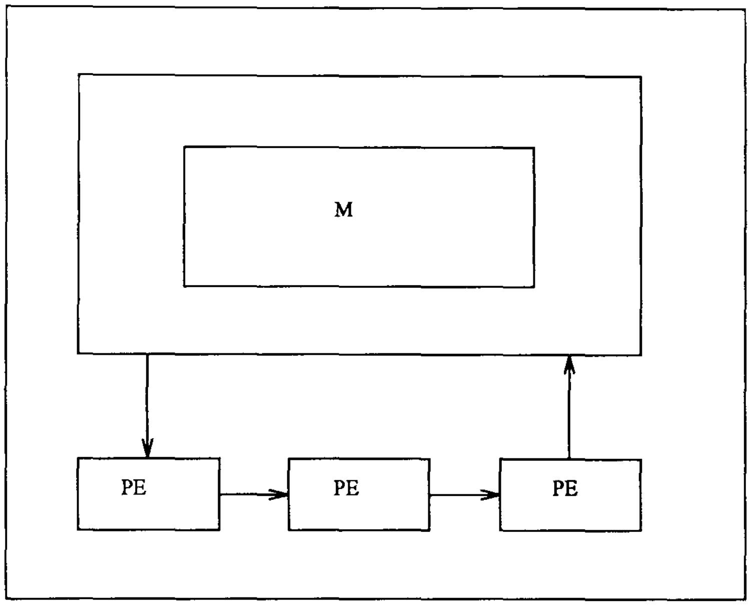

Figure 2.7: Systolic flow of data to and from memory

links.

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

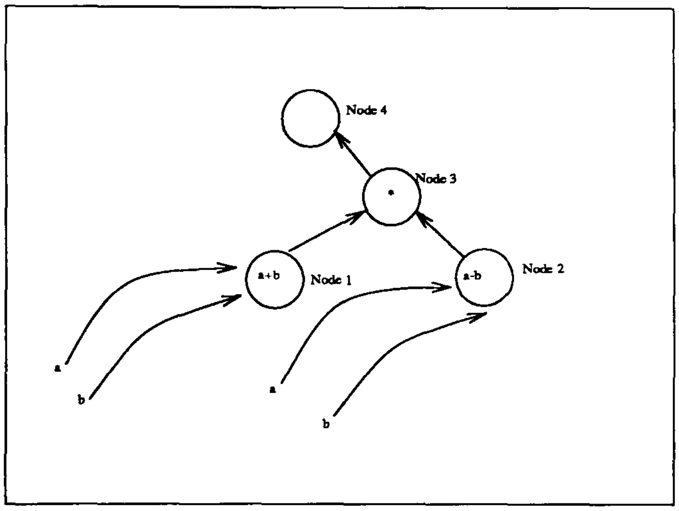

Figure 2.8: Dataflow Graph

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 $a * b$ 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

SISD	IBM 7090 [52]
SIMD	Illiac IV [25]
MISD	
Multiprocessors(Shared Memory)	Encore Multimax [35]
Multicomputers(Distributed Memory)	Transputers [29]
Systolic Arrays	Carnegie Mellon's Warp [36-38]
Dataflow	MIT Tagged Token Dataflow [39]
Reduction	North Carolina's FFP [41]

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.

```
/*
** A parallel program using multiple threads for multiplying matrices.
*/
PROGRAM1
========
#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\n %3d %3d %3d\n %3d %3d %3d\n",
      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)
    break;
        count++;
        THREADvsem(sem);
    }
    THREADvsem(sem);
    return(i);
}
```

The critical sections are protected by the statements **THREADpsem(sem)** and **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.

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 **sender**, and the process receiving it is the **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 **par** file which describe a multi-process task to the parallel loader is also shown. Each process calls the function, **csn_init()** to initialise the Computing Surface Network (CSN). This is followed by a call to **csn_open()** that creates a connection between the process and the CSN. This connection is called a **Transport**. Each Transport on the CSN has an associated address, called a **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("Farenheit     Celsius\n"); fflush( stdout );
   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 processses '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 Asynchronous. 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 determines 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, attectiveness, 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 = Speedup

T_s = running time of fastest sequential algorithm and

T_p = 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 ∞ 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 $\overline{p_1 p_2}$ is defined by $\beta p_1 + (1-\beta)p_2$ 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^{T_r}.

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 α_1, α_2, $\ldots$, α_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 $p = (\alpha_1 p_1 + \alpha_2 p_2 + \ldots + \alpha_k p_k)$ 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 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 R^d, the set of points $p = (\alpha_1 p_1 + \alpha_2 p_2 + \cdots + \alpha_k p_k)$ where $(\alpha_j \in R\ ,\ \alpha_j > 0,\ \sum_{j=1}^{k} \alpha_j = 1)$ is the convex set generated by $p_1, p_2, \ldots, p_k$ and its convex combination is p. A domain D in R^d is convex if, for any two points p_1 and p_2 in D, the segment $\overline{p_1 p_2}$ is entirely contained in D.

Hyperplane: A hyperplane H is the set of points $X = (x_1, x_2, \cdots, x_d)$ which satisfy an equation represented in the form $\sum_{i=1}^{d} \alpha_i x_i - \beta = 0$, where not all α_i are zero. A hyperplane H separates the space 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 $S \subset R^d$ 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 $(\hat{d}.\hat{v}_i) \geq 0$ for $i = 1, \cdots, d$ or the inner product $(\hat{d}.\hat{v}_i) \leq 0$ $i = 1, \cdots, m$; $\hat{d}$ 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 $S \subseteq R^d$ 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.

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

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 $\mid S \mid$ 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) $\| kx \| = \mid k \mid \| x \|$ 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

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

Orthonormal Set: If $(x, x) = \| x \|^2 = 1$, the vector x is said to be normalised. If a set of vectors $x_1, \cdots, x_d$ 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 $O(nlog_2n)$ 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 $O(nlog_2n)$ 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 $O(nlog_2h)$ 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 $O(nlog_2n)$. 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

[82] which cover most of the variations.

3.2.1 Graham's Algorithm:

Graham in [64] presented one of the first $O(nlog_2n)$ 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 $O(nlog_2n)$, it follows that finding the convex hull by Graham's algorithm requires $O(nlog_2n)$ steps. The algorithm can be summarised as follows:

Step 1: An internal point O is chosen arbitrarily (e.g. centroid of three non colinear points). At worst case this can be done in c_1n 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_2n operations, c_2 a constant.

Step 3: The elements $\rho\exp(i\theta_k)$ are sorted in terms of increasing θ_k such that the set of points $S = \{r_1exp(i\psi_1), \ldots, r_nexp(i\psi_n)\}$ with $0 \leq \psi_1 \leq \ldots \psi_n \leq 2\pi$ and $r_i \geq 0$. This is possible in $O(nlog_2n)$ 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 $S' = \{r_1exp(i\psi_1), \cdots, r_{n'}exp(i\psi_{n'})\}$ where $n' \leq n$. This elimination can be done in less than n comparisons.

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

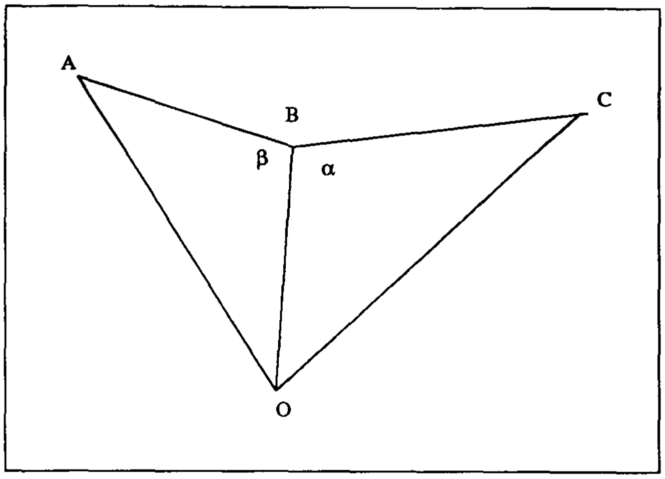

Figure 3.1: Angle Between Three Points

1. If $(\alpha + \beta) \geq \pi$, delete the point $r_{k+1}exp(i\psi_{k+1})$ from S' since it cannot be an extreme point and return to the beginning of step 5 with the points $r_kexp(i\psi_k)$, $r_{k+1}exp(i\psi_{k+1})$, $r_{k+2}exp(i\psi_{k+2})$ replaced by $r_{k-1}exp(i\psi_{k-1})$, $r_kexp(i\psi_k)$, $r_{k+2}exp(i\psi_{k+2})$.
2. If $(\alpha+\beta) < \pi$, return to beginning of step 5 with points $r_kexp(i\psi_k)$, $r_{k+1}exp(i\psi_{k+1})$, $r_{k+2}exp(i\psi_{k+2})$ replaced by $r_{k+1}exp(i\psi_{k+1})$, $r_{k+2}exp(i\psi_{k+2})$, $r_{k+3}exp(i\psi_{k+3})$.
   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

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
x	4	12	7	5	6	9	2	8	10	15	11	17	16	14	4	13
y	10	2	9	4	16	12	7	5	8	6	14	15	3	17	13	11

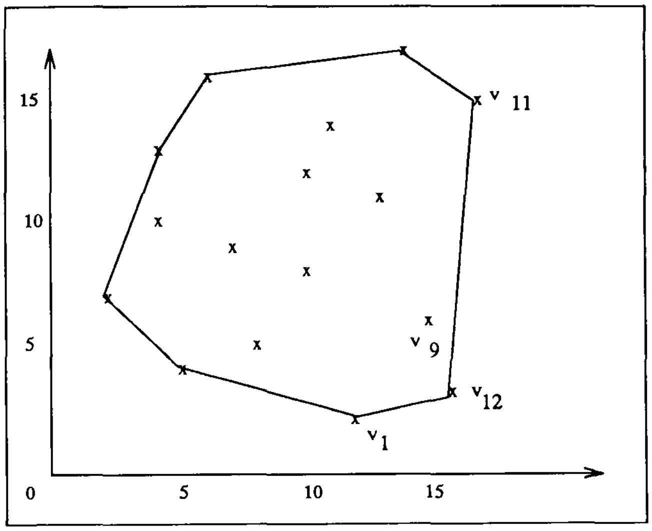

Figure 3.2: 2-D Convex Hull Illustration

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 $\overline{v_1 v_{12}}$ is on the convex hull, i.e all the remaining points must lie on one side of the directed line containing $\overline{v_1 v_{12}}$. After v_{12} has been found, the same technique is applied to locate the next point v_{11} such that $\overline{v_{12} v_{11}}$ 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 $O(nlog_2n)$ 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.

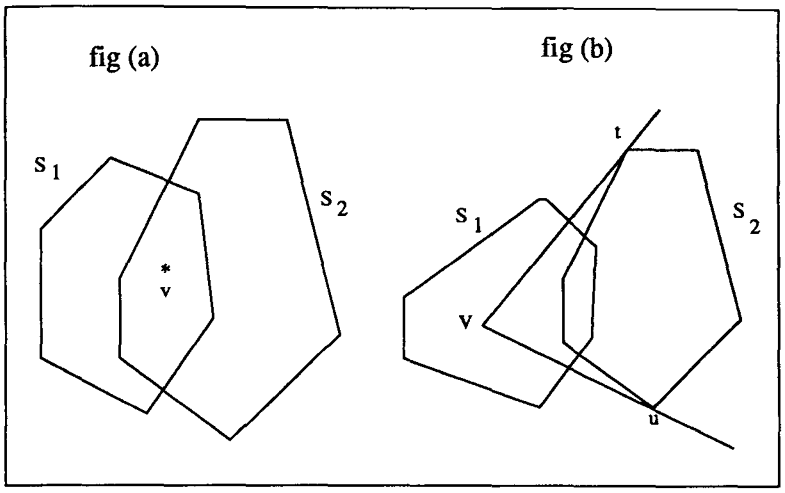

Figure 3.3: 2-D Convex Hull For Divide-And-Conquer

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 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 $S_1 \cup S_2 = S$;
        $S' :=$ Convex Hull($S_1$);
        $S'' :=$ Convex Hull($S_2$);
        $T :=$ 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

Dimension	Author	Time
2-D	R. L. Graham 64]	$O(nlog_2n)$
	R. A. Jarvis [60]	$O(nH)$
	Divide-and-Conquer[82]	$O(nlog_2n)$
3-D	Preparata and Hong [62]	$O(nlog_2n)$
	Johansen and Cram [65]	$O(nF)$

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 $S \subset R^d$ 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 $P^I \in S_c$. The hyperplane H, $x_1 = p^I$, 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 $(\hat{e}.\hat{d}\ ) = 0$, $(\hat{e}.\hat{v}_i\ ) = 0$, $P_i \in S_b$, $\hat{v}_i$ being a unit vector along $\vec{p^I p^i}$ and $(\hat{e}.v_k) \geq 0$, $P_k \in S_c$.

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

$$\frac{\lambda_k}{\mu_k} = -\frac{\hat{e}.\hat{v}_k}{\hat{d}.\hat{v}_k}$$

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 λ and μ 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 by product.

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 $O(logn)$ time, a worst case

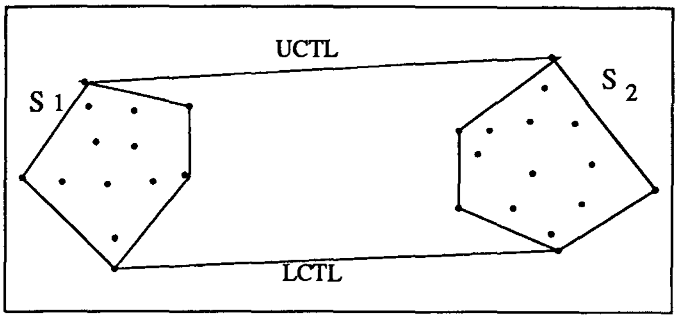

Figure 3.4: Upper and lower tangent lines between S_1 and S_2

algorithm for the pyramid, tree machine, and mesh-of-trees finishes in $O(log^3 n/(loglogn)^2)$ 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 $O(logn)$ 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.

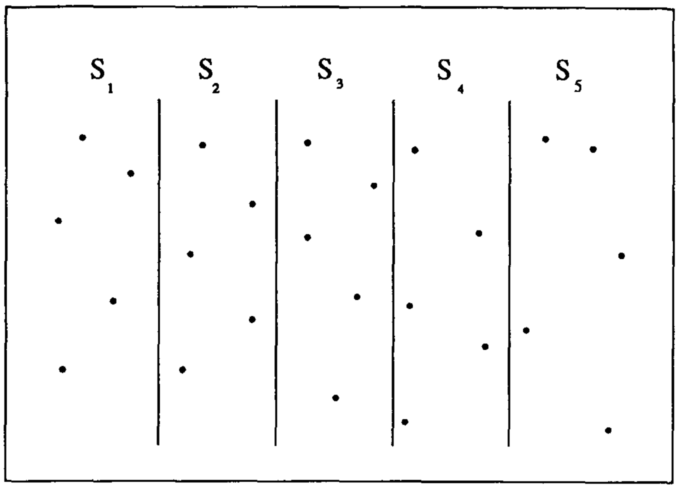

Figure 3.5: A partitioning of S into 5 subsets

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 $i \in \{1,2,...,\sqrt{n}\}$ 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 $CH(S_j)$.

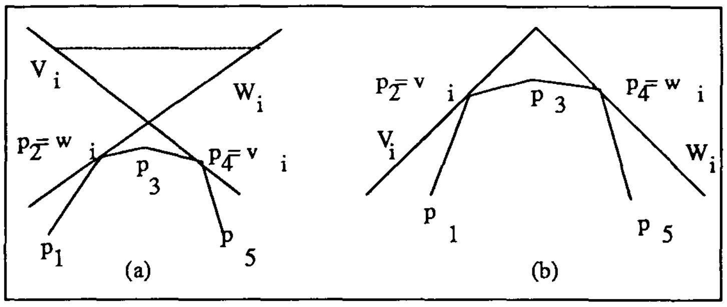

Figure 3.6: Illustration Of Merge Procedure

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 π, 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 $i \in \{1,2,3,...,\sqrt{n}\}$ 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 etal. [85] also achieved $O(logn)$ 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 $O(logn)$ time using $O(n/logn)$ 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 $O(logn)$ 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

Architecture	Author	Complexity	Processors
Systolic array	Chazelle [78]	$O(n)$	n
CREW PRAM	Chow [73, 86]	$O(log_2^2 n)$	n
Multiprocessors	Akl [74]	$O(log_2 n)$	n^3
SIMD	Akl [76]	$O(n^e log_2 n)$	n^{1-e} $0 < e < 1$
Square mesh	Miller and Stout [68]	$O(\sqrt{n})$	n
Hypercube		$O(log_2 n)$	
Tree machine		$O(log_2^3 n/(log_2 log_2 n)^2)$	
Mesh of trees			
Pyramid			
Reconfigurable mesh		$O(log_2^2 n)$	
AKS network		$O(log_2 n)$ worst case	
Pyramid (ordered input)		$O(log_2 n)$ worst case	
CREW PRAM	Goodrich and Atallah [69]	$O(log_2 n)$	$O(n)$
CREW PRAM	Goodrich [70]	$O(log_2 n)$	$O(n/log_2 n)$
Mesh array (OIA)	Holey and Ibarra [80]	$O(n)$	$(n-2)$
Cellular array		$O(n)$	$O(n)$
D-Cellular array		$O(n^{1/d})$	$O(n)$
OCA		$O(n)$	n

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 $O(nlog_2n)$ 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 $O(nlog_2n)$.

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, $N = \mid S \mid$, 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 $O(Ndf + d^3f^2 + Nd!f)$. 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 sub facets, 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 **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.

Algorithm affine_hull()

Input: $d > 1, S \subseteq R^d$

Output: $A \subseteq S$ the first points of S which form a maximal set of affinely independent points, k is the dimensionality of S.

```
function affine_hull($S, d$) : ($A, k$);

    $A := \emptyset$;    (1)
    $p_0$ := first point in $S$;    (1)
    $r$:= 0;    (1)
    For $p \in S - \{p_0\}$ do
        {check if $\vec{p_0 p}$ is representable in terms of elements of $A$}
        $v := \vec{p_0p}$;    $(| S | d)$
        for $x \in$ A do    $(| S | k)$
            $v := v - \alpha.x$;  $\{\alpha$ a scalar$\}$    $(| S | kd)$
        end
        if($v \neq 0$) $\{\vec{p_0p}$ is not representable$\}$    $(| S |)$
            then $A := A \cup v$    $(kd)$
    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 $kd \mid S \mid$ 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

$$u_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix}, \quad u_2 = \begin{bmatrix} 2 \\ -1 \\ -1 \\ 1 \end{bmatrix}, \quad u_3 = \begin{bmatrix} -1 \\ 2 \\ 2 \\ 1 \end{bmatrix}$$

we set $v_1 = u_1$ and then choose α 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

$$v_2 = \frac{3}{4} \begin{bmatrix} 3 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

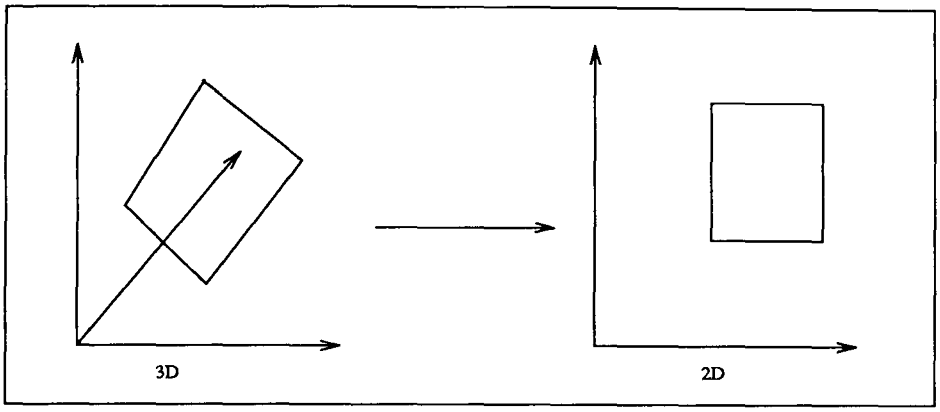

Figure 4.1: Transformation Of 3-D To 2-D square

Similarly

$$v_3 = u_3 - \beta v_1 - \gamma v_2$$

giving $\beta = \frac{1}{2}$ 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

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

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.

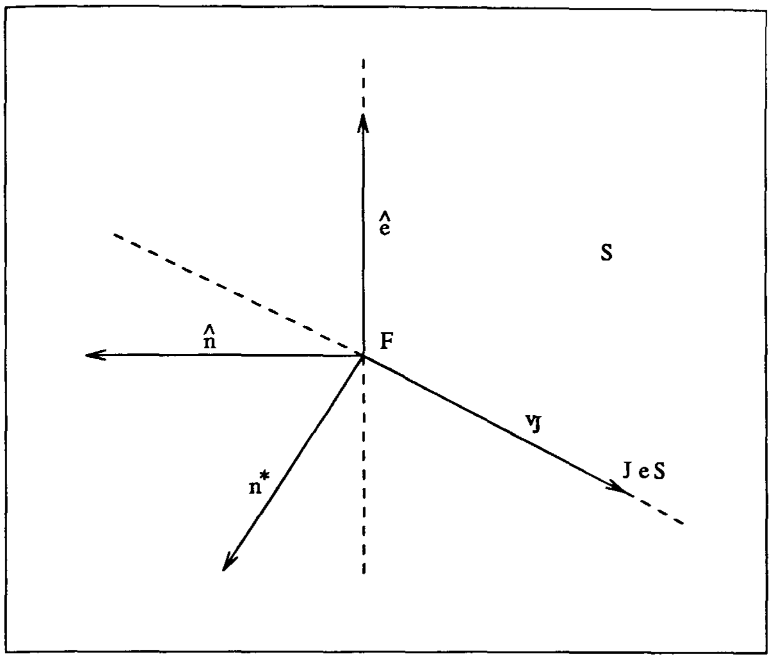

Figure 4.2: Set S Projected Onto The Plane Of $\hat{e}$ And $\hat{n}$

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, $F = \{p_0, \cdots, p_k\}$, 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 $\hat{e} = (e_1, e_2, \cdots, e_d)$ 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 $v_p = \vec{p_0 p}$, 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:

$$\hat{e}.\hat{n} = 0$$

$$\hat{e}.v_i = 0$$

where

$$v_i = \vec{p_0 p_i}, \quad i = 1, 2, \cdots, k.$$

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 θ_p between v_p and $\hat{n}$ by

$$\tan\theta_p = -\frac{\hat{e}.v_p}{\hat{n}.v_p}.$$

The vector orthogonal to v_J is given by

$$n^* = (v_J.\hat{n})\hat{e} - (v_J.\hat{e})\hat{n}$$

$$n^*.v_q \leq 0 \ \ for\ any\ q \in S$$

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 = affine_hull(S, d); F a $(j+1)$-membered subset of AS, s.t. $j < k-1$ and 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 $\hat{e} \in$ affine($S$) orthogonal to $F$ and $\hat{n}$}
  $AS' := \{\vec{p_0p} : p \in AS - \{p_0\}\}$      $(kd)$
  $F^* := (j+1)$ by $k$ matrix of the vectors $\{\vec{p_0p} : p \in F - \{p_0\}\} \cup \{\hat{n}\}$ 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 $S$;      $(| S | d)$
  $J$ = 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 $\hat{n}.\vec{p_0p} > 0$ for any $p \in S$          {i.e. check the orientation of the outward normal}
     then $\hat{n}^* := -\hat{n}^*$;
```

```
    return $(J, \hat{n}^*)$;
end
```

Normally, once a facet is known, a single call of **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 rotate is given as $jkd + d \mid S \mid$. The two major steps that contributes to the running time of this routine are the formation of the $(j+1)$ by k matrix and the computation of the angle. The code for **rotate ()** is given in Appendix C.1.9.

Algorithm initial_face

Input: $d \geq k > 1, S$ a k-dimensional subset of R^d and AS = 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;     $(d)$
  $F$ := set of points in $S$ with minimal $i$th coordinate;     $(\mid S \mid d)$
  $F$ := affine_hull$(F, d)$     $(\mid S \mid kd)$
    $\hat{n}$ := projection of the outward normal $(0, 0, \ldots, -1, 0, \ldots, 0)$
      { i.e. vector with -1 in the $i$th component } onto affine$(AS)$     $(kd)$
  while $\mid F \mid \leq k - 1$ do     $(k)$
    $(P, \hat{n})$ := rotate$(S, d, F, \hat{n})$;     $(k(kkd + d \mid S \mid))$
    $F := F \cup P$;     $(kd)$
    end;
  return $(F, \hat{n})$     $(d)$
  end
```

The running time of initial_facet() is given as $kd \mid S \mid + k^3 d$. 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 = affine_hull(S, d).

Output: $CH \subseteq S$, a set of vertices of $CH(S)$. FA a family of sets of affinely independent points $\{FA_1, \dots, FA_{f_{d-1}}\}$ such that affine(FA_i) is a hyperplane containing the ith 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 := $p \in S$ s.t. $\vec{p_0 p}.\vec{p_0 p_1}$ is minimised;    ($| S | d$)
        max := $p \in S$ s.t. $\vec{p_0 p}.\vec{p_0 p_1}$ 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 $(S, \{FA \subseteq F : | F | = k\})$    ($kd$);
    $(F, \hat{n})$ := 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    ($f_{d-1}$)
        $FA := FA \cup \{F\}$;    ($f_{d-1} kd$)
        Pick a point $p_0 \in F$;    $(f_{d-1})$
        $F' := \{p \in S : \vec{p_0 p}.\hat{n} = 0\}$;    $(f_{d-1} \mid S \mid d)$
        (FCH,FFA) := convex_hull($F', F, d, k-1$); $(\sum_i T(F_i', k-1))$
        CH := $CH \cup FCH$;    $(f_{d-1} \mid S \mid d)$
    while(FFA $\neq \emptyset$) do    $(2f_{d-2})$
        {remove facet(E,-) from FFA;    $(2f_{d-2} kd \log f_{d-2})$
        if (E $\in$ EdgeList) remove facet(E,-) from EdgeList;
            else add $(E, \hat{n})$ to EdgeList; }
    if(EdgeList $\neq \emptyset$)    $(f_{d-1})$
        { Pick an $(E, \hat{n})$ from the edge list;    $(f_{d-1} kd)$
        $(P, \hat{n})$ := rotate($S, AS, d, k, E, \hat{n}$);    $(f_{d-1}(k^2 d + \mid S \mid d))$
        $F = E \cup \{P\}$; }    $(f_{d-1} d)$
    }while(EdgeList $\neq \emptyset$);
    return (CH,FA);
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

$$T(S,k) = O(k^2 d f_{d-1} + dn f_{d-1} + kd f_{d-2} log f_{d-2} + \sum_i T(F_i', k-1))$$

where f_k is $f_k(P)$ and F_i' is the set of points sharing the hyperplane with the ith facet of P. The boundary conditions are $T(S,k) = \Theta(k\ d)$ if $\mid S \mid$ = k+1, and $T(S,1) = \Theta(n)$. The main driving routine in the program is the function **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 $CH(S)$ 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 line. 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_facet()** 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 $\mid S \mid = 3 = k + 1$, 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

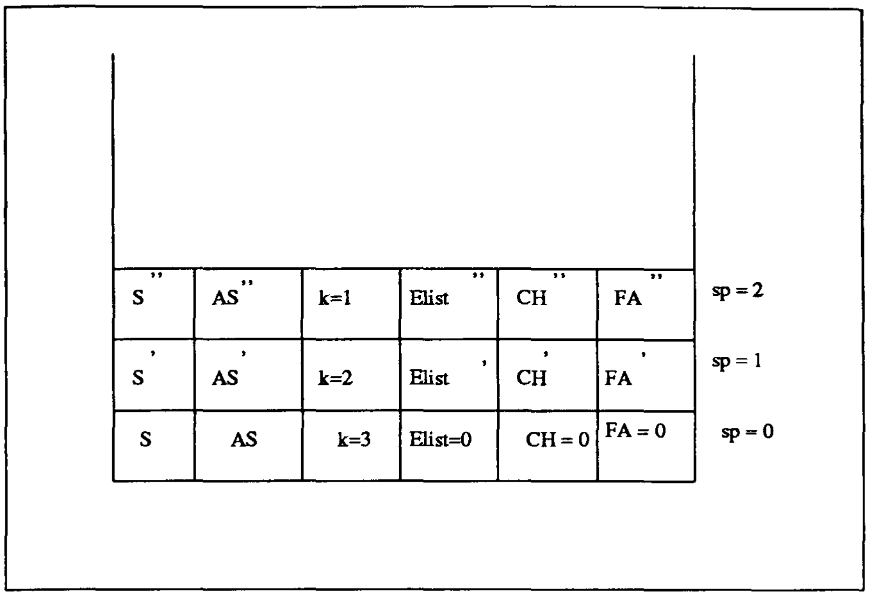

Figure 4.3: Stack Implementation For 3-D

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 is. 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);

```
{ /* 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 $\vec{p_0p_1} \cdot \vec{p_0p_1}$ is minimised
          max = $p \in$ S such that $\vec{p_0p_1} \cdot \vec{p_0p_1}$ is maximised
          return ({max,min} , {max}, {min});
          }
        if ($| S |= k+1$)
          return $(S, \{F \subseteq S :| F |= k\})$;   /* check for a simplex */
        else
            {
            (F,$\hat{n}$) = initial facet(S,AS,n,k)
            FA = FA $\cup$ { F }
            $F' = \emptyset$;
            Pick a point $p_0 \in F$;
            $F' = \{p \in S : \vec{p_0p}.\hat{n} = 0\}$ ;
            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
```

```
                else add (E,-) to Stack[sp-1].Elist
                }
            Delete lists: Stack[sp].S; Stack[sp].AS;
            Stack[sp].k = 0; sp = sp-1;
            }
        if (EdgeList $\neq \emptyset$) /* get next face */
            {
            Pick an $(E, \hat{n})$ from EdgeList;
            $(p, \hat{n})$ = rotate(S,AS,n,k,E,$\hat{n}$);
            F = E $\cup$ {p};
            FA = FA $\cup$ { F }
            Pick a point $p_0 \in$ F
            $F' = \{p \in S : \vec{p_0 p_1}.\hat{n} = 0\}$;
        /* stack the face */
            sp = sp +1; Stack[sp].S = $F'$; Stack[sp].AS = (F,$\hat{n}$);
            Stack[sp].k = Stack[sp-1].k-1;
            }
    }while(EdgeList $\neq \emptyset$ 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 $\hat{n}$. This representation follows because the pair $(E, \hat{n})$ 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 Appendies 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) examines 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.

Table 4.1: Test Data

Points	*Vertices*	*Dimension*
25–4000	3	2-D
25–4000	4	
25–4000	6	
25–4000	16	
50–4000	26	
25–4000	3	3-D
25–4000	4	
25–4000	6	
25–3000	12	
25–4000	3	4-D
25–4000	4	
25–4000	6	

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.

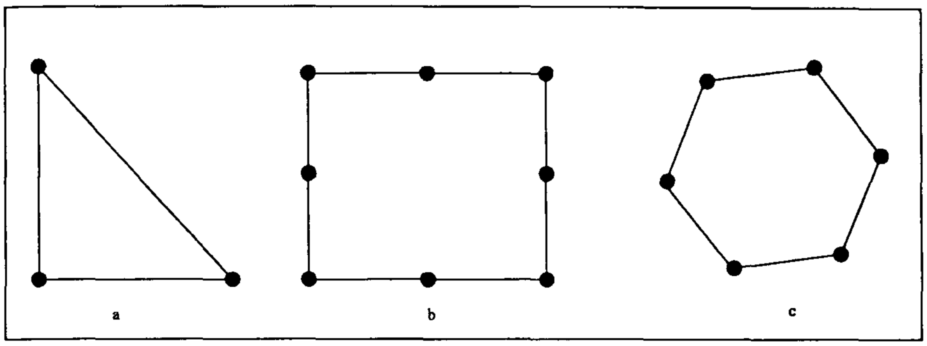

Figure 4.4: Types Of Shapes In 2-D

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

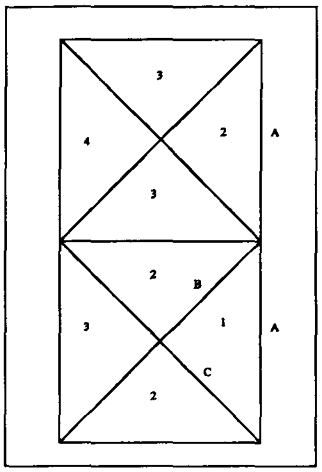

Figure 4.5: Illustration of 3-D Circular Shape

constant radius r. With the starting radius, and rotating in an anticlockwise direction, an angle θ is formed with another point say, p_i such that arc pp_i subtends an angle θ 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

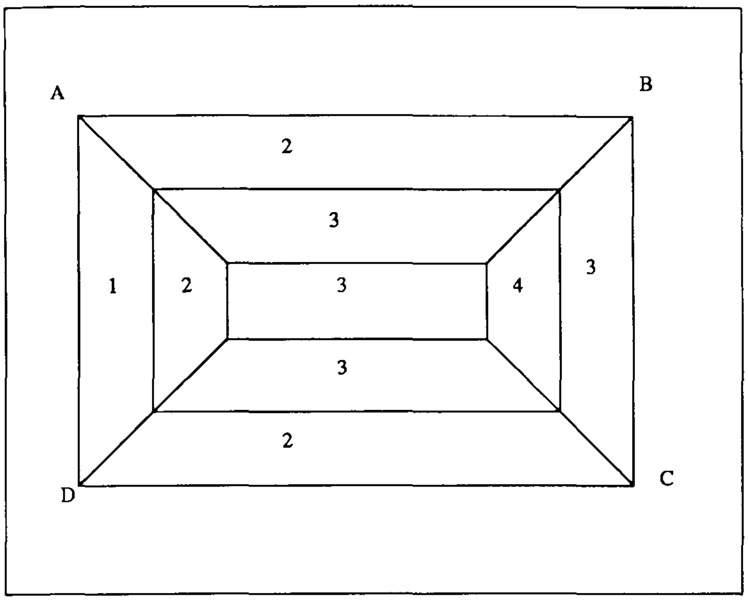

Figure 4.6: Illustration of 3-D Pyramid Shape

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 approaches 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 $\Omega(fNd + d^3f^2 + d!df)$ 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 $N >> f$, 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 Mergel.

Mergel

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

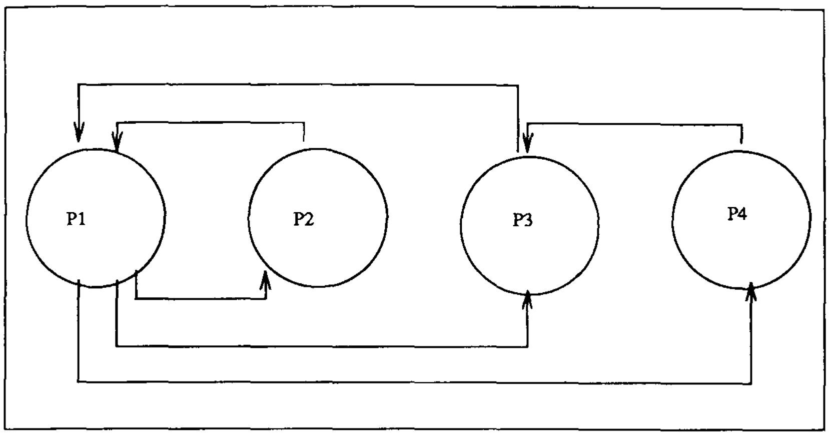

Figure 5.1: Illustration Of Hybrid Approach

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 $\mathcal{S}$ 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 = $\lceil p/2 \rceil$
    if $procs$ odd $CH(R_{p+1}) = \emptyset$
    while(procs $\neq$ 1) /*   loop 1   */
    {
        For $i = 1$ to procs
        {
            $CH(R_i) = CH(R_i) \cup CH(R_{\lceil p/2 \rceil + i})$
            $CH(R_{\lceil p/2 \rceil + i}) = \emptyset$ /*   loop 3   */
            Generate_Hull($CH(R_i)$) /*   loop 2   */
        }
    procs = $\lceil procs/2 \rceil$
    }
```

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($R_i$) $\neq$ TRUE)
    if($R_i$ $\neq$ GetNext_Point($R_i$)
        $R_i$ = Remove_Duplicate_Points($R_i$, n)
        Quick_Sort(GetNext_Point($R_i$), GetPrev_Point($R_i$), n)
        Affine_Hull($R_i$, n, AS, k)
        Convex_Hull($R_i$, AS, n, k, $CH(R_i)$, FA)
    else
        Return single point as $CH(R_i)$
else
    MakeEmpty_Plist($CH(R_i)$)
    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 $CH(R_i)$ need not be considered any further because they cannot be vertices of $CH(S)$. This algorithm could be summarised as follows:

```
    $R_1, R_2, \ldots, R_p = partition(S, d)$
/* 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
```

```
else
    procs = procs/2 + 1
while(procs $\neq$ 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 $\Omega((\log_2 p + 1)(f_{max}nd + d^3 {f_{max}}^2 + d!df_{max}))$ 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 $\Omega((\log_2 p+1)(f_{max}dn/p+d^3 {f_{max}}^2 + d!df_{max}))$ 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

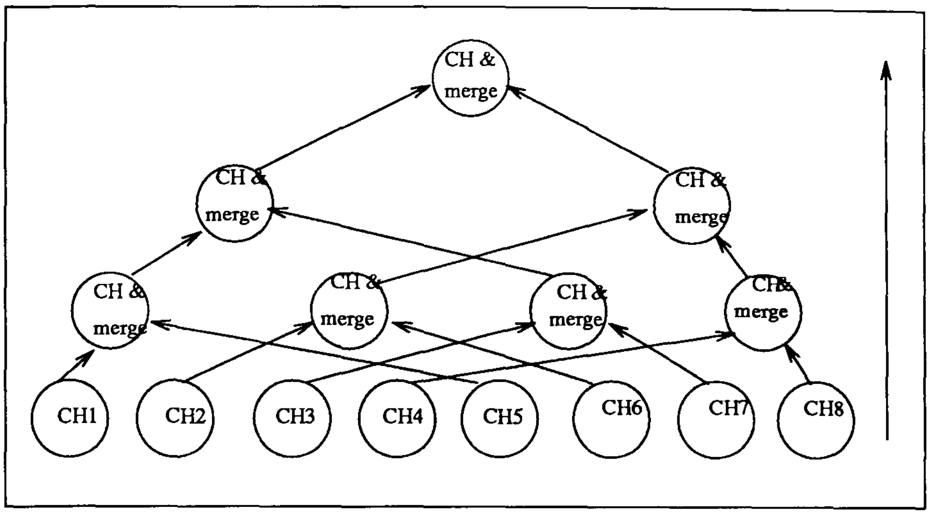

Figure 5.2: Merge Tree For Eight Subconvex Hulls

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 achieveable. 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 book-keeping 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 inital 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

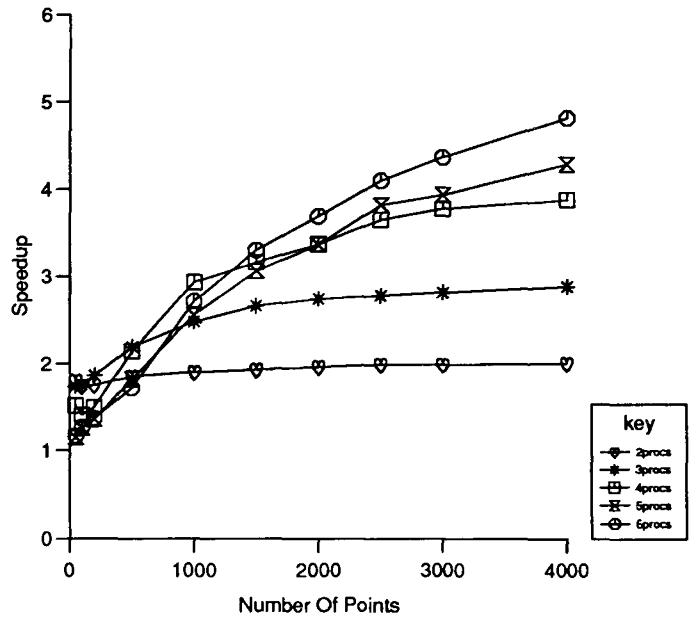

Figure 5.3: Recursive Version 2-D 26 Vertices Using Threads

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.

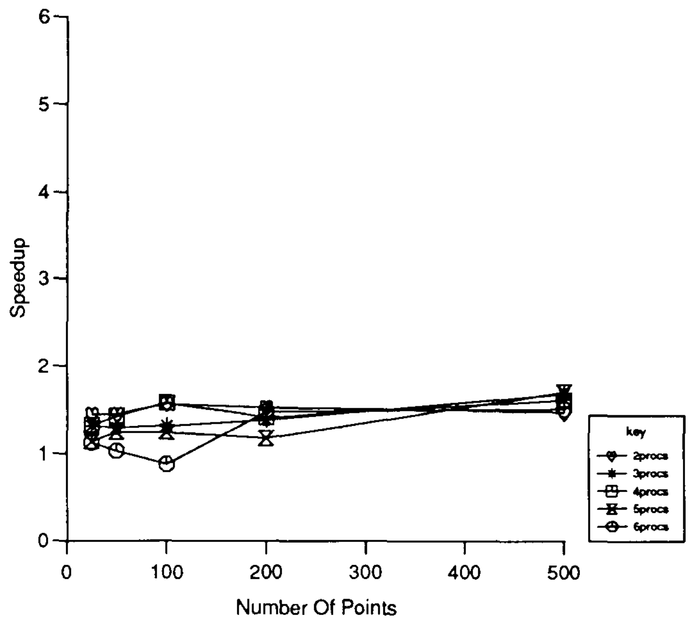

Figure 5.4: Recursive Version 4-D 6 Vertices Using Threads

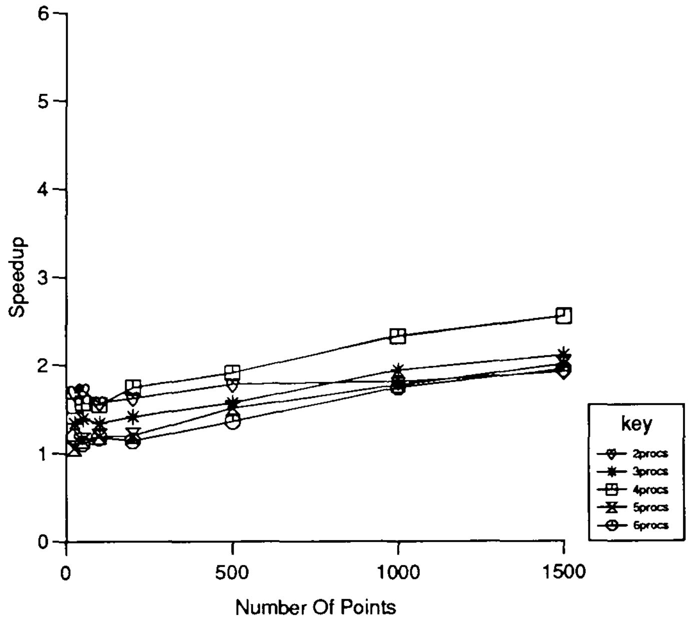

Figure 5.5: Recursive Version 3-D 12 Vertices Using Microthreads

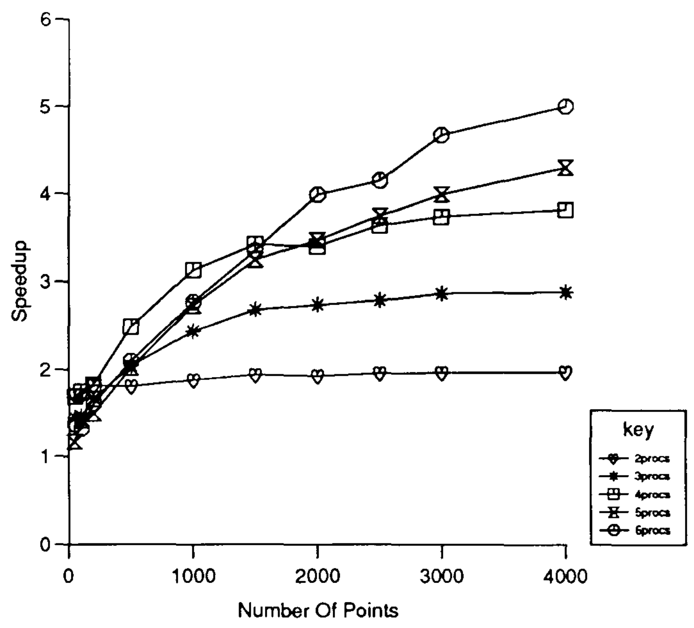

Figure 5.6: Stack Version 2-D 26 Vertices Using Threads

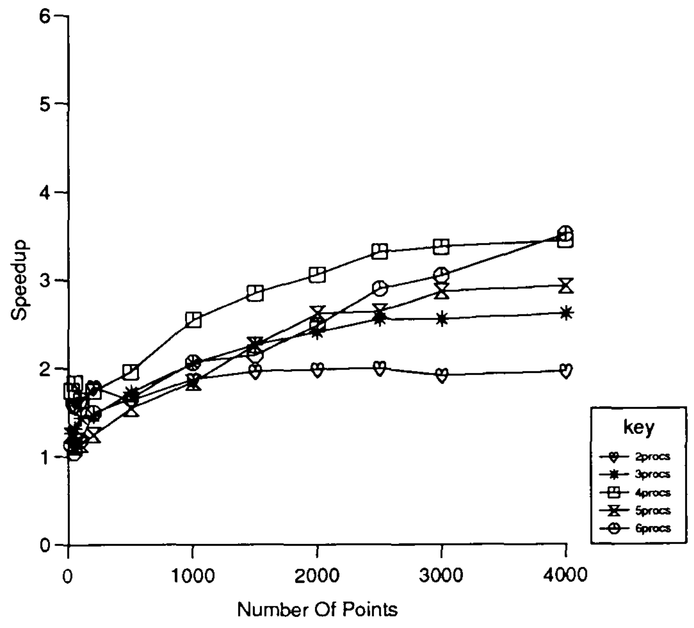

Figure 5.7: Stack Version 4-D 6 Vertices Using Threads

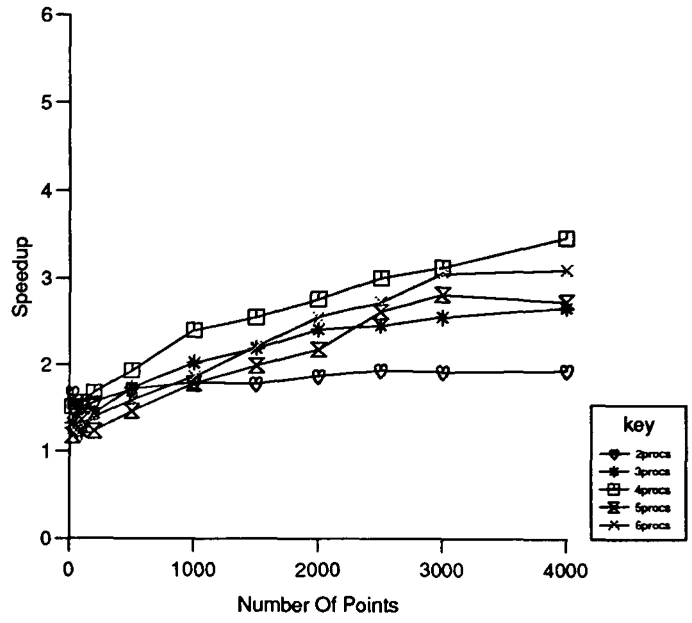

Figure 5.8: Stack Version 3-D 12 Vertices Using Microthreads

- Pipelined or Fixed Size Tree.

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:

```
$R_1, R_2, \ldots, R_p = partition(S, d)$
For $i = 1$ to procs    /* send list to slave i */
{
    csn_tx(masterchan, 0, toslave_id[i], &status, sizeof(status));
    Transmit_Plist($R_i$,n, masterchan ,toslave_id[i]);
}
For $i = 1$ to procs    /* get result from slave i */
{
    csn_tx(masterchan, 0, toslave_id[i], &status, sizeof(status));
    Receive_Plist(&$CH(R_i)$,&m, masterchan ,&fromslave_id[i]);
}
while(procs > 1)   /* loop 1 */
{ /* send list to slaves */
    For $i = 1$ to $\lceil procs/2 \rceil$
    {
        csn_tx(masterchan, 0, toslave_id[i], &status, sizeof(status));
        Transmit_Plist($CH(R_i)$,n, masterchan ,toslave_id[i]);
        Transmit_Plist($CH(R_{\lceil procs/2 \rceil + i})$,n, masterchan ,toslave_id[i]);
    }
    For $i = 1$ to $\lceil procs/2 \rceil$ /* master gets results from slaves */
    {
        csn_tx(masterchan, 0, toslave_id[i], &status, sizeof(status));
        Receive_Plist(&$CH(R_i)$, &m, masterchan,&fromslave_id[i]);
        For $i = 1$ to $\lceil procs/2 \rceil$ do { $R_i = R_i \cup R_{\lceil procs/2 \rceil + i}$;
        $R_{\lceil p/2 \rceil + i} = \emptyset$}
    procs = $\lceil procs/2 \rceil$;
```

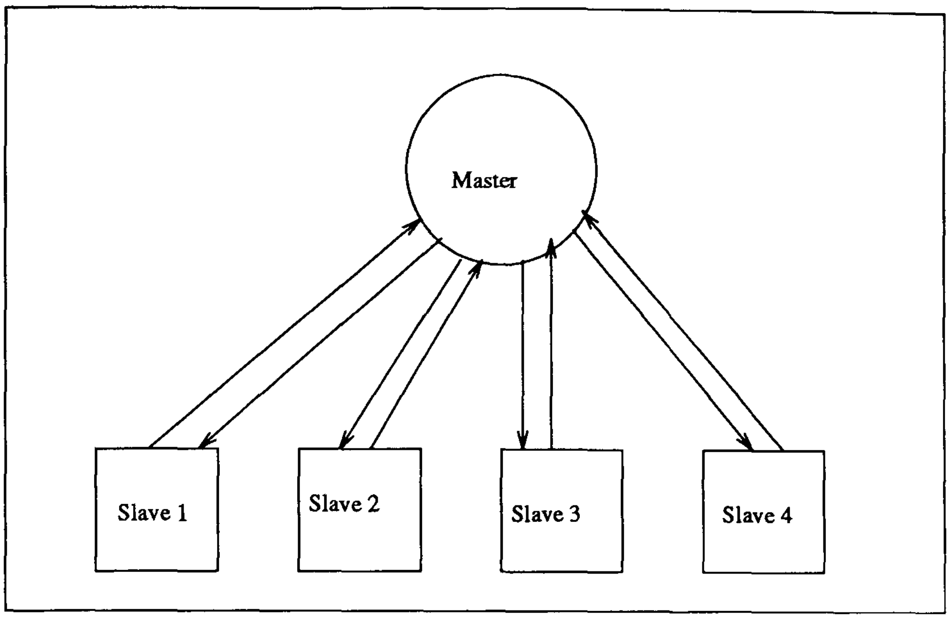

Figure 5.9: A Simulated Tree Implementation

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

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:

```
    $R_1, R_2, \ldots, R_p = partition(S, d)$
    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:

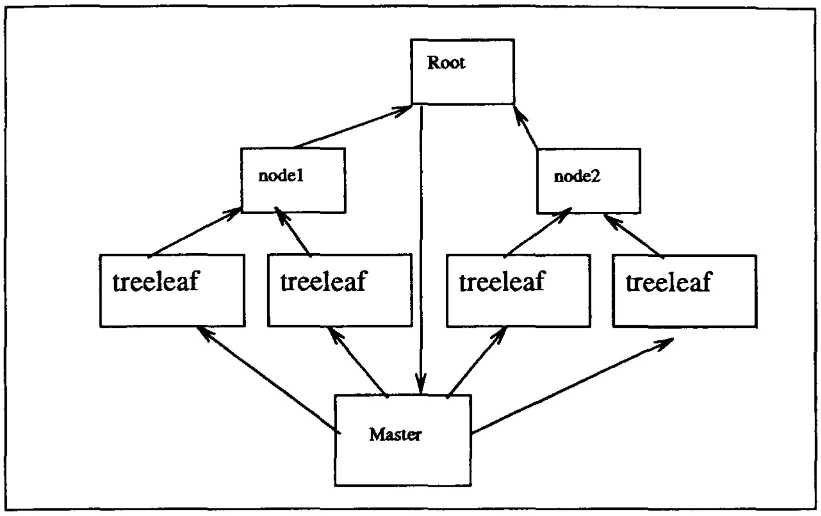

Figure 5.10: Tree In A Distributed Machine

```
Transputer_Hull()
/* get list from the master */
   Receive_Plist(&$R_i$, &n, leaf_in ,NULL);
/* compute the sub convex hull */
   $CH(R_i) = \emptyset$
   if(IsEmpty_Plist($R_i$) = FALSE)
      Generate_Hull($R_i$, n, &CH($R_i$), &FA)
/*  send result to node  */
   Transmit_Plist($CH(R_i)$,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(&$CH(R_i)$, &n1, leaf_in_1 ,NULL);
    Receive_Plist(&$CH(R_t)$, &n2, leaf_in_2 ,NULL);
    if(IsEmpty_Plist($CH(R_i)$) == FALSE)
    {
        while(IsEmpty_Plist($CH(R_t)$) == FALSE)
        {
            CH($R_i$) = $CH(R_i) \cup CH(R_t)$
            $CH(R_t) = \emptyset$
        }
    }   Generate_Hull($CH(R_i)$)
/* send result to the next lower node */
    Transmit_Plist($CH(R_i)$,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 $\bar{p}$ partitions where $\bar{p} >> p$ the number of leaf processors. The treeleaf 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:

```
/* 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 */
```

```
h = (int) (log10(procs)/log10(2)) + 1 ;   /* height of tree -1 */
status = 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
```

```
          {
           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 $\bar{p} >> p$ is used to pipeline partitions through the architecture in blocks of size

$\bar{p}/p$. 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

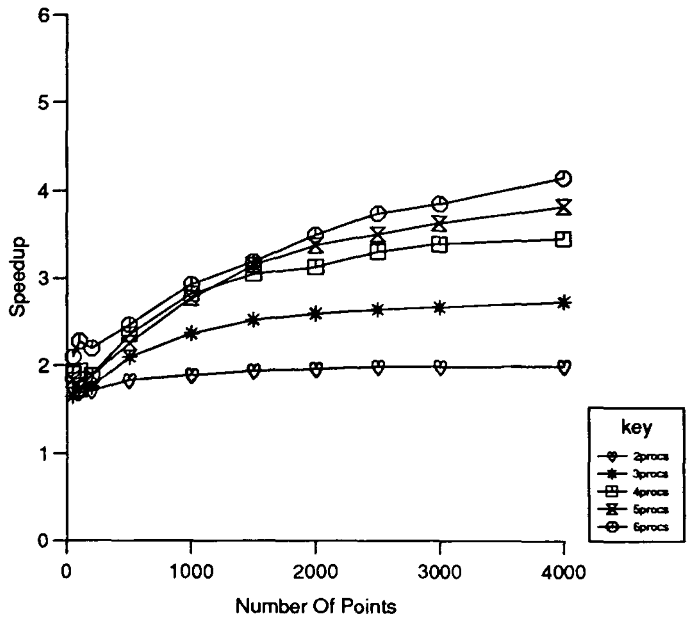

Figure 5.11: Recursion Version 1 2-D 26 Vertices Using Transputer

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

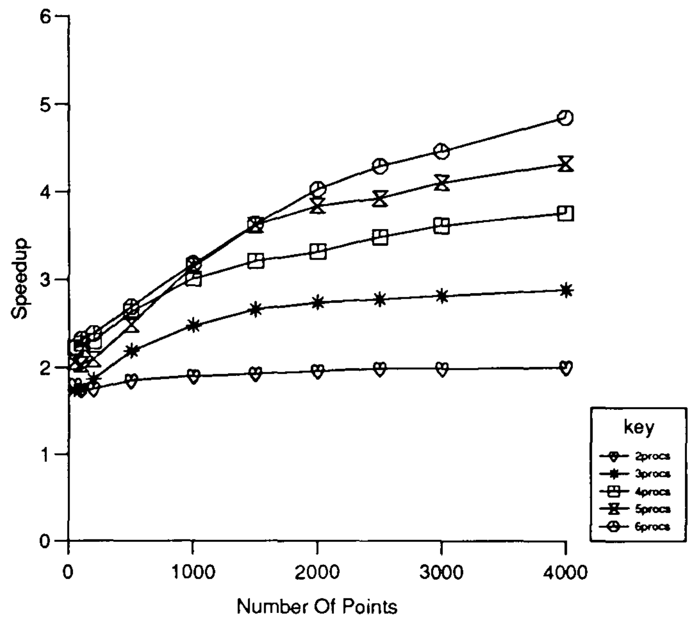

Figure 5.12: Recursion Version 2 2-D 26 Vertices Using Transputer

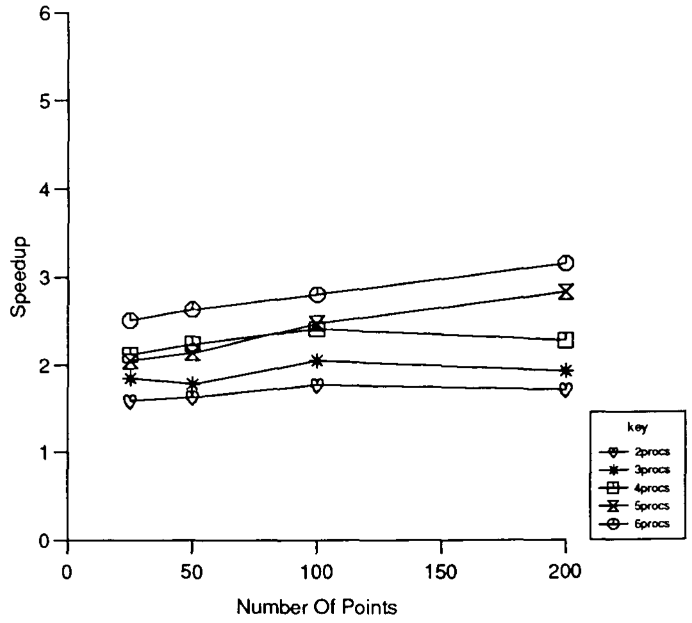

Figure 5.13: Recursion Version 2 4-D 6 Vertices Using Transputer

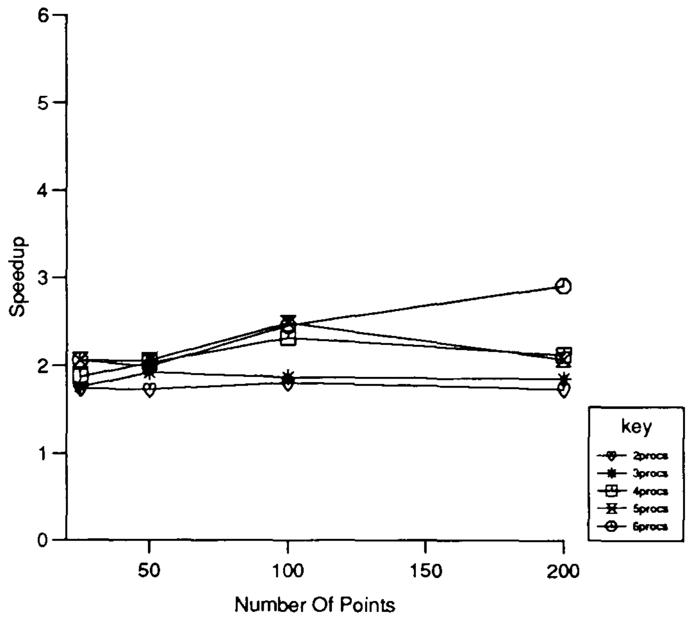

Figure 5.14: Recursion Version 1 4-D 6 Vertices Using Transputer

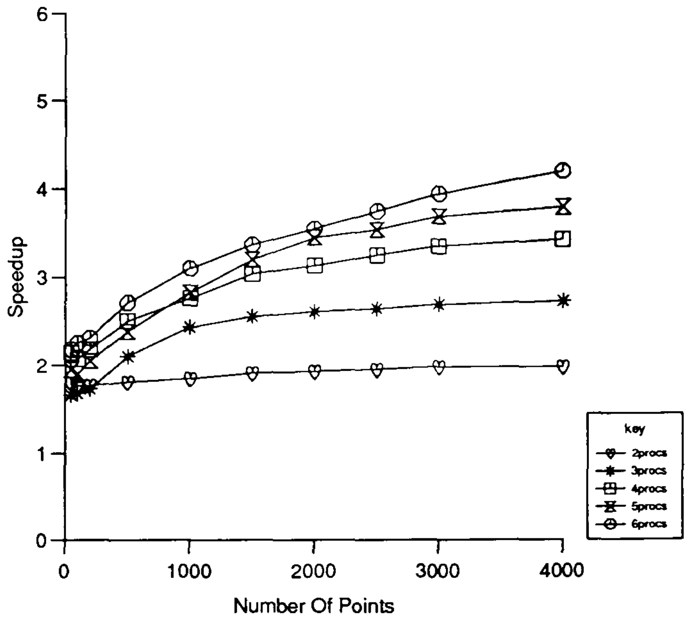

Figure 5.15: Stack Version 1 2-D 26 Vertices Using Transputer

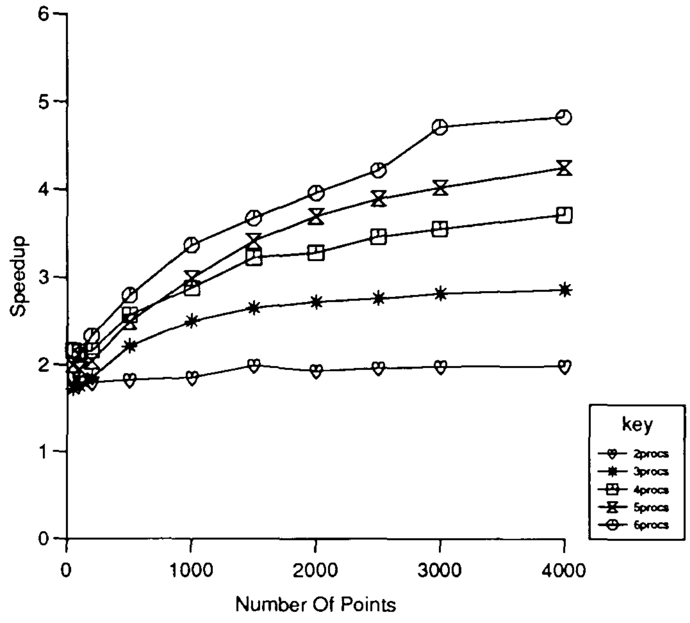

Figure 5.16: Stack Version 2 2-D 26 Vertices Using Transputer

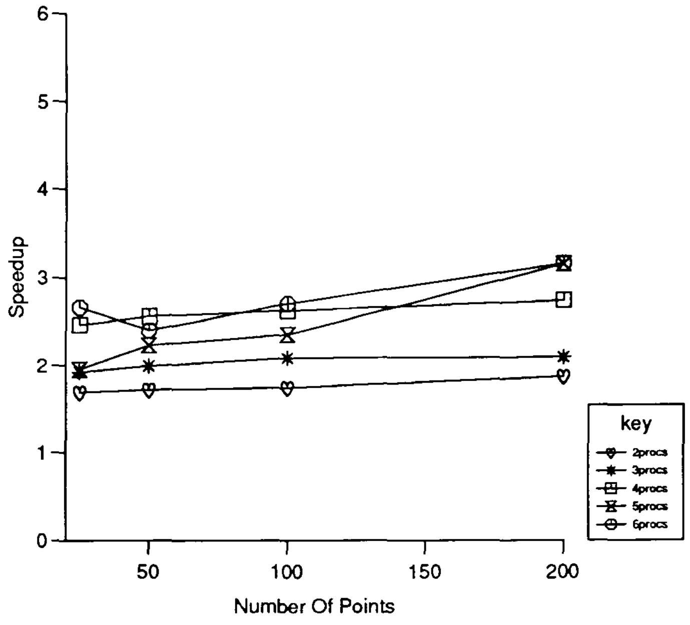

Figure 5.17: Stack Version 2 4-D 6 Vertices Using Transputer

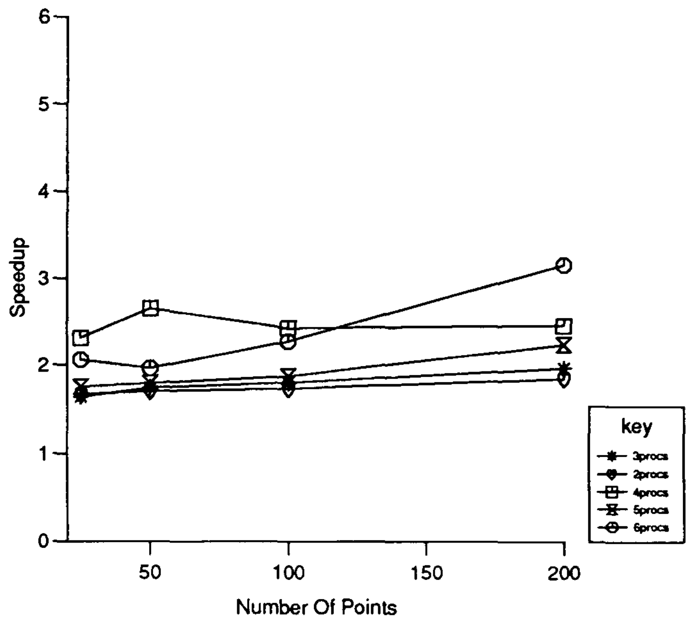

Figure 5.18: Stack Version 1 4-D 6 Vertices Using Transputer

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 ith 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.

```
{
   Quick_Sort(S,n)   /* put $S$ into lexicographic order */
   For i = 0 to MAXPARTITIONS-1
      {
      CHlist[i] = $\emptyset$;  FAlist[i] = $\emptyset$;
      Pointcount[i] = 0;
      }
   For i = 0 to parts-1
      Slist[i] = $\emptyset$
   i = 0;
   while($S \neq \emptyset$)     {
      Take the next point from $S$
      Add point to Slist[i]
      Pointcount[i] = Pointcount[i] + 1
      i = $(i+1)$%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

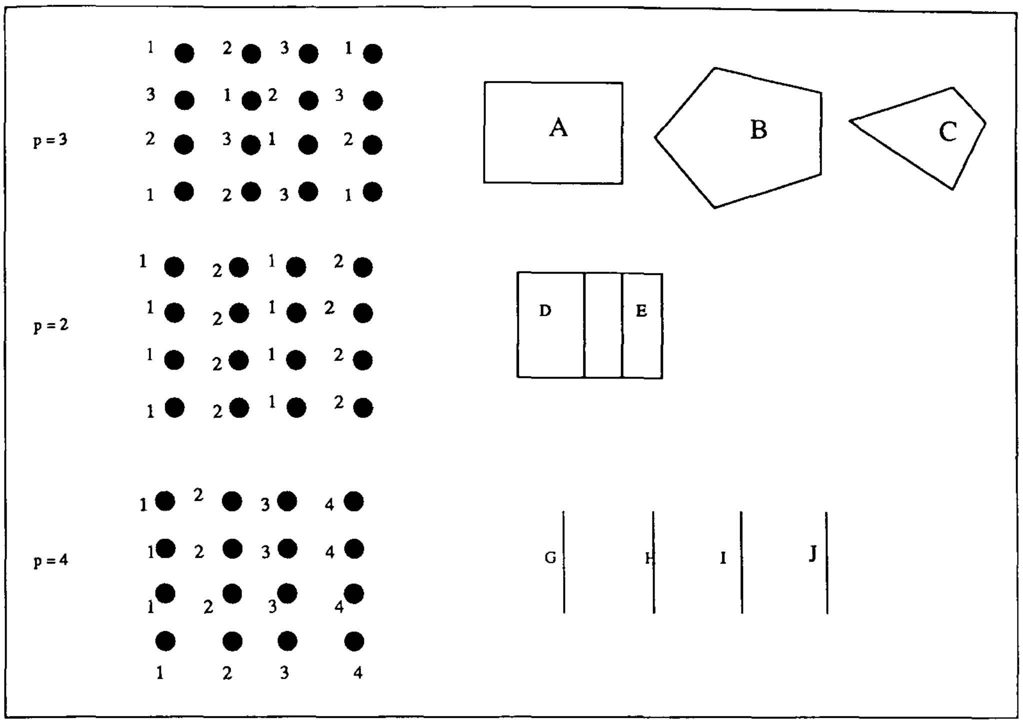

Figure 5.19: Point Allocation In Lex Partitioning

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 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)parts-1.

```
{
   For i = 0 to MAXPARTITIONS-1
   {
      CHlist[i] = $\emptyset$; FAlist[i] = $\emptyset$;
   }
```

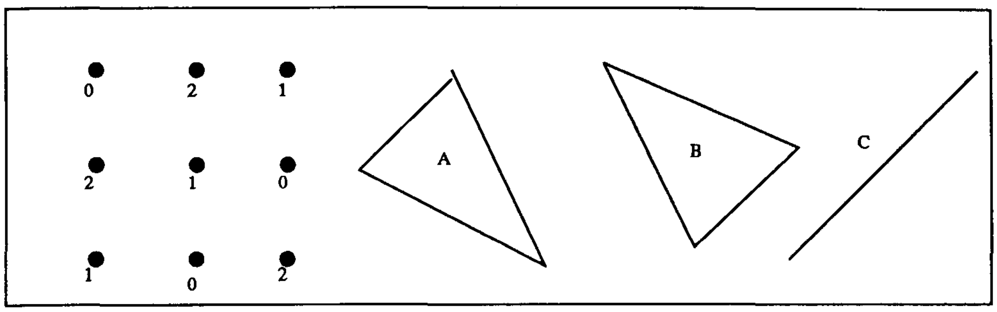

Figure 5.20: Allocation Of Points In Random Colouring

```
    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

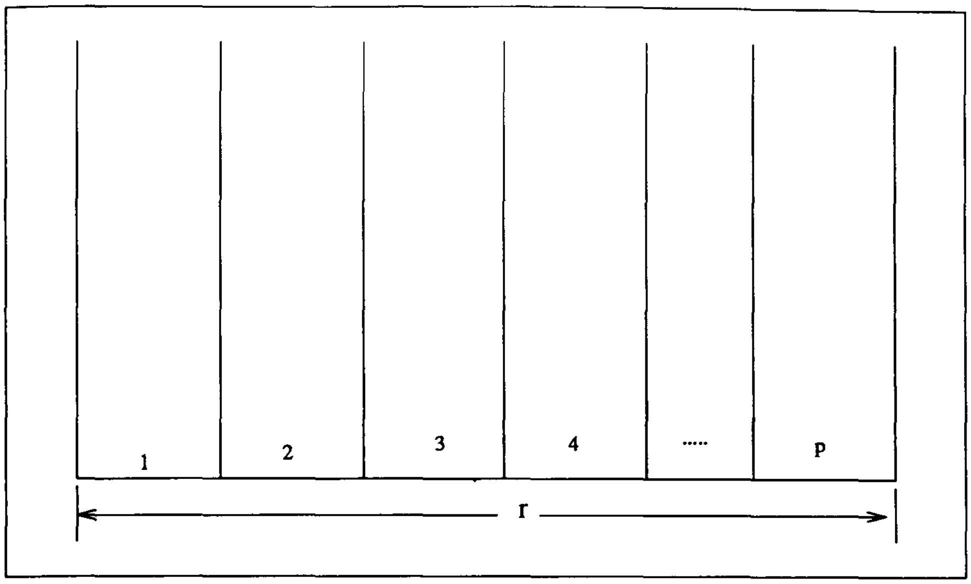

Figure 5.21: Partitioning of 2-D Plane

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 $c_1 n$ steps. The longest euclidean distance r between c and points $x \in S$ is calculated and shell i

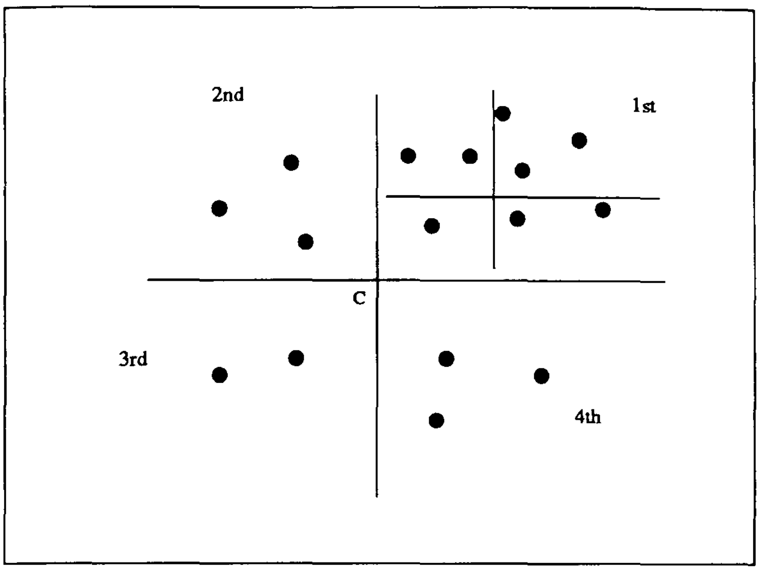

Figure 5.22: Distribution Of Points Into Buckets

is determined according to the bounds $(i-1)r/p$ and ir/p. This will require $c_2 n$ steps. Points are allocated to partitions according to the shell they inhabit and this can be accomplished in $c_3 n$ 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

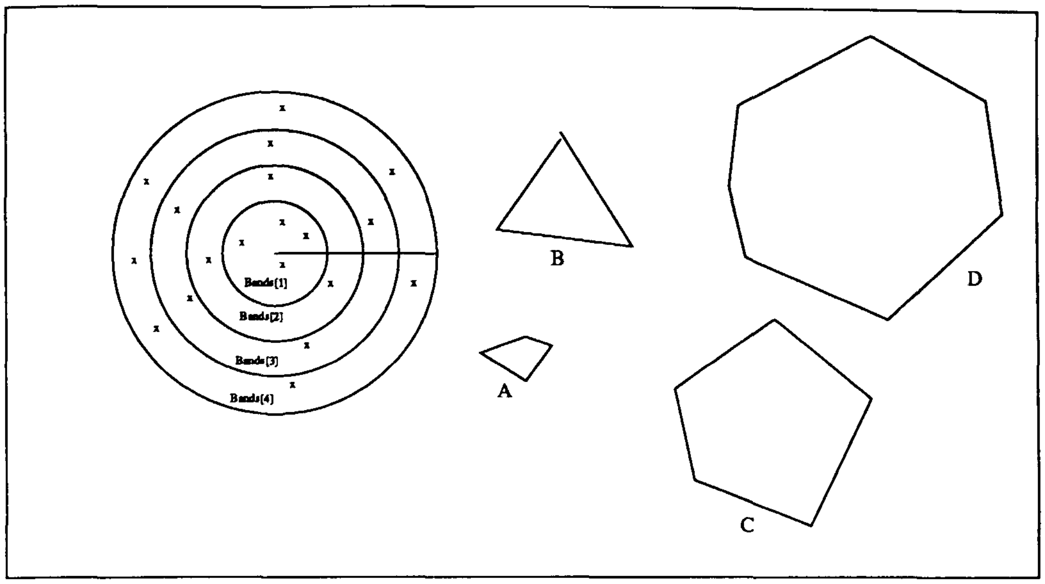

Figure 5.23: Allocation Of Points To Shells With Convex Hull On One Band

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].

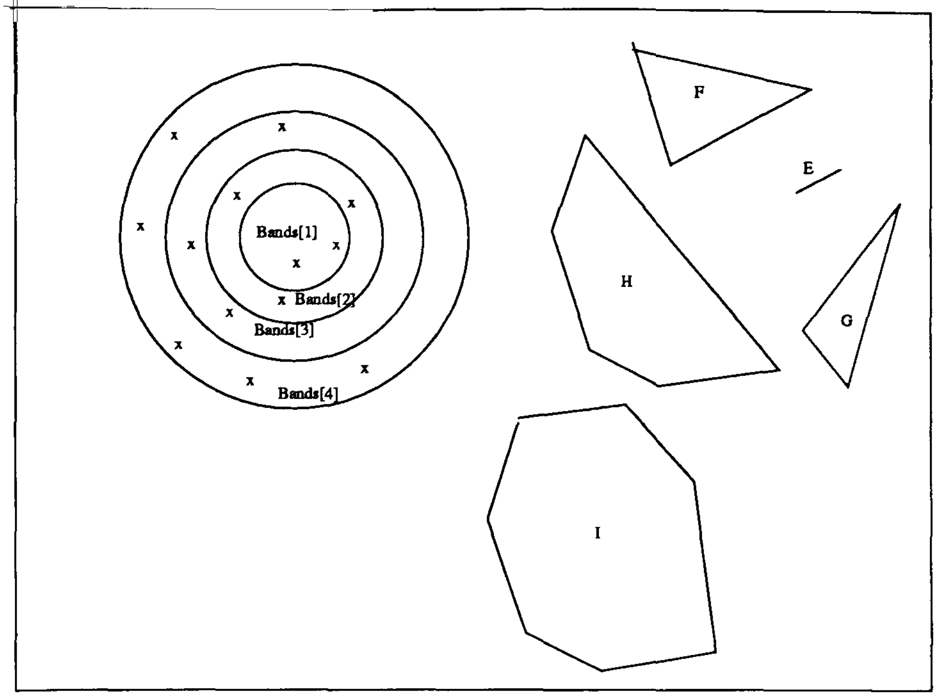

Figure 5.24: Allocation Of Points To Shells With Convex Hull Across Bands

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(max)/parts;
   Bands[0] = 0;
   For i = 1 to parts-1
      Bands[i] = Bands[i-1] + r;
   For i = 0 to MAXPARTITIONS-1
   {
      CHlist[i] = $\emptyset$;  FAlist[i] = $\emptyset$;
   }
   For i = 0 to parts-1
   {
        Slist[i] = $\emptyset$
        Pcount = 0;
    }
/* Partition $S$ according to distance Bands[i-1] <= r < Bands[i] inserted into Slist[i-1] */
    while($S \neq \emptyset$)
    {
        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

$$Vol = \frac{\sqrt{\pi^n}}{\Gamma(\frac{n}{2}+1)} R^n$$

where n is the space dimension, R the radius of sphere and Γ 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

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

Methods	*Partitions*									
	2		3		4		5		6	
	2D	4D	2D	4D	2D	4D	2D	4D	2D	4D
Lex (sp)	1.89	1.79	2.57	2.14	3.12	2.49	2.87	2.12	2.83	1.89
Rand (sp)	1.61	1.78	2.53	2.00	3.47	2.30	2.68	1.67	1.86	1.71
Shell (sp)	1.74	1.29	1.17	1.56	1.86	1.73	1.73	1.46	1.86	1.80
New_Shell (sp)	1.05	0.98	1.28	1.02	1.66	0.97	1.86	0.94	2.02	0.97
Bucket (sp)	1.41	1.11	1.29	1.03	2.38	1.18	2.04	1.07	2.27	1.47

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 5.2: Statistics For Partitioning Methods From Table 39

Methods		2		3		4		5		6	
		2D	4D	2D	4D	2D	4D	2D	4D	2D	4D
Lex	SD	0.0	0.0	0.58	0.58	0.0	0.0	0.0	0.0	0.52	0.52
	$\bar{X}$	500	500	333.3	333.3	250	250	200	200	166.7	166.7
Random	SD	0.0	0.0	14.7	14.7	0.0	0.0	11.18	11.18	13.37	13.37
	$\bar{X}$	500	500	333.3	333.3	250	250	200	200	166.7	166.7
Shell	SD	58	147.1	221.5	141.5	146.8	145.5	115.6	116.8	106.1	100.9
	$\bar{X}$	500	500	333.3	333.3	250	250	200	200	166.7	166.7
New_Shell	SD	473.8	688.7	261.0	532.7	196.9	430.0	154.1	367.1	126.51	319.4
	$\bar{X}$	500	500	333.3	333.3	250	250	200	200	166.7	166.7
Bucket	SD	158.4	418.6	243	407.5	68.6	376.5	126.6	337.3	139.6	190.7
	$\bar{X}$	500	500	333.3	333.3	250	250	200	200	166.7	166.7

(*Partitions* spans the columns 2–6.)

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. Altenatively, 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

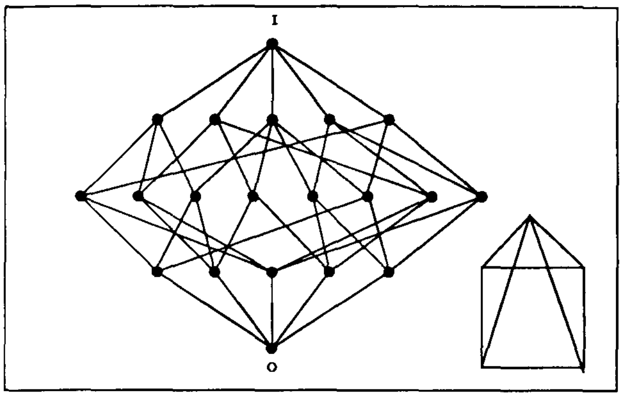

Figure 6.1: Facial lattice of a pyramid over a square

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

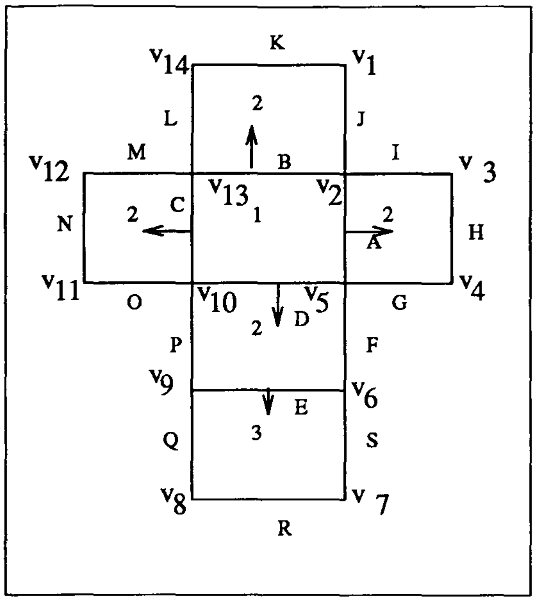

Figure 6.2: Illustration Of Parallel Execution Of 3D Convexhull by FLE

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

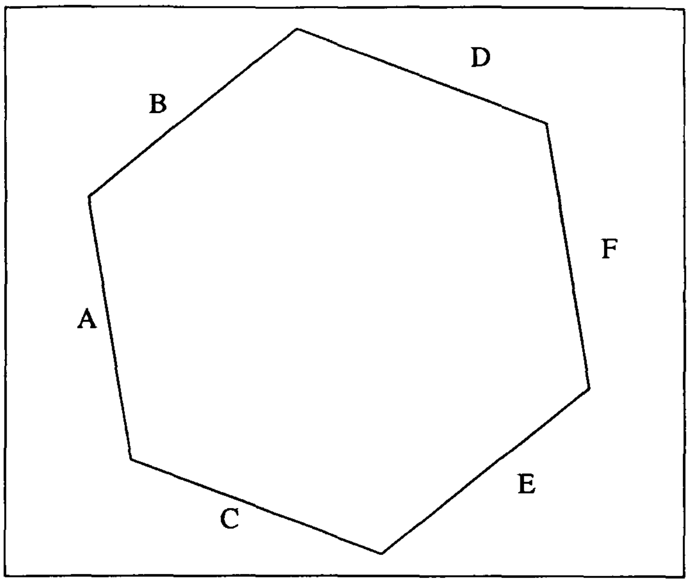

Figure 6.3: Edge Computation In A 2-D Problem

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

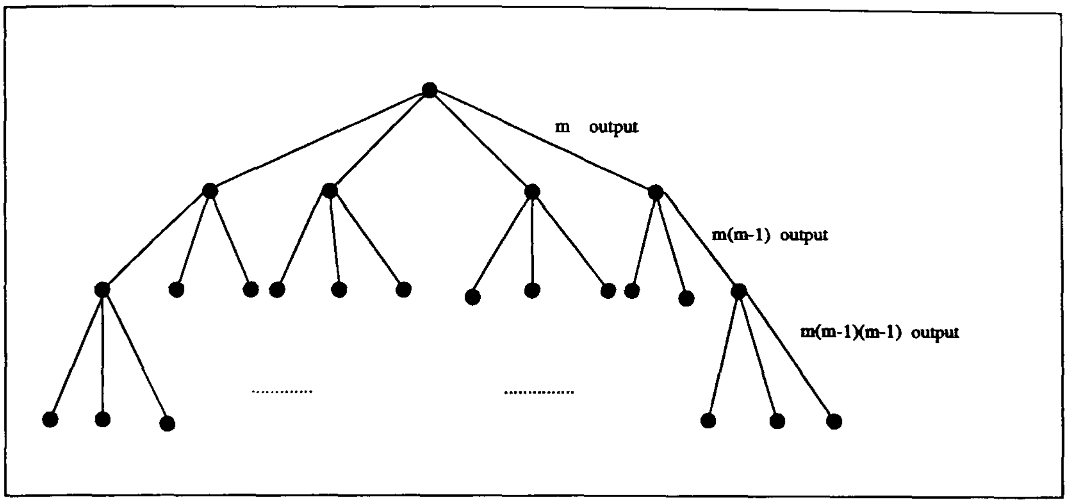

Figure 6.4: Exploiting Parallelism With Unlimited Number Of Processors

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_Elist 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 = { $p_0, p_1$ };
    min = $p \in$ S such that $\vec{p_0p_1}.\vec{p_0p_1}$ is minimised
    max = $p \in$ S such that $\vec{p_0p_1}.\vec{p_0p_1}$ is maximised
    return ({max,min} , {max}, {min});
    }
  if ($|S| == k+1$)
    return ($S, \{F \subseteq S : |F| = k\}$); /* check for a simplex */
  else
      {
      (F,$\hat{n}$) = initial facet(Stack[sp].S,Stack[sp].AS,n,Stack[sp].k)
      Stack[sp].FA = Stack[sp].FA ∪ { F }
      $F' = \emptyset$;
      Pick a point $p_0 \in F$;
      $F' = \{p \in S : \vec{p_0p_1}.\hat{n} = 0\}$ ;
      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 = ∅; Stack[sp].CH = ∅; Stack[sp].FA = ∅;
      Stack[sp].Pend_Elist = ∅;
      }
  }
  /* unstack the face completed by master */
  if (Stack[sp].CH ≠ ∅ and Stack[sp].FA ≠ ∅)
    while(Stack[sp].Elist = ∅ and Stack[sp].e == 0 and sp > 0)
         {
         while(Stack[sp].CH ≠ ∅)
         { /*  add new vertices found  */
         pick a point p;
         if (p ∉ Stack[sp-1].CH)
           Stack[sp-1].CH = Stack[sp-1].CH ∪ { p };
         }
    while(Stack[sp].FA ≠ ∅)
    { /*  add new edges found using norm of complete face  */
    pick an Edge E;
    if (E ∉ Stack[sp-1].Pend_Elist
      {
      if (E ∈ Stack[sp-1].Elist
        { remove E from Stack[sp-1].Elist
        and add to Stack[sp-1].Pend_Elist }
      else
          Stack[sp-1].Elist = Stack[sp-1].Elist ∪ { E }
      }
    }
     /*  Unstack the face  */
    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 $\neq \emptyset$) /* get next face for the master */
   {
   Pick an Edge E;
   (p,$\hat{n}$) = rotate(S,AS,n,k,E,$\hat{n}$);
   F = F $\cup$ { p };
   Stack[sp].Pend_Elist = Stack[sp].Pend_Elist $\cup$ {E}
   Stack[sp].FA = Stack[sp].FA $\cup$ {F}
   Pick a point $p_0 \in$ F
   $F' = \{p \in S : \vec{p_0 p_1}.\hat{n} = 0\}$;
   /* 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 = $\emptyset$;
   Stack[sp].Elist = $\emptyset$; Stack[sp].FA = $\emptyset$; Stack[sp].CH = $\emptyset$;
   }
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 $\neq \emptyset$)
        lsp = i;  break;
      }   if (Stack[lsp].Elist $\neq \emptyset$)
      {
        Pick an Edge E;
        Stack[lsp].Pend_Elist = Stack[lsp].Pend_Elist $\cup$ {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 = $\emptyset$;
                Slaves[count].FA = $\emptyset$;
                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 $\neq \emptyset$ 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   */
  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]);
  while(status $\neq$ STOP)
    {
    if(status > START and status < RESULT)
      {
    if(S $\neq \emptyset$ )
      (p,$\hat{n}$) = rotate(S,AS,n,k,F,$\hat{n}$);
      F = F $\cup$ {p};
      $F' = \emptyset$;
      Pick a point $p_0 \in$ F;
      $F' = \{p \in S : \vec{p_0 p}.\hat{n} = 0\}$;
      CH = $\emptyset$; FA = $\emptyset$;
      Convex_Hull($F'$,F,n,k-1) : (CH,FA);
    /*  signal result is valid   */
      THREADpsem(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 6.1: Movement Of Edges Into Pend_List

List	Master	Slave 1	Slave 2	Slave 3
Edgelist	ABCD			
Pend_List				
Faces	(ABCD)			
Vertices	$v_2 v_{13} v_{10} v_5$			
Edgelist	BCD	A		
Pend_List	A			
Faces	(ABCD)			
Vertices	$v_2 v_5 v_{10} v_{13}$			
Edgelist	CD	A	B	
Pend_List	AB			
Faces	(ABCD)	AIHG		
Vertices	$v_2 v_5 v_{10} v_{13}$	$v_2 v_3 v_4 v_5$		
Edgelist	D	A	B	C
Pend_List	ABC			
Faces	(ABCD)	AIHG	BJKL	
Vertices	$v_2 v_5 v_{10} v_{13}$		$v_2 v_1 v_{14} v_{13}$	
Edgelist	IHGPEF		B	C
Pend_List	ABCDG			
Faces	(ABCD)(DPEF)(AIHG)		BJKL	CMNO
Vertices	$v_2 v_5 v_{10} v_{13} v_6 v_9 v_3 v_4$		$v_2 v_1 v_{14} v_{13}$	$v_{13} v_{12} v_{11} v_{10}$
Edgelist	IHPEJKL			C
Pend_List	ABCDGJ			
Faces	(ABCD)(DFEP)(AIHG)(BJKL)			(CMNO)
Vertices	$v_2 v_5 v_{10} v_{13} v_6 v_9 v_3 v_4 v_1 v_{14}$			

Table 6.2: Movement Of Edges Into Pend_List (Table 6.1 Cont.)

List	Master	Slave 1	Slave 2	Slave 3
Edgelist	HPEKLMNO			
Pend_List	ABCDGJM			
Faces	(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)			
Vertices	$v_2v_5v_{10}v_{13}v_6v_9v_3v_4v_1v_{14}v_{12}v_{11}$			
Edgelist	EKN	H		
Pend_List	ABCDGJMP			
Faces	(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)			
Vertices	$v_2v_5v_{10}v_{13}v_9v_4v_1v_{12}v_{11}$			
Edgelist	KN	H	E	
Pend_List	ABCDGJMP			
Faces	(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)	EKNH		
Vertices	$v_2v_5v_{10}v_{13}v_9v_4v_1v_{12}v_{11}$			
Edgelist	NEKNH		E	K
Pend_List	ABCDGJMPN			
Faces	(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)(EKNH)		EPDF	
Vertices	$v_2v_5v_{10}v_{13}v_9v_4v_1v_{12}v_{11}$			
Edgelist	EKHEPDF			K
Pend_List	ABCDGJMPNEH			
Faces	(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)(EKNH)			KJBL
Vertices	$v_2v_5v_{10}v_{13}v_9v_4v_1v_{12}v_{11}$			
Edgelist	KFKJBL	H	E	
Pend_List	ABCDGJMPNEKH			
Faces	(ABCD)(DFEP)(AIHG)(BJKL)(CMNO)(ENKH)			
Vertices	$v_2v_5v_{10}v_{13}v_9v_4v_1v_{12}v_{11}$			

Table 6.3: Results For FLE Version 1

Sequential	Procs	Parallel	Remarks
19039752	2	18558925	4D 6 vertices Using 1000 points
	3	17933964	
	4	18911185	
	5	18722954	
	6	18811114	
24953393	2	20752284	3D 29 vertices Using 1000 points
	3	19728550	
	4	19115759	
	5	18871582	
	6	18993580	
16168211	2	14596503	2D 26 vertices Using 1000 points
	3	14756803	
	4	14766466	
	5	15038233	
	6	15154157	

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

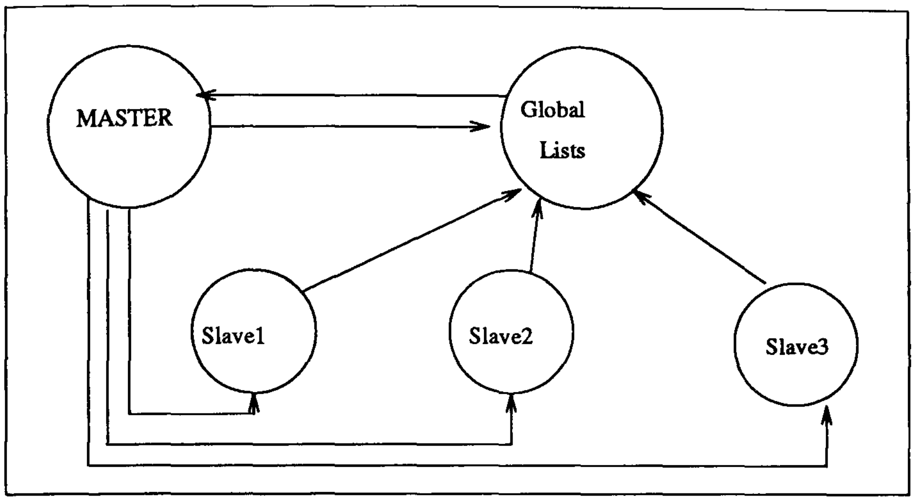

Figure 6.5: FLE Implementation Using Global Lists

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, P0, P1, 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;
                    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;
     };
    THREADvsem(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, P0;
  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 \n")
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

Points	*Seq.*	*Processors*				
		2	3	4	5	6
10	61946	85050	97208	108145	118386	138746
20	165693	165057	182230	198029	211846	241288
30	291058	287509	300129	318960	350037	342996
50	645248	574529	609818	624019	636293	650159
100	2146766	1773448	1843447	1867255	1881780	1918853
150	4516176	3679362	3772718	3823652	3913955	3851436
250	12437120	9728746	9854043	9987508	9755902	10081848
350	23144612	18363471	18626296	18832709	18764585	18461995

Table 6.5: Timing for 3-D Circular Structure

Points	*Seq.*	*Processors*				
		2	3	4	5	6
10	568741	459748	416767	396144	404662	405506
30	1899102	1614474	1441846	1232050	1164047	1202158
50	3606038	3126744	2621719	2286960	2050852	2124603
100	10273413	8604516	7080557	6295752	5684705	5523024
120	15126292	11963384	9356036	8160797	7555822	7388448

Table 6.6: Timing for 3-D Pyramidal Structure Generating Rectangle In Levels

Points	*Seq.*	*Processors*				
		2	3	4	5	6
40	1578688	1213436	1052820	932388	926521	894041
80	5005574	3674417	2784585	2498091	2289742	2190818
120	10405410	7076664	5339403	4647789	4067981	3812792

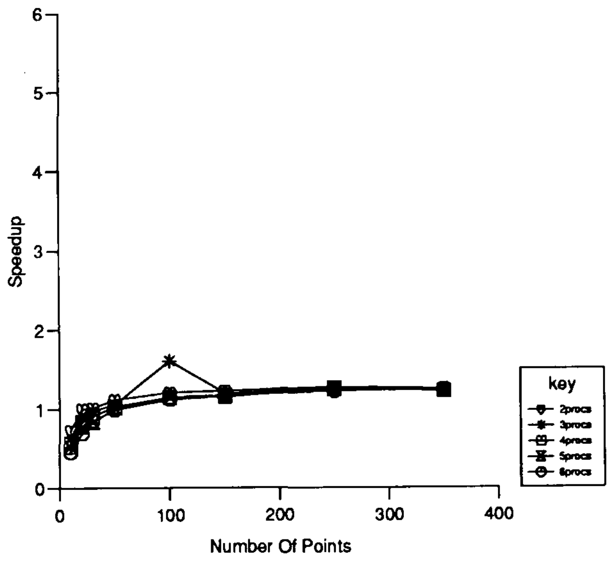

Figure 6.6: FLE For 2-D On Shared Memory

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 preceeding 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

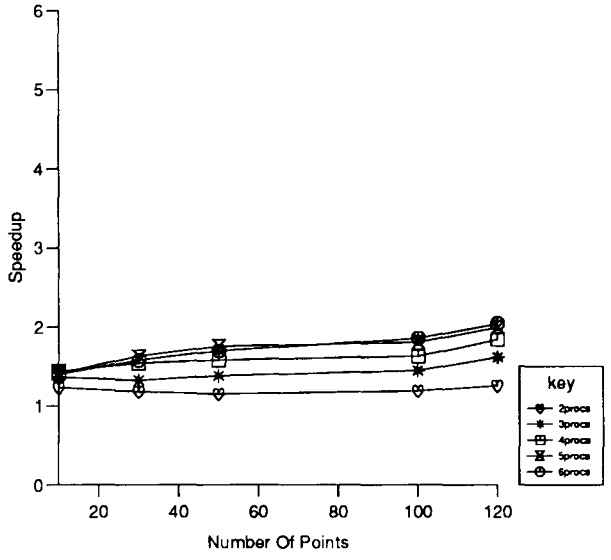

Figure 6.7: FLE For 3-D Circular Shape On Shared Memory

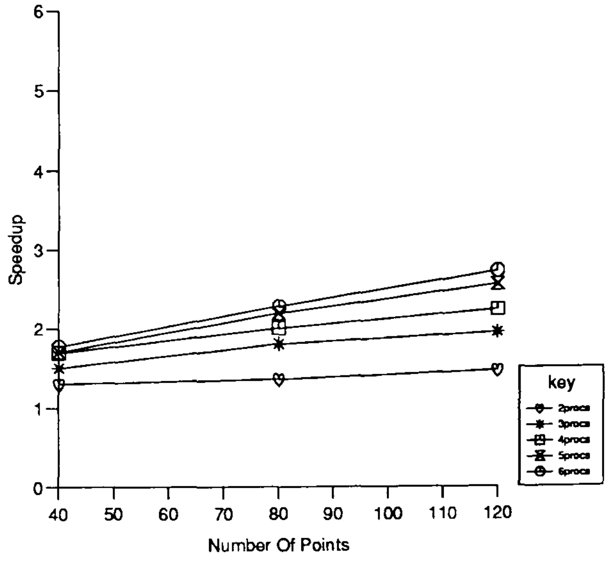

Figure 6.8: FLE For 3-D Pyramidal Shape On Shared Memory

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

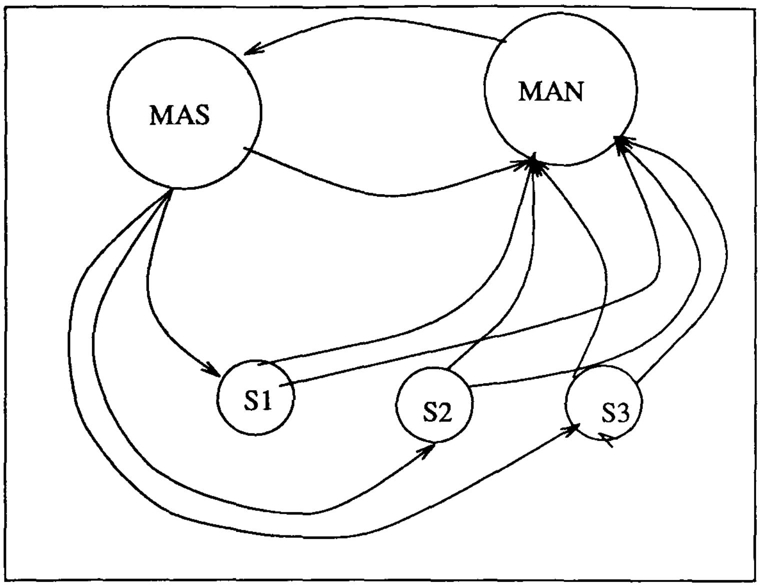

Figure 6.9: Communication Between Processes on Different Transputers

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.

```
/* 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 */
```

```
  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);
      };

     /* run out of edges and synchronise slaves   */

     while( slave_count != parts)
       {
          csn_rx(masterchan, NULL, &status, sizeof(status));
          Svector[slave_count] = status;
          slave_count = slave_count + 1;
        };

     /* ask manager for new edge list and set new_face flag */

     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));

     /*  master completes computation   */

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


   /* recieve result and shutdown */

   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)
     {
      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 */

      csn_rx(slavechan_from_manager, NULL, &new_face, sizeof(new_face));
   /*   manager checked norm list  and process it   */

      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);
  /* check orignal 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_Plist(&GCH, &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,
                                              sizeof(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 */

 for(i=0; i<p; i++)
 if (junk_id == toslave_id[i]) break;
 switch(status)
  {
   case CHECKNM : /* check global norm list */
   csn_rx(aux_managerchan, &junk_id, &norm, sizeof(norm));
   if (IsMember_Plist(GNORMS, n, norm) == FALSE)
    {
      new_face = TRUE;
      GNORMS = Insert_Point(GNORMS, n, norm);
    }
   else
    {
     new_face = FALSE;
     csn_tx(managerchan, 0, master_id, &i, sizeof(i)); /* free 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;
    };
  };
```

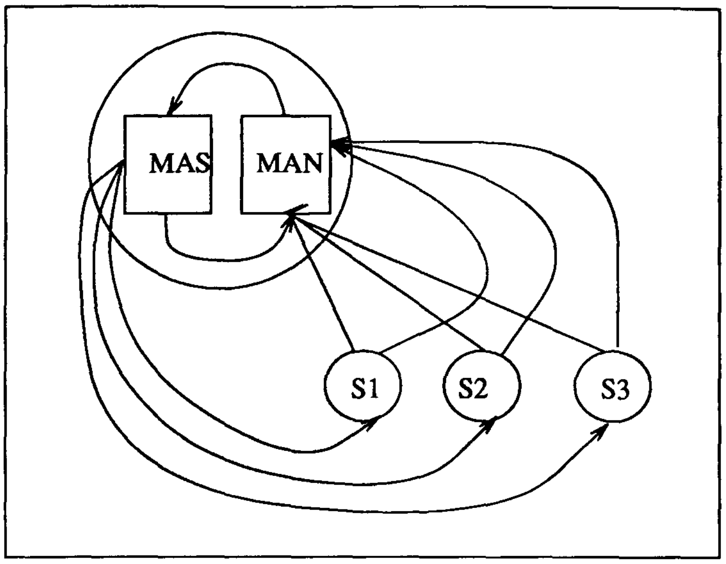

Figure 6.10: Master And Manager Run On Same Transputers

```
 }while(status != STOP);
}
```

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

Table 6.7: Timing for 2-D

Size	*Sequential*	*Parallel*	*Processors*				
			3	4	5	6	7
10	92160	Ver 1	167616	150528	151232	161216	170048
		Ver 2	150016	156416	162752	169088	175424
20	232896	Ver 1	376512	324864	324544	341120	356736
		Ver 2	324096	334336	343936	354880	365760
31	433920	Ver 1	646976	545920	545152	569856	591232
		Ver 2	545280	559808	574208	589952	603776
50	951232	Ver 1	1317504	1075584	1073216	1112512	1144768
		Ver 2	1074880	1096832	1119488	1138240	1163712
100	3136320	Ver 1	3926016	3079296	3077120	3152576	3211328
		Ver 2	3078528	3120640	3155392	3200704	3247488
350	32472128	Ver 1	36438336	27231552	27229184	27488960	27675840
		Ver 2	27799104	27695232	27534272	27373760	27231360

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 6.8: Timing for 3-D Pyramidal Structure

Size	*Sequential*	*Parallel*	*Processors*				
			3	4	5	6	7
40	2448640	Ver 1	3015744	2061120	1827584	1753024	1903680
		Ver 2	2124864	1816384	1896192	1802752	1989632
80	7668288	Ver 1	8721792	5818432	5053056	5012352	4941760
		Ver 2	5905664	5313536	4913472	4940032	5074176
120	15429632	Ver 1	16701248	10794688	9264832	9319040	9371904
		Ver 2	11174464	5963136	9363584	9366848	9878272

Table 6.9: Timing for 3-D Circular Structure

Size	*Sequential*	*Parallel*	*Processors*				
			3	4	5	6	7
12	804736	Ver 1	1079552	739648	638592	656064	625536
		Ver 2	771200	689664	671360	629504	636224
33	2952446	Ver 1	3905664	2564224	2196096	2113536	2065472
		Ver 2	2720960	2364800	2254016	2104704	2016448

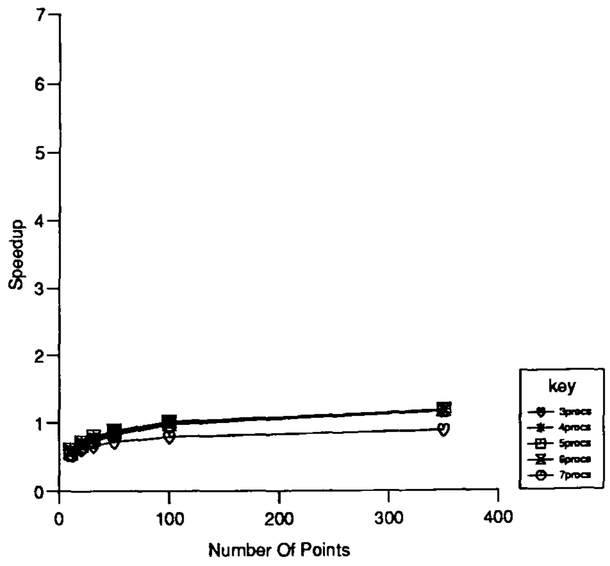

Figure 6.11: FLE For 2-D On Transputer (Ver 1)

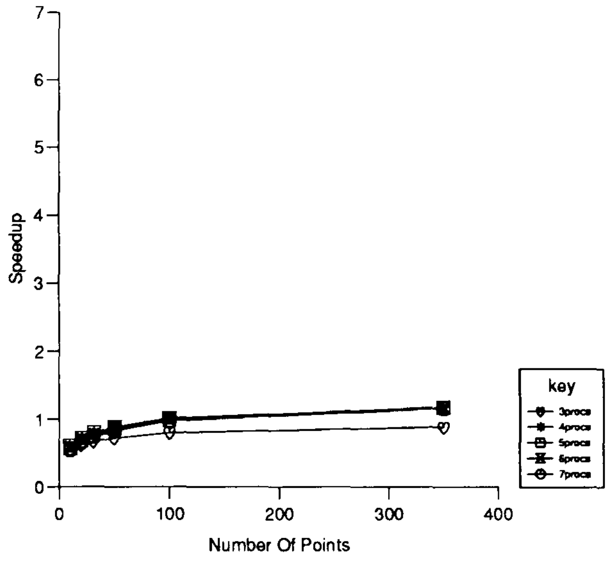

Figure 6.12: FLE For 2-D On Transputer (Ver 2)

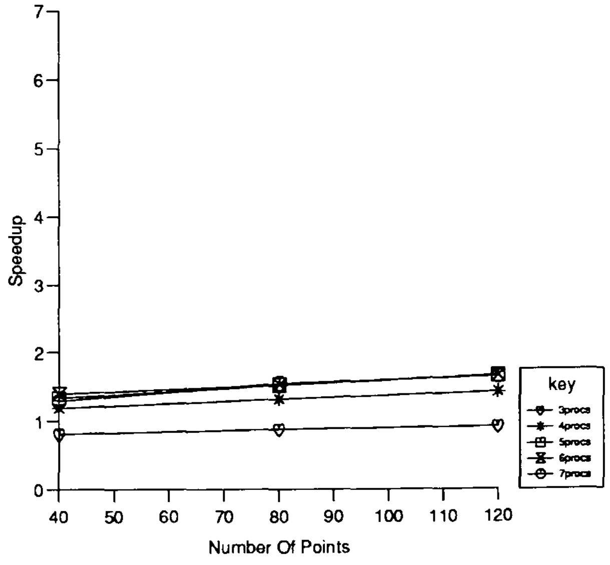

Figure 6.13: 3-D Circular Shape On Transputer (Ver 1)

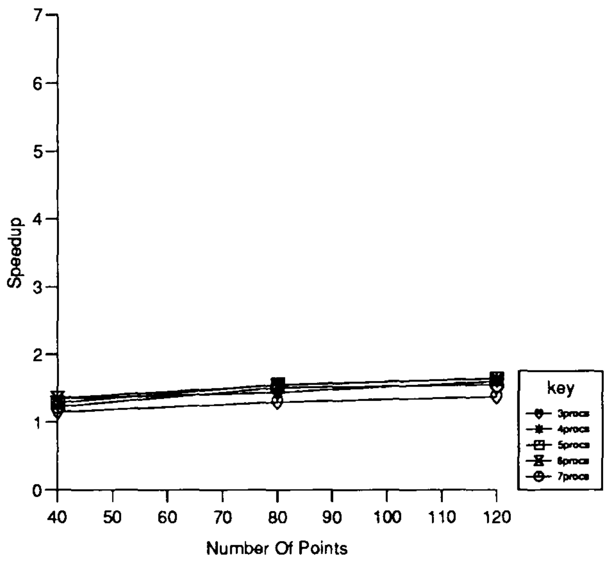

Figure 6.14: 3-D Circular Shape On Transputer (Ver 2)

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 summaried 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 straight forward 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

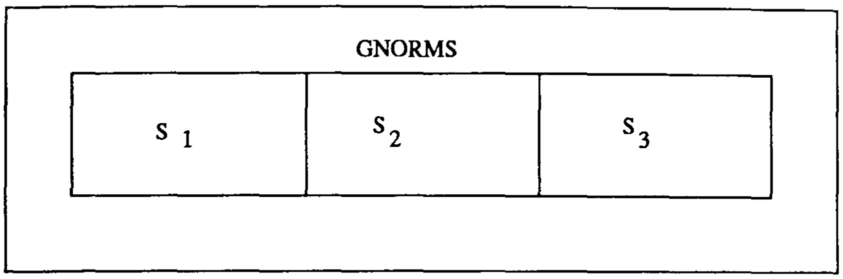

Figure 7.1: Partitioning of GNORMS into three sublists

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

[1] Chand,D.R.; and Kapur,S.S.; *An Algorithm For Convex Polytopes*, Journal Of ACM Vol.17 (1) (1970) pp 78–86

[2] Swart, G.; *Finding The Convex Hull Facet By Facet*, Journal Of Algorithms 6 (1985) pp 17–48.

[3] Lee D. T.; and Preparata F. P.; *Computational Geometry - A Survey*, IEEE Trans. On Computers C-33 (12) (1984) pp 1072-1101.

[4] Kay T. L.; and Kajiya J. T.; *Ray tracing complexes scenes*, Computer Graphics Vol.20(4) (1986) pp 269–278.

[5] Freeman H.; *Computer processing of line drawing images*, Computing Surveys Vol.6 (1974) pp 57–97.

[6] Megson,G.M.; and X.Chen; *Partitioning And Mapping For Lower Dimensional Given Regular Arrays*, Proceedings Euromicro Workshop On Parallel And Distributed Processing, Malaga, Spain (Jan. 1994) ppl 49–155.

[7] Flynn,M.J.; *Very High Speed Computing Systems.* Proceeding Of The IEEE , Vol.54, No.12, Dec. 1966, pp 1901–1909.

[8] Decegama,A.L.; *The Technology Of Parallel Processing - Parallel Processing Architectures And VLSI Hardware Vol. 1* , Prentice Hall 1989.

[9] Duncan,R.; *A Survey Of Parallel Computer Architectures.* Computer , Vol.23 .No.2 IEEE Feb. 1990, pp 6–16.

[10] Skillicorn,D.B.; *A Taxonomy For Computer Architectures.* Computer , Vol.21, No.11 IEEE Nov. 1988.

[11] Feng,T.Y.; *Parallel Processors and Processing,* ACM Computing Surveys, Vol. 9 No. 1 1977.

[12] Reddi,S.S.; and Feurstel,E.A.; *A Conceptual Framework For Computer Architectures.* Computing Surveys , Vol.8, No.2 June 1976, pp 277–300.

[13] Händler,W; *The Impact Of Classification Schemes On Computer Architecture.* Proc. Int. Conf. On Parallel Processing , Aug.1977, pp 7–15.

[14] Evans,D.J.; *Parallel Processing — Its Use In All Levels Of Processing Operations.* Data Processing , Vol.28, No.10, Dec. 1986, pp 529–542. [Butterworths & Co. (Publishers) Ltd.]

[15] Kung,H.T.; *Why Systolic Architectures?* Computer , Vol.15, No.1, Jan.1982, pp 37–46.

[16] Neumann,J.von.; *The Computer and the Brain,* Yale University Press, New Haven, 1958.

[17] Hayes,J.; Mudge,T.N.; Stout,Q.F.; Colley,S.; and Palmer,J.; *A Microprocessor-Based Hypercube Supercomputer.* IEEE Micro , Vol.6, 1986, pp 6–17.

[18] Gustafson,J.L.; Hawkinson,S.; and Scott,K.; *The Architecture Of A Homogeneous Vector Supercomputer.* Proc. Int. Conf. On Parallel Processing , 1986, pp 649–652. [IEEE Computer Society Press.]

[19] Veen,A.H.; *Dataflow Machine Architecture.* ACM Computing Surveys , Vol.18, No.4 Dec.1986, pp 365–396.

[20] Treleaven,P.C.; Brownbridge,D.R.; and Hopkins,R.P.; *Data-Driven And Demand-Driven Computer Architecture.* ACM Computing Surveys , Vol.14, No.1, 1982, pp 93–143.

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

[22] Hockney,R.W.; and Jesshope,C.R.; *Parallel Computers — Architecture And Algorithms.* Adam Hilger, Bristol 1981.

[23] Hwang,K.; and Briggs,F.A.; *Computer Architecture and Parallel Processing.* McGraw Hill, New York 1984.

[24] Batcher,K.E.; *Design of Massively Parallel Processors*, IEEE Trans. Computers, Vol. C-29 Sept. 1980 pp 836–844.

[25] Stolfo,S.J.; and Miranker,D.P.; *DADO: A Parallel Processor For Expert Systems.* Proceedings 1984 IEEE International Conference On Parallel Processing, pp 74–82, IEEE Computer Society Press, 1984.

[26] Hayes,J.P; etal., *Architecture Of A Hypercube Supercomputer*, 1986 IEEE International Conference On Parallel Processing, pp 653–660, IEEE Computer Society Press, 1986.

[27] Snyder L.; *Introduction To The Configurable Highly Parallel Computer.* Computer , Vol. 15, No. 1, pp 47–56, Jan. 1982.

[28] Maples,C.; *Pyramids, Crossbars And Thousands Of Processors*, 1985 IEEE International Conference On Parallel Processing, pp 681–688, IEEE Computer Society Press, 1985.

[29] *Transputer Development System*, Prentice Hall International (UK) Ltd, 1988.

[30] Gollakota,N.S.; and Gray,F.G.; *Reconfigurable Cellular Architecture*, 1984 IEEE International Conference On Parallel Processing, pp 377–379, IEEE Computer Society Press, 1984.

[31] Siegel etal.; *The PASM Parallel System Prototype*, 1985 IEEE COMPCON, pp 429–434.

[32] Gottlieb,A.; *An Overview Of The NYU Ultracomputer Project*, Ultracomputer Note #100, July 1986, Ultracomputer Research Lab. New York University, New York.

[33] Pfister,G.F.; etal. *The IBM Research Parallel Processor Prototype (RP3): Introduction And Architecture*, 1985 IEEE International Conference On Parallel Processing, pp 764–771, IEEE Computer Society Press, 1985.

[34] Gurd,J.R; Kirkham,C.C.; and Watson,I.; *The Manchester Prototype Dataflow Computer*, Communications of the ACM. Vol.28, No.1 pp 34 - 52, 1985.

[35] *Multimax Technical Summary*, Encore Computer Corporation, Jan 1989.

[36] Annaratone,M.; etal. *Architecture of Warp*, Proceedings of 14th Annual International Conference On Computer Architecture, pp 264–267, 1987.

[37] Bruegge,B.; etal. *Programming Warp*, Proceedings of 14th Annual International Conference On Computer Architecture, pp 268–271,1987.

[38] Annaratone,M.; etal. *Applications And Algorithm Partitioning On Warp*, Proceedings of 14th Annual International Conference On Computer Architecture, pp 272–275, 1987.

[39] Arvind,V.K.; and Culler,D.E.; *Dataflow Architectures*, MIT Laboratory for Computer Science, MIT, LCS/TM-294, 1986

[40] Gurd,J.; and Watson I.; *Data Driven Systems For High Speed Parallel Computing*, Computer Design, Parts I and II, June/July 1980.

[41] Mago,G.; *A Cellular Computer Architecture For Functional Programming*, Proceedings of IEEE Computer Society COMPCON, pp 179–187 Spring 1980.

[42] Andrews,G.R.; and Schneider,F.B.; *Concepts And Notations For Concurrent Programming*, Computing Surveys, 15, pp 34–43, 1983.

[43] Brinch-Hansen,P.; *The Architecture Of Concurrent Programs*, Prentice Hall Int. 1977.

[44] Lampson,B.; and Redell,D.; *Experiences With Processes And Monitors In Mesa*, CACM, Vol 23, No 2, February 1980.

[45] Jones,G.; *Programming In Occam*, Prentice Hall, 1985.

[46] Carriero,N.; and Gelernter,D.; *How To Write Parallel Programs : A Guide To The Perplexed*, ACM Computing Surveys, Vol 21, No 3, pp 323–357, September 1989.

[47] Brookes,S.D.; and Hoare,C.A.R.; *A Theory Of Communicating Sequential Processes*, Journal of the ACM, Vol 31, No 3, July 1984.

[48] Hoare,C.A.R.; *Communicating Sequential Processes*, CACM, Vol. 21, No 8, August 1978.

[49] Brinch-Hansen,P.; *Edison : A Multiprocessor Language*, Software Practice And Experience, Vol. 11, pp 325–361, 1981.

[50] Brinch-Hansen,P. *Distributed Processing : A Concurrent Programming Concept*, CACM, Vol 21, pp 934–940, 1978.

[51] Gehani,N.; *Ada : Concurrent Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[52] Tabak,Daniel; *Multiprocessors*, Prentice-Hall International, NJ, 1990.

[53] Wirth,N.; *Modula: A Language For Modular Multiprogramming*, Software Practice And Experience 7, 1977, pp 33–35.

[54] *Encore Parallel Threads Manual*, Encore Corporation, ref 724-06210, 1988.

[55] Feldman,J.A.; *High Level Programming For Distributed Computing*, Communication Of The ACM 22, 6 (June), 1979, pp 353–368.

[56] Fanstini,A.A.; and Wedge,W.W.; *An Eductive Interpreter For The Functional Languages Lucid.* Second ACM Conference On Functional Programming And Computer Architecture, Nancy, France, 1985.

[57] Keller,R.M.; *Rediflow Architecture Prospectus.* Technical Report No. UUCS-85-105, April 1986 Dept. Of Computer Science, University Of Utah.

[58] Quinn,M.J.; *Designing Efficient Algorithms For Parallel Computers* Mcgraw-Hill International Editions, 1987.

[59] Harel,D.; *Algorithmics, The Spirit Of Computing, 2nd Edition* Addison-Wesley Publishing Company, 1992.

[60] Jarvis,R.A.; *On The Identification Of The Convex Hull Of A Finite Set Of Point In The Plane*, Information Processing Letters Vol.2 (1973) pp 18–21.

[61] Lee,D.T; and Preparata,F.P.; *Computational Geometry - A Survey*, IEEE Transactions On Computers C-33 (12) (1984) pp 1072–1101.

[62] Preparata,F.P.; and Hong,S.J.; *Convex Hull Of A Finite Set Of Points In Two And Three Dimensions*, Commun. ACM Vol.20(2) (1977) pp 87–93

[63] Day,A.M.; *The Implementation Of An Algorithm To Find The Convex Hull Of A Set Of 3-D Points.* ACM TOG Vol. 9(1) (1990) pp 105–226

[64] Graham,R.L.; *An Efficient Algorithm For Determining The Convex Hull Of A Finite Planar Set*, Information Processing Letters 1 (1972) pp 132–133

[65] Johansen G.H.; and Cram C.; *A Simple Algorithm For Building The 3-D convex Hull*, BIT Vol. 23 (1983) pp 146-160.

[66] Bentley,J.L.; and Shamos,M,I,; *Divide and Conquer For Linear Expected Time*, Information Processing Letters 7 (1978) pp 87–91.

[67] Miller,R.; and Stout,Q.F.; *Computational Geometry On A Mesh-Connected Computer*, 1984 IEEE International Conference On Parallel Processing, pp 66–73, IEEE Computer Society Press, 1984.

[68] Miller,R.; and Stout,Q.F.; *Efficient Parallel Convex Hull Algorithms*, IEEE Trans. On Computers Vol 37(12) (1988) pp 1605–1618.

[69] Atallah,M.J.; and Goodrich,M.J.; *Efficient Solution To Some Geometric Problems*, Journal of Parallel And Distributed Computing 3 (1986) pp 492–507.

[70] Goodrich,M.J.; *Finding The Convex Hull Of A Sorted Point Set In Parallel*, Information Processing Letters 26 (1987/88) pp 173–179

[71] Akl,S,G,; *A Constant Time Parallel Algorithm For Computing Convex Hulls*, BIT 22 (1982) pp 130–134.

[72] Nath,D.; Maheshwari,S.N.; and Bhatt, P.C.P.; *Parallel Algorithm For The Convex Hull Problem In Two Dimensions*, Technical Report EE 8005, Department Of Electrical Engineering, Indian Institute Of Technology, New Delhi, India (October 1980).

[73] Chow,A.L.; *A Parallel Algorithm For Determining Convex Hull Of Sets Of Points In Two Dimensions*, Proceedings Of The 19th Allerton Conference On Communication, Control And Computing, Monticello, Illinois (1981) pp 214–223.

[74] Akl,S.G.; *Optimal Parallel Algorithms For Selection, Sorting And Computing Convex Hulls*, Computational Geometry (Editor G.T.Toussaint), Elsevier Science Publishers, (1985) pp 1–22.

[75] Kirkpatrick,D.G.; and Seidel,R.; *Planar Convex Hull Algorithms?*, SIAM Journal Of Computing 15(1) (1986) pp 287–299.

[76] Edelsdrunner,H.; Kirkpatrick,D.G.; and Seidel,R.; *On The Shape Of Set Of Points In The Plane* IEEE Transactions On Information Theory 29, (1983) pp 551–559.

[77] Day,A.M.; *Parallel Implementation Of 3-D Convex Hull Algorithm*, Computer-aided Design Vol. 23 (1991) pp 177–188

[78] Chazelle, B.; *Computational Geometry On A Systolic Chip*, IEEE Trans. On Computers C-33(9) (1984) pp 774–785.

[79] Miller,R.; and Stout,Q.F.; *Mesh Computer Algorithm For Computational Geometry*, IEEE Trans. On Computers Vol.38 (1989) pp 321–340.

[80] Holey,J.A.; and Ibarra,O.H.; *Iterative Algorithms For The Planar Convex Hull Problem On Mesh-Connected Arrays*, Parallel Computing 18 (1992) pp 281–296.

[81] Sedgewick,R.; *Algorithms* (2nd Edition), Addison-Wesley, New York, (1988).

[82] Preparata,F.P.; and Shamos, M.I.; *Computational Geometry - An Introduction* Springer-Verlag, New York, (1985).

[83] Holey,J.A.; and Ibarra,O.H.; *Triangulation In A Plane And 3-D Convex Hull On Mesh-Connected Arrays And Hypercubes* Int. Conference On Parallel Processing (1992) pp 10–17.

[84] Reif,J; and Sen,S.; *Optimal Parallel Algorithms For 3-Dimensional Convex Hull And Related Problems*, SIAM Journal On Computing 21:3 (1992) pp 466–485.

[85] Aggarwal,A.; Chazelle,B.; Guibas,L.; Dúnlaing,C.Ó,; and Yap,C.; *Parallel Computational Geometry*, Algorithmica 3(3) (1988) pp 293–328.

[86] Chow,A.; *Parallel Algorithms For Geometric Problems*, Ph.D. Thesis, Comp. Sc. Dept., Univ. Of Illinois, 1980.

[87] Yao,A.C.; A Lower Bound To Finding Convex Hulls, JACM 28 (1981) pp 780–787.

[88] Grünbaum,B.; *Convex Polytopes*, Wiley, New York, 1967

[89] Day,A.M.; *Experiments in the Parallel Computation of 3D Convex Hulls* Computer Graphics forum Vol.13 (1994) number 1, pp 21–36.

[90] Goodman,S.E. and Hedetneimi,S.T.; *Introduction to the Design and Analysis of Algorithms* McGraw-Hill, 1977.

[91] Horowitz, Ellis; and Sahni, Sartaj; *Fundamentals Of Computer Algorithms*, Computer Science Press, Inc, 1978

[92] Rabhi,F.A.; and Manson,A.; *Divide-and-Conquer Parallel Graph Reduction*, Parallel Computing 17 (1) pp 189–205 (1991).

[93] McMullen,P.; and Shephard,G.C.; *Convex Polytopes and the Upper Bound Conjecture*, Cambridge University Press, Cambridge (1971).

[94] Aho,A.V.; Hopcroft,J.E.; and Ullman,J.D.; *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., (1974).

[95] Megson,G.M; and Eyoh,E.O; *Implementation And Evaluation Of Parallel n-D Convex Hull Algorithms*, Parallel Computing: Trends and Applications Joubert,G.R.; Trystram,D.; Peters,F.J.; and Evans,D.J.; (Editors) pp 169–176 1994 Elsevier Science B.V

[96] Frierbera,A; Megson;G.M.; etal. *Solving COmbinatorial Optimization problems in Parallel (SCOOP)*, HCM Grant From EC, 1994.

[97] C-S Tools Meiko Manual, S0205-20T101.02, 1983.

Appendix A

Tables

Table 7: Timing Of Recursive Version For 2D With 3 Vertices (Multimax)

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	166643	226598	395592	841881	2888055	9944076	36344589	79394541	1378888505
	Thread	145477	174203	283449	531775	1581603	4998430	18087512	39788809	68566957
	Microthread	119575	151613	245838	503796	1530946	4959641	17759369	39146672	67444355
3	Sequential	250455	259281	408478	700583	2415931	7381996	24643418	52916117	92943296
	Thread	255714	245774	315450	426747	1090612	2843493	8571150	18351091	31479976
	Microthread	208585	192197	268140	394880	1017109	2729174	8317713	17939287	31025329
4	Sequential	297844	338180	486876	883342	2347805	6649440	20961733	43864862	74988769
	Thread	298106	311715	409316	566718	916117	1969556	5901357	11178784	18907401
	Microthread	257928	266216	340557	469771	843114	1903718	5688291	10887830	18424953
5	Sequential	318418	407157	513174	855599	2145561	5799237	17139867	34310048	59230580
	Thread	463863	459578	541558	699127	1039039	1819594	4656267	7973657	13107170
	Microthread	356237	423753	470971	583695	873298	1715513	4378911	7751366	12625087
6	Sequential	347294	425128	605007	901621	2115442	5029615	15210449	31765370	53226559
	Thread	503843	547304	662944	757207	1146337	1679135	3579314	6525964	10219182
	Microthread	442964	429084	573459	652567	987300	1576188	3289031	6203045	9711809

Table 8: Running Time Of Stack Version For 2D With 3 Vertices (Multimax)

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	171795	220795	405292	800480	2805394	9416219	35165824	76264562	131861840
	Thread	128067	160722	259339	479444	1508454	4927765	18215514	39169851	67262466
	Microthread	118362	140034	247320	470177	1483651	4923643	18378435	39275085	66840088
3	Sequential	230080	272634	461340	743921	2424615	7186961	23885017	52160976	90594770
	Thread	225864	248969	326768	439353	1103186	2727143	8531503	18461144	31236412
	Microthread	183634	216023	306004	421202	1084138	2637875	8394068	18287225	31136970
4	Sequential	280428	353605	575213	893349	2498857	6405496	20256477	42276904	72281665
	Thread	240641	292351	387192	461127	973430	1910593	5806972	10996424	18361041
	Microthread	239551	249390	351870	438841	870968	1836431	5715211	11038432	18334049
5	Sequential	374792	500755	543308	853976	2178488	5665507	16981020	34106313	57401634
	Thread	394556	463900	530366	607584	987637	1818106	4388859	7923885	12935666
	Microthread	360002	448842	434042	529114	971841	1735659	4292449	7760416	12870669
6	Sequential	353168	442014	619913	892631	2182652	5121904	15149259	31427771	52130207
	Thread	443582	490571	590469	690288	1113235	1561610	3417766	6465608	9993635
	Microthread	379444	480216	576132	644633	1071175	1584105	3338281	6256069	9727668

Table 9: Running Time Of Recursion Version For 2D With 4 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	184718	256787	404946	837163	3086369	10387073	37339400	81452728	139086980
	Thread	153726	207290	298654	501634	1678617	5281474	18400466	39577255	67727610
	Microthread	135439	190973	273922	477257	1631857	5110611	18580781	39305244	67017867
3	Sequential	247632	339463	551195	898768	2723183	7939021	25898307	54332578	95859112
	Thread	265885	303179	412607	567280	1206506	2966390	9129147	18696411	32013841
	Microthread	220720	245900	344514	508256	1171959	2948384	8979744	18434301	31704243
4	Sequential	316932	429847	568383	940243	2718568	7042684	22459330	45490372	76796234
	Thread	285591	408423	419467	603010	1153803	2183848	6005215	11773635	19146855
	Microthread	264546	369890	383428	530410	995708	2052921	5811840	11623861	19027283
5	Sequential	380775	445247	704988	1039649	2419476	6000231	17652364	36200510	60498770
	Thread	443970	519756	683598	832369	1061796	1895604	4527614	8484806	13116768
	Microthread	397882	440698	604610	714095	961593	1959000	4394591	8190441	12855870
6	Sequential	408523	487766	740164	1230741	2577658	5903105	16380658	33477002	54592502
	Thread	563195	625305	757138	894526	1369915	2099523	3941789	6838396	10395177
	Microthread	460624	553136	664903	826341	1198693	1876853	3735809	6543242	10198214

Table 10: Running Time Of Stack Version For 2D With 4 Vertices Multimax

		Size Of Set								
Part	*Time*	25	50	100	200	500	1000	2000	3000	4000
2	Sequential	179937	246330	401684	896571	3148521	10061855	35246600	75542696	132227836
	Thread	133073	172573	249834	494699	1663914	5152896	18134939	37861602	66423436
	Microthread	124118	147559	234823	487940	1643575	5080126	17967554	38245979	66867522
3	Sequential	258929	347523	593902	998718	2727935	7795451	25251701	52203333	91777099
	Thread	232112	277962	402711	548566	1152339	2869871	8855241	18108688	30953067
	Microthread	198650	243254	384149	506839	1111128	2878764	8841232	18185660	31117805
4	Sequential	328139	439211	645478	935660	2735482	6995230	21567480	43911145	74595295
	Thread	262010	328902	409402	536057	1043818	2111955	5712353	11335829	19108059
	Microthread	227745	291614	369991	472544	1010210	2090919	5670572	11495215	19143499
5	Sequential	338587	399897	649008	1002435	2322423	5908421	16987047	35224746	58496400
	Thread	385796	430921	576295	765984	1110811	1967018	4346752	8447888	12778839
	Microthread	342270	375946	526294	696474	1063812	1847302	4371905	8271329	12711139
6	Sequential	362989	474592	724406	1132590	2544260	5801328	16283151	32323302	52838541
	Thread	490819	562849	723758	885897	1255013	1961690	3701654	6567896	10329716
	Microthread	442047	526553	634937	774331	1141670	1805265	3670439	6452282	10119331

Table 11: Running Time Of Recursion Version For 2D With 6 Vertices Multimax

		Size Of Set								
Part	*Time*	25	50	100	200	500	1000	2000	3000	4000
2	Sequential	221998	284216	475811	844084	2994355	10074251	38507491	81983467	142852560
	Thread	176022	215691	326818	560735	1615788	5296834	19528552	40108865	69735024
	Microthread	159331	207183	309624	560825	1593358	5242129	18919027	39982323	68984396
3	Sequential	297073	343671	601389	943006	2902141	8146052	7269586	57574391	97345795
	Thread	277173	305834	420827	545904	1258780	3026240	9773776	20391250	32460103
	Microthread	244954	270467	388251	527734	1198566	2996836	9633546	19583982	32202725
4	Sequential	319250	432511	599318	951104	2575469	6888964	22043638	46214546	80110130
	Thread	328587	371911	475127	551431	1066793	2026243	6038707	11647296	19780583
	Microthread	270854	361611	427553	532695	973770	2087041	5923266	11584230	19629361
5	Sequential	385861	513460	683076	1105969	2498052	6188444	18236801	37617244	62691278
	Thread	465381	577280	692530	818183	1240429	2100728	4841619	8398059	13866013
	Microthread	406235	526190	643591	760641	1203854	1995165	4658336	8429404	14036697
6	Sequential	495790	568149	779717	1238328	2721480	6106542	17666738	34107467	56227909
	Thread	542052	622397	714767	880753	1341999	2098050	4379085	6844348	10952141
	Microthread	495053	560969	694309	853716	1290812	2048214	4010596	6863816	10417525

Table 12: Running Time Of Stack Version For 2D With 6 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	219703	282218	452471	874563	3098384	9902401	35938228	77616206	134590474
	Thread	171561	208862	292862	531426	1698052	5039327	18131809	39472836	67895813
	Microthread	155103	195200	284284	522818	1679954	5058126	18175808	39461650	68000178
3	Sequential	276994	392818	562226	942158	2636662	7741089	26440531	54617805	93102282
	Thread	283200	344730	415738	538480	1217950	2907186	9326959	19049083	31785887
	Microthread	252478	305324	379204	515239	1174462	2891341	9295835	19081092	31697839
4	Sequential	346713	447768	572729	897743	2566478	7127616	21925298	45040262	75847007
	Thread	313368	357773	422339	527712	1042473	2157542	5734172	11657229	19631788
	Microthread	293014	351504	401716	497407	951755	2122063	5708770	11619756	19461919
5	Sequential	438973	575829	716454	1116581	25799534	6113786	17776446	37094961	62944229
	Thread	469163	561835	699586	806569	1231890	2135698	4648554	8619235	13908491
	Microthread	437648	513183	600356	771537	1111605	2034998	4495085	8520865	13980592
6	Sequential	438154	577205	792405	1271968	2452412	5892925	16800228	32790522	54431487
	Thread	553973	637605	746262	873162	1251918	1966603	4117385	6710347	10488974
	Microthread	444930	570824	649353	829752	1181233	1854700	3937502	6604640	10232065

Table 13: Running Time Of Recursive Version For 2D With 16 Vertices

Multimax

		Size Of Set							
Part	*Time*	50	100	200	500	1000	2000	3000	4000
2	Sequential	602471	835479	1469125	4229507	12402870	40618124	85707559	150378853
	Thread	405774	531614	866955	2308325	6290290	20032395	42359593	74305470
	Microthread	360991	465392	819964	2228885	6122574	20041050	42008423	72830190
3	Sequential	662965	842963	1293993	3334217	9022123	29559550	60664435	105490322
	Thread	485752	623511	804473	1527234	3529152	10570816	20916199	35977255
	Microthread	412542	551983	714245	1462163	3323032	10333311	20749627	35828266
4	Sequential	786429	989894	1486063	3312612	8417969	25513497	50994652	85283854
	Thread	635058	752697	984605	1535809	2810454	7038562	13619740	22376597
	Microthread	528816	603506	853768	1356001	2681696	6787691	13578626	22187587
5	Sequential	845834	1044532	1545487	3329581	7391584	20961062	41263157	71001479
	Thread	848205	981892	1226905	1793451	2710403	5620218	10023203	16128231
	Microthread	676471	854621	1076408	1592007	2494333	5447993	9767438	15781184
6	Sequential	1007704	1221947	1874625	3478913	7425055	19835133	38479566	64631080
	Thread	930181	1103334	1348552	1767130	2744056	5220778	8511904	12395989
	Microthread	726209	942612	1123241	1632563	2489998	4659696	8178321	12204448

Table 14: Running Time Of Stack Version For 2D With 16 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	438237	501098	688407	1206813	3781394	11288503	39382710	82586303	143232614
	Thread	273809	327065	435688	707819	2065308	5857332	19965172	41947280	71997072
	Microthread	262787	313203	412283	672180	2028574	5908308	20063185	42189504	72255027
3	Sequential	540370	616302	795805	1234391	3260153	8818599	34191495	59812347	101186146
	Thread	456423	485827	580193	810903	1510498	3592190	10108609	20783700	35300877
	Microthread	415623	423394	530172	778839	1499541	3593458	10089207	20833155	35254412
4	Sequential	619046	722180	988565	1501872	3393357	8320267	24637632	49397993	81469924
	Thread	465538	499980	626410	790614	1378202	2649174	6847562	13168099	21326094
	Microthread	415924	466732	598099	738900	1303722	2637578	6794975	13145470	21335850
5	Sequential	672044	830658	1018513	1445755	3275966	7468584	20449382	40699345	67130694
	Thread	692004	738367	856461	1103499	1759247	2863324	5641694	10158813	15535053
	Microthread	591768	628544	796601	959053	1640498	2743912	5496743	10079398	15267890
6	Sequential	805349	907354	1201049	1644466	3409271	7126086	19331759	37549010	62311681
	Thread	769638	824613	894559	1171371	1739702	2489209	4514578	8057934	12314018
	Microthread	626740	745269	883191	1062892	1560810	2316508	4613527	7804343	12256118

Table 15: Running Time Of Recursive Version For 2D With 26 Vertices
Multimax

Part	Time	*Size Of Set*							
		50	100	200	500	1000	2000	3000	4000
2	Se	964889	1160424	1899973	4837268	12829542	42508855	87288169	150142739
	Thd	575609	723871	1103277	2594791	6473626	21288419	42923848	73579814
	Mic	529471	659029	1033137	2529473	6402651	21098194	42617297	72638377
3	Se	1022317	1221381	1805385	3957516	10199550	30671482	62225972	104756362
	Thd	709530	849220	1166013	1946468	4246614	11227680	21508762	35742436
	Mic	645742	752913	987639	1775792	4134344	10812395	21246392	35138608
4	Se	1170221	1408673	1925895	3947911	9071274	26139869	51320751	85651257
	Thd	768549	1000718	1281315	1843590	3091455	7759988	13585782	22006890
	Mic	727521	843694	1156380	1758960	2941975	7536789	13172829	21862406
5	Se	1304737	1545678	2033306	3822352	8122054	22044446	42451833	70543433
	Thd	1135889	1231019	1498220	2093196	3168640	6553279	10781561	16393182
	Mic	962679	1151950	1368687	1915379	2878962	6019937	10259267	15980502
6	Se	1485323	1773226	2302073	4063619	8465524	20734042	38921146	63413802
	Thd	1270880	1391464	1678968	2366798	3129037	5621141	8915234	13116447
	Mic	1031737	1087402	1431080	1969049	2904042	5457167	8426992	12551836

Table 16: Running Time Of Stack Version For 2D With 26 Vertices Multimax

		Size Of Set							
Part	*Time*	50	100	200	500	1000	2000	3000	4000
2	Se	799626	1018554	1616582	4211996	12099486	41075182	84993169	146502630
	Thd	468751	594439	889918	2322021	6426176	21298879	43167069	74045495
	Mic	499363	591368	946001	2339214	6547105	21512137	43641390	74481206
3	Se	955540	1135145	1684747	3897608	9732913	30609698	60793146	103046696
	Thd	657030	771407	1008305	1910711	3987107	11188300	21204006	35654079
	Mic	729229	781073	1105368	1849045	4083197	11170921	21673246	36135443
4	Se	1081779	1264601	1891106	3838060	9179551	26401304	51889124	84503948
	Thd	640842	723185	1034944	1542973	2929600	7767234	13875375	22073110
	Mic	721574	787835	1084781	1547949	3027627	7798447	14075527	22390803
5	Se	1274056	1600812	2046954	3829524	8402885	22202521	42320349	69545423
	Thd	1088321	1130568	1368516	1894319	3081545	6394846	10580379	16124866
	Mic	910606	1042344	1286985	1847069	2955581	6326362	10426058	16106770
6	Se	1441501	1632284	2347862	4175950	8269892	20513379	39222263	62389904
	Thd	1051740	1235442	1458794	1991132	2990148	5145894	8382625	12431726
	Mic	999598	1071338	1460061	1956984	2784946	5188213	8842198	12339234

Table 17: Running Time Of Recursive Version For 3D With 3 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Seq	191002	257505	422981	907270	3311387	11262728	41646847	90963577	158090182
	Thd	147057	176926	285245	569361	1784776	5736607	20801350	46191565	79149567
	Micro	124955	167571	264300	535801	1691438	5634501	20411169	45110666	77989612
3	Seq	281662	270256	446775	772714	2701048	8339643	27793519	60746721	105842702
	Thd	262261	236420	344696	444811	1172869	3200211	9615183	21229381	36180296
	Micro	223739	193277	299745	421406	1130563	3081471	9391500	20558236	35387207
4	Seq	337128	383227	557649	988080	2651712	7611609	24032760	50264628	86095773
	Thd	296197	321281	414554	534011	953907	2214881	6470076	12691111	21340911
	Micro	268700	281930	365666	508142	879827	2143965	6366797	12513896	21089096
5	Seq	374438	457316	597845	979171	2401118	6341535	19384828	39074948	66937214
	Thd	412845	453437	558529	641171	1116176	2000253	5111176	8952028	14569480
	Micro	401533	407290	483835	579773	984947	1791846	4958971	8753109	14502298
6	Seq	397678	485684	674168	1035264	2384366	5733392	17267740	36735811	60281419
	Thd	465541	507082	653816	733550	1130847	1780661	3913172	7242708	11511981
	Micro	452704	464233	567087	645442	106515	1697762	3710122	7042643	10994920

Table 18: Running Time Of Stack Version For 3D With 3 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	164261	230552	426877	892198	3201575	10842843	40298028	87692368	151524478
	Thread	150744	180104	297298	534238	1727143	5664150	21055835	45053790	76916959
	Microthread	127222	155432	279483	519014	1731914	5661155	21105611	45285034	76744759
3	Sequential	227730	291267	493370	805211	2698044	8032566	27228649	59481820	103313185
	Thread	231516	271860	373627	499642	1248264	3137940	9705098	21094198	35736292
	Microthread	187059	233711	319585	429613	1227727	3053498	9555944	20976719	35653243
4	Sequential	272936	355076	581550	961489	2726522	7279003	23450202	48368051	83130233
	Thread	272024	296529	429222	514919	1073314	2168768	6601624	12541863	20930652
	Microthread	237440	260158	386297	476331	1017961	2083785	6441090	12509020	20856008
5	Sequential	501454	483300	553496	907848	2376180	6390708	19148047	38718325	65756050
	Thread	506365	513106	558656	668767	1105444	1895065	4950537	8958928	14611194
	Microthread	443543	464680	442418	614421	1052246	1970731	4795839	8788986	14482455
6	Sequential	349277	429281	634935	937718	2384980	5863562	17322716	35780457	59591289
	Thread	478600	510883	679019	758801	1237021	1820303	3805581	7394280	11546585
	Microthread	404445	453944	634708	673364	112570	1609410	3718570	7039703	11147451

Table 19: Running Time Of Recursive Version For 3D With 6 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	1408992	2045367	3123531	4794280	9537390	24459785	69419005	140138731	235748086
	Thread	1019090	1498559	2180051	3254645	5358834	12849886	36589913	74900019	121829580
	Microthread	922762	1375346	2056412	3121691	5238302	12456578	36352647	73535602	120502781
3	Sequential	1765916	3022200	4013869	6222328	14181379	30785002	70003050	122739191	198105746
	Thread	1795665	2458018	3372433	4688611	8588337	16031583	28991650	47200861	74096213
	Microthread	1611633	2270763	3145186	4440276	8218655	15229455	28172232	46357493	72717403
4	Sequential	2301323	4189155	6312317	9763828	15881987	31083664	66840105	114759296	177561500
	Thread	1777621	3277518	4279028	6340170	8036408	13235591	24132584	37205074	55411393
	Microthread	1740002	3030974	4288062	5670915	8203960	13168767	23099406	36319282	54094344
5	Sequential	2597564	4858734	6750602	10147752	16860846	30651408	62963046	103075953	159240981
	Thread	2603970	4999464	6269377	9027274	14437509	19537096	30325360	38812347	54848687
	Microthread	2060013	4282070	6239949	8734159	13003050	17916020	28498004	36926320	54165774
6	Sequential	2429977	4660491	8034779	11084349	20729449	36907585	70798109	112962003	
	Thread	2857940	4419704	6707080	9057511	15846982	23112309	31370218	42167355	
	Microthread	2767652	4364910	6242605	9019551	14295803	20911524	30773925	40093255	

Table 20: Running Time Of Stack Version For 3D With 6 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	1354126	1964372	3467641	5249766	10233543	47777153	75116996	140522946	231929393
	Thread	931053	1406728	2207168	3380621	5788846	24352283	38252659	71635451	116792935
	Microthread	926152	1404746	2136772	3394829	5780546	24637839	38380556	73642918	118595412
3	Sequential	1808409	3100231	4227923	6420597	14133709	30487417	72208457	118884854	192784718
	Thread	1501117	2573954	3521707	4848839	7578600	14363838	29582509	44189544	70741649
	Microthread	1506685	2428275	3496390	5019037	7595814	14121249	29755374	46058247	70712140
4	Sequential	2373251	4394728	6623836	10605187	16644702	30529141	70771712	117472593	181922875
	Thread	1545363	2922992	4191561	6224911	8178610	12684899	24097498	35978730	54799339
	Microthread	1586404	2717748	4268620	6241414	79233898	12369110	24203162	35481027	53871123
5	Sequential	2208959	4237917	6979341	10569777	18574668	32550786	67347138	108829496	167334070
	Thread	2131256	3979656	6906242	8989477	13468181	18498185	30304653	40112169	55881617
	Microthread	1961248	3675145	6829397	8585202	12522565	18298623	27938634	35704247	53451278
6	Sequential	2470822	4510592	7892396	11174624	20174141	36720752	72470059	114156170	166816567
	Thread	2562205	4181715	6299489	8918670	12223090	20503996	28997871	36807894	46852393
	Microthread	2158121	4279180	6541125	8552091	12471590	19516881	29972876	36541452	45726394

Table 21: Running Time Of Recursive Version For 3D With 4 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	1029209	1731725	2063612	2450055	6959136	19682131	61075993	123679083	201868635
	Thread	736165	1180139	1349724	1423903	4301364	11186286	32395962	62181250	102696619
	Microthread	729565	1158003	1333289	1447564	4280487	11139758	32137025	61732493	101661372
3	Sequential	1516177	2404234	3140973	4581890	8984569	21582399	53598594	99891860	159672945
	Thread	1196174	1849343	2456583	3377019	5529202	11408487	24418535	41303516	61465074
	Microthread	1124898	1774696	2331521	3230932	5425084	11156196	23873380	40941737	61142249
4	Sequential	2036716	3072537	4906458	5799374	11161337	22115273	5834495	102123876	150737625
	Thread	1492149	2559347	3192184	3468030	5577790	9499841	19837317	30685058	45029581
	Microthread	1371296	2585033	3277687	3449376	5501486	9484761	19622241	30210972	44328131
5	Sequential	2210800	4123100	5228460	7399654	13432186	26495203	56590499	90549336	136447324
	Thread	2235279	3469332	4238592	6420875	10597099	15388740	23516016	31196291	42602577
	Microthread	2102834	3574987	4280684	6204155	10382109	14881230	23194596	31226483	42210981
6	Sequential	2075348	3625572	5578632	7713982	14018784	24588080	54745205	92710057	127654696
	Thread	2091257	3409623	4922741	6836127	10322681	15838850	23905220	33416037	37962118
	Microthread	2125613	3300676	5267435	5615746	9940959	14727413	23824410	31992792	37378900

Table 22: Running Time Of Stack Version For 3D With 4 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	824448	1317836	1966149	2831352	6965447	19082476	60973772	116750103	196545184
	Thread	625051	981317	1418046	1757316	3813474	10022249	31209276	60277176	100516366
	Microthread	555864	896421	1301346	1642382	3574871	9911399	31131740	59979838	100807104
3	Sequential	1649136	2403866	3088686	4906667	9723408	22093569	52185938	97832498	159636227
	Thread	1256806	2160935	2459137	3566299	6104928	11294064	24240601	41185120	66953905
	Microthread	1260156	2052497	2354820	3474618	6012278	11053363	23230891	41094742	65225880
4	Sequential	2078128	3127831	5424324	5388741	12588846	24287117	62390064	102100480	153532173
	Thread	1251415	2169715	3160214	3586279	5453281	9996009	21978173	31463286	48043550
	Microthread	1226353	2039686	3285314	3440695	5390922	10207619	21811502	31262556	49494975
5	Sequential	2222092	4221255	5649386	8067373	12939023	23445026	57163054	90122919	133755173
	Thread	2113062	4036456	5031962	6017084	9478750	13514314	23912694	32290905	41120167
	Microthread	2027396	3668457	4895662	5857628	8953232	13447734	24549103	32155788	39843737
6	Sequential	2092445	3532993	5550338	7806060	13907603	26016431	55474547	89311795	122732678
	Thread	2039055	4036238	4515347	6004118	9624268	15930748	22749017	32905282	39662904
	Microthread	2167988	3890213	4694520	5919338	10253676	14643373	23436801	29568532	34890424

Table 23: Running Time Of Recursive Version For 3D With 12 Vertices
Multimax

Part	*Time*	*Size Of Set*							
		25	50	100	200	500	1000	1500	2000
2	Se	2230274	2837968	3951825	5497084	12868619	31860599	54763524	86671992
	Thd	1494974	1821453	2681349	3603937	7497893	18112834	29192735	46157369
	Mic	1322588	1663876	2522490	3401606	7217238	17623925	28590517	45439081
3	Se	2879735	3516638	5390896	8102545	17059070	35444910	57230207	80008248
	Thd	2365613	2922105	4421350	6242234	11234806	19117964	27417510	37305975
	Mic	2144717	2531592	4018140	5765831	10853025	18234510	27136816	36825423
4	Se	3304689	5000448	7043008	10466587	19803319	41318174	62387147	86665659
	Thd	2261120	3602718	5125209	6642256	10962011	19555123	25285284	32447824
	Mic	2146794	3187145	4555571	6019811	10390158	17727574	23485229	30574548
5	Se	3689174	5184579	7219759	11424423	23687109	41849833	60230266	86049520
	Thd	3727365	4912099	6918398	10713618	16488925	24729878	31029290	38646778
	Mic	3107662	4494727	6092362	9487920	15729572	23706920	30029516	37836727
6	Se	3830610	5453476	8692048	11954175	24687434	44105853	63431709	85573980
	Thd	3095672	4930355	7932046	11083089	19459897	28135740	34541676	
	Mic	3222731	4907094	7337385	10474654	18153202	25323758	32589889	38414012

Table 24: Running Time Of Stack Version For 3D With 12 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	2204041	2741851	3735085	56011642	14234881	29213943	8366037	151141884	244313804
	Thread	1251693	1762498	2270701	3389339	8126989	15815902	43928584	77795509	124963041
	Microthread	1321214	1862647	2470408	3596146	8299166	16356070	58312752	78835444	125766382
3	Sequential	2779650	3317128	4701923	818988	16567745	36879165	78073921	138545877	208982002
	Thread	2146009	2708450	3730842	5540172	9704841	17306836	32131949	53602113	77533099
	Microthread	2102714	2724695	3893413	5709344	9570865	18301103	32411269	54203214	78167892
4	Sequential	3136157	4711118	6619404	9971261	21130785	41385047	88176575	144502151	221044594
	Thread	1921481	2874589	4173276	5723226	10624790	17295682	31561535	44645276	63125342
	Microthread	2082639	3027677	4248592	5923726	10945494	17238292	31965270	46238221	63469361
5	Sequential	3704253	5274559	7787765	10502310	22837574	40605383	83235725	138417672	
	Thread	2807329	4203521	5814783	8122623	15398649	22318513	37684960	52448457	
	Microthread	3138806	4171137	5730686	8521258	15647516	22781468	38212674	49069180	
6	Sequential	3866709	5610750	8577267	13233042	26211257	45865123	90951005	144167198	
	Thread	3127962	4966923	6609025	9697545	16379433	24727389	36506680	48382486	
	Microthread	3110675	4576035	6654482	9505082	16495974	24717599	35483472	47188209	

Table 25: Running Time Of Recursion Version For 4D With 4 Vertices Multimax

		Size Of Set								
Part	*Time* 25	50		100	200	500	1000	2000	3000	4000
2	Sequential	1158206	1966674	2741508	3985138	8326177	22027647	68450236	140037467	229948679
	Thread	802472	1357565	1797885	2424096	4881826	11976947	36028564	69232122	116352606
	Microthread	776741	1342427	1872418	2480479	4800511	12107501	36373235	68679960	116086347
3	Sequential	1805074	2629180	3829272	6605513	13267715	24262032	66073331	118906314	191218448
	Thread	1371249	2051285	2761506	4162198	7544741	12002554	31611004	52193217	75663140
	Microthread	1318712	2053119	2809943	4112512	7779136	12251590	31275451	51906021	75915160
4	Sequential	2481469	3580083	4766951	7212221	13697578	24612912	65320061	109822524	169744488
	Thread	1497734	2527076	3353795	4347110	6382623	10445176	21879559	32619503	48976054
	Microthread	1450719	2626278	3378771	4357245	6467682	10134720	21448268	33348190	48322766
5	Sequential	2713999	4097572	5777287	8414176	14974888	26163343	60572885	103505265	152359908
	Thread	2145057	3115028	4877886	6924146	10690652	14493183	25651936	39688430	47026264
	Microthread	2092092	2994206	4596052	6707321	10278986	14244952	25618457	39013947	45618124
6	Sequential	2294691	4018381	6447497	9143431	17483216	27693143	73955452	99453434	144488390
	Thread	2477229	4013943	5747545	7783671	12965054	16899753	29431648	38116715	45373326
	Microthread	2193537	3906841	5760220	7770783	13206322	16049395	27362659	34268775	39929238

Table 26: Running Time Of Stack Version For 4D With 4 Vertices Multimax

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	1138077	1697442	2617576	4425604	9058828	22288278	70210909	136039653	223048907
	Thread	825432	1352566	1872348	2899550	5176467	11369985	37317550	69857589	112596087
	Microthread	733512	1253308	1747722	2692085	5117828	11484210	37384943	69952883	112741702
3	Sequential	1814229	2774631	4511882	6813202	12002137	24901228	64235226	117365598	179434591
	Thread	1446208	2210376	3412172	4268580	6937759	12194278	31185031	46086170	69172729
	Microthread	1343525	1997912	3120188	3916779	6713808	12238967	29888780	45526258	68942788
4	Sequential	2072888	2966513	4681063	7512854	14616840	24109518	63150015	109422245	169201164
	Thread	1402305	2466578	3596980	4786488	7404375	9976870	21406417	33650175	49509870
	Microthread	1295292	2292970	3139602	4570776	7085635	9616745	20559884	33216084	49027371
5	Sequential	2521494	3961067	5819348	9343878	17023370	25459597	56521139	102935522	149439460
	Thread	2040500	3582381	4862166	6523997	10679989	13497818	26173280	39980457	45529606
	Microthread	1910114	3127204	4866249	6091358	9882265	12422160	23310568	37894962	44483526
6	Sequential	2252262	4321293	8422066	9016757	17771167	29679813	63770801	99576673	144591169
	Thread	2751466	4439935	6156647	7083331	12014048	17260445	28858599	34324986	42825180
	Microthread	2325774	3818927	5673450	7613544	9920228	16037868	27182858	30506228	42110908

Table 27: Running Time Of Recursion Version For 4D With 6 Vertices Multimax

Part	*Time*	*Size Of Set*				
		25	50	100	200	500
2	Sequential	1415454	2256424	3144069	4473580	11097172
	Thread	973649	1557434	2007314	2923587	7516094
	Microthread	966602	1555640	2009687	2941250	7652470
3	Sequential	2177069	3840606	4717484	6378258	14027001
	Thread	1649357	2954981	3561955	4598603	8297328
	Microthread	1651806	2862417	3831109	4170080	80166570
4	Sequential	2151635	3950716	6350195	7634326	15262782
	Thread	1622152	2765458	4019103	5390021	9418362
	Microthread	1605438	2879478	3872044	5276979	9019382
5	Sequential	3185643	5701038	7681109	10242403	19516214
	Thread	2785837	4557850	6164982	8647973	11383711
	Microthread	2539810	4206738	5762590	7696241	13002992
6	Sequential	2994899	5527919	81120022	12216067	22663236
	Thread	2656984	5359347	9141272	8226553	14939240
	Microthread	2711542	4860165	6786919	9210576	14843606

Table 28: Running Time Of Stack Version For 4D With 6 Vertices (Multimax)

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	1483262	2052189	2832361	4257537	12001020	27840025	76940957	146168404	247081506
	Thread	930070	1306623	1756920	2382976	7324872	14779193	38675072	76259056	125303875
	Microthread	945196	1299912	1738383	2449510	7401634	14843552	38934447	74717331	125549171
3	Sequential	1964238	3127499	5083690	6037713	14415293	28364695	72529471	136880584	205456060
	Thread	1541639	2370795	3532133	4098230	8312304	13769341	30026110	53372827	78535576
	Microthread	1512782	2311715	3430170	3949422	8256607	13489688	30636134	52830140	78582997
4	Sequential	2822031	4772036	6550691	7837972	15723008	33434190	73621468	122705343	195679459
	Thread	1612849	2591325	3989683	4514043	8017037	13119550	24065039	36314714	56799570
	Microthread	1652581	2586407	3796719	4658620	8220714	13049250	24173440	37309307	57876617
5	Sequential	2829477	4815503	6611360	9390307	18812688	34078983	71784306	120775183	178484126
	Thread	2271872	4346821	5788596	7540462	12152854	18379908	27418719	42151713	60889963
	Microthread	2258483	4130430	5529576	7724367	10741914	16391612	28562629	41638107	59436165
6	Sequential	2976810	4515871	7558337	11817052	22267167	37359059	74153346	118722189	170168512
	Thread	2615702	4297267	6347670	7895590	13399292	18021789	29828200	38930694	48354522
	Microthread	2394701	4191436	6361005	7200881	13339700	17853546	27780024	39339894	49408206

Table 29: Running Time Of Stack For 2D With 26 Vertices Using Transputer

Part	*Time*	*Size Of Set*							
		50	100	200	500	1000	2000	3000	4000
2	Sequential	1023744	1326912	2149824	5820992	16314944	53836672	110340480	188417792
	Version 1	569600	751040	1217536	3219264	8824128	27911808	55785088	94943104
	Version 2	552000	736512	1201472	3201216	8804992	27886464	55760832	94915008
3	Sequential	1203008	1474304	2278464	5516160	13938352	41854656	82758848	137646976
	Version 1	724032	878592	1315200	2624960	5733376	16046016	30732288	50425408
	Version 2	697792	840640	1228672	2502720	5560832	15361536	29353280	47952384
4	Sequential	1360640	1565312	2493184	5185792	12353920	34703808	67940800	110584124
	Version 1	635712	754432	1144448	2073024	4473984	11078400	20309212	32234560
	Version 2	628608	727808	1156672	2015360	4294016	10588864	19113600	29723456
5	Sequential	1594432	1951872	2733824	5385792	11798848	31590592	59183872	95910976
	Version 1	815872	1025024	1335040	2261184	4172608	9148864	16036928	25216256
	Version 2	794944	998528	1340736	2159488	3949440	8565504	14710272	22499008
6	Sequential	1758912	2105024	3091520	5789120	11576064	28451136	53320896	84442880
	Version 1	805504	935168	1337792	2135552	3734208	8011264	13545664	20076480
	Version 2	812160	997504	1327104	2073728	3441408	7181056	11921152	17460608

Table 30: Running Time Of Stack For 2D With 16 Vertices Using Transputer

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	530368	636608	922432	1672192	5219456	15275520	51156864	107220992	184807232
	Version 1	304960	390912	568064	976512	2932224	8032834	26113152	54241472	92326400
	Version 2	295168	377408	551296	963072	2915584	8014208	26092288	54217984	92300096
3	Sequential	642816	746240	1009664	1707968	4649088	12603392	38899008	80630592	134504128
	Version 1	425664	474688	620992	1000960	2222144	5510080	14948608	30203840	49794432
	Version 2	376128	433920	569792	872064	2057920	5242944	14151104	28757184	47204544
4	Sequential	750336	901696	1269376	1972352	4659520	11373376	32809856	65035840	106478976
	Version 1	367616	444736	604544	851136	1826432	3958976	10184896	19205440	30339584
	Version 2	343424	400832	550936	830720	1757888	3792064	9730432	17825152	28457536
5	Sequential	822784	1018304	1306048	1955328	4654272	10855744	28995648	57138240	93126528
	Version 1	473024	546944	714112	998272	1960192	3827904	8761344	15502848	24351232
	Version 2	418752	534016	650624	845248	1757760	3547072	7691264	13822592	21229056
6	Sequential	938304	1081152	1500608	2162688	4638784	9918720	26757312	51416768	83520576
	Version 1	486144	546560	705920	1037824	1775872	3321984	7201600	12576000	19990272
	Version 2	430016	496640	665792	914688	1658048	2951808	6369024	11107712	17152320

Table 31: Running Time Of Stack For 3D With 6 Vertices Using Transputer

Part	*Time*	*Size Of Set*			
		25	50	100	200
2	Sequential	1613888	2518720	4328000	7016704
	Version 1	887680	1490496	2342848	4148544
	Version 2	876160	1473920	2318016	4118016
3	Sequential	2118656	3753984	5316224	8355520
	Version 1	1365952	2313344	3315584	5001344
	Version 2	1099072	1956480	2525248	3624576
4	Sequential	2801024	5355328	8620032	16652480
	Version 1	1204992	2271040	3594496	5754752
	Version 2	1251840	2190080	3635456	5475136
5	Sequential	2689472	5117632	9590400	18109696
	Version 1	1352000	2704128	5608320	6496896
	Version 2	1238976	2294336	3270528	5327488
6	Sequential	3044608	5767680	11829888	19591680
	Version 1	1607744	2872064	4289280	6328704
	Version 2	1406848	1976192	3920896	5390208

Table 32: Running Time Of Stack For 3D With 12 Vertices Using Transputer

Part	Time	*Size Of Set*			
		25	50	100	200
2	Sequential	2707840	3503232	4792960	7417216
	Version 1	1405120	2054848	2584704	3931520
	Version 2	1395584	2038912	2561152	3896512
3	Sequential	3626176	4275904	6152640	11295680
	Version 1	12238592	2921280	3739584	5685376
	Version 2	2113024	2379008	3346304	5478080
4	Sequential	4134080	5941632	8870336	15343680
	Version 1	1801408	2654592	3607168	5258624
	Version 2	1854912	2526784	3475392	4946240
5	Sequential	4945856	7091968	11083964	18197056
	Version 1	2257536	3128192	4706880	6652160
	Version 2	2485568	3754688	5240256	6336448
6	Sequential	4924864	6885056	12945408	25192704
	Version 1	2480960	3320128	4620864	7003008
	Version 2	2097856	2915008	4395648	6429568

Table 33: Running Time Of Stack For 4D With 6 Vertices Using Transputer

		Size Of Set			
Part	*Time*	25	50	100	200
2	Sequential	1837056	2602432	3740224	5868160
	Version 1	1097088	1528064	2167360	3173312
	Version 2	1086272	1512960	2147264	3144384
3	Sequential	2411904	4051264	6427584	7928320
	Version 1	1468352	2325504	3569792	4021888
	Version 2	1256960	2031168	3090496	3785088
4	Sequential	3520768	6054336	8215808	10488384
	Version 1	1523264	2288000	3387648	4261504
	Version 2	1429248	2363200	3132288	3833792
5	Sequential	3485632	6222080	8864256	13702720
	Version 1	1988736	3469376	4748096	6113152
	Version 2	1786752	2794560	3770176	4352064
6	Sequential	3565056	5774528	10247040	17050240
	Version 1	1734720	2946624	4520576	5384192
	Version 2	1341440	2407232	3800000	5399488

Table 34: Running Time Of Recursion For 2D With 26 Vertices Using Transputer

Part	*Time*	*Size Of Set*							
		50	100	200	500	1000	2000	3000	4000
2	Sequential	1086912	1405568	2441024	6365888	16630848	53746624	109602624	186876416
	Version 1	624832	829504	1425664	3478720	8802688	27438976	55138496	93090304
	Version 2	604096	807168	1394752	3448768	8746816	27392896	54911104	92911936
3	Sequential	1141184	1436928	2235456	5372736	13847936	41110336	81103680	135746432
	Version 1	693248	829056	1251840	2575936	5879168	15846656	30328192	49576128
	Version 2	656448	819136	1204608	2456704	5587392	15012032	28777792	47001728
4	Sequential	1315264	1611072	2363584	5141504	11864320	33711232	65517888	108934336
	Version 1	692416	834432	1251008	2184064	4210304	10780416	19281280	31350144
	Version 2	592320	708800	1029824	1953280	3937024	10149184	18082432	28910528
5	Sequential	1467840	756544	2473344	5016128	11137344	30450496	57514880	94030656
	Version 1	804160	1014016	1314624	2225856	4019712	8996928	15780416	24513600
	Version 2	713152	865344	1181568	2011776	3535424	7932736	13983744	21673728
6	Sequential	1672640	2037504	2751424	5279616	11241024	27797888	51681280	82819712
	Version 1	778432	904704	1256896	2155904	3856512	7952064	13371968	19856448
	Version 2	737920	879296	1153024	1964800	3533760	6902528	11560192	16994176

Table 35: Running Time Of Recursion For 2D With 16 Vertices Using Transputer

Part	*Time*	*Size Of Set*								
		25	50	100	200	500	1000	2000	3000	4000
2	Sequential	524672	680960	1000192	1917888	5597696	15944512	50916928	107224896	183449024
	Version 1	312448	421504	615680	1131200	3096128	8390400	26081856	54026432	91068608
	Version 2	284224	390464	595904	1101952	3079168	8317440	25979584	53880384	90893632
3	Sequential	578752	721984	995264	1619392	4495680	12371968	38701312	79100736	134215424
	Version 1	391104	433792	591680	917504	2133504	5212480	15108864	29689984	49289344
	Version 2	361920	447104	609408	853952	2052928	5064320	14374208	28314496	46901952
4	Sequential	670784	833856	1093376	1780416	4323136	11003648	32586048	64841600	105794752
	Version 1	369152	468032	607936	930688	1836672	3942976	9965440	18709632	29949312
	Version 2	321280	399360	506240	798144	1628800	3562368	9359488	18129216	28799808
5	Sequential	775808	915840	1184256	1870720	4454144	10226496	28328320	55726720	92217344
	Version 1	450816	550976	715328	982592	1843968	3421824	8363648	14984768	23963392
	Version 2	401728	461120	603584	835072	1672512	3117120	7419456	13268608	21041920
6	Sequential	862784	1050880	1394496	2105664	4529216	9566976	26142976	51012736	82154368
	Version 1	439872	513216	683392	969408	1719872	3103360	7178816	12498496	19456448
	Version 2	394496	459584	625920	916800	1618816	2728832	6228288	10932800	16627200

Table 36: Running Time Of Recursion For 3D With 6 Vertices Using Transputer

		Size Of Set			
Part	*Time*	25	50	100	200
2	Sequential	1555456	2280896	3749312	5983808
	Version 1	902464	1319104	2066560	3390400
	Version 2	892544	1302784	2042432	3358720
3	Sequential	2096320	3490944	5038720	7524544
	Version 1	1420352	1845952	2823744	4061376
	Version 2	1166464	1890560	2585536	3609728
4	Sequential	2659008	4895104	7717056	12508160
	Version 1	1264512	2113216	3134912	4577472
	Version 2	1387264	2125056	3167488	4840512
5	Sequential	2989824	5634880	8094208	13096640
	Version 1	1472832	2787136	4387648	6222464
	Version 2	1214592	2474112	3263872	5000640
6	Sequential	2704320	5427840	9974400	
	Version 1	1644928	2295936	3680704	
	Version 2	1132288	2059072	3854016	

Table 37: Running Time Of Recursion For 3D With 12 Vertices Using Transputer

		Size Of Set			
Part	*Time*	25	50	100	200
2	Sequential	2584512	3331840	4770560	6927104
	Version 1	1380736	1761216	2694848	4052160
	Version 2	1289536	1738688	2042432	3777024
3	Sequential	3185856	4122624	6402240	9962112
	Version 1	1982720	2553536	3836608	5496256
	Version 2	1802816	2289664	3408768	4964736
4	Sequential	3666368	6046784	8745664	13497216
	Version 1	1670016	2411200	3779456	5432576
	Version 2	1623680	2515072	3610240	5287744
5	Sequential	4245184	5951104	8616064	
	Version 1	2361408	3002112	4565184	
	Version 2	2149312	2970432	4199808	
6	Sequential	4228992	6436160	10964032	
	Version 1	2207168	2984640	4672576	
	Version 2	1858944	2541184	4352576	

Table 38: Running Time Of Stack For 3D With 6 Vertices Using Transputer

Part	*Time*	*Size Of Set*			
		25	50	100	200
2	Sequential	1803392	2635200	3635584	5958848
	Version 1	1036736	1526720	2012480	3430080
	Version 2	9244096	1503488	1990848	3405184
3	Sequential	2402816	4357568	5746752	8037376
	Version 1	1363456	2272512	3075712	4326144
	Version 2	1255936	2337216	2594240	3977152
4	Sequential	2533952	4580736	7030464	9396416
	Version 1	1355904	2256000	3030400	4413824
	Version 2	1155264	2044352	2966720	4221824
5	Sequential	3601536	6072960	8953344	12732544
	Version 1	1745600	2949568	3596096	6116096
	Version 2	1668992	2911040	3581376	4460864
6	Sequential	3312768	5658688	9662528	16160768
	Version 1	1611776	2849664	3935168	5529920
	Version 2	1255104	2088128	3363840	4992960

Table 39: Partitioning Of 2-D – 26 vertices and 4-D – 6 vertices using 1000 points Between 2 – 6 Partitions

2D																								
Lex					Random					Shell					New_Shell					Bucket				
2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6
500	334	250	200	167	500	328	250	207	172	459	200	115	79	57	835	611	459	357	299	388	166	166	166	166
500	333	250	200	167	500	322	250	199	176	541	589	344	211	143	165	296	376	370	312	612	222	222	222	0
	333	250	200	167		350	250	208	182		211	406	365	259		93	100	151	224		612	309	0	222
		250	200	167			250	205	156			135	243	330			65	76	72			303	309	0
			200	166				181	146				102	128				46	58				303	309
				166					168					83					35					303
4D																								
Lex					Random					Shell					New_Shell					Bucket				
2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6
500	334	250	200	167	500	328	250	207	172	604	337	220	152	99	987	948	892	851	810	204	28	0	0	0
500	333	250	200	167	500	322	250	199	176	396	473	384	333	278	13	47	95	120	138	796	176	204	121	28
	333	250	200	167		350	250	208	182		190	338	268	227		5	11	23	39		796	0	83	176
		250	200	167			250	205	156			58	219	246			2	4	8			796	0	0
			200	166				181	146				28	135				2	3				796	398
				166					168					15					2					398

Appendix B

Graphs

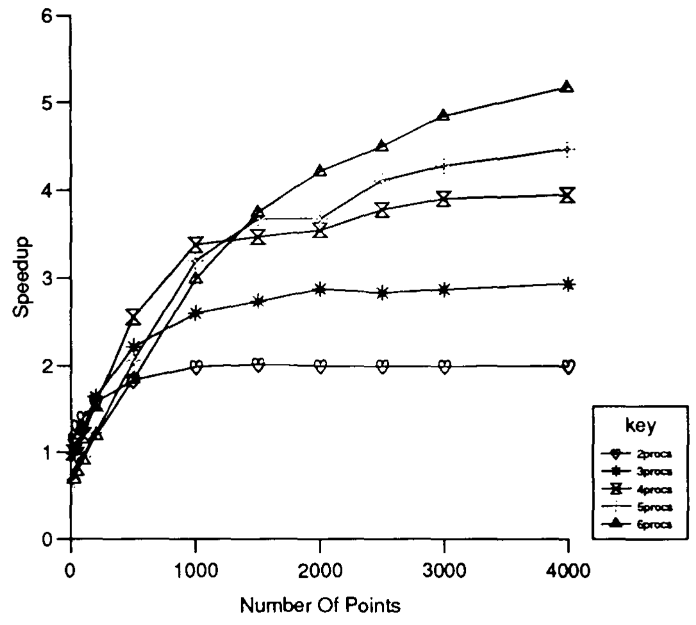

Figure B.1: Recursive Version 2D 3 Vertices Using Threads

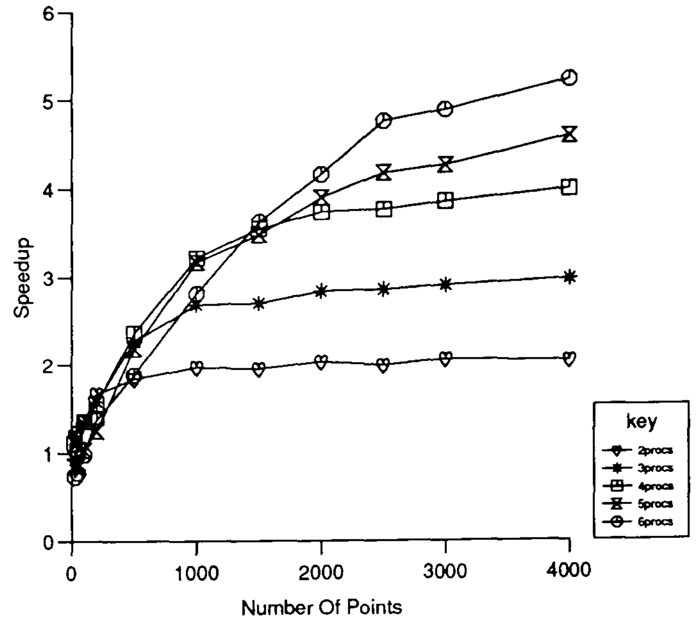

Figure B.2: Recursive Version 2D 4 Vertices Using Threads

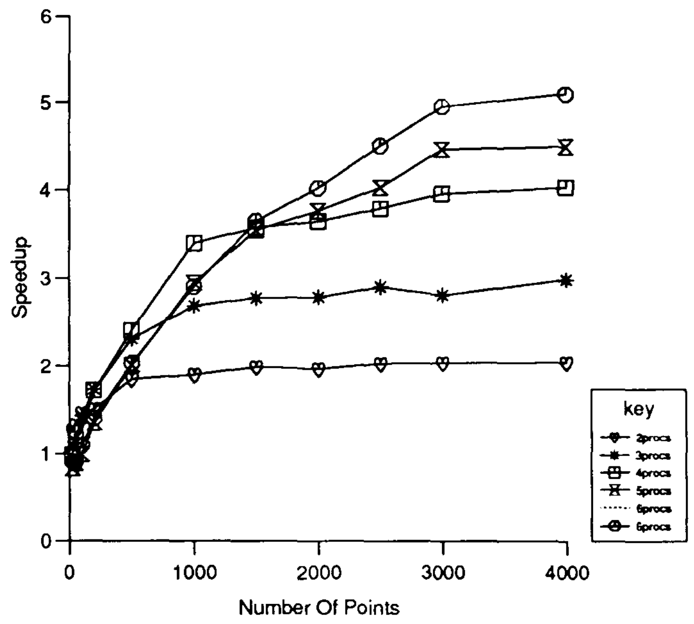

Figure B.3: Recursive Version 2D 6 Vertices Using Threads

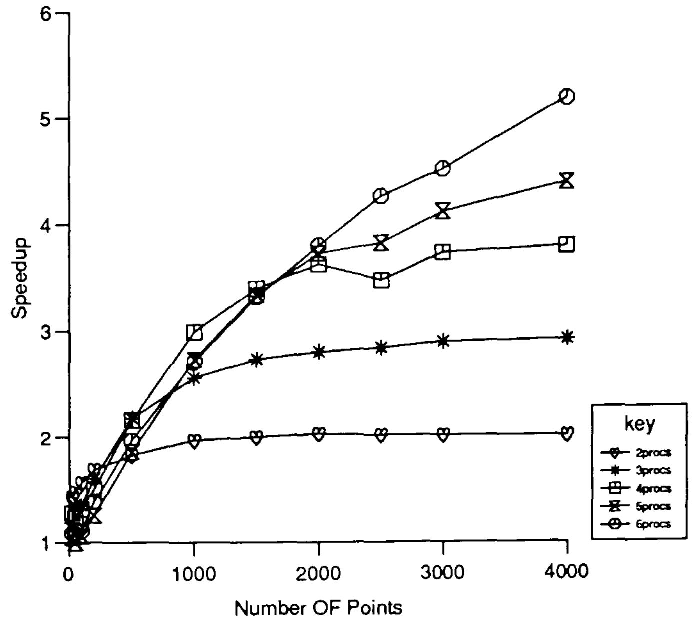

Figure B.4: Recursive Version 2D 16 Vertices Using Threads

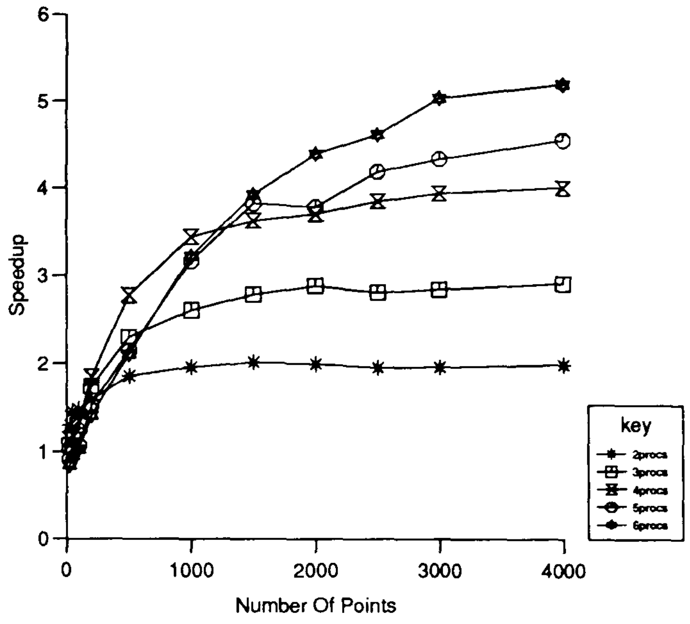

Figure B.5: Recursive Version 3D 3 Vertices Using Threads

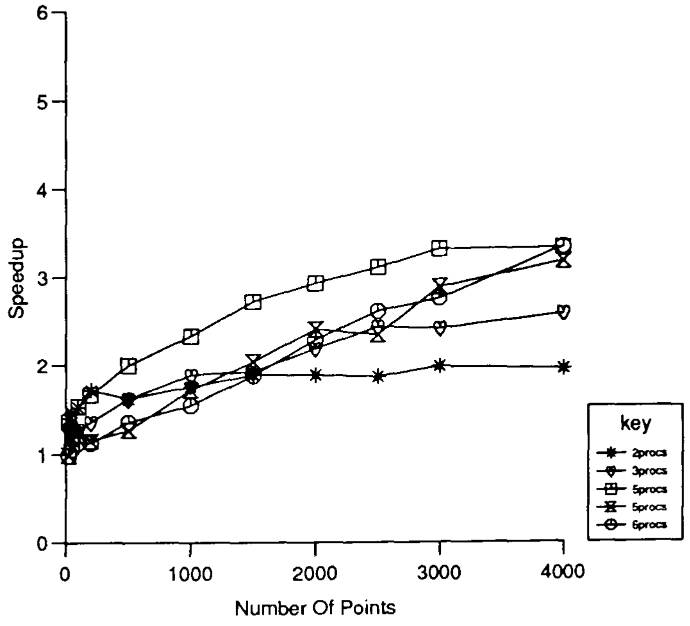

Figure B.6: Recursive Version 3D 4 Vertices Using Threads

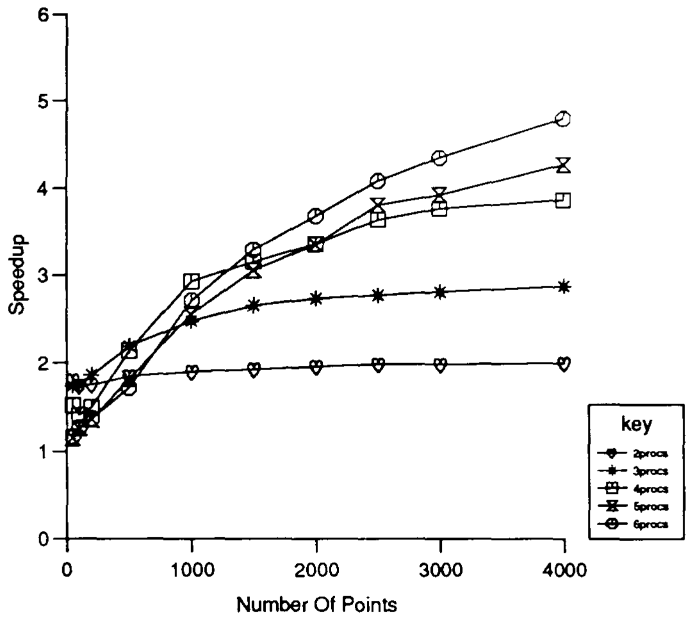

Figure B.7: Recursive Version 3D 6 Vertices Using Threads

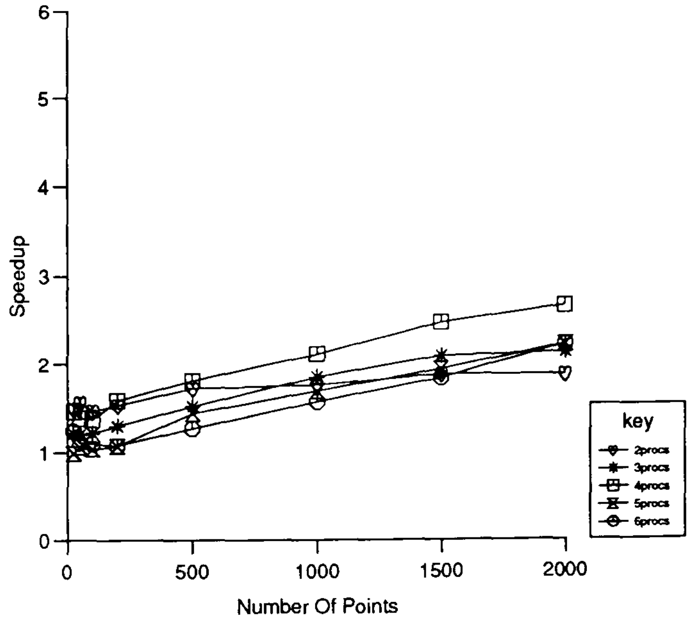

Figure B.8: Recursive Version 3D 12 Vertices Using Threads

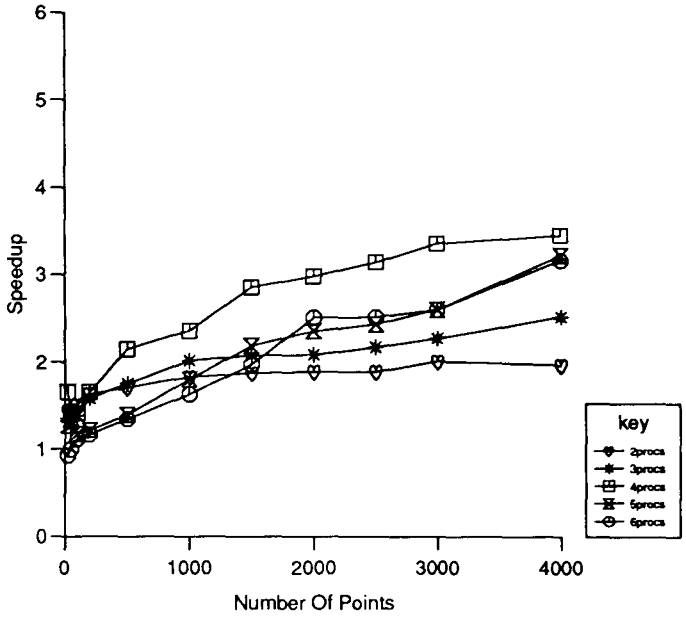

Figure B.9: Recursive Version 4D 4 Vertices Using Threads

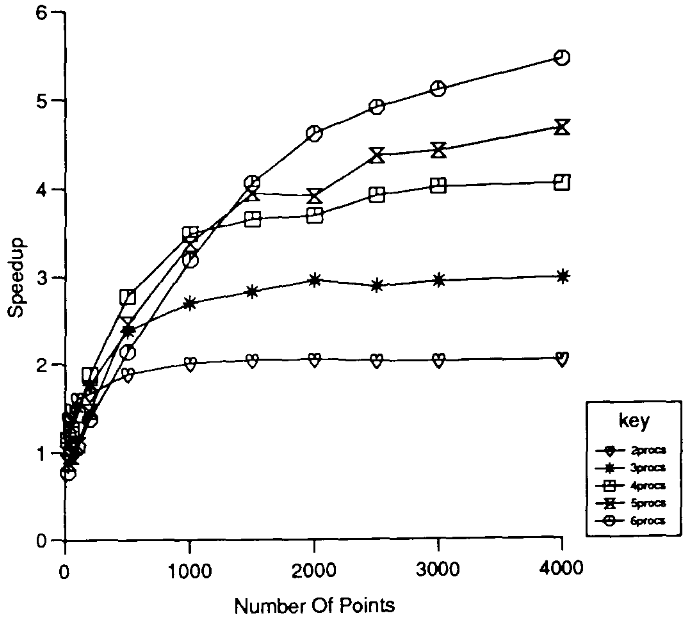

Figure B.10: Recursive Version 2D 3 Vertices Using Microthreads

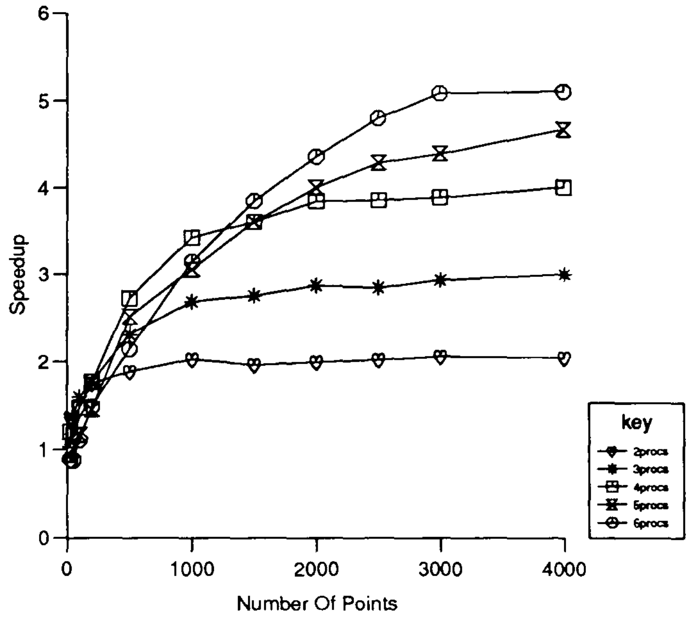

Figure B.11: Recursive Version 2D 4 Vertices Using Microthreads

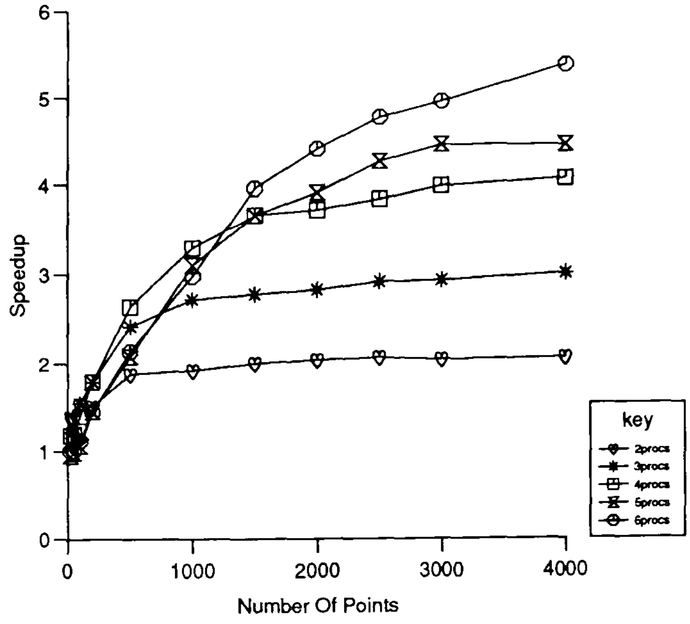

Figure B.12: Recursive Version 2D 6 Vertices Using Microthreads

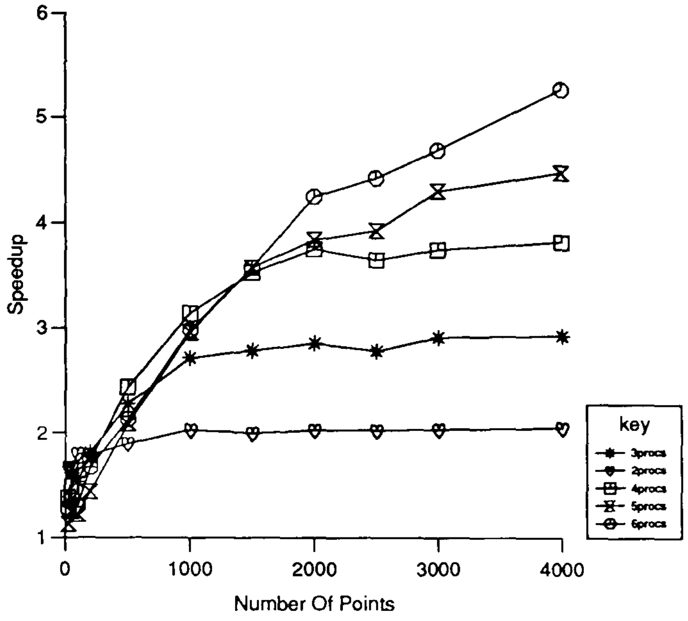

Figure B.13: Recursive Version 2D 16 Vertices Using Microthreads

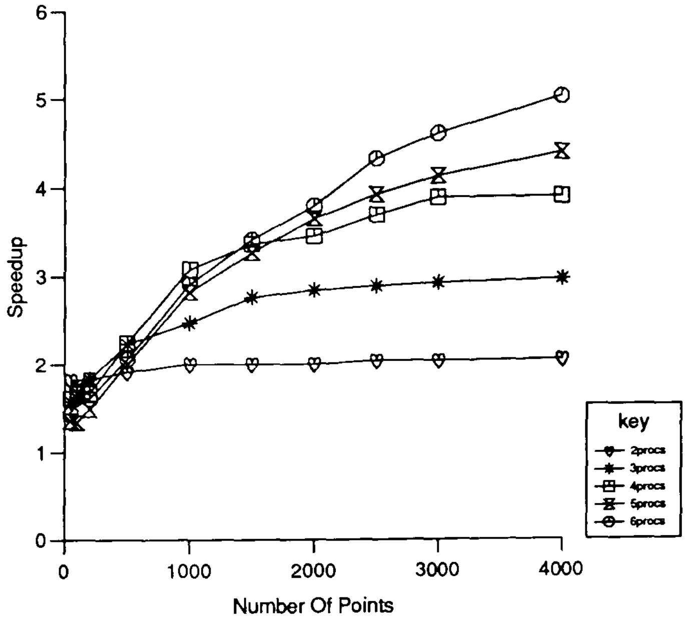

Figure B.14: Recursive Version 2D 26 Vertices Using Microthreads

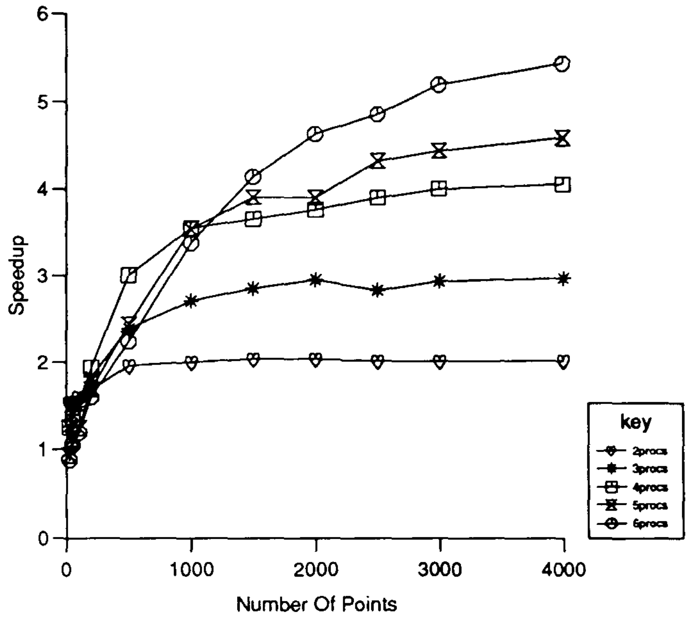

Figure B.15: Recursive Version 3D 3 Vertices Using Microthreads

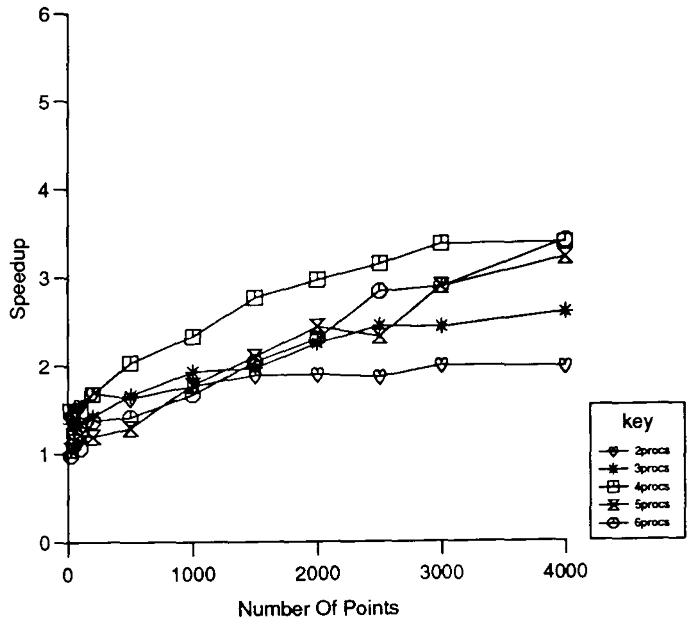

Figure B.16: Recursive Version 3D 4 Vertices Using Microthreads

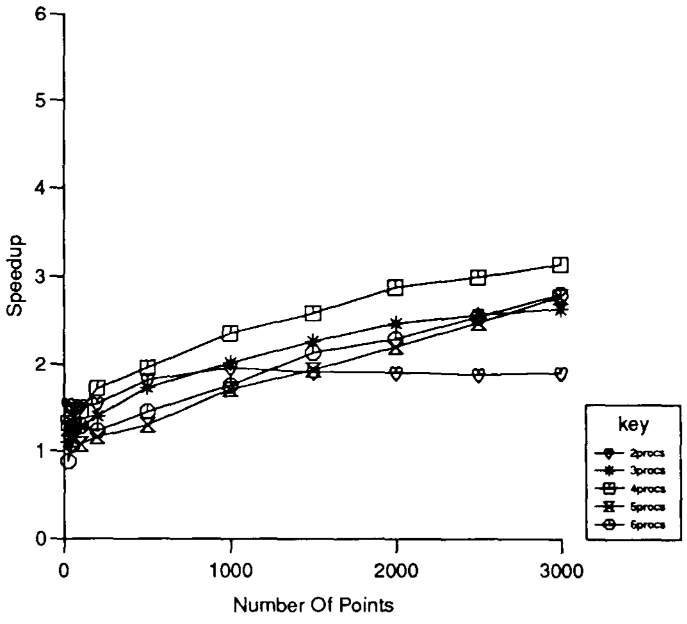

Figure B.17: Recursive Version 3D 6 Vertices Using Microthreads

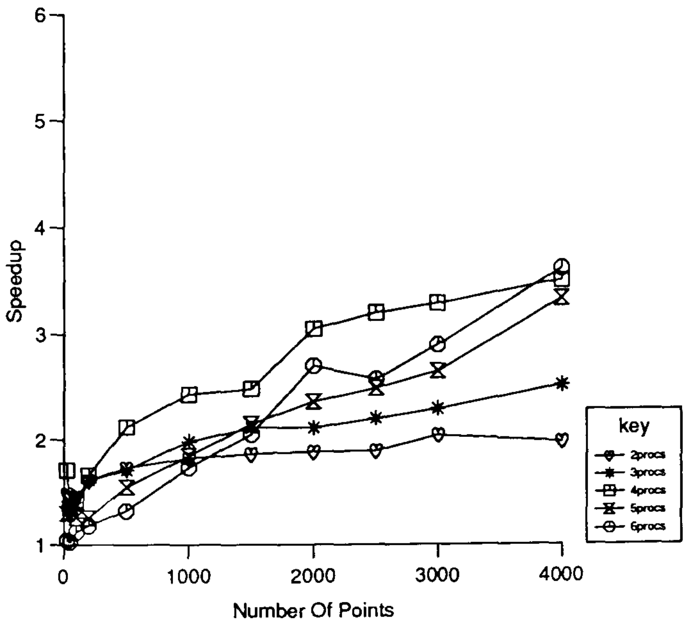

Figure B.18: Recursive Version 4D 4 Vertices Using Microthreads

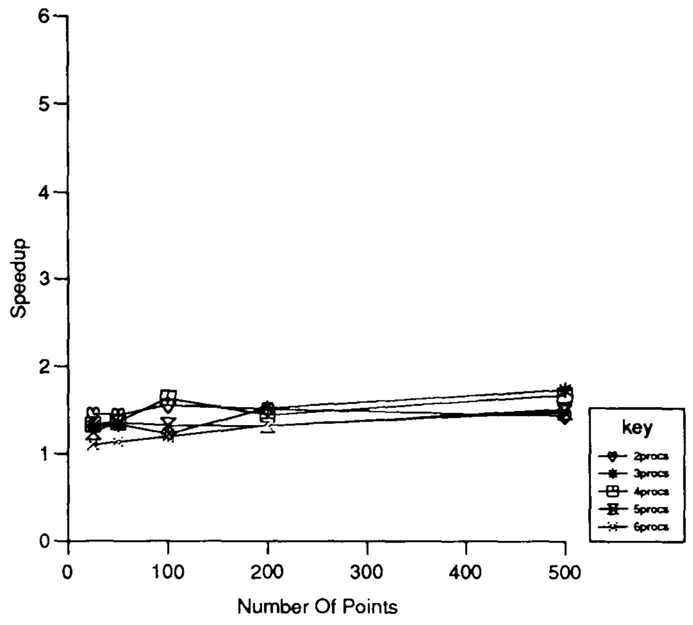

Figure B.19: Recursive Version 4D 6 Vertices Using Microthreads

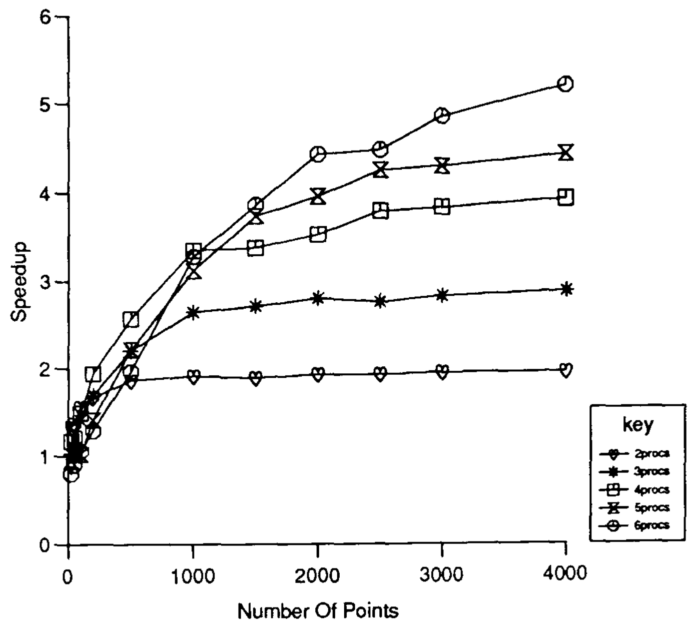

Figure B.20: Stack Version 2D 3 Vertices Using Threads

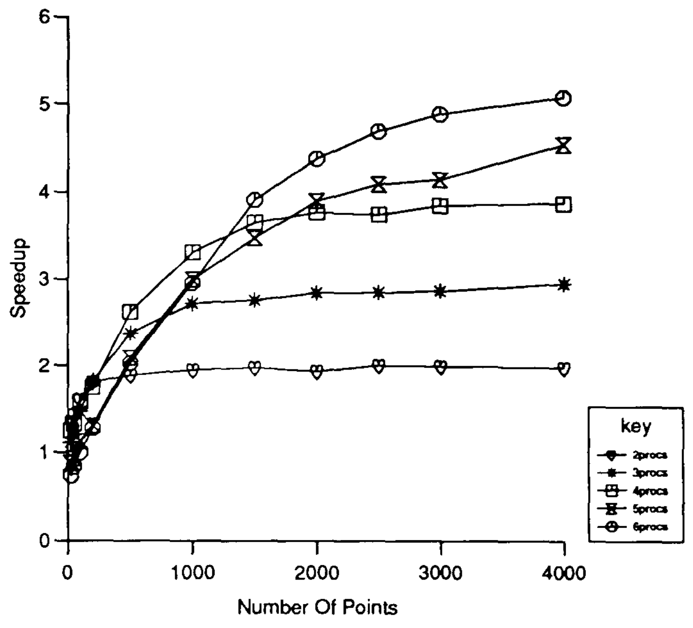

Figure B.21: Stack Version 2D 4 Vertices Using Threads

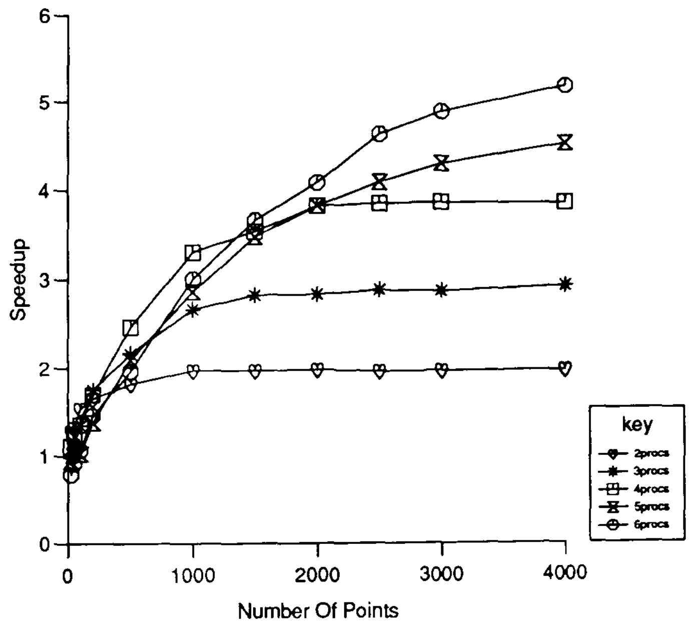

Figure B.22: Stack Version 2D 6 Vertices Using Threads

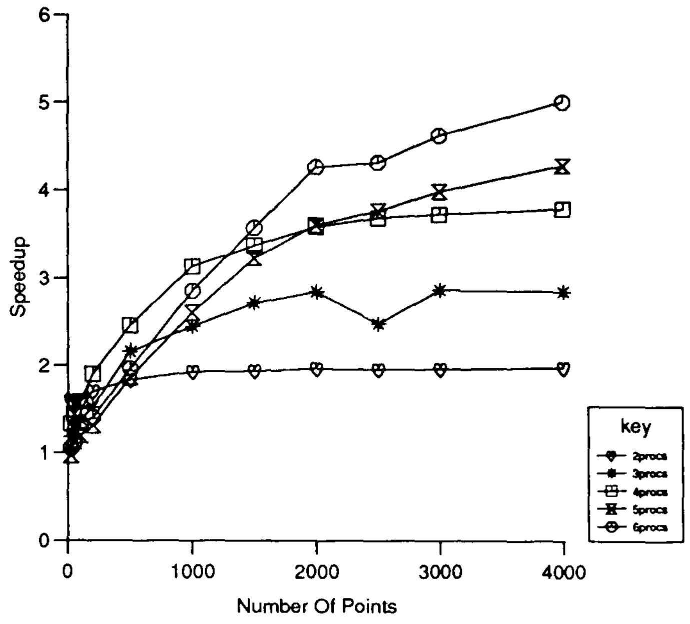

Figure B.23: Stack Version 2D 16 Vertices Using Threads

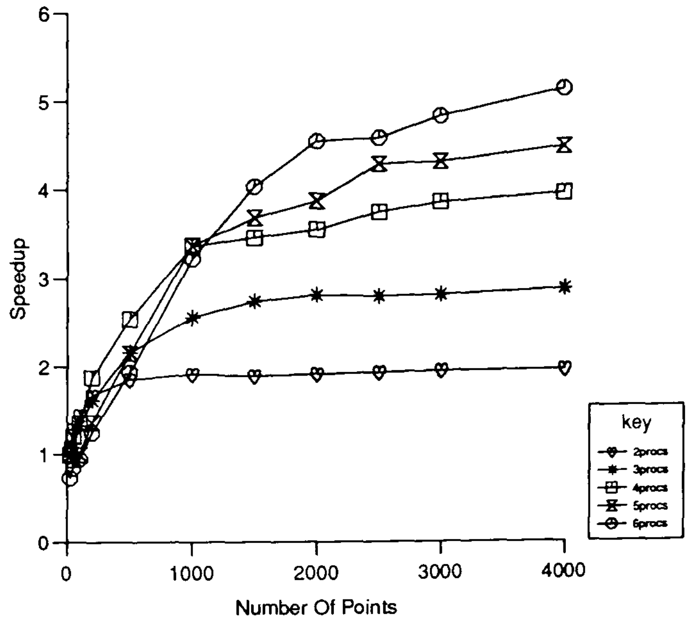

Figure B.24: Stack Version 3D 3 Vertices Using Threads

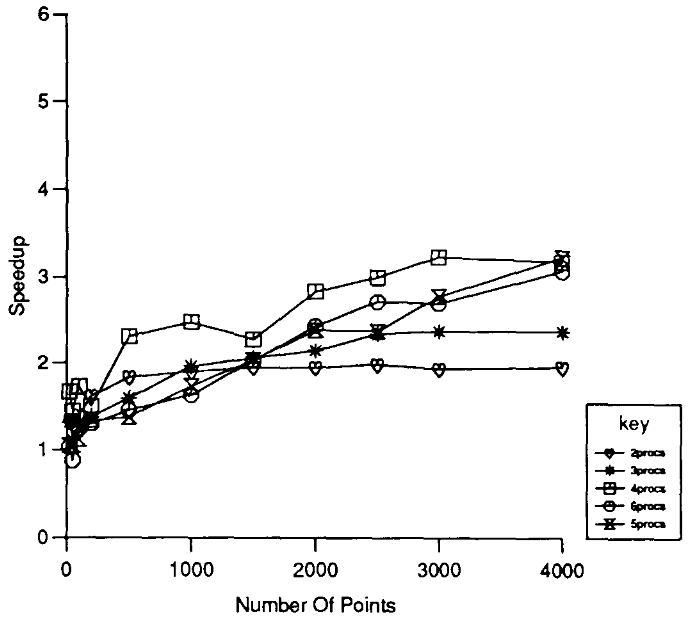

Figure B.25: Stack Version 3D 4 Vertices Using Threads

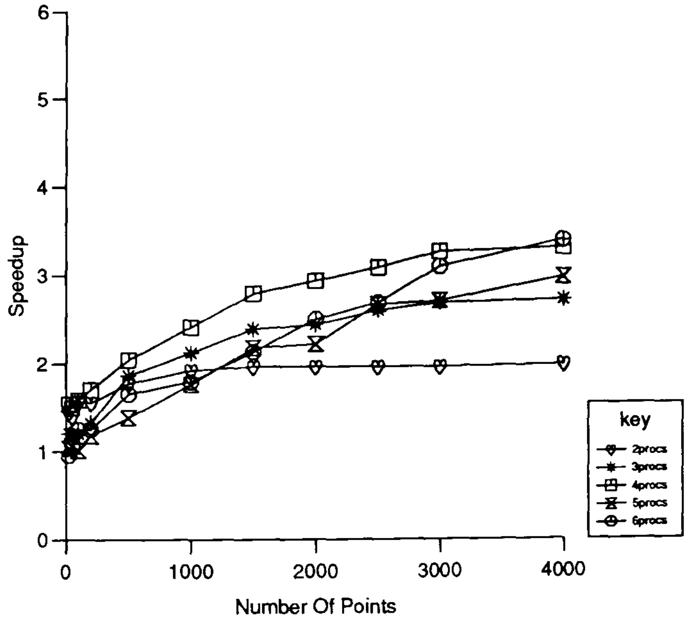

Figure B.26: Stack Version 3D 6 Vertices Using Threads

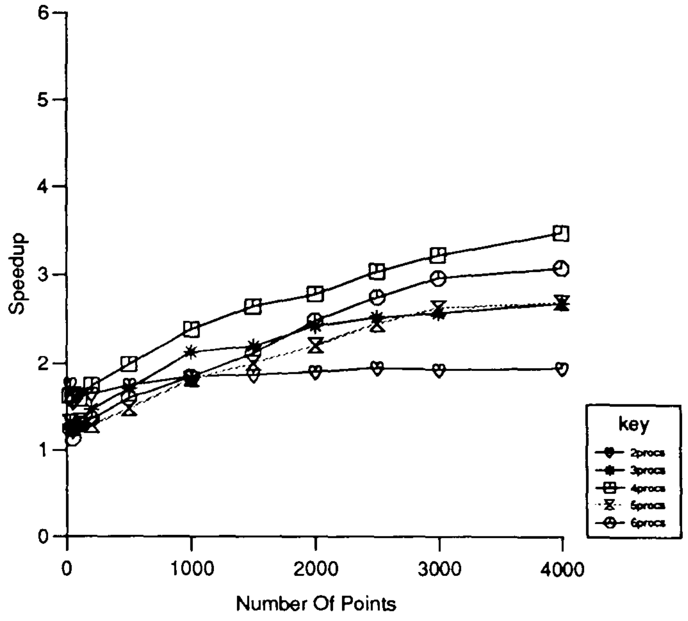

Figure B.27: Stack Version 3D 12 Vertices Using Threads

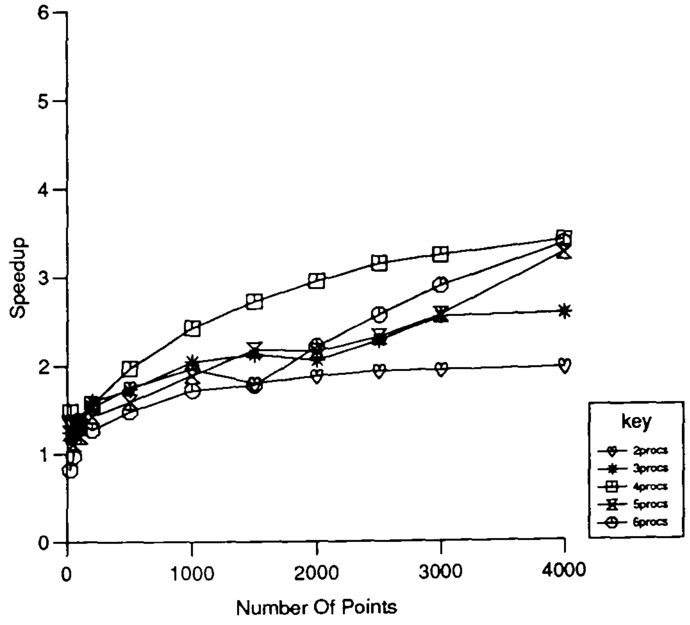

Figure B.28: Stack Version 4D 4 Vertices Using Threads

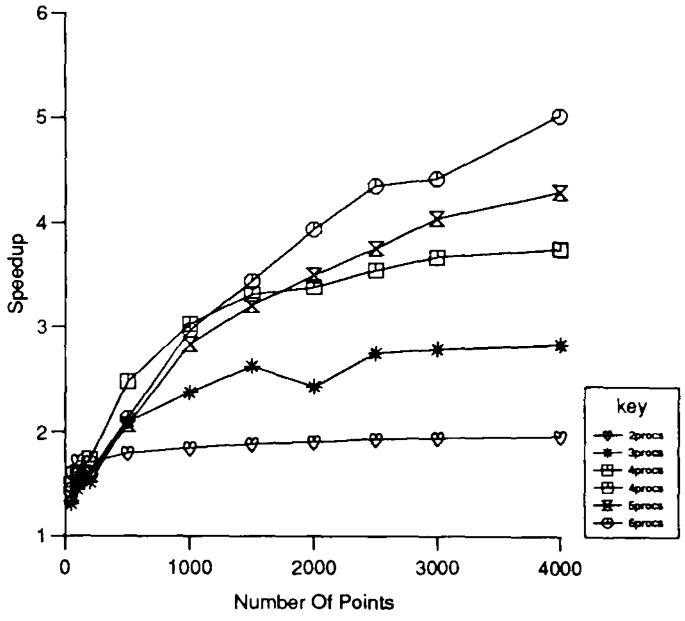

Figure B.29: Stack Version 2D 26 Vertices Using Microthreads

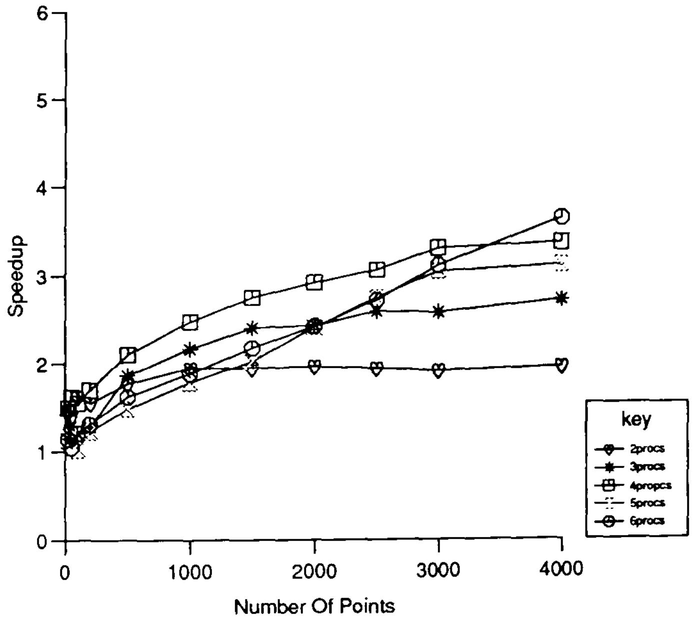

Figure B.30: Stack Version 3D 6 Vertices Using Microthreads

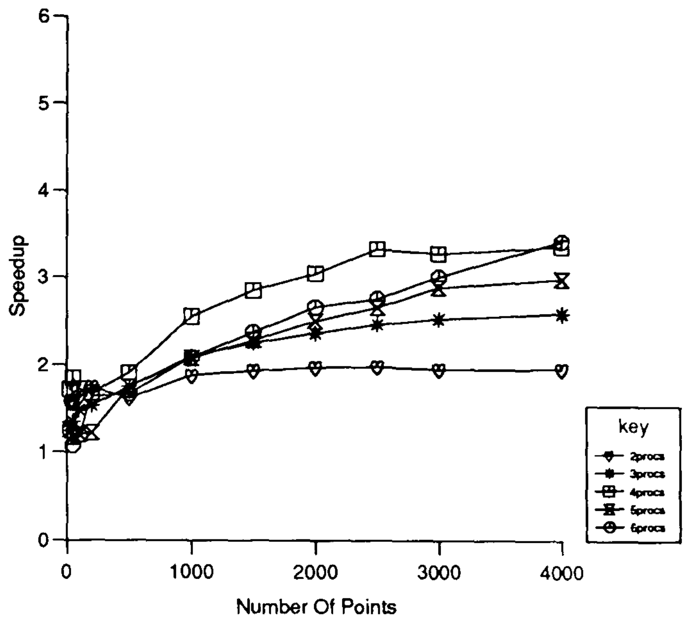

Figure B.31: Stack Version 4D 6 Vertices Using Microthreads

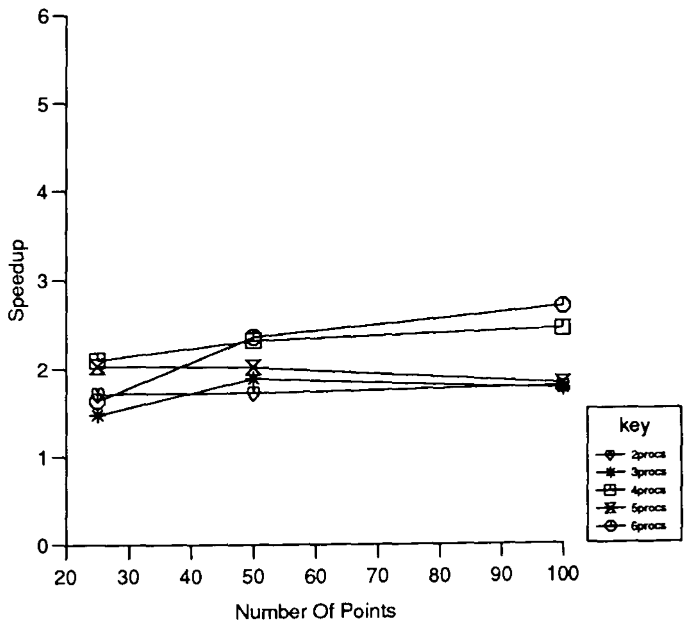

Figure B.32: Recursion Version 1 3D 6 Vertices Using Transputer

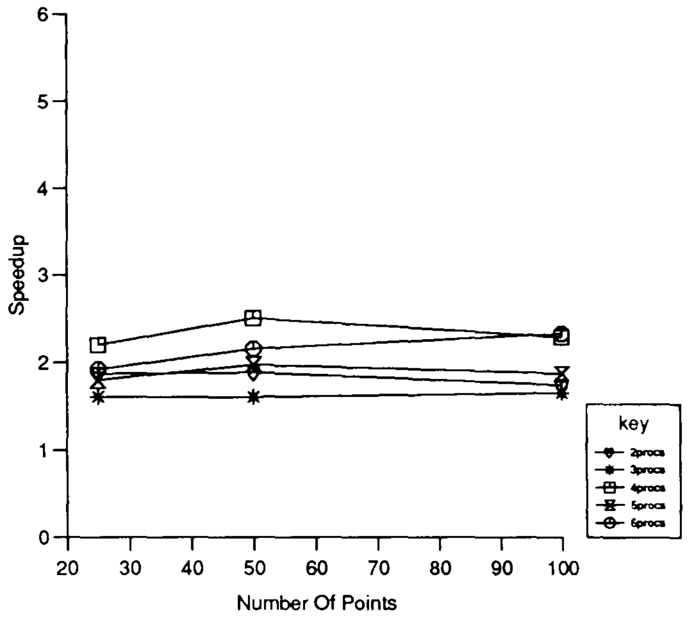

Figure B.33: Recursion Version 1 3D 12 Vertices Using Transputer

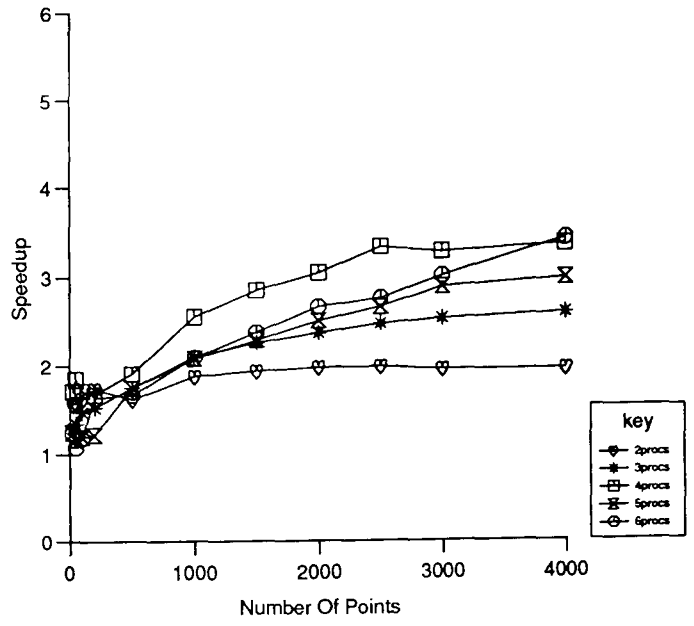

Figure B.34: Stack Version 4D 6 Vertices Using Microthreads

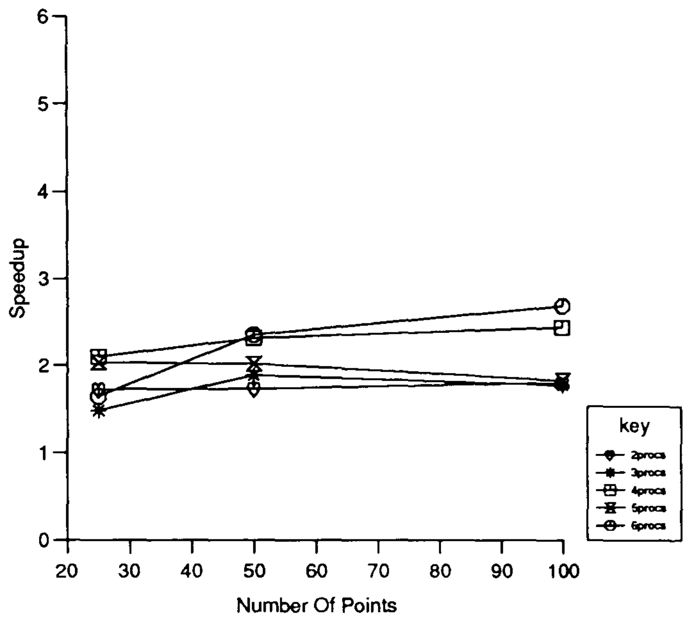

Figure B.35: Recursion Version 1 3D 6 Vertices Using Transputer

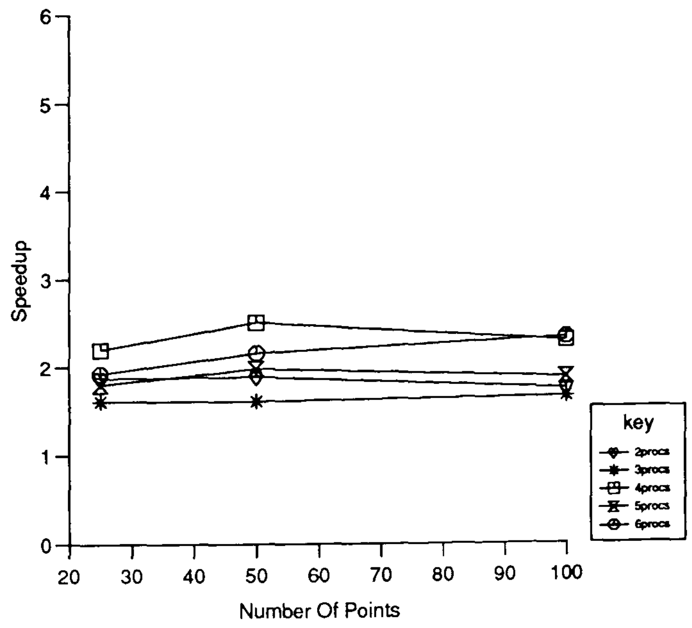

Figure B.36: Recursion Version 1 3D 12 Vertices Using Transputer

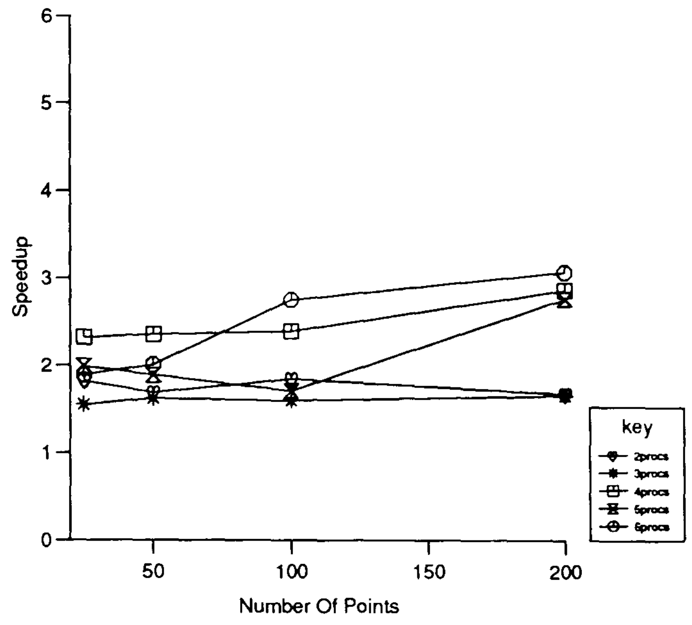

Figure B.37: Stack Version 1 3D 6 Vertices Using Transputer

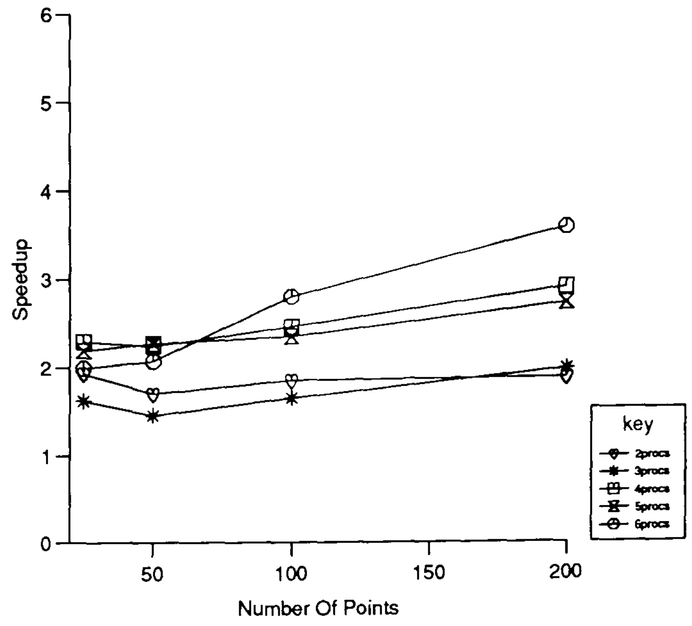

Figure B.38: Stack Version 1 3D 12 Vertices Using Transputer

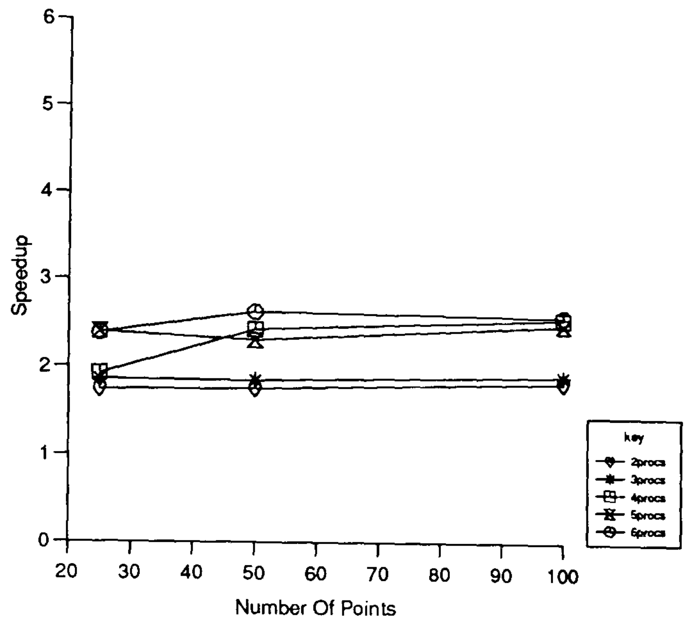

Figure B.39: Recursion Version 2 3D 6 Vertices Using Transputer

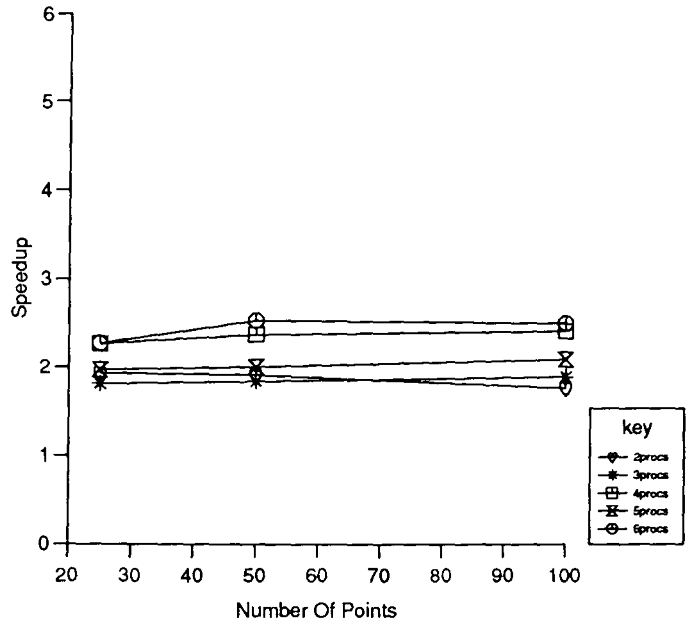

Figure B.40: Recursion Version 2 3D 12 Vertices Using Transputer

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;
}
   int 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 GetNext_Point(P)
POINTS P;
{
    if (IsEmpty_Plist(P) == TRUE) return P;
    else return P->next;
}
    POINTS GetPrev_Point(P)
POINTS P;
{
if (IsEmpty_Plist(P) == TRUE) return P;
else return P->prev;
}
    Read_Point(P, n, v)
POINTS P;
Vector v;
int n;
{
    int i;
    if (IsEmpty_Plist(P) == FALSE)

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

```
    int Compare_Points(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 IsMember_Plist(P, n, v)
POINTS P;
Vector v;
int n;
{
    POINTS Q;
    int i;
    Vector w;
    int Match;
    if ( IsEmpty_Plist(P) == TRUE )
        Match = FALSE;
    else
    {
        Q = P;
        do{
            Read_Point(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 = GetNext_Point(Q);
        } while ( (P != Q) && (Match == FALSE));
    };
    return Match;
}
```

```
  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;
}
```

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;
}
```

```
    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 {
            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)))
```

```
                    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 ((flag == TRUE) || (i == GetNext_Point(j)) || (j == f))
                    flag = TRUE;
                else
                    flag = FALSE;
            } while ((swap == FALSE) && (flag == FALSE));
            if (flag == FALSE)
            {
                Write_Point(i, n, v2);
                Write_Point(j, n, v1);
            };
        } 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
Facets 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);
         S = Delete_Point(S);
         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)

```
POINTS S;
int 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)

```
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

```
/* matrix times a vector n*m 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;
        };
}
    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;
/* Givens triangularisation of an nxp matrix with rhs right handsides
stored in columns p+1 ... p+rhs, trangularise 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] = r;
                                M[j][i] = 0.0;
                            }
                        for(k=i+1; k<=p+rhs; k++)
                            {
                                r = M[j-1][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 subsitution of an nxp matrix with nxn upper triangular
portion and rhs v - result is in px1 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)
                if (fabs(M[i][i]) < TOL)
                    e[i] = 1.0;
                else
                    e[i] = 0.0;
            else
                if(fabs(M[i][i]) < TOL)
                    {
                        e[i] = 1.0;
                        for(j=i+1; j<=p; j++)
                            e[j] = 0.0;
                    }
                else
                    e[i] = (r) / M[i][i];
        };
}
```

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-1 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, r, 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];
            R = GetNext_Point(R);
        };

    /* insert normal */

    j = j + 1;
    for(i=1; i<=n; i++)
        Basis[i][j] = norm[i];
    Reduce(Basis, n, m, j-m);

    /* 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<=j; i++)
        {
            for(t=1; t<=k; t++)
                v[t] = Basis[t][i];
            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_Plist(S) == 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++)
                {
                    minJ[i] = P[i];
                    maxJ[i] = P[i];
                };
        }
    else
        {
            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++) maxJ[i] = P[i];
                    }
                else
                    if (temp < min)
                        {
                            min = temp;
                            for(i=1; i<=n; i++) minJ[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] = maxJ[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] = minJ[i];
                };
            Normalize(newnorm, n);
/* check this norm */
            sign = Check_Plane(S, n, newnorm, P0);
        };
/* check validity of norm */
    if ( sign == 0 )
        {
            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. */
POINTS S, AS, *F;
Vector norm;
int n, k;
{
    POINTS Q, P, Abar;
    Vector v, v1, v0;
    double x, r;
    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));
/* 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];
        Normalize(v,n);
        r = 0.0;
        for(j=1; j<=n; j++)
            r = r + v1[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 |= 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)
                {
/* 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 (IsEmpty_Plist(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 */
    sign = 0;
    R = S;
    do {
        Read_Point(R, n, P);
        t = 0.0;
        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);
    MakeEmpty_Plist(&T);
    T = Insert_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);
    initstate(seed, state, 256);
    srandom(seed);
    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;
/* to generate decimal places - prime number better */
                        frac = frac - (int) frac;
```

```
                        v[j] = (random()% ((int)temp + 1)) + small[j] + frac;
                    }
                    else
                    {
                        v[j] = small[j];
                    };
                };
            }while ((IsMember_Plist(*CH,n,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{
        r = r+1;
        T = GetNext_Point(T);
    }while (T != *CH);
    if(r != *npts)
    {
        PrintWarn( "Generate test" , "multiple points in test");
        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", &x, &y, &z, &r, &pts);
/* calculate number of points */
    count = 0;
    step = 2*pi/pts;
    t = 0.0;
    do{
        count = count + 1;
        t = t + step;
    }while(t < (2*pi));
/* print header for result file */
    printf("3 %d ", count+2);
/* generate points */
    t = 0.0;
    do{
        printf("%f %f %f ", x + r*cos(t), y + r*sin(t), z);
        t = t + step;
    }while(t < (2*pi));
    printf("%f %f %f ", x, y, z+r);
    printf("%f %f %f ", x, y, z-r);
}
```

C.2.3 Test To Generate Rectangles In Levels

```
#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; */
main()
{
    int i, count;
    float x, y, z, t;
    float r, adj, levels;
    float offset, step, theta;
    scanf("%f %f %f %f %f %f", &x, &y, &z, &r, &levels, &theta);
    count = 8*levels;
    printf("3 %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);
/* move to new level */
        r = 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

```
#define VALID 3
#define INVALID -2
#define SYNC 1
#define STOP 0
#define HULL 1
#define MERGE 2
struct mess_st {
        int data_flag;
        Vector data_v;
        ;
struct mess_st 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);
  *n = m;
  *P = R;
}

  Transmit_Elist(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 mess_st local_message;
```

```
    MakeEmpty_Elist(&R);
    do{
        csn_rx(channel, chan_id, (char *) &local_message, sizeof(local_message));
        if (local_message.data_flag != INVALID)
        {
            Receive_Plist(&P, &m, channel, chan_id);
            R = Insert_Edge(R, m, P, local_message.data_v);
        };
    }while( local_message.data_flag != INVALID);
    *E = R;
    *n = m;
}
```

C.3.2 Build Files For Partitioning Method

```
#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 =      if (2*parts-1>= MAXPROCS)
        {
            printf("Not Enough processors available ");
            exit(1);
        }
/* build process objects */
    masterGRP_ptr = cs_group(NULL, "masterGRP");
    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 processes */
    cs_exe( masterGRP_ptr, "treemaster", "treemaster", "int arg", parts, 0);
    for(i=0; i< parts; i++)
        cs_exe( leafGRP_ptr[i], "treeleaf", "treeleaf", "int arg", i+parts, 0);
    for(i=1; i<parts; i++)
        cs_exe( nodeGRP_ptr[i], "treenode", "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

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");
    managerGRP_ptr = cs_group(NULL, "managerGRP");
    for(i=1; i<= parts-2; i++)
```

```
        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 hull 2.c

```
    /* 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( masterGRP_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_option( masterGRP_ptr, "commit", "transputer");
     for(i=1; i<= parts-1; i++)
         cs_option( slaveGRP_ptr[i], "commit", "transputer");
     printf("Go ");
 /* load computing surface */
     cs_load();
     printf("stop ");
 }
```