9/3910814

MAPPING UNSTRUCTURED MESH CODES ONTO LOCAL MEMORY PARALLEL ARCHITECTURES

Beryl Wyn Jones

A thesis submitted in partial fulfilment of the requirements of the University of Greenwich for the degree of Doctor of Philosophy.

September 1994

This research programme was funded by SERC

Centre for Numerical Modelling and Process Analysis
School of Mathematics, Statistics and Computing
University of Greenwich
London U.K.

Table of Contents

Acknowledgements

I would like to thank my supervisors Professor Martin Everett and Professor Mark Cross for their invaluable advice, encouragement and guidance received during the course of this research.

I would also like to thank Steve Johnson for the indispensable discussions and assistance at various stages of the research.

Peter Lawrence and Kevin McManus are also gratefully acknowledged.

Thanks also go to the staff at the School of Mathematics, Statistics and Computing and to the postgraduates at the Centre for Numerical Modelling and Process Analysis of the University of Greenwich for providing a good working environment.

Thanks also to Frank for those unforgettable three days at Prague.

Finally, the financial support provided by the Science and Engineering Research Council is gratefully acknowledged.

Abstract

Initial work on mapping CFD codes onto parallel systems focused upon software which employed structured meshes. Increasingly, many large scale CFD codes are being based upon unstructured meshes. One of the key problem when implementing such large scale unstructured problems on a distributed memory machine is the question of how to partition the underlying computational domain efficiently. It is important that all processors are kept busy for as large a proportion of the time as possible and that the amount, level and frequency of communication should be kept to a minimum.

Proposed techniques for solving the mapping problem have separated out the solution into two distinct phases. The first phase is to partition the computational domain into cohesive sub-regions. The second phase consists of embedding these sub-regions onto the processors. However, it has been shown that performing these two operations in isolation can lead to poor mappings and much less optimal communication time.

In this thesis we develop a technique which simultaneously takes account of the processor topology whilst identifying the cohesive sub-regions. Our approach is based on an unstructured mesh decomposition method that was originally developed by Sadayappan et al [SER90] for a hypercube. This technique forms a basis for a method which enables a decomposition to an arbitrary number of processors on a specified processor network topology. Whilst partitioning the mesh, the optimisation method takes into account the processor topology by minimising the total interprocessor communication.

The problem with this technique is that it is not suitable for dealing with very large meshes since the calculations often require prodigious amounts of computing processing power.

The problem can be overcome by creating clusters of the original elements and using this to create a reduced network which is homomorphic to the original mesh. The technique can now be applied to the image network with comparative ease. The clusters are created using an efficient graph bisection method. The coarseness of the reduced mesh inevitably leads to a degradation of the solution. However, it is possible to refine the resultant partition to recapture some of the richness of the original mesh and hence achieve reasonable partitions.

One of the issues to be addressed is the level of granuality to obtain the best balance between computational efficiency and optimality of the solution. Some progress has been made in trying to find an answer to this important issue.

In this thesis, we show how the above technique can be effectively utilised in large scale computations. Results include testing the above technique on large scale meshes for complex flow domains.

To Dad

Gresyn blodeuyn mor deg

Ei ffoi cyn fo'i adeg

Chapter 1
Introduction

1.1 Introduction

Many large scale computational problems are based on unstructured computational domains. By using unstructured meshes, this allows the code to cater for completely general geometries and hence a wide range of problems in both two and three space dimensions. Examples include unstructured grid calculations based on finite volume methods in computational fluid dynamics, or structural analysis problems based on finite element approximations.
Software packages have been developed with the intention of using the results of the analysis for solving such problems. Analysis is carried out for the selected input parameters and the results are interpreted for optimising a design. This iterative procedure requires interpretation of results and also uses a vast amount of time for solving a given problem. To reduce the computation time, various optimisation procedures have been incorporated into the code. One practicable approach is to use parallel computation techniques. Therefore, there is a demand for parallel computers and the development of parallel algorithms to execute on these computers.

One of the important problems to be addressed in this situation is to devise means of actually employing a sufficiently high fraction of the raw computational power of a parallel computer. Overheads due to interprocessor synchronisation and communication, processors sitting idle due to contention for shared hardware resources, and uneven load balancing in the distribution of computational load can lead to poor overall performance. To optimise the speedup of a parallel program on a parallel computer requires the mapping of the parallel tasks of the program among the processors such that the computational load is distributed as evenly as possible and at the same time minimising the amount of communication between the processors.

This thesis investigates mapping the tasks associated with the solution of unstructured grid problems to the processors of a parallel computer such that the execution time is minimised.

1.2 Outline of Thesis

In the remainder of Chapter 1, we define terminology and notation for graph theory that is used throughout the thesis. We then discuss various parallel architectures and various configurations that can be used. The mapping problem is discussed with a short summary of existing methods.

In Chapter 2, we give a brief outline of some of the existing techniques for graph partitioning and embedding. These are methods that we have looked at extensively and discussions of the analysis of each method is given.

Chapter 3 discusses the Recursive Clustering algorithm which is a method based on the Kerninghan-Lin mincut algorithm [KL70]. We have modified the Recursive Clustering algorithm so that our needs are catered for and descriptions of these modifications are discussed in Chapter 4.

This new modified algorithm gives reliable decompositions but one drawback is the time taken to decompose the meshes. We have overcome this problem and discussions of how this is done can be seen in Chapter 5.

Finally, Chapter 6 shows the parallel efficiency of the decompositions used together with conclusions and discussions of further work.

1.3 Graph Theory

The following terminology and notation is used throughout this thesis [Wil85], [BM76]. A graph G is a pair of sets [V,E] where V is non-empty and E is a set of unordered pairs of elements of V. The elements of V are called the *vertices* of G and the elements of E are called the *edges* of E. V_G is used to represent the vertices of G and E_G is used to represent the edges of G. The symbols υ_G and ε_G are used to denote the number of vertices and edges in G. If only one graph is being considered, then the letter G will be omitted from the symbols, and therefore we use V, E, υ and ε instead of V_G, E_G, υ_G and ε_G.

Two graphs G and H are said to be *isomorphic* if there is a one-one correspondence between their vertices which has the property that two vertices are joined by an edge in one graph if and only if the corresponding vertices are joined by an edge in the other.

Two vertices u, v of a graph G are *adjacent* if there is an edge joining them i.e. $<u,v> \in E$.

With each $<u,v> \in E_G$, let there be associated an integer $c(<u,v>)$, called its *edge weight*, and with each $v \in V_G$, let there be associated an integer w(v) called its *vertex weight*. Then G, together with these edges and vertex weights is called a *weighted graph*.

A vertex v and an edge e are *incident* if v is one of the vertices of e.

The degree $\rho_G(v)$ of a vertex v in G is the number of edges incident with v.

Figure 1.1 shows a graph G where $\upsilon = 8$ and $\varepsilon = 14$.

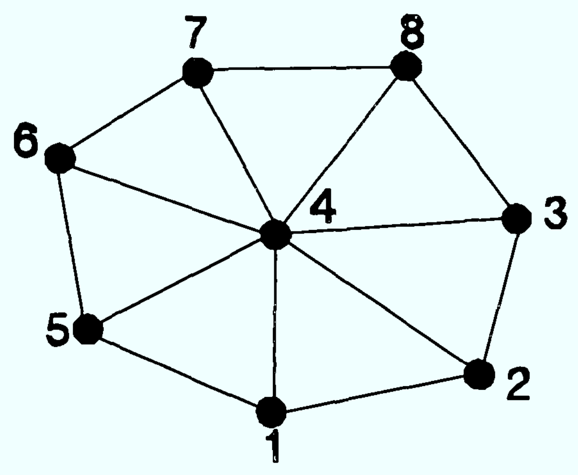


Figure 1.1: A graph G with 8 vertices

To any graph G, there corresponds an *adjacency matrix*. This is the $\upsilon \times \upsilon$ matrix $A(G)=[a_{ij}]$, where a_{ij} is the number of edges joining v_i and v_j. The Laplacian matrix of a graph G is defined as $L(G)=[l_{ij}]$ where $l_{ij}=a_{ij}$ for $i \neq j$ and $l_{ij} = -\rho_G(v_i)$ for each $v_i \in V$.

Figure 1.2 shows the adjacency matrix and the Laplacian matrix for the graph G shown in Figure 1.1.

$$A(G) = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad L(G) = \begin{bmatrix} -3 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & -3 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -3 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & -7 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & -3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & -3 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & -3 \end{bmatrix}$$

Figure 1.2 Adjacency matrix and Laplacian matrix of the graph G shown in Figure 1.1

A *directed graph (digraph)* D=(V,E) is a graph whose edges are ordered pairs of vertices. With each digraph D we can associate a graph G on the same vertex set; corresponding to each directed edge of D, there is an edge of G with the same ends.

A network N is defined to be a weighted digraph with two distinguished subsets of vertices, X and Y, which are assumed to be disjoint and nonempty.
The vertices in X are the *sources* and those in Y are the *sinks* of N. The edge weight C of each edge is a non-negative integer called the *capacity*.

A *cutset* in a network N is a set of edges which when removed disconnects the source nodes from the sink nodes.

The *weight* of a cutset is equal to the sum of the capacities of the edges in the cutset. The Max-Flow Min-Cut theorem [FF62] states that the value of a maximum flow in a network is equal to the weight of a minimum cutset of that network.

1.4 Parallel Architectures

The availability of relatively cheap and efficient microprocessors has produced a tremendous upsurge in the development of parallel computers [Car88], [Cri88], [Duc86]. These computers now consist of numerous (up to thousands) of processors. These processors usually have reduced instruction sets and are frequently referred to as processing elements (PEs). This section gives a brief overview of some of the models that exist.

The SISD (Single Instruction Stream, Single data Stream) is the original von Neumann model of computation where only one instruction is processed at a time on a single item of data. Some parallelism may occur in the internal operations of such machines, for example, parallel loading and storing of data items along with actual arithmetic operations.

The MISD (Multiple Instruction Stream, Single Data Stream) performs several instructions simultaneously on a single stream of data. Strictly speaking, this category could contain the operation of internally parallel SISD architectures and pipeline processors, but since the user's understanding of computer architectures is in our interest, neither is included.

Computer architectures such as the SIMD (Single Instruction Stream, Multiple Data Stream) commonly known as vector or pipeline computer architectures. A SIMD computational model corresponds to a single stream of instructions each of which is applied to multiple data items.

A broad definition of a vector processor is where each processing element allows a sequence of identical operations at the same time but acts upon different sets of data. This type of operations is often featured in operations involving vectors of data.

With pipeline processors, overlapping in the execution of instructions is permitted. The data enters the pipeline at the processing element performing the first stage of the operation, passing through the other processing elements until finally arriving at the last one for the final stage of the operation. Parallelism is achieved when several data items pass through such a pipeline, but with each item passing through different stages at the same time. It is important that every processing element in the pipeline is kept busy in order to achieve a significant speed-up. This is accomplished by passing several data items that need the same overall operation to be performed on them through the same pipeline. This is typical for vector operations where the data passing through the pipeline consists of each consecutive element of the vector(s) concerned.

1.5 MIMD Multiple Instruction Stream, Multiple Data Stream

This type of machine is the one that we are focusing on and it typically consists of a number of processing units each capable of executing its own program on separate sets of data. All the processing units are interconnected and to achieve parallelism, the overall task must be broken down into a group of many sub-tasks .

There are various designs of MIMD machines with a major distinguishing feature being the interconnection network. The two extreme classes of machines are discussed, namely the shared memory systems and distributed memory systems [Cri88].

Shared memory systems use a shared global memory that is accessible from every processing unit via a communication bus. The processors can be considered identical (providing the processors are of the same type) and the programmer need not be concerned with the issue of mapping which task of the computation onto which processor since communication between any pairs of processors is the same. Problems occur with such systems when large number of processors are used since the communication bus hardware becomes a bottleneck when many processors request access to the global memory. Another disadvantage is that the bus only permits one processor to access the global memory at any time. Thus if many pairs of processors require interaction on a pair-wise basis, they will have to do so in sequence rather than in parallel. Figure 1.3 shows an example of a shared memory system.

Distributed memory systems consists of processing elements which have their own local memory unit. The processors are joined by an interconnection network so an overall task can be performed on by many processors and data can be sent from one processor to another. With this type of machine, the processors do not have to fight for access to the shared global memory and the communication bus does not become a bottleneck, but data traffic bottlenecks can occur with a large processor network. Unlike the shared memory system, task to processor allocation is not arbitrary and a task should be placed on a processor that either holds the data to be accessed or can access the data through as short as possible a communications route. The program data should, if possible, be divided over all the local memories with a minimum of duplication to ensure efficiency of such a system. Figure 1.4 shows an example of a distributed memory system.

PVM (Parallel Virtual machine) [SHH94] from ORNL has become a de-facto standard for message-passing systems and because it is freely available, it has spread all over the academic community and beyond. PVM has been ported to a big variety of currently available machines ranging from workstations to MPP-systems. The highlight of PVM is its usability in heterogeneous environments. However, its functionality is limited.
As a consequence, the international initiative MPI (Message Passing Interface) [Hem94] was started in 1992 by the Center for Research in Parallel Computing at Rice University and Oak Ridge National laboratory. The goal is to define a message passing interface which will then be implemented and supported by all hardware manufacturers. It was not the design goal to support low-level features to be used by parallelising compilers. The focus of MPI is the point-to-point communication between pairs of processors, and collective communication within process groups. More advanced concepts allow creating those groups, and giving them topological structure.

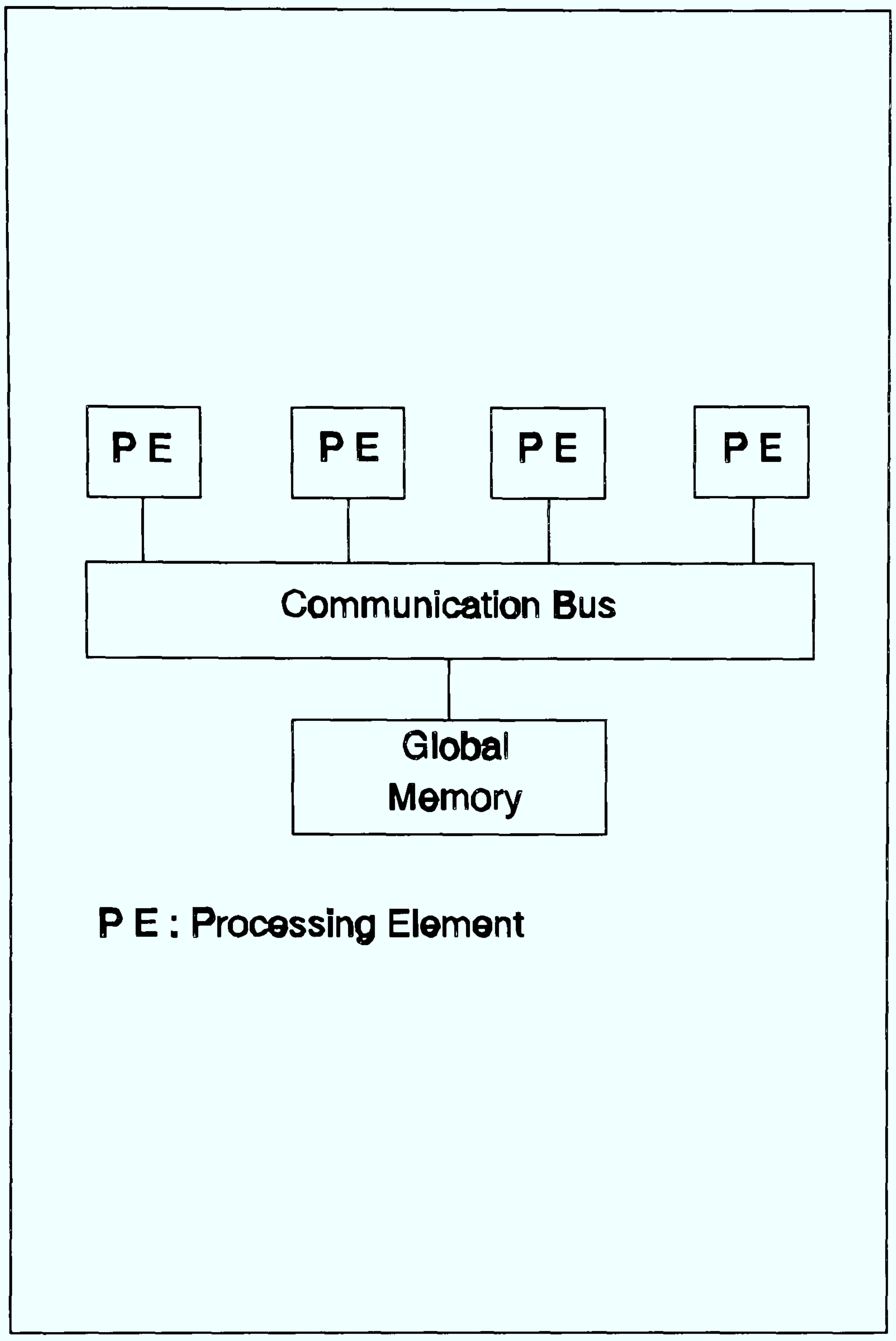


Figure 1.3: Shared Memory System

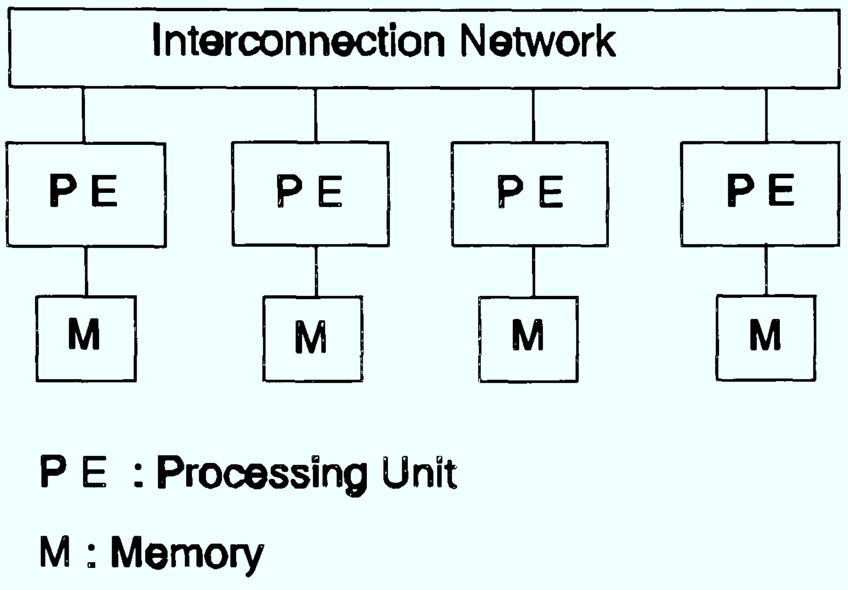


Figure 1.4: Distributed Memory System

1.6 Processor Configurations

The processor network used can be in a number of different configurations [Car88]. The configuration chosen should be influenced by the data access structure of the code concerned. The amount of communication time acquired can be minimised by a sensible choice of network configuration. Examples of network configurations are shown in Figure 1.5.

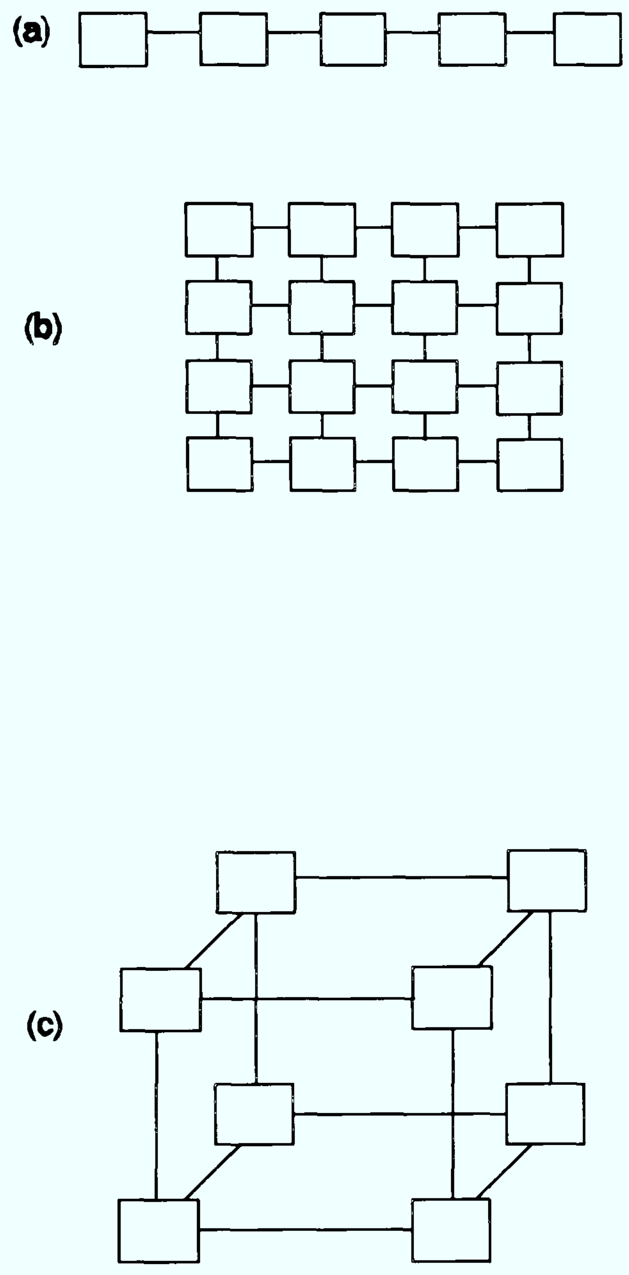


Figure 1.5: Processor Configurations
(a) Chain; (b) Grid; (c) 3 Way Hypercube

1.7 Parallel Performance Measurement

There are two practical ways in which performance of software on parallel systems can be measured. The first is speedup (Sp) which is defined as:-

$$Sp = \frac{Time\ on\ a\ single\ processor}{Time\ on\ p\ processors}$$

Sp gives the number of times faster the software executes on p processors as opposed to the execution on a single processor [HJ81]. However, there are two possible single processor times that can be used, both carrying slightly different information about the software.

Firstly, if the single processor time is that of the best serial version, using optimal serial algorithms, then the speedup signifies the advantage of using a parallel machine rather than a serial machine. If the algorithms used for the parallel version are different to those in serial, then the speedup figure can be reduced because the serial performance may be sacrificed for the parallel nature of the new algorithm. The second single processor time that can be used is that of the parallel version being run on a single processor. This speedup represents the performance of a parallel machine as more processors are used and not performance over serial because any serial version should always use the best serial algorithms available.

Efficiency is the second measure of performance of software on MIMD machines and this is a measure of how well an application uses the available computer power. Again, there are two types that can be used. The first is known as efficiency percentage (Ep) and it is given by :

$$Ep = \frac{Sp}{p} * 100 = \frac{Speedup\ on\ p\ processors}{p} * 100$$

Ep indicates the percentage of available processor time which has been beneficially used, providing that the speedup uses the best serial execution time. Some of the efficiency that is lost here is mainly due to processor idle time, communication time and less efficient parallel algorithms.

The second type of efficiency is calculated using processor time and is given by :-

$$Ep = \left(1 - \frac{\textit{Total Idle Time}}{\textit{Total Processor Time}}\right) * 100$$

Ep gives the measure of the percentage of time spent performing some form of operation.

1.8 Control Volume Unstructured Grid Methods

1.8.1 Introduction

In order to solve continuous partial differential equations (PDE's) using computational methods, the equations under consideration need to be transformed into algebraic difference equations using a discretisation scheme. Many discretisation schemes are available among which the most well known are the Finite-Element (FE), Boundary-Element (BE) and the Control-Volume (CV). Of these methods the CV is probably the most widely used in the context of fluid flow problems, because it is computationally cheap and it preserves continuity of the dependent variables over cells.

In the following sections, control volume based unstructured mesh methods are considered because the aim of this study is to produce meshes on which other researchers at the University of Greenwich can use their CV based methods.
In general the CV-UM method can be categorised into two approaches, one being a vertex centred approach, the other cell centred. The classification of the two methods lying somewhere between the finite element method (in terms of mesh) and the CV

method in terms of numerical integration.

1.8.2 Vertex Centred Approach

This approach is also generally known as the control-volume based finite element mesh (CV-FE), [BP83],[BP88],[Sch88],[LW89]. Domain discretisation involves subdividing the solution domain into a number of smaller regions. The connections between the nodes and the subregions is known as a mesh. The subregions of the domains are called elements and the vertices of these elements nodes. There are many possible element configurations using a FE mesh of which the most commonly used are quadrilaterals and triangles, using four or three nodes respectively. In fact a combination of the two element types or more can be used when discretising a problem domain, giving excellent flexibility in representing complex problem geometries.

A typical finite element mesh is shown in Figure 1.6. As previously mentioned nodes are located at every element corner, where all of the problem unknowns are located. (velocity, temperature, etc).

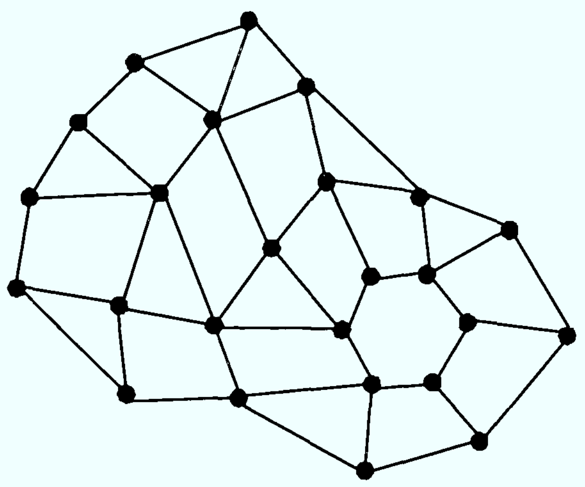


Figure 1.6: Finite element mesh

When working with finite elements, it is convenient to use local co-ordinates in order to homogenise the treatment of individual classes of elements irrespective of how distorted any element is in terms of the global co-ordinates. Conservation in the finite element method is expressed over the global domain and hence these elements need to be related to their global positions. This is done using 'shape functions', which relate local variations to their global equivalent [Zie77].

In the vertex-centred solution mesh, each node is associated with one control volume, whilst the surface of a control volume (CS), is defined from the centroid of an element to the midpoint of one of its sides, as shown in Figure 1.7.

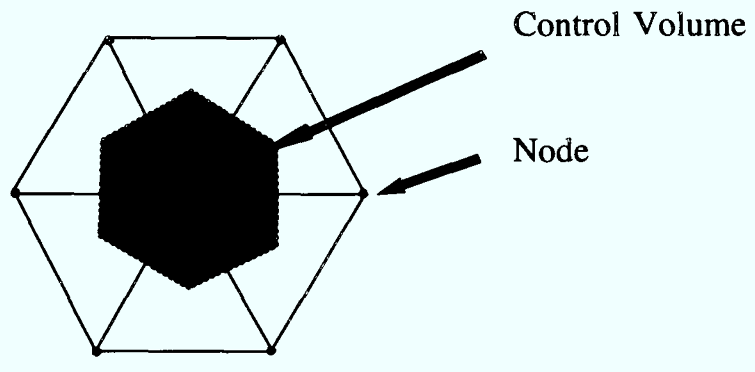


Fig 1.7: Vertex-Centred Mesh-Control Volume

Every element is therefore divided into a number of quadrants by these control surfaces. The quadrants are called sub-control-volumes (SCV's), and a control volume is therefore made up of a number of sub-control volumes of polygonal shape. A CV based discretisation on this mesh involves expressing fluxes across these control surfaces. In algebraic form these fluxes are determined by evaluating integrals at the midpoints of the control surfaces, these are known as integration points and are illustrated in Figure 1.7 above.

1.8.3 Cell-Centred Approach

This method is known as the Irregular control volume method (ICV) and is an extension of the standard control volume method [CC92]. In order to distinguish between the vertex centred approach the previous terminology is replaced by a more control volume orientated terminology. The element is now called a cell or control volume and the vertices of the elements are now called grid points, see Figure 1.8. Nodes are now defined to be at the cell centroids, where all relevant information concerning the dependant variables are stored.

Discretisation of any transient terms in the equation follows the same procedure as that for the vertex centred method. However discretisation of terms involving spacial derivatives differs from that of the previous method in that these fluxes will be evaluated at the midpoints of the control volume surfaces, which are in different positions since the control volume is not subdivided into a number of sub-control volumes.

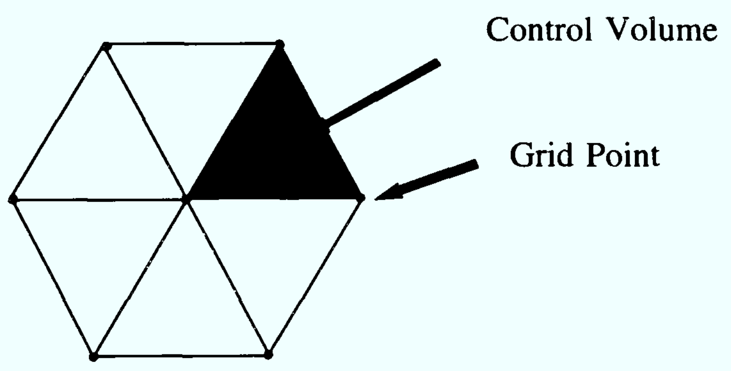


Fig 1.8: Cell-Centred Mesh-Control Volume

Whilst both methods have their relative advantages and disadvantages, they share the finite element quality of excellent geometric representation and the control volume benefits associated with cell-wise conservation of the dependent variables. For a comprehensive description of the discretisation involved using both methods see p32-60 of reference [Cho93] and reference [CC92].

1.9 The Mapping Problem

1.9.1 Statement of Problem

The processor allocation problem is one of utmost importance in the effective utilization of large-scale parallel computers and distributed computer networks. The task allocation problem requires the allocation of multiple interrelated tasks of a single application program in order to minimize the completion time of the parallel program on the parallel computer system. Execution of the parallel program involves a number of iterations, where each iteration involves some computation by each node followed by communication by each node to other nodes.

In many multi-processor systems, there is no direct link between every pair of processors. When assigning nodes of large computational problems onto processors, pairs of nodes that have to communicate with each other should be placed on processors that are directly connected. The mapping problem consists of minimising the number of pairs of communicating nodes that fall on pairs of disconnected processors.

Let the graph of a problem be characterized by a task graph $G = (V_G,E_G)$, whose vertices V_G represent the tasks of a program, and edges E_G characterize the data communication between tasks.

The parallel computer is represented as a graph $P = (V_P,E_P)$ where the vertices represents the processors and the edges represents the communication between the processors.

The mapping problem consists of finding $\Phi : V_G \rightarrow V_p$ such that we minimize

$$\text{time}_{exec} = \text{time}_{comm} + \text{time}_{comp}$$

Undoubtedly, if Φ maps all tasks onto one processor, then the communication cost would be zero but the computational costs would be extremely high.

Assumptions that we make is that all tasks have equal amounts of computations, all data communicated between tasks is equal and the computer is assumed to be homogeneous i.e. tasks execute equally well on all processors.

These are reasonable assumptions since the application area to which this work applies include computational fluid dynamics and structural mechanics. The problems to be solved are nearly always initial or boundary value problems for coupled systems of PDE's. We associate a task with each point in the discretizing grid. The tasks have similar communication and computation requirements and the amount of computation exceeds the communication. Finally, many of the parallel computers being developed today are networks of homogeneous processors.

1.9.2 Objectives

Since parallel architectures provide significant raw processing power it is not surprising that there has been a significant effort by the CFD community to exploit such systems. It is self-evident that in mapping any scalar application onto a local memory parallel architecture it is vital to decompose it in such a way as to:

- Keep all processors busy for as large a proportion of the time as possible (i.e. balance the computational load). It is important that all the processors have approximately the same amount of work to do. We don't want processors having to sit idle waiting for the other processors to finish their jobs.

- Minimise the amount, level and frequency of communication between processors and wherever possible constrain the distance of communication to nearest neighbours,

- Distribute the data evenly over the whole processor array.

1.9.3 Complexity

Bokhari [Bok81] showed that it is unlikely that an exact polynomial time algorithm exists for solving the mapping problem. For a mapping algorithm to be practical, the time taken to decompose the mesh should be a small percentage of the time taken to solve the application in parallel. This will ensure a gain over solving the application in serial.

Initial work on mapping CFD codes onto parallel systems focused upon software which employed structured meshes. Effectively, the approach involved decomposing the mesh and mapping the constituent submeshes onto the processor array in such a way that communication was usually restricted to nearest neighbour. Various approaches for two and three dimensional mapping are described by a number of groups in reference [RSEHP92]. One such approach is described by Johnson and Cross [JC92] who implemented the CFD code, FLOW3D, onto transputer and i860 based systems. Here, the mesh is decomposed into (i,j) slabs such that each processor has an equal number. The processors are then configured as a simple pipeline and the solution proceeds as in scalar, except that periodically information is exchanged between neighbouring processors to send or receive latest values of solution variables on adjacent (i,j) slabs. Because, each processor has (roughly) the same amount of work to do at each stage, the communications can be synchronised at very little cost to the efficiency of the parallel implementation. On transputer based systems efficiencies of 80%+ have been reported on 50 processor systems running problems with 40,000+ nodes [JC92].

The next stage in exploiting parallel architectures for CFD involves codes based upon unstructured meshes. Although, the approach for such codes should be analogous, the key new problem to be addressed involves the strategy for decomposing the mesh. For structured meshes (even block structured meshes) the strategy is fairly obvious. However, for unstructured mesh codes the decomposition is problem dependent and so algorithms are required which will partition the mesh onto a given processor topology to meet the objectives stated in section 1.9.2.

1.10 Summary of Existing Methods

It is known that finding an exact solution to the mapping problem is NP-complete and so any method for finding an exact solution will almost certainly require an exorbitant amount of computation. Heuristics attempt to find a sub-optimal solution in a reasonable amount of time.

Proposed techniques for solving the mapping problem have separated out the solution into two distinct phases. The first phase is to partition the computational domain into cohesive sub-regions. The second phase consists of embedding these sub-regions onto the processors. Partitioning followed by embedding can be viewed as a heuristic or as an initial mapping to be improved by an iterative heuristic.

1.10.1 Graph Partitioning Problem

Let G = (V,E) be an undirected graph with a cost C(u,v) associated with each edge (u,v) $\in$ E. The graph partitioning problem is to partition the vertices of G into two subsets such that the cut set has minimum cost. i.e. the sum of the cost of all those edges with end points in different subsets is minimum. The graph partitioning problem is known to be NP-Complete [GJ79] therefore heuristics have been used to find acceptable solutions.

Kerninghan and Lin [KL70] consider the problem of partitioning the nodes of a graph into subsets of given sizes to minimize the sum of the costs of all edges cut. The work is influenced by two applications. The first problem is placing the components of an electrical circuit onto printed circuit boards so as to minimise connections between the boards. The other application is an attempt to improve the paging properties of programs for use in computers with paged memory organisation. Objects such as subroutines, procedure blocks, data items etc. are assigned to pages of memory and the problem is to minimise the reference to objects that reside on different pages. Experimentally, they determine that the time complexity of this heuristic for finding a 2-way partition of a graph with n nodes is O(n^2). The technique is also extended to perform k-way partitions, using the 2-way procedure as a tool.

Fiduccia and Mattheyses [FM82] improved the Kerninghan-Lin algorithm. They use efficient data structures and vertex displacements instead of exchanges to derive a linear time heuristic for improving 2-way graph partitions.

Gilbert and Zmijewski [GZ87] found low cost partitions for factorisation of sparse matrices by developing a parallel version of the Kerninghan-Lin algorithm. These partitions are used to compute ordering for factoring matrices and the resulting orderings lead to good processor utilization and low communication overhead.

Pothen , Simon and Lieu [PSL90] partition the graphs of sparse matrices using the Recursive Spectral Bisection (RSB). The RSB method for graph partitioning uses the eigenvector x_2 corresponding to the largest eigenvalue λ_2 of the Laplacian matrix of the graph to find vertex separators (see Section 2.3). The special properties of the eigenvector x_2 have been investigated by Feilder [Fei73], [Fei74]. Their results show that the spectral partitions obtained compare favourably with partitions obtained by previous algorithms.

Simon [Sim91] investigates three algorithms for the partitioning problem for unstructured domains. All three algorithms considered are recursive and are Recursive Coordinate Bisection (RCB), Recursive Graph Bisection (RGB) and Recursive Spectral Bisection (RSB). The main result is that RSB is a significant improvement over the other two algorithms.

Simulated Annealing is a combinatorial optimization technique to minimise/maximise an objective function (see section 2.4.3). Johnson et al [JH89] made a critical evaluation for the performance of simulated annealing to the graph bisection problem and compared its performance with that of the Kerninghan-Lin approach. In general, simulated annealing is time consuming, but it has been successfully applied to many combinatorial optimization problems.

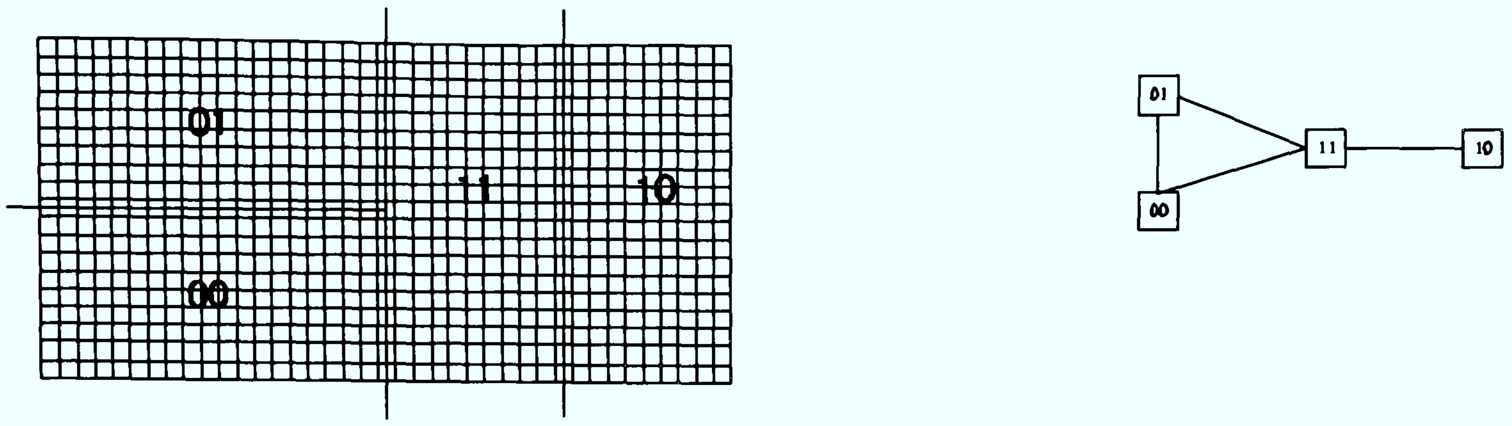
Tabu Search is a fairly new approach to combinatorial optimization (see Section 2.4.5). Tabu search algorithms are generally slower than other problem-specific heuristics but they have been successfully applied to many problem domains. Tao et al [TZTS92] propose two new algorithms based on Simulated Annealing and Tabu Search and compare the effectiveness of these two algorithms with that of an algorithm based on the Kerninghan-Lin method. They show that their two algorithms provide better solutions than the Kerninghan-Lin based algorithm but with longer running time.

Lim and Chee [LC91] consider a graph partitioning approach based on the tabu search heuristic. Their experimental results show that their algorithm consistently out-performed the Fiduccia-Matheyeses [FM82] version of the Kerninghan-Lin method. The speed of their algorithm is also comparable. They also compared their tabu search algorithm with the simulated annealing algorithm and they state that their algorithm does not perform well in terms of quality of solution on random graphs.

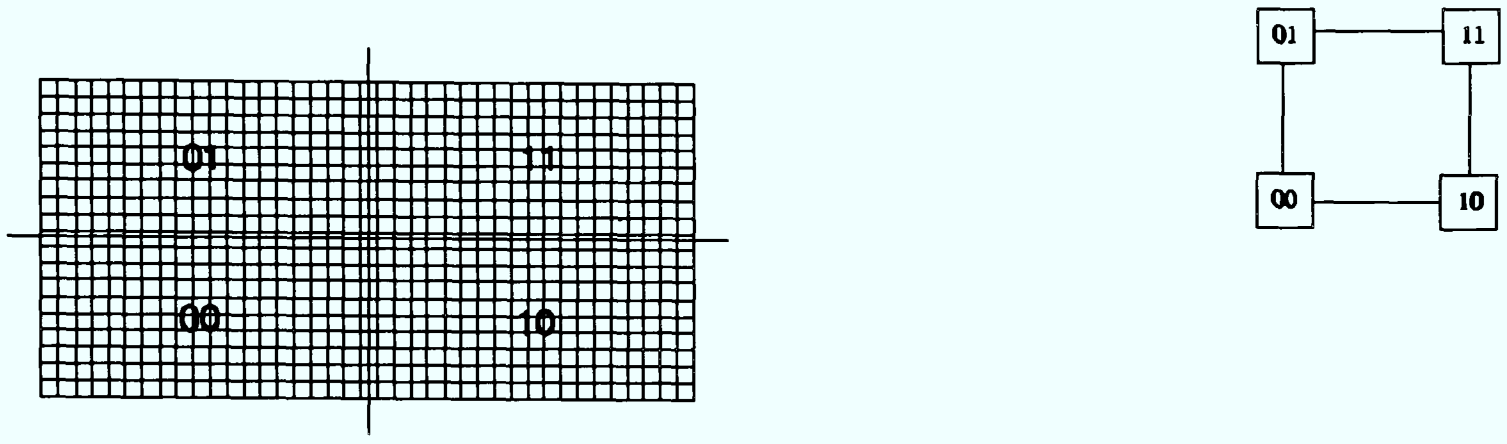
1.10.2 Graph Embedding Problem

Let the graph of the problem to be mapped be denoted by $G = (V_G,E_G)$ where V_G represents the tasks of the problem and edges E_G characterize the data communication between tasks. Let the processor graph be denoted by $P = (V_P,E_P)$ where V_P represents the processors and E_P the communication between processors. The graph embedding problem is to map G onto P such that the maximum distance that data travels should be minimsed.

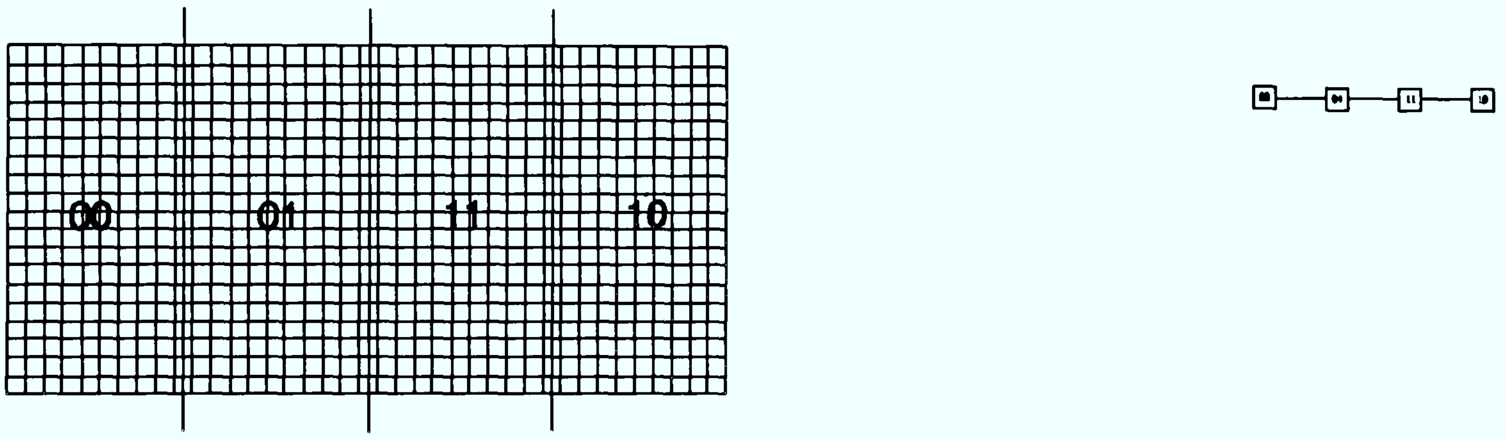
One can first partition the task graph which can then be embedded onto the processor graph. Ercal et al [ERS88] argue that this may not be a good approach to solving the mapping problem. They claim that performing the two operations in isolation can lead to poor mappings and much less than optimal communication time. Ercal et al uses the following example to illustrate the disadvantage of performing the partitioning and processor assignment independently in two phases. Consider a simple regular task graph with 800 nodes, interconnected in a 20 x 40 rectangular mesh. The first partition to minimise the cut separates it into two 20 x 20 meshes. After the first partition is made, there are 4 possible choices for the second level partitions. Three of these choices can be seen in Figures 1.9(a-c). All four partitions are optimal - the load is balanced and they share the same number of nodes. However, they are not equal from the embedding perspective. With the partitions obtained In Figure 1.9(a), it is impossible to assign the clusters to processors of a 2-dimensional hypercube so that all communication is between directly connected neighbouring processors. Therefore, the minimum total communication cost that can be achieved is 70. However, the two second level bisections shown in Figures (b) and (c) performed identical cuts and the total communication cost that results is only 60.



(a) Communication cost = 70



(b) Communication cost = 60



(c) Communication cost = 60

Figure 1.9: Possible 4-way partitions of a 40x20 grid with processor topology.

1.10.3. Prior work on the Mapping Problem

This section gives a brief overview of some of the more important work on the mapping problem presented in chronological order.

Stone [Sto77] develops a heuristic for the mapping problem using the Ford-Fulkerson algorithm [FF62] as modified by Edmonds and Karp [EK72]. He uses a network flow algorithm as a "black box" utility to map a task graph onto a two-processor system. A network representation of the mapping problem is produced and fed to a network flow algorithm.

The construction of a network representation N of the two-processor mapping problem is as follows:

1. N = G.

2. Add nodes labelled s_1 and s_2 to V_G representing the two processors. s_1 is the unique source and s_2 is the unique sink [FF62].

3. For each $v \in V_G$, add an edge $\langle v,s_1\rangle$ and $\langle v,s_2\rangle$ to E_N.

4. Let $C(\langle v,s_1\rangle)$ be the estimated time to execute task v on processor s_2 and $C(\langle v,s_2\rangle$ be the estimated time to execute task v on processor s_1.

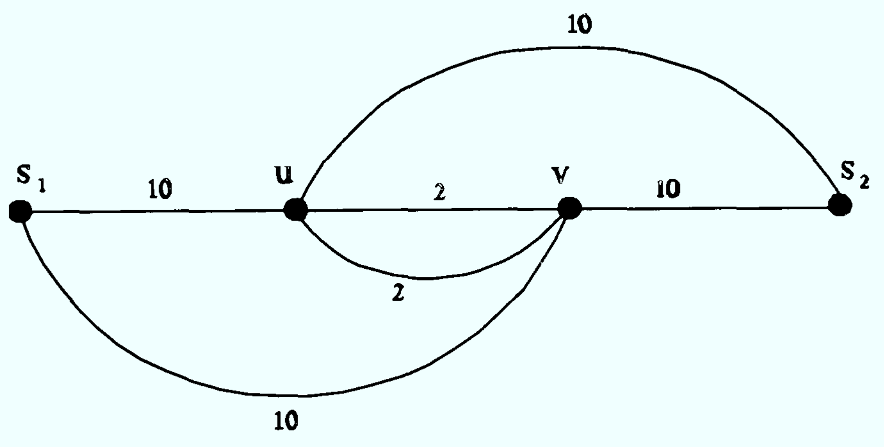


Figure 1.10: A network flow graph constructed from a task graph with two vertices.

The edge weights are chosen so that the weight of a cutset of N is equal to the execution time of the corresponding task-to-processor mapping. An optimal mapping of tasks to two processors is found by finding a minimum weight cutset, and assigning tasks to the processor on the same side of the cut.

For example, in Figure 1.10, a task graph $G=(V_G,E_G)$ is shown where $V_G=\{u,v\}$ and $E_G=\{\langle u,v\rangle,\langle v,u\rangle\}$. The edge weights are $C(\langle u,v\rangle)=2$ and $C(\langle v,u\rangle)=2$. A network is constructed by adding the vertices s_1 and s_2 and the edges $\langle u,s_1\rangle$, $\langle u,s_2\rangle$, $\langle v,s_1\rangle$ and $\langle v,s\rangle$ with weights $C(\langle u,s_1\rangle)=10$, $C(\langle u,s_2\rangle)=10$, $C(\langle v,s_1\rangle)=10$ and $C(\langle v,s_2\rangle)=10$.

Therefore, the tasks execute equally well on either processor and five times as much time will be spent computing as communicating.

However, good solutions to the mapping problem are not always provided by using network flow algorithms. In the above example, the maximum flow algorithm assigns both u and v to the same processor since there are two minimum weight cutsets with weight 20, $\{\langle u,s_1\rangle,\langle v,s_1\rangle\}$ and $\{\{\langle u,s_2\rangle,\langle v,s_2\rangle)$. Putting both tasks onto one processor results in a running time of 20; only one task computes at a time. The communication time is zero since both tasks are on the same processor. However, if the computations can be done in parallel and the communications are completed serially, then the running time when the tasks are mapped to different processors is 14. Constructing a network in this manner does not account for the concurrency in the two tasks. The result is a mapping that requires more execution time than if the tasks were mapped onto different processors. Also, using a network flow based algorithm heuristic to solve the mapping problem is computationally expensive. Efficient Max-Flow Min-Cut algorithms are of complexity $O(e_N v_N \log v_N)$. [GMG82]

Stone [Sto77] generalizes this approach to υ_P processor networks. Although he does not give a complete efficient algorithm. He shows that a single source network flow algorithm can give information about the minimal weight cutset in a υ_P processor graph. Let $S=\{s_1,\ldots,s_\upsilon\}$ be the distinguished nodes representing υ_P processors. For $i=1,\ldots,\upsilon_P$, run a single source network flow algorithm using s_i as the source node and $S\backslash s_i$ sinks. Stone proves that if some v is associated with s_i by the two-processor flow algorithm then v is associated with s_i in a minimum cost cutset in a υ_P processor network. Unfortunately, one can construct examples in which some v is mapped onto a processor in the υ_P processor cutset, but fails to be associated with that processor by the two-processor cutset. Therefore, even after v_P applications of the two-processor network flow algorithm, some subset of V_N may not be mapped to a processor.

Bokhari [BOK81] develops a heuristic algorithm which consists of pairwise interchanges alternating with random interchanges. The quality of a mapping is determined by the number of graph edges E_G that fall on processor edged E_P. This number is called the cardinality of the mapping.

First, an initial assignment of tasks to processors is made.

Next, loop for each node:

1. Examine the pairwise exchange of this node with all other nodes.
2. Select the one which leads to the largest gain in the cardinality of the mapping.

In this loop, only one pair of nodes can be exchanged at each iteration. If at least one exchange is made through the loop over all nodes, then the loop is repeated. If no exchange is made, the current mapping is saved and a random jump to a nearby mapping is made by interchanging *n* randomly selected pairs of nodes. If the new mapping is poorer than the saved mapping, then the saved mapping is kept and the heuristic stops. If the new mapping gives better results than the saved mapping, the new mapping replaces the saved mapping and the loop is repeated until no further improvement is made. The complexity of the outer iteration for this heuristic is $O(n^2)$ and may not be suitable for large problems.

Lee and Aggarwal [LA87] develop a deterministic-iterative heuristic mapping strategy for parallel processors using a more accurate quantification of the communication overhead. Their mapping scheme has been tested using the hypercube as the processor graph. They introduce three objective function to evaluate the quality of a mapping. The first objective function is the sum of the communication overheads of all problem edges. However, this in only appropriate if no two communications occur at the same time. The second objective function is the maximum number of communication overheads which is appropriate if all communication occur simultaneously. The third objective function is the sum of the maximum number of communication overheads at each stage which is appropriate if the communications occur at different stages. To evaluate the objective function, one must assign task graph edges to processors graph edges every time the mapping is changed. An initial assignment of tasks to processors is made using a one-pass approach which attempts to match the communication requirements of tasks to the communication capacities of the processors. The execution of this initial procedure takes O(v_G^2) time [LA87]. They then try to improve the mapping by performing serial pairwise exchanges and checking whether it gives a better mapping, or not, using the appropriate objective function. The pairwise exchange used is similar to the one used by Bokhari[Bok81]. The objective function is evaluated for every exchange and the pairwise exchange that results in the largest decrease in the objective function is made.

Berger and Bokhari [BB87] consider mapping refined grids onto different multiprocessors, namely a mesh-connected array, a binary tree machine and a hybercube. The task graphs that they consider are initially regular grids and are refined by superimposing fine grid patches over an underlying global coarse grid. The objective function used here is to maximise the cardinality of the mapping and a one-pass algorithm to map tasks to processors. The task graphs are partitioned into load balanced, disjoint subgraphs. This is done by placing a horizontal or vertical line such that half the vertices lie on either side of it. Each half is then bisected in the same manner by a line orthogonal to the previous partitioning line. This is done recursively until the number of partitions equals the number of processors. The partitions are then embedded in the processor graph. They achieve lower cost mappings on the hypercubes and meshes than the binary tree interconnections. However, the results for the hypercube are only marginally better than the results for the mesh.

Ercal et al [ERS88] use a recursive task-allocation scheme based on the Kerninghan-Lin mincut bisection heuristic. They compare simulated annealing to their recursive bisection method and their objective function is to minimise the number of adjacent vertices in the task graph that are mapped onto processor graph vertices. They show that, on average, their recursive bisection scheme reduces the objective function almost as well as the simulated annealing approach, but requires approximately two orders of magnitude less time to achieve the results.

Williams [Wil91] compares three parallel algorithms for mapping unstructured meshes among the processors of a distributed-memory machine. The three algorithms are orthogonal recursive bisection (ORB), recursive spectral bisection (RSB) and a simple parallelization of simulated annealing which is a well known heuristic. These have been implemented for load balancing a dynamic unstructured triangular mesh on 16 processors of an NCUBE machine. He shows that execution time of the RSB is larger than the ORB and the parallel implementation of simulated annealing takes 20 times longer to run than RSB for his test cases. After mapping the three methods, the running time of each application is measured and using a mesh with 5772 nodes, the execution time using the simulated annealing is the fastest and the ORB mapping is the slowest. Even though simulated annealing ran significantly longer that ORB, the running time for the simulated-annealing partitioned (best mapping) was only 21% less than the running time for the ORB partitioned application (worst mapping).

Sadayappan et al [SER90] compare two heuristic approaches for mapping grids onto hypercubes, a nearest-neighbour approach and a recursive clustering scheme. The recursive clustering method is based on the Kerninghan-Lin algorithm. They show that the nearest-neighbour approach is found to be more effective when using regular grids and that the recursive clustering approach is more effective than the nearest-neighbour at reducing communications for systems with high message start-up costs.

Search techniques are exact algorithms that have been used to solve the mapping problem.
Shen [ST85] considers an optimal task assignment in which communicating tasks are required to reside in the same or neighbouring processors. An ordered search is used to search the space of feasible assignments.

Sinclair [Sin87] uses a state space reduction technique (branch-and-bound-with underestimates) to find optimal assignments. The disadvantage with search techniques is if we have V tasks and P processors, then there are V^P possible assignments of tasks to processors. For very large problems, searching is not very feasible since in the worst case it needs a completer enumeration of all possibilities and this can prove to be very expensive.

Lo [Lo88] extends Stone's network approach. Recall that in a homogeneous system, an optimal network solution will map all tasks to one processor. here, a penalty function is incurred to distribute the tasks to multiple processors. This approach is not very practical for very large problem since repeated use of the Max-Flow Min-Cut algorithm is too expensive.

Chapter 2
Overview of Existing Techniques

2.1 Introduction

In Chapter 1, a brief outline of existing techniques to graph partitioning and embedding was given. In this chapter, some of the methods that we looked at extensively are described in more detail. At the end of each section, we discuss the analysis of the method described.

2.2 Nearest Neighbour

2.2.1 Introduction

The nearest neighbour algorithm was proposed in [SE87] as an approach to mapping finite element meshes onto processors. A nearest neighbour mapping is one where elements which share a node are mapped onto the same or neighbouring processors. The idea behind the nearest neighbour mapping is that if neighbouring elements are mapped onto neighbouring processors then the total communication costs should be low. Furthermore, by aiming at a load-balanced distribution, computation costs can also be minimized. Starting with an initial nearest neighbour mapping, successive incremental modification of the mapping is done to improve the load balancing whilst still maintaining the nearest neighbour property.

The nearest neighbour mapping uses a two-step procedure :

1) Creation of an initial nearest-neighbour partition by grouping nodes into clusters and the clusters allocated to processors so that any two nodes which share an edge are mapped onto the same or neighbouring processors.

2) Iterative boundary refinement involving modification of the initial mapping where nodes are reassigned among the processors to improve the load balancing but maintaining the nearest neighbour property.

2.2.2 Regular Grids

We first describe a scheme for creating nearest neighbour mappings for regular graphs. These are graphs that are made up of four-node finite elements that can be embedded onto a uniform two-dimensional grid in such a way that any two vertices of the finite element graph which share an edge are mapped onto adjacent (vertically or horizontally) mesh points. There are two types of partitioning methods for the regular graphs. The first is a one dimensional strip partitioning which provides perfectly load-balanced nearest neighbour partitions. The second is a two dimensional strip partitioning which satisfies the nearest neighbour property but does not always satisfy the load-balancing property. It is with the two dimensional scheme where the boundary refinement procedure is applied. One example of nearest neighbour partitioning is strip partitioning.

2.2.2.1 One Dimensional Strip Partitioning

The essential idea behind this method is to partition the graph into "strips", where each column (or row) encompasses one or more contiguous columns (or rows) of the graph. The graph is covered by strips in such a way so that each strip is adjacent to at most two other strips, one on either side of it. The number of strips equals the number of processors and each strip can be made so that each strip contains the same number of vertices.

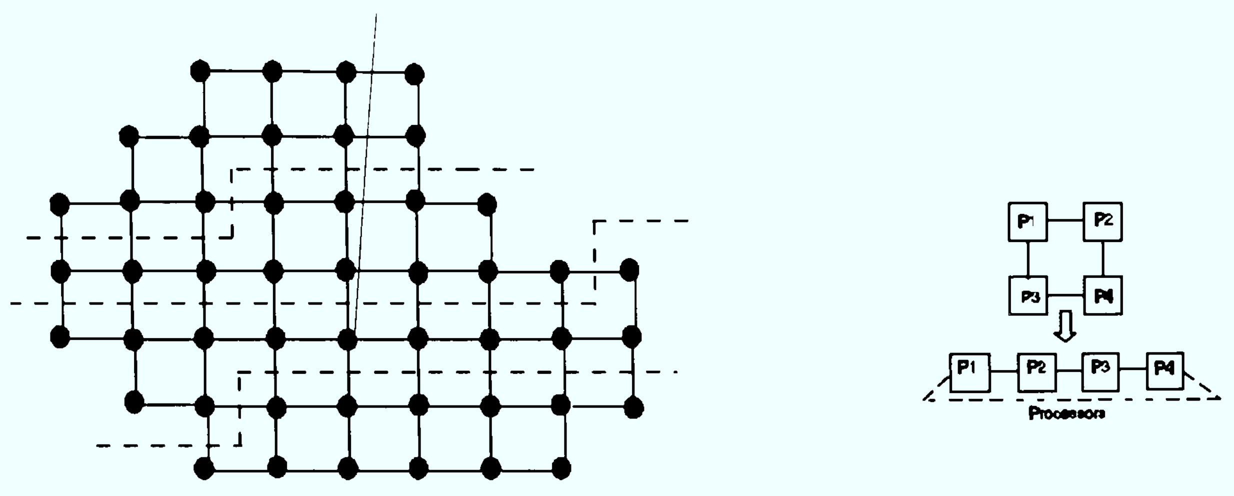


Fig 2.1: Example of one-dimensional strip partitioning

It is easy to assign the strip partitions of the finite element mesh to the processors. Starting with the leftmost node in the uppermost row of the regular graph, the nodes are assigned to a processor by proceeding across the row of the regular graph. When the first row is completely exhausted we then move on to the left end of the next row and continue assigning nodes to the processors. When the processor has been assigned the required number of nodes, we begin assigning the remaining nodes to the next processor in the linear chain of processors. Proceeding in this manner, all nodes are assigned to the processors.

An example of the one dimensional strip partitioning is given in Figure 2.1. Here, we have a graph with 48 nodes which has been partitioned into 4 strips, each strip containing 12 nodes and is mapped onto a 2 x 2 processor mesh by treating it as a 1 x 4 linear chain. A similar procedure can be used to create a vertical one dimensional strip partitioning of the same graph.

2.2.2.2 Two Dimensional Strip Partitioning

When the regular graph is not one dimensional strip partitionable into as many strips as the number of processors (i.e. when we have a matrix of processors rather than a chain of processors), it is often possible to partition the graph into the number of processors on one side of the processor mesh. The idea behind the two dimensional strip partitioning is to create two independent one-dimensional strip partitios, one in the horizontal direction and the other in the vertical direction. This is illustrated in Figure 2.2 with a mesh graph with 40 nodes to be mapped onto a 4 x 2 processor mesh. Figure 2.2(a) shows the graph partitioned into 4 strips in the horizontal direction and each strip contains 10 nodes.

A similar procedure is used to partition the graph in the vertical direction and it is partitioned into two vertical strips, each strip containing 20 nodes. By overlapping the two orthogonal strip-partitions generated, and forming the intersections, we can generate a number of regions which equals the number of processors in the processor mesh. The nature of the construction guarantees that the partitions generated satisfy the nearest-neighbour property. Even though the two independent partitions are each individually balanced, the intersection partitions are not generally load balanced as can be seen in Figure 2.2(c). However, this partition is used as an initial partition and is then refined by the load-balancing boundary refinement procedure.

One way of balancing the computational loads between the processors is to reassign some of the nodes among the processors and this is done by transferring nodes from overloaded processors to underloaded processors. For example by transferring one task from P_{22} to P_{12} and one task to P_{21}, and also transfer one task from P_{31} to P_{32} as shown graphically in Figure 2.2(d) as a Load Transfer Graph (LTG).

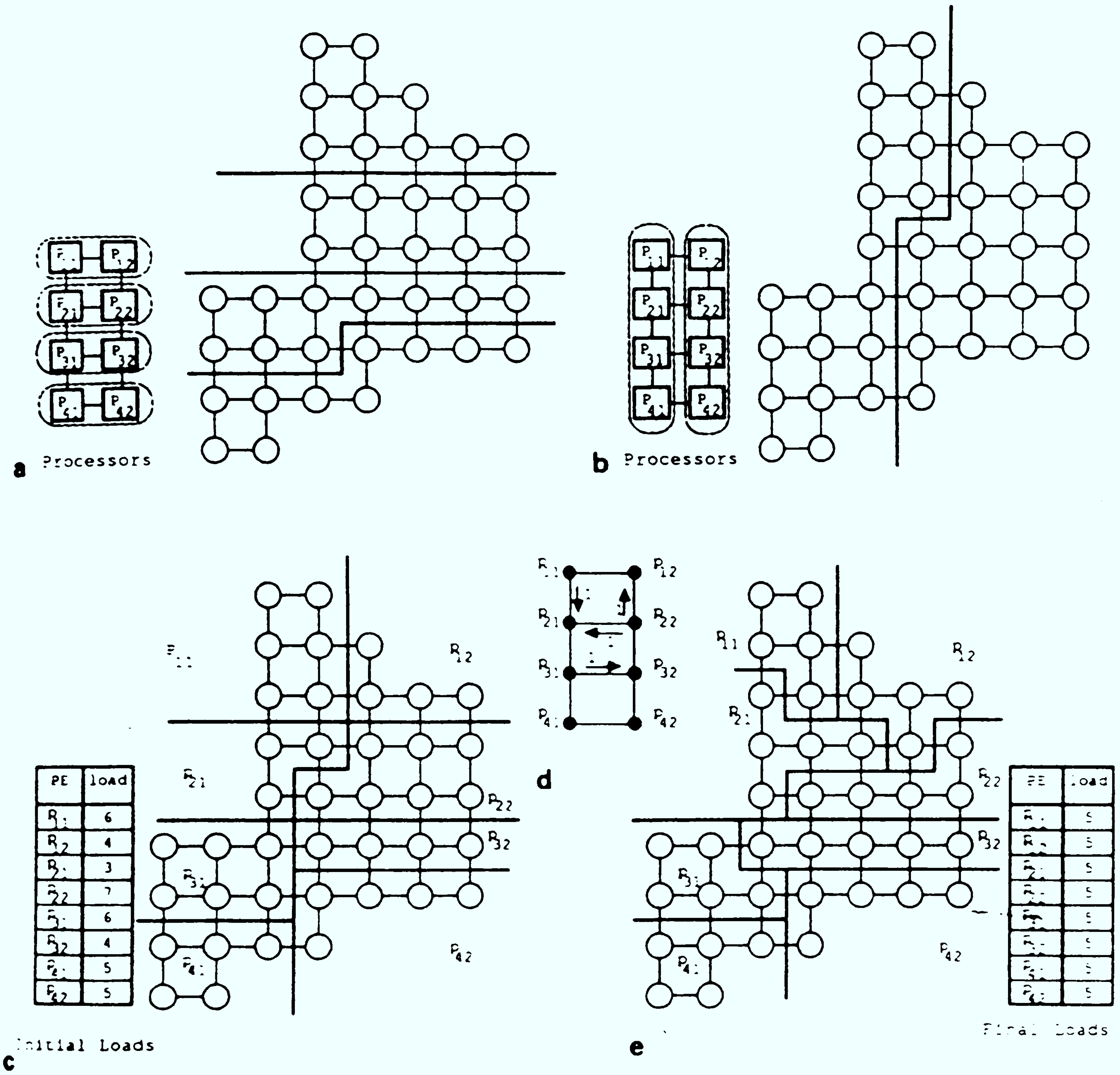


Fig 2.2: Two dimensional strip partitioning

(a) Horizontal strip partitioning; (b) Vertical strip partitioning

(c) Initial two dimensional mapping (d) Load transfer graph

(e) Final mapping after boundary refinement

A heuristic boundary-refinement procedure which is described in detail in [SE87] iteratively attempts to transfer tasks between the processors using the load transfer graph. An "active" processor list is formed and is sorted in decreasing order of current task-load, containing all processors that source an edge in the LTG. Looking at the processor which has the greatest load, a task is found (preferably on the partition boundary) which can be transferred to a neighbouring processor in the LTG. At this stage we must make sure that the nearest neighbour constraint is not violated. If a task cannot be found, then we scan the sorted active processor list in decreasing order of processor loads until a processor which has a transferable task is found. The task is transferred and the LTG is updated by decreasing the relevant edge-weight by one and removing it if its edge-weight becomes zero.

The algorithm proceeds iteratively in an incremental fashion to refine the mapping by rearranging the boundaries of the partitions to improve the load balancing. With the example used in Figure 2.2, the final mapping after using the boundary-refinement procedure can be seen in Figure 2.2(e).

2.2.3 Non Regular Grids

This section describes methods for generating nearest neighbour partitions for general finite element graphs. A generalization of the one dimensional strip partitioning used for regular graphs is possible for non-regular graphs. The basic idea is to cover the graph with strip-like regions, so that if a node lies on a certain strip then all other nodes which share an edge with this node lie on the same or adjacent strip. Due to the regularity of the regular graphs, the process of generating strips was simplified and the nodes could easily be grouped into columns and rows. Unfortunately, this is not possible with the non-regular graphs. The nodes have to be grouped by using a levelization process and each node is assigned a level, this is illustrated in Figure 2.3. Starting with a randomly selected node or set of nodes (preferably on the boundary) these are assigned a level 1. All nodes which are connected to these level 1 nodes and which have not previously been assigned a level are assigned a level 2. The same procedure is carried out with these level 2 nodes until all the nodes in the graph have been assigned a level.

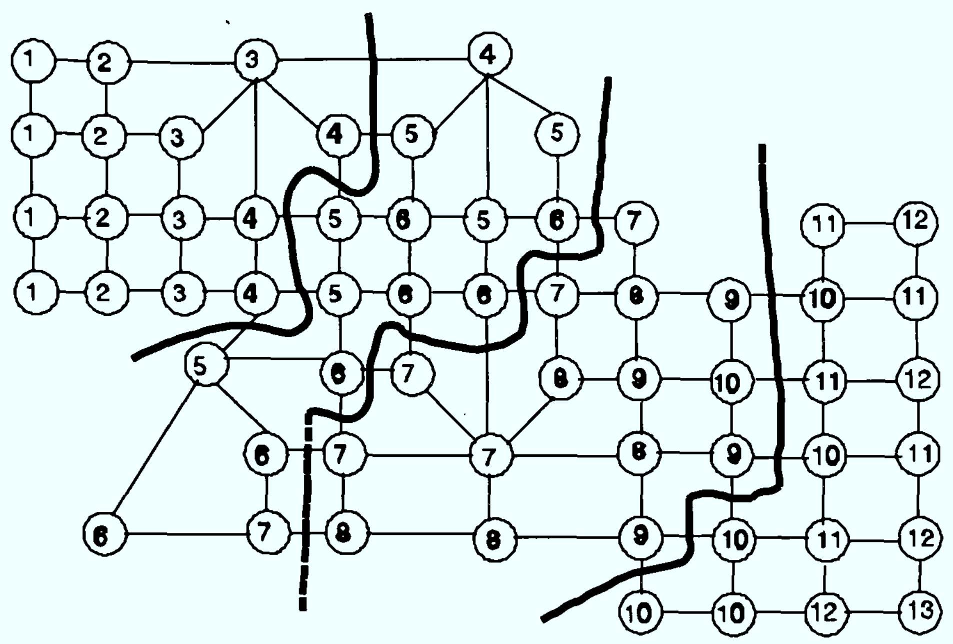


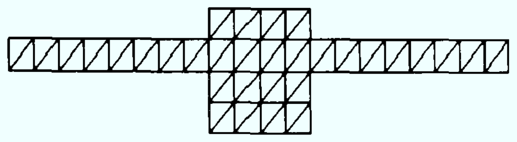
Figure 2.3: Example of a one-dimensional non-regular graph

The nature of this levelization procedure ensures nearest neighbour communication since if any node is assigned a level *i*, then any neighbouring node will be assigned a level *i*, *i* - 1 or *i* + 1. The strip partitioning can now be achieved using the levels similar to using the rows or columns used for mesh graphs as described earlier. Figure 2.3 shows a non-regular graph containing 60 nodes and this is partitioned into 4 strips, each containing 15 nodes. We start with the level 1 nodes and allocate these to the first strip and when all level 1 nodes have been used we move on to level 2 nodes and so on until we have allocated 15 nodes to the first strip.

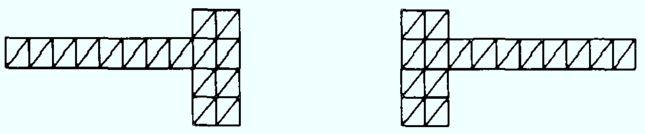
In the case of regular graphs, mapping onto an *m* X *n* processor mesh was achieved by performong two independent partitions, one in the vertical direction and the other in the horizontal and overlapping these two partitions.

2.2.4 Analysis of Method

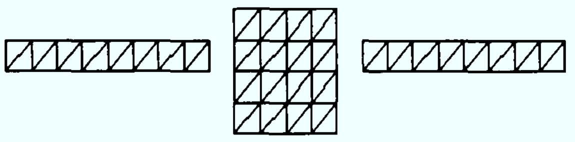
We investigated the nearest neighbour method and wrote code to perfrom the partitioning strategies.Even though the method can be quite effective, unfortunately, the strip partitioning procedure does not always work well, particularly on meshes that are very unstructured. As such, its partitions can be arbitrarily poor. The key reason for this poor performance, is that essentially the nearest neighbour strategy is a geometric approach, whereas in reality the task of mesh decomposition is a topological one.



a) Original mesh



b) Split into two using nearest neighbour



c) Ideal decomposition with minimal communicaton cost

Figure 2.4: A simple mesh illustrating the limitation of the nearest neighbour technique

This is demonstrated in the example shown in Figure 2.4. Figure 2.4(b) shows the simple split into two equal sub-meshes produced by the nearest neighbour approach of the mesh shown in Figure 2.4(a). Unfortunately, this split, though viable, does not minimise the communication cost between the sub-meshes. The split that achieves this is shown in

Figure 2.4(c). Although, the two arms are located on the same processor and have no connection, the total amount of communication is less than the nearest neighbour split. However, the split shown in Figure 2.4(c) is also nearest neighbour because we can see that all neighbouring elements are placed on the same or neighbouring processors. Even though the split contains a disconnected sub-mesh, the two arms do not have to communicate with each other and only have to communicate with the other sub-mesh which is placed on a neighbouring processor. Although the mesh in Figure 2.4 is trivial, it demonstrates the one type of limitation experienced by all approaches which are essentially based upon geometrical rather than topological considerations.

2.3 Recursive Spectral Bisection

2.3.1. Introduction

The recursive spectral bisection algorithm is derived from a graph bisection strategy developed by Pothen, Simon and Lieu (PSL90) and has been developed and explored separately by both Williams (Wil91) and Simon et al (S91), (VSB91). The approach is based on the computation of a specific eigenvector of the Laplacian matrix of the connected graph G.

2.3.2 The Laplacian Matrix

Recall that the Laplacian matrix $L(G) = (l_{ij}),i,j = 1....n$ is defined by

$$l_{i,j} = \begin{cases} +1 & \text{if } (v_i,v_j) \in E \\ -\deg(v_i) & \text{if } i=j \\ 0 & \text{otherwise} \end{cases}$$

The Laplacian matrix $L(G) = -D + A$ where A is the adjacency matrix of the graph and

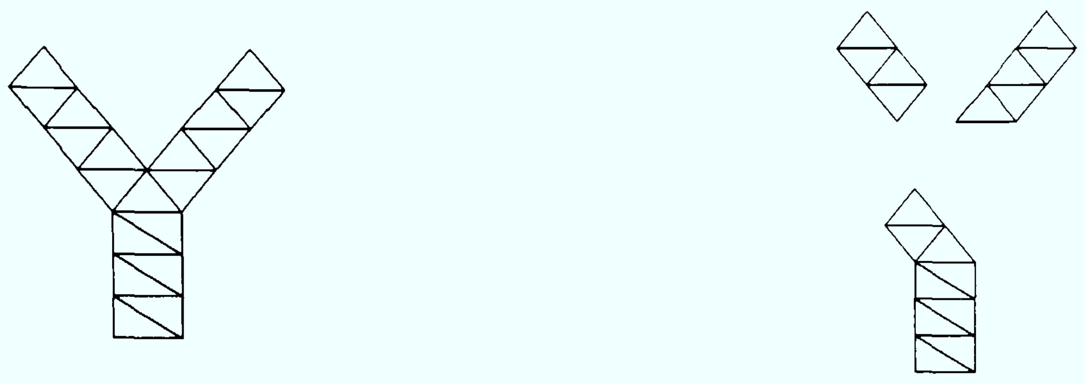
D is the diagonal matrix of the vertex degrees. This matrix has a number of important algebraic properties [Moh88] which reflect some of the basic structure of the graph. For example, all the cofactors are equal and have a value whose modulus equals the number of spanning trees. Since the matrix is obviously singular, then zero is an eigenvalue. In fact, the largest eigenvalue λ_1 is zero and that the associated eigenvector x_1, is the vector of all ones. If G is connected then λ_2, the second largest eigenvalue is negative. The magnitude of λ_2 is a measure of connectivity of the graph.

2.3.3 The Fiedler Vector

The eigenvector x_2 associated with λ_2 gives some directional information on the graph. If the components of x_2 are associated with the corresponding vertices of the graph, they yield a weighting for the vertices. Differences in this weight gives a distance information about the vertices of the graph. The graph can then be partitioned by sorting the vertices according to their weight. This eigenvector is called the Fiedler vector since the special properties of x_2 have been investigated by Fiedler (Fei73), (Fei75).

The RSB algorithm works as follows:

1. Compute the second largest eigenvalue λ_2 and corresponding eigenvector x_2 (the Fiedler vector).
2. Sort the vertices of the graph so that they correspond monotonically to their entries in the Fiedler vector.
3. Half of the vertices are assigned to each subdomain.
4. Repeat recursively.



a) Original mesh **b) Partition into two using RSB**

Figure 2.5: A simple mesh illustrating the RSB producing disconnected sub-domains

```
To find a $2^n$ way parition of a graph G
$P^0_0 = V_G$
do i = 0,..., n - 1
    do j = 0, 1,..., $2^{i+1}$ -1
        compute $x_2$ of $P^i_j$
        sort out the components of $x_2$
        assign half of the vertices and edges corresponding to the smallest
        component in $x_2$ to $P^{I+1}_{2j}$ and those corresponding to the other half
        to $P^{i+1}_{2j+1}$
    enddo
enddo
```

P^n_i, i =0, 1, 2,..., 2^n - 1 are the subgraphs of G.

Figure 2.6: Recursive Spectral Bisection Algorithm.

2.3.4 Analysis of Method

Venkatakrishnan et al [VSB91] show impressive mesh decomposition produced by the RSB algorithm, notably for external flow regions.

The RSB algorithm assumes that the decomposition will produce connected sub-meshes but this is not always guaranteed. For example, using the mesh given in Figure 3.5(a) and splitting into two we can see by looking at Figure 3.5(b) that one of the sub-meshes is disconnected. The theory applies to connected sub-meshes but disconnected sub-meshes can be re-connected but it is unclear what effect this has since in a sense, it is against the 'spirit' of the method.

Moreover, when the domains are connected then the resultant partition is obviously nearest neighbour and has all the advantages and suffers the same disadvantages as the general nearest neighbour strategy. So, for example, if the RSB is applied to the graph in Figure 2.5(a) then it cuts the graph into two blocks down the central block as in Figure 2.5(b).

2.3.5 Multilevel Recursive Spectral Bisection

Barnard and Simon [BS93] describe a multilevel implementation of RSB that achieves about an order-of-magnitude improvement in run time.

The multilevel RSB method requires three components to be added to the basic single-level RSB algorithm. The first component is contraction where a series of smaller graphs are created such that the global structure of the original large graph is retained.

The second is interpolation where given a Fiedler vector of a contracted graph, this vector is interpolated to the next larger graph in a way that provides a good approximation to next Fiedler vector. The last component is refinement. Given an approximate Fiedler vector for a graph, a more accurate vector is computed efficiently.

The performance advantage of the multilevel algorithm over the single level algorithm is dependent on the problem size. The speedup decreases as the number of domains increases because the multilevel algorithm must spend more time partitioning smaller subproblems.

2.4 Combinatorial Optimisation Methods

2.4.1 Introduction

Solving a combinatorial optimization problem [Law76] amounts to finding the best solution among a finite collection of alternative solutions. At first it may seem that this is a simple problem; the solution can be found by just examining each of the alternatives in turn and selecting the best. The problem is that the number of alternatives can be numerous.

A classic example of a combinatorial optimisation problem is the travelling salesman problem [GJ79]. Given a list of *N* cities and a means of calculating the cost of travelling between any two cities, one must plan a route, which will pass through each city only once and return finally to the starting point, minimising the total cost.

All exact methods for determining an optimal route require a computing effort that increases exponentially with *N*, so that in practice exact solutions can be attempted only on problems involving a few hundred cities or less. An algorithm which depends upon a power of *N* is called a polynomial time algorithm.

Theoretically combinatorial optimization problems can be divided into two classes: those which can be solved by a polynomial time algorithm and those which cannot. In principle polynomial time algorithms are easy to solve on a computer and those which are not polynomial are very difficult. The theory of NP-completeness [GJ79] has been developed in an effort to gain insight into this mysterious class of problems.

A non-deterministic algorithm is an algorithm in which there are choices. We assume that when such an algorithm is executed, the computer fortunately makes the right choice on each occasion.(It goes without saying that no such machine exists). If a problem can be solved in polynomial time by such an algorithm then we say it is in the class NP (Non-deterministic Polynomial). The travelling salesman problem belongs to this class. An alternative way of thinking about this classification is that a problem belongs to the class NP if, given a possible solution to the problem, we can check whether it is a solution or not in polynomial time.

We say that any problem which can be solved by a polynomial time algorithm is in the class P. Obviously $P \subset NP$. It is not known whether P=NP or not. A problem is said to be hard if finding a polynomial time algorithm for a hard problem we are at the same time looking for a solution to all the problems in NP. It can be shown that certain hard problems are also in NP, these are called NP-complete problems. Unfortunately, it is not possible to find complete solutions to these problems in reasonable amounts of computation time. In trying to solve such problems we have two choices, either go for an exact optimal solution and restrict the size of the problem which can be solved or look for approximate or nearly optimal solutions for a larger class of problem. We are interested in this second class of algorithms and these are known as heuristic algorithms. The mapping problem can be regarded as a combinatorial optimization problem. Two heuristic algorithms that have been used to try and solve the mapping problem are Simulated Annealing [RGV83] and Tabu Search [Glo89], [Glo90].

2.4.2 General Formulation

In this section we give a general formulation of combinatorial optimization problems and give examples relating to graphs.

Formally, a combinatorial optimisation problem consists of a cost function f with domain S and co-domain $\mathbb{R}$ the set of real numbers

$$f: S \rightarrow \mathbb{R}$$

S is a finite set and is called the solution space. Problems can either be maximisation or minimisation problems.

For minimisation the problem is to find $x^* \in S$ such that $f(x^*) \leq f(x)$, for all $x \in S$.

In the case of maximisation we require $f(x^*) \geq f(x)$.

Since we can transform a maximisation problem into minimisation by multiplying the cost function by -1, it is sufficient to think of all problems as minimisation problems. Whilst this formulation gives the general structure, certain consideration must be given to the construction of S and F.

To illustrate this, consider the problem of partitioning a graph into two. Let G be a graph where G = (V,E) with a cost of 1 i.e. c(u,v) = 1 associated with each edge (u,v) $\in$ E. The problem is to partition the vertices of G into two subsets of equal sizes such that the cut set has minimum cost i.e. the sum of the cost of all those edged with end points in different subsets is minimum.

Therefore, we can define the set of feasible solutions S to be

$$S = \{(S_1, S_2) \mid S_1 \cup S_2 = V, |S_1| = |S_2|\},$$

and the cost function f(x), $x=(S_1,S_2) \in S$ to be

$$f(x) = \sum_{u \in S_1, v \in S_2} c(u,v)$$

2.4.3 General Purpose Algorithms

One important problem shared by all combinatorial optimisation problems is that the solution space is ragged and unpredictable. Classical optimisation problems can often be solved by hill climbing techniques. We optimise a function by always going up, when we can no longer go up we have reached the summit and this we hope, is the best solution. In a complex landscape we may have chosen the wrong hill to climb and although we have reached the top, there is a higher hill elsewhere. The complexity of the landscape for combinatorial optimisation problems makes the scenario almost inevitable. We are therefore forced to search for our optimum value by taking a more heuristic strategy. We shall now examine the general principles of the most successful techniques.

In examining general methods, we require the concept of a neighbourhood. Given a cost function f and a member x of the solution space S then a neighbourhood N(x) of x is a member of S which is close to x in some sense. A neighbourhood structure is a mapping which defines a set of neighbourhoods for every x in S. To illustrate how this may work, consider again the problem of partitioning a graph. A neighbourhood N(x) can be reached from x via a single pair-exchange between members of the two different subsets of x. Naturally, the best neighbour of x corresponds to the member in N(x) that gives us the highest decrement in cost.

It is illuminating to consider how the concept of a neighbourhood can be applied to obtain a standard hill climbing method. We first generate a random start and evaluate the cost. We next examine a neighbourhood of the random start and see if the cost has improved. If there is an improvement, we move to that position and examine the neighbourhood of the new position. If the cost deteriorates, then we examine a different neighbourhood. We repeat this strategy iteratively until there is no improvement. The problem with this method is that we can easily find ourselves in a position in which no improvement is possible but we have reached an optimum. The method is rather like trying to find the lowest point in a range of mountains. First randomly pick a starting point and select a direction to walk. Check that the direction is downhill, then walk in that direction until you are no longer going down. Repeat the procedure until you are forced to stop. The chances of this method getting you to the lowest point are virtually zero. The most likely is that you would be trapped in the bottom of a small valley. To find a way out of the solution it would be necessary to climb upwards in order to go down further in the future. The methods of Simulated Annealing [KGV83] and Tabu search [Glo80], [Glo90] are based on this concept.

2.4.4 Simulated Annealing

2.4.4.1 Introduction

The Simulated annealing approach is based on ideas from statistical mechanics. It can be viewed as an enhanced version of the iterative improvement, in which an initial solution is repeatedly improved by making small local changes until no such change gives a better solution. Simulated Annealing randomizes this procedure in a way that allows for occasional uphill moves (changes that worsen the solution) in an attempt to reduce the probability of becoming stuck in a poor but locally optimal solution.

2.4.4.2 Methodology

The difficulty with local optimisation is that it has no way of backing out of unattractive local optima. A move to a new solution is never made unless the direction is downhill, that is, a better value of the cost function.

Simulated Annealing is an approach that avoids getting trapped in local minima by allowing an occasional uphill move. This is done using a random number generator and a control parameter called the *temperature*. As typically implemented, the Simulated Annealing approach involves a pair of nested loops and two additional parameters, *a cooling ratio r, 0 < r < 1,* and an *integer temperature length L.* Figure 2.7 shows the steps of the Simulated Annealing algorithm. In Step 3 of the algorithm, the term "freezes" refers to a state in which no further improvement in cost *S* seems likely.

The core of this procedure is the loop at Step 3.1. The random number $e^{-\Delta/T}$ will be a number between 0 and 1, where Δ and *T* are positive, and can rightfully be interpreted as a probability that depends on Δ and *T*. The probability that accepting an uphill move of size Δ decreases as the temperature cools down, and, for a fixed temperature *T*, small uphill moves have higher probabilities of being accepted than larger ones.

```
1. Obtain an initial solution $S$.
2. Obtain an initial temperature $T > 0$.
3. While not yet frozen do :
        3.1 Perform the following loop $L$ times.
                3.1.1 Let $S'$ be a random neighbour of $S$.
                3.1.2 Let $\Delta$ = cost ($S'$) - cost ($S$).
                3.1.3 If $\Delta \leq 0$ (downhill move)
                        Then set $S = S'$.
                3.1.4 If $\Delta > 0$ (uphill move)
                        Then set $S = S'$ with probability $e^{-\Delta/T}$.
        3.2 set $T = rt$ (reduce the temperature)
4 Goto Step 1
```

Figure 2.7: Simulated Annealing algorithm

2.4.4.3 Addressing Graph Partitioning using Simulated Annealing

Recalling that in a graph partitioning problem, we have a graph $G = (V, E)$ and we want to partition the graph such that the partition $V = V_1 \cup V_2$ of V into two equal sized sets has the minimum number of edges being shared by the sets V_1 and V_2. Using the simulated annealing method, a solution will be any partition $V = V_1 \cup V_2$ of the vertex set and not just a partition into equal sizes sets. Two partitions are neighbours if one can be obtained from the other by moving a single vertex from one of its sets to the other set.

Therefore, if (V_1, V_2) is a partition and $v \in V_1$, then $(V_1 - \{v\}, V_2 \cup \{v\})$ and (V_1, V_2) are neighbours.

The cost of a partition (V_1, V_2) is defined to be :

$$\mathrm{Cost}\,(V_1, V_2) = \mid \{\, \{ u, v \} \in E : u \in V_1 \,,\, v \in V_2 \} \mid \; + \; \alpha \,(\mid V_1 \mid - \mid V_2 \mid)^2$$

where $|V_1|$ is the number of elements in set V_1 *and* α is a parameter known as the imbalance factor.

Although this method allows infeasible partitions to solutions, it carries a penalty cost according to the square of the imbalance. Consequently, at low temperatures, the sets tend to be perfectly balanced. The penalty function approach is common to many implementations of simulated annealing and it is often effective because the extra solutions that are allowed give new escape routes out of local optima.
The initial solution is randomly generated. If the final solution stays unbalanced, then a greedy heuristic is used to put it into balance. The heuristic then repeats the following operation until the two sets of the partition contain the same number of vertices: find a vertex in the larger set with the least increase in cutsize, and move it. The best feasible solution found is noted, be it the modified final solution or some earlier feasible solution encountered earlier on.

2.4.2.4 Analysis of Method

Simulated annealing has proved succesful in certain practical domains. However, there are certain areas of potential difficulties for the approach.
One question that needs to be addressed is the running time of the algorithm. It has been observed by many researchers that the simulated annealing requires large amounts of computational time to perform well. For partitioning unstructured meshes, the computational time for partitioning is just as important as the sub-meshes obtained since the time taken to partition the mesh must be a small fraction of the overall solution time of the problem.
Johnson et al [JAMS89] discusses simulated annealing for graph partitioning. They showed that for sparse random graphs, it tended to outperform the Kerninghan-Lin method [KL70]. However, it did not perform so well on a graph that was generated with

a built-in geometric structure.

Williams [Wil91] also compared the simulated annealing method with two other methods. One method was the recursive spectral bisection and the other was recursive orthogonal bisection. Recursive Orthogonal Bisection partitions a planar graph by placing a horizontal or vertical line such that half the vertices lie on either side of it. Williams showed that the parallel implementation of simulated annealing took 20 times longer than the recursive spectral bisection for the test cases that he used. The running time of an application was measured after being mapped by the three methods he was comparing. Using a graph with 5722 nodes, the running time of an application was fastest using simulated annealing. Even though simulated annealing ran significantly longer than the recursive orthogonal bisection method, the running time for the simulated annealing partitioned application (which was the best mapping) was only 21% less than the running time for the recursive orthogonal bisection (which was the worst mapping).

2.4.5 Tabu Search

2.4.5.1 Introduction

Tabu search is a fairly new approach to combinatorial optimisation where it is very similar to Simulated annealing in that it accepts "bad" moves in hope that there exits a better solution later on. It is characterised by aggressive local search during each iteration, and avoiding cycling in the solution space by keeping a short history of the recent solution [Glo89],[Glo90].

2.4.5.2 Methodology

There are two aspects in which Tabu search differs from Simulated annealing.

1. It is more aggressive. The whole neighbourhood is searched for each iteration of the current solution and it is usually searches exhaustively to find the best candidate moves.

2. It is deterministic. The above exhaustive search for best candidate moves is repeated for each iteration. If a candidate move does not cause cycling in the solution space then this candidate move is made to avoid cycling no matter what sign its gain has. A "tabu list" is usually used to record the most recent move history.

Figure 2.8 shows the general framework of tabu search where S is used to represent the solution. C is used to represent the cost function and t is the length of the tabu list. The first solution is a random solution, and the algorithm repeats the loop at step 2 until some criteria for stopping has been met. During each iteration, the algorithm makes an exhaustive search of the solution space in the neighbourhood of the current solution which has not been traversed in the last t ($t > 1$) iterations. The current solution with the best cost. Some of the main points for the tabu search algorithms are :

1. The design of the neighbourhood system effects the selection procedure. Usually, each iteration is made more aggressive if there is a large neighbourhood and this can prove to be very time consuming.

2. The design of the contents of the tabu list. If a current solution is transformed to S by a move m, then some attributes of S or m should be captured by the corresponding cell of the tabu list, so that S will not be traversed again in the next t steps. At one extreme, the solution S can be stored directly in the tabu list. However, in practice, in order to save memory space and checking time, some attributes of S will be stored in the tabu list to prevent m or m^{-1} to be used in the next t iterations. If a more detailed set of attributes of a solution or of a move in each cell of the tabu list is used, then more memory space and checking time will be incurred during the search of the solution space, and the searches will be less restrictive because less solutions (as well as the ones visited in the last t iterations) will be tabued. However, if a more simplified set of attributes of a solution or move in each cell of the tabu list is used, then the implementation will make more efficient use of space and time for each iteration, and since extra solutions will be tabued, the searches will be more restrictive.

3. The design of the aspiration level function. To make more efficient use of space and time, most designs of the contents of the tabu list will tabu too many solutions as well

as those that have been visited in the last t iterations therefore increasing the risk of losing good move candidates. However, an aspiration level A(m,S) can be defined for each pair of move m and solution S such that if C_3 (m(S)) > A(m,S) the tabu status of m for the current solution S can be outweighed to capture the common properties of the earlier applications of m to solutions sharing the same attribute values of S.

4. The design of the length t of the tabu list. The length of the moved history that is saved in the tabu list is determined by the parameter t. Suppose that we have a local optimum S and that it needs at least t^l consecutive "downhill" moves to get to another local optimum S^l. Therefore, a necessary condition for S to reach S^l is that $t \geq t^l$. Generally, the longer the tabu list, the more time for tabu status checking for each move, and the more restrictive the search process. However, by having a tabu list that is too short can risk introducing cycling in the solution space. The parameter t can be a constant or a variable during the execution of the algorithm. Glover [Glo89] states that a tabu list of length 7 is appropriate for many applications.

```
1. Get a random initial solution S
2. If stop criteria not met, DO
        2.1 Let $S^l$ be neighbouring solution of S maximising
                $\Delta = C_3 (S^l) - C_3 (S)$ and that
            this neighbourhood has not been visited in the last t iterations.

        2.2 Let S = S'

3. Return the best solution S visited.

$C_3$ : Objective function
```

Figure 2.8: Tabu search algorithm

Tao et al [TZTS92] propose the following description for their tabu search algorithm for multiway graph partition:

For the tabu list design, they use a circular list to maintain vertices that have been moved in the last t (t > 1) iterations. They find that by having a more detailed characterization of past known moves commonly traps the search process in a small subspace of the solution space. For their problems, a constant tabu list of length 5 gave the best performance.

For their aspiration level A (m,s) for all pairs of m and s, they used the cost of the best visited solution, based on the same observation as pointed out above, more "flexible" searches implemented by a more sophisticated aspiration level definition tend to limit the real search freedom in the solution space.

2.4.5.3 Analysis of Method

Lim et al [LC91] compare the tabu search algorithm with the Simulated annealing algorithm for graph partitioning. They found that tabu search did not perform as well in terms of quality of solutions on random graphs, even though in most cases the results were close. However, they found that their tabu search algorithm was faster by two to three orders of magnitude.
Tao et al [TZTS92] studies show the importance of the design of the solution neighbourhood structure. They believe that the running time of their algorithms can be greatly reduced if they combine the aggressive search in the tabu search approach with the stochastic search in the simulated annealing approach. While the former is critical to finding "good" solutions in practical time frames, the latter is effective in avoiding cycling in the solution space.
As well as combining Tabu search with simulated annealing, other approaches have been looked at. For example, Tabu search has been modified so that recent implementations do not search the whole solution space but accept the first downhill move encountered. This improves speed but does not detract from the solution process.

In general, Tabu search algorithms are slower than other problem-specific heuristics, but they have been successfully applied to many problem domains. The relative performance of Simulated annealing and Tabu Search is primarily problem dependent.

We experimented with different neighbourhood structures and cost function with some of our simple meshes. Our preliminary findings were in accord with those mentioned above [ELJC93].

For example, one strategy taken was to use a cost function which was the correlation of the adjacency matrix of the mesh with an ideal structure matrix. In particular, if we wish to split the mesh into two sub-meshes, the ideal structure matrix has the form

$$\begin{pmatrix} E & 0 \\ 0 & E \end{pmatrix}$$

where E is matrix of all 1's
and 0 is a matrix of all 0's

Given a particular partition of a mesh the correlation between the re-arranged partitioned matrix and the ideal structure matrix gives a measure of how good a partition we have achieved.

This process was used on the example shown in Figure 2.4(a) and we obtained the partition that is shown in Figure 2.4(c).

Unfortunately, the time taken to achieve this partition on a 486 PC was approximately 20 minutes. Clearly, this method, whilst capable of producing excellent partitions is not feasible for any realistic problems.

Chapter 3
Recursive Clustering Algorithm

3.1 Introduction

The recursive clustering method was devloped by Sadayappan et al [SER90] in order to split and map graphs onto a local memory machine with a hypercube interconnection tolopgy. The algorithm explicity attempts to minimise the communication volume through the use of an iterative improvement heuristic based on the Kernighan-Lin mincut algorithm [KL70]. Kerninghan-Lin propose an efficient mincut bisection heuristic with an experimental determined time complexity of $O(n^{2.4})$. Their algorithm is based on finding an advantageous series of vertex-exchanges between the two partitions to minimise the communication between the two. The method is considered superior to other simple local search heuristics since it is endowed with the "hill-climbing" ability because of the swapping of a series of vertex-exchanges as opposed to simple perturbations.
The Kerninghan-Lin method is described in the next section.

3.2 Kerninghan-Lin Graph Bisection method

3.2.1 Definition of Problem

Given a graph G with costs on its edges, partition the nodes of G into subsets so as to minimise the total costs of the edges cut. The simplest problem to be partitioned is that of a graph G with 2n vertices which is to be split into 2 sub-sets with minimal cost between the subsets which will both contain n vertices each.

Let G be a set with 2n vertices and an associated cost C_{ij}, for each edge connecting vertices i and j.
We wish to partition G into 2 subsets A and B, each with n vertices, such that the "external cost" (i.e. cost between the two subsets)

$$K = \sum C_{ab} \text{ for all } a \in A \text{ and } b \in B \text{ is minimised.}$$

The method is as follows :

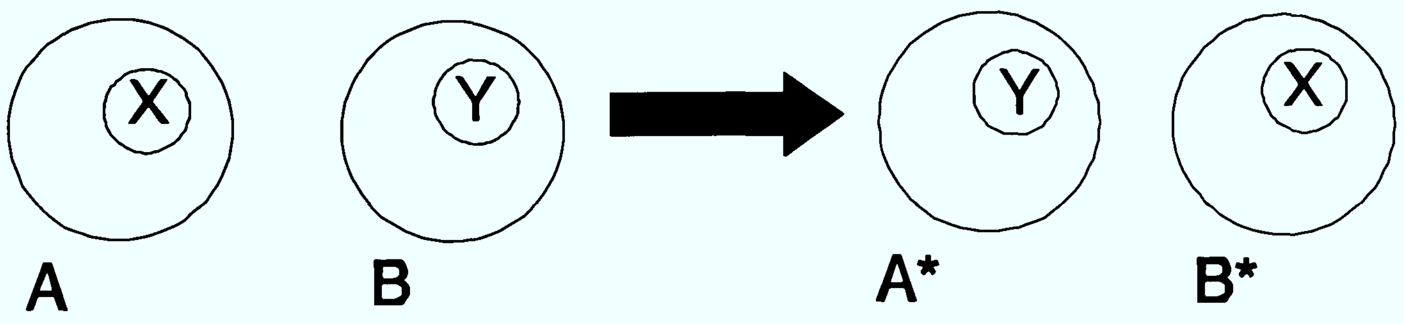
1. Arbitrarily assign each vertex to one of two subsets A and B of G.

2. Try to decrease the initial external cost between the two subsets by a series of interchanges of subsets A and B.

The subsets to be chosen are to be described . When it is not possible for further improvements, the resulting subsets A^1 and B^1 have a local minimum cost between them. Kerninghan and Lin state that the resulting partition also has a fairly high probability of being a globally maximum partition.

The procedure can then be repeated in a recursive manner, so that we can obtain 2^n subsets.

Given G and C_{ij}, suppose A* and B* is a minimum cost 2-way partition. Suppose A and B is any initial arbitrary 2-way partition. It is clear therefore that there exists $X \subset A$, $Y \subset B$ with $|X| = |Y| \leq n/2$ such that interchanging X and Y produces A* and B* as shown in Figure 3.1.



A* = A-X+Y

B* = B-Y+X

Figure 3.1: Interchanging sets X and Y between two subsets A and B.

However, the problem is to identify X and Y from A and B without having to consider all possible choices.

The process to identify X and Y is described below:

For each $a \in A$, we define an external cost E_a by

$$E_a = \sum_{y \in B} C_{ay}$$

and an internal cost by

$$I_a = \sum_{x \in A} C_{ax}$$

Similarly, for each $b \in B$, we define an external cost E_b by

$$E_b = \sum_{x \in A} C_{bx}$$

and an internal cost by

$$I_b = \sum_{y \in B} C_{by}$$

Let the difference between external and internal costs be :

$$D_z = E_z - I_z \quad \text{for all } z \in G$$

It is stated that by considering any $a \in A$, $b \in B$, if a and b are interchanged, then the gain (i.e. the reduction in cost) is precisely

$$D_a + D_b - 2C_{ab}$$

This can easily be seen as follows:

Let t be the total cost for all connections between A and B that do not involve a or b. Then the external cost K is

$$K = t + E_a + E_b - C_{ab}$$

Exchange a and b; let K^I be the new cost. We obtain

$$K^I = t + I_a + I_b + C_{ab}$$

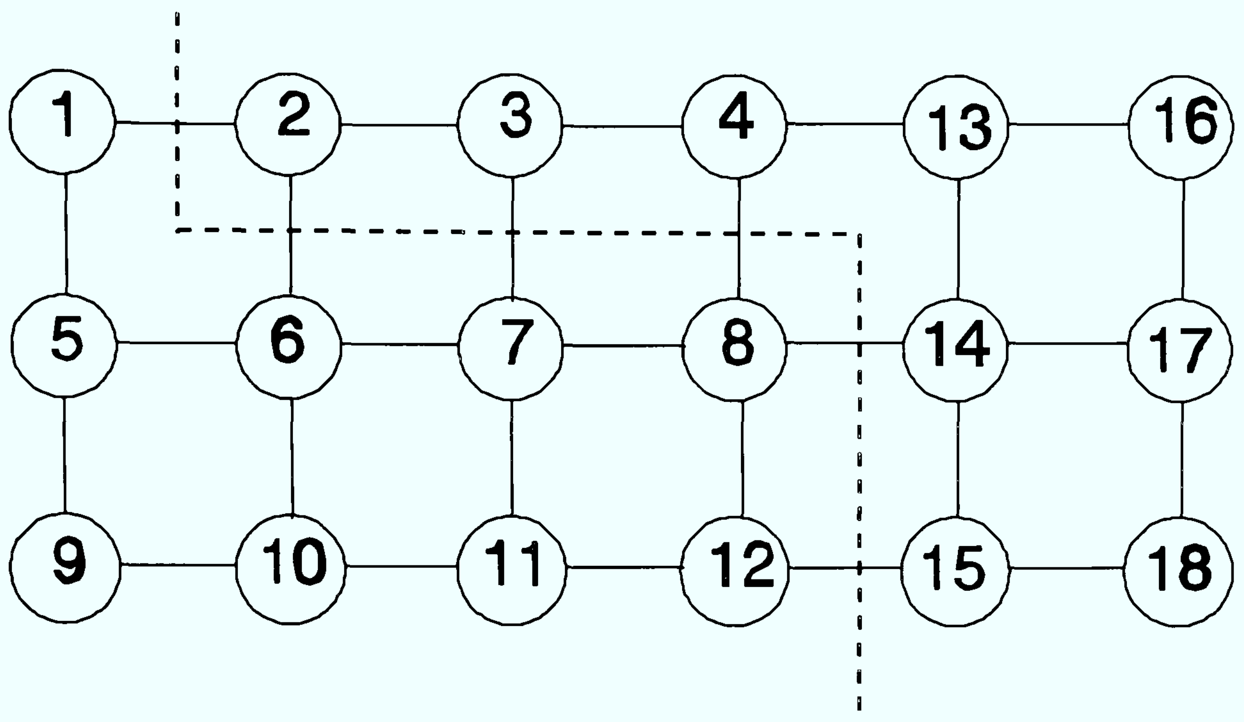
Therefore,

$$\begin{aligned} \text{Gain} &= \text{old cost - new cost} \\ &= K - K^I \\ &= E_a + E_b + \quad I_a + I_b - C_{ab} - C_{ab} \\ &= D_a + D_b - 2C_{ab} \end{aligned}$$

This is illustrated in the example given below:

<u>Example</u>

Suppose we have a graph with 18 vertices, whose initial arbitrary split can be seen in Figure 3.2, each subset containing 9 vertices. If we assume that each edge carries a cost of 1, then the external cost (i.e number of edges shared) between A and B is 6.



A = {1, 5, 6, 7, 8, 9, 10, 11, 12}

B = {2, 3, 4, 13, 14, 15, 16, 17, 18}

Figure 3.2: Example of graph with initial arbitrary split

If vertices numbered 2 and 8 are interchanged then calulating :

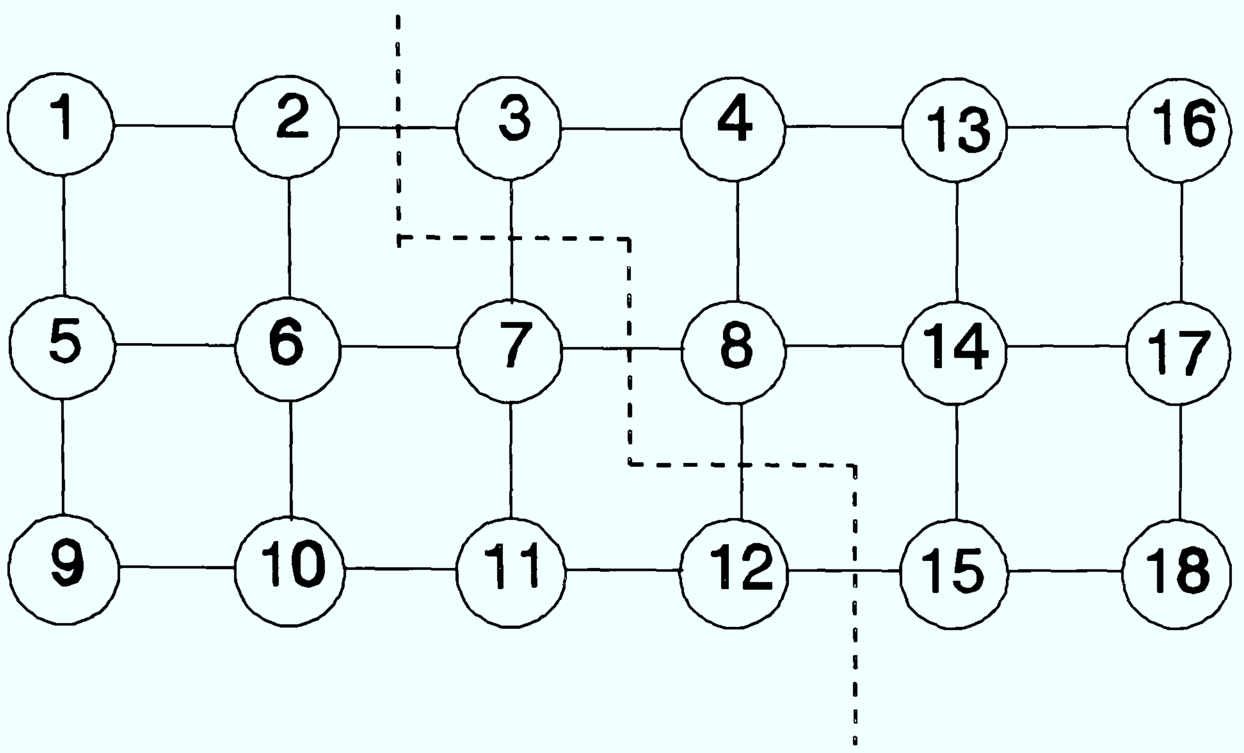
$E_2 = 2$ $\quad I_2 = 1$ $\quad$ So, $D_2 = 2\text{-}1 = 1$

and $E_8 = 2$ $\quad I_8 = 2$ $\quad$ So, $D_8 = 2\text{-}2 = 0$

Hence, using

$$\begin{aligned}\text{Gain} &= D_2 + D_8 - C_{ab} \\ &= 1 + 0 - 0 \\ &= 1\end{aligned}$$

Therefore, there is a gain i.e reduction in cost of 1 by interchanging vertices numbered 2 and 8. This can be seen in figure 3.3, where the vertices have been swapped and we can see that the number of edges shared by subsets A and B is now 5, i.e. a reduction in cost of 1 from the split shown in Figure 3.2



$A = \{1, 2, 5, 6, 7, 9, 10, 11, 12\}$

$B = \{3, 4, 8, 13, 14, 15, 16, 17, 18\}$

Figure 3.3: A reduction in cost of 1 by interchanging vertices 2 and 8

The algorithm proceeds as follows:

1. Compute the D values for all vertices of the graph.

2. Choose $a_i \in A$, $b_j \in B$ such that

$$g_1 = D_{ai} + D_{bj} - 2C_{aibj}$$

is maximum.

a_i and b_j correspond to the largest possible gain from a single interchange.

3. a_i and b_j are set aside temporarily, and are called a_1 and b_1 respectively.

4. Recalculate the D values for the vertices of $A - \{ a_1 \}$ and for $B - \{ b_1 \}$ by using :

$D_x' = D_x + 2C_{xai} - 2C_{xbj}$ $\qquad$ $A - \{ a_1 \}$

$D_y' = D_y + 2C_{ybi} - 2C_{yai}$ $\qquad$ $B - \{ b_1 \}$

These expressions can be easily verified:

The edge connecting vertices x and a_i is counted as internal in D_x. It is counted as being external in D_x^{l}, therefore C_{xai} must be added twice to make this correct.
Similarly, C_{xbj} must be subtracted twice to convert the edge joining x and b_j from external to internal.

5. The second step is now repeated by choosing a_2^{l}, b_2^{l} from $A - \{ a_1 \}$ and $B - \{ b_1 \}$ such that :

$$g_2 = D_{a2}^{l} + D_{b2}^{l} - 2C_{a2|b2|} \qquad \text{is maximum}$$

(a_1^{l} and b_1^{l} are not considered in this choice)

Therefore, g_2 is the additional gain when the vertices a_2^{l} and b_2^{l} are interchanged as well as having exchanged a_1^{l} and b_1^{l}.
This additional gain is maximum, given the previous choices.

6. a_2^{l} and b_2^{l} are set aside and the algorithm continues until all the vertices are exhausted, identifying (a_3^{l}, b_3),....(a_n, b_n^{l}) and the corresponding maximum gains g_3, g_4,....g_n.

As each pair (a^{l}, b^{l}) is identified, it is removed from contention from further choices. Therefore, the size of the subsets decrease by one each time a pair (a^{l}, b^{l}) is chosen.

If $X = a_1^{l}, a_2^{l}, ..., a_n^{l}$,

and $Y = b_1^{l}, b_2^{l}, ..., b_n^{l}$

then the decrease in cost when sets X and Y are interchanged is

$$\text{Cost} = g_1 + g_2 + ... + g_n.$$

Of course, if every pair of elements were swapped then the cost would be 0 i.e.

$$Cost = \sum_{1}^{n} g_i = 0$$

Obviously, some of the g_i's are negative because by swapping certain vertices can increase the cost. i.e. a reduction in cost of -2 means that there is an extra cost of 2.

7. *kbest* is chosen to maximise the partial sum :

$$\sum_{i=1}^{kbest} g_i = G$$

If G > 0, then a reduction in cost of value G can be made by interchanging the sets X and Y.

8. When this is done, the resulting partition is treated as the initial partition and the algorithm is repeated from Step 1.

If G = 0 (i.e. not worthwhile to make any more swaps), then we have arrived at a locally optimum partition.

Figure 3.4 shows a flow-chart for the algorithm described above.

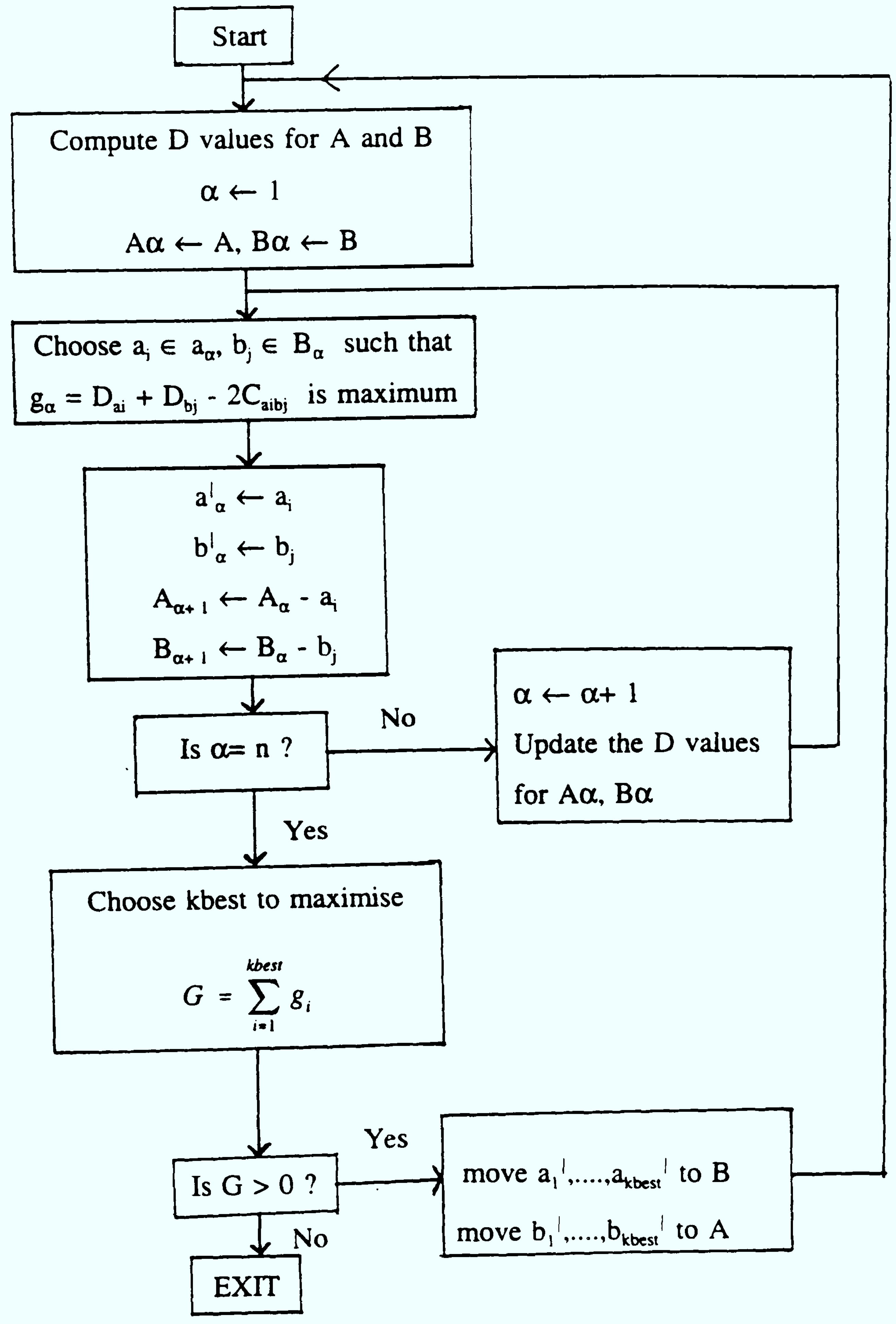

Start
Compute D values for A and B
$\alpha \leftarrow 1$
$A\alpha \leftarrow A, B\alpha \leftarrow B$
Choose $a_i \in a_\alpha$, $b_j \in B_\alpha$ such that
$g_\alpha = D_{ai} + D_{bj} - 2C_{aibj}$ is maximum
$a'_\alpha \leftarrow a_i$
$b'_\alpha \leftarrow b_j$
$A_{\alpha+1} \leftarrow A_\alpha - a_i$
$B_{\alpha+1} \leftarrow B_\alpha - b_j$
Is $\alpha = n$?
No
$\alpha \leftarrow \alpha + 1$
Update the D values
for $A\alpha$, $B\alpha$
Yes
Choose kbest to maximise
$G = \sum_{i=1}^{kbest} g_i$
Is G > 0 ?
Yes
move $a_1', \ldots, a_{kbest}'$ to B
move $b_1', \ldots, b_{kbest}'$ to A
No
EXIT

Figure 3.4: Flow chart showing steps of recursive clustering method

For example, using the graph which contains 18 vertices that is given in Figure 3.2

i	a_i^l	b_i^l	g_i	$G=\sum g_i$	kbest
1	2	8	1	1	1
2	12	3	2	3	2
3	11	4	-1	2	2
4	7	13	-2	0	2
5	10	16	-1	-1	2
6	6	17	-1	-2	2
7	1	14	1	-1	2
8	5	18	-1	-2	2
9	9	15	0	-2	2

Table 3.1: Table showing best interchanges

As we can see from Table 3.1, the value of G, which means a reduction in cost of 3, it at its highest when i = 2, therefore kbest = 2. Hence, for this particular example, the subsets X and Y contain the following vertices :

X = 8, 3

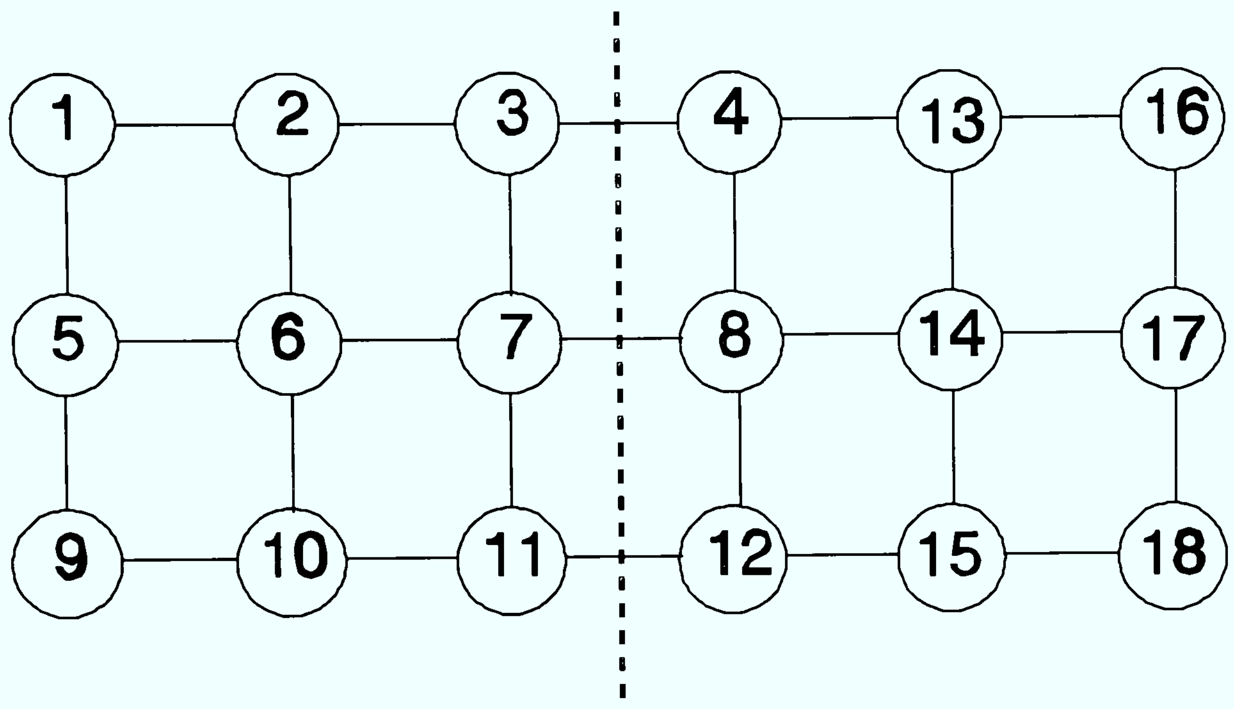
Y = 2, 12

And these two subsets are interchanged between A and B which gives us the partitions:

A = {1, 5, 6, 7, 9, 10, 11, 8, 3}

B = {4, 13, 14, 15, 16, 17, 18, 2, 12}

The resulting partition can be seen in Figure 3.5:



A = {1, 2, 3, 5, 6, 7, 9, 10, 11}

B = [4, 13, 16, 8, 14, 17, 12, 15, 18}

Figure 3.5: Resulting partition

If the algorithm is run again, then the value of G is equal to 0 since the partition shown in Figure 3.5 is optimal and a smaller cost could not be found.

3.2.2 Analysis of Method

Given such problems as above, one approach to solve these problems is to find the best

exchange involving, say, β pairs of vertices. for some β that has been specified in advance. By using a small value of β, the difficulty that is met is to identify good exchanges, but as β increases, the computational effort required grows rapidly.
The Kerninghan-Lin method sequentially finds an approximation to the best exchange of β pairs. To make the improvement as large as possible, β is chosen and is not specified in advance.
Since a sequence of gains g_i, i = 1,..., n is constructed and the 'maximum' partial sum is found, the process does not terminate immediately if any g_i is negative. Therefore, the process can sequentially identify sets for which the exchange of only a few vertices would actually increase the cost, while the interchange of the entire sets produces a net gain.

3.2.3 Running Time of The Algorithm

The operations involved in making one cycle of identification $(a_1^l, b_1^l),...,(a_n^l, b_n^l)$ and the selection of the subsets X and Y that are to be exchanged can be defined as a 'pass'. The total time taken to make a pass can be recalculated as follows:

The initial calculation of the D values is an n^2 procedure, because for each vertex of G, all other vertices of G have also to be considered.
To update the values of D, the time required is proportional to the number of values that need to be updated. Therefore, the total updating time in one pass grows as

$$(n - 1) + (n - 2) + ...+ 2 + 1$$

which is proportional to n^2.

3.3 Using Recursive Clustering to Partition Unstructured Meshes.

In the above section, the recursive clustering technique was described for partitioning graphs. However, we want to be able to partition unstructured meshes so we need to be able to view an unstructured mesh as a graph.

Since elements of the mesh communicate via shared nodes, then the cost between submeshes depends on the number of nodes shared by these two submeshes.

The unstructured mesh can be viewed as a Task Interaction Graph (TIG). The vertices of the TIG represent the elements of the mesh and the edges represent communication requirements between elements with edge-weights reflecting the relative amounts of communication involved.

This is demonstrated using the simple mesh shown in Figure 3.6. This can be transformed into a Task Interaction graph which can be seen in Figure 3.7.

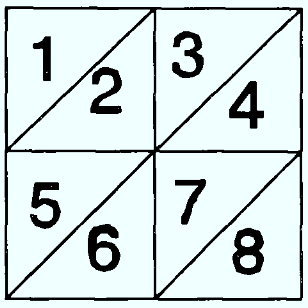


Figure 3.6: Simple mesh containing 8 elements

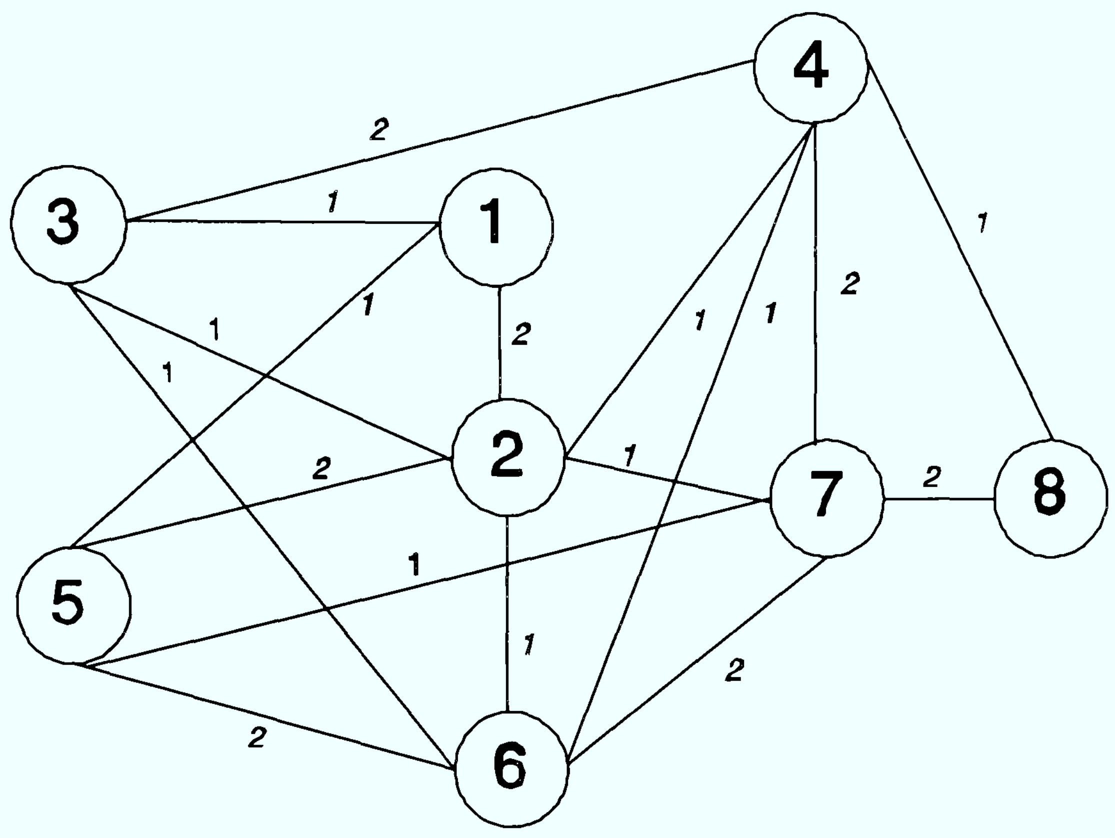


Figure 3.7: Task Interaction Graph

The information of the Task Interaction Graph can be stored in matrix form where the number of rows and columns equals the number of elements. For example, the TIG shown in Figure 3.7 could be stored as the following matrix:

$$\begin{bmatrix} 0 & 2 & 1 & 0 & 1 & 0 & 0 & 0 \\ 2 & 0 & 2 & 1 & 2 & 1 & 1 & 0 \\ 1 & 2 & 0 & 2 & 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 0 & 1 & 1 & 2 & 1 \\ 1 & 2 & 1 & 1 & 0 & 2 & 1 & 0 \\ 0 & 1 & 1 & 1 & 2 & 0 & 2 & 1 \\ 0 & 1 & 1 & 2 & 1 & 2 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 1 & 2 & 0 \end{bmatrix}$$

In order to find the amounts of data communication between two elements, for example, elements numbered 3 and 4, we can look up row 3, column 4 in the above matrix and we can see that the amount of data communication between these two elements is 2. i.e. they share 2 nodes.

3.4 Cost Function

An assumption that is made throughout this thesis is that the amount of data communiated via each node is homogeneous.

The cost function in our recursive clustering algorithm is slightly different to the one described for the Kerninghan-Lin algorithm.

The cost function is calculated by counting the number of nodes shared by the two submeshes.

If a mesh M with n elements has ben split into two submeshes A and B. We have a matrix C(a,b) = 1 if node a $\in$ A is the same node as node b $\in$ B, else C(a,b) = 0. Therefore,

$$Cost = \sum_{i=1}^{n} C(a_i,b_i)$$

This is illustrated in the example given in Figure 3.8. Looking at Figure 3.8(a), we can see that eight nodes are being shared between the two submeshes, therefore the communication cost between the two is equal to eight. However, if elements numbered 2 and 3 are swapped, then by looking at Figure 3.8(b), we can see that now only six nodes are being shared by the two submeshes. Hence, the communication cost has been reduced by two from eight to six. Therefore, it would be advantageous to implement this swap .

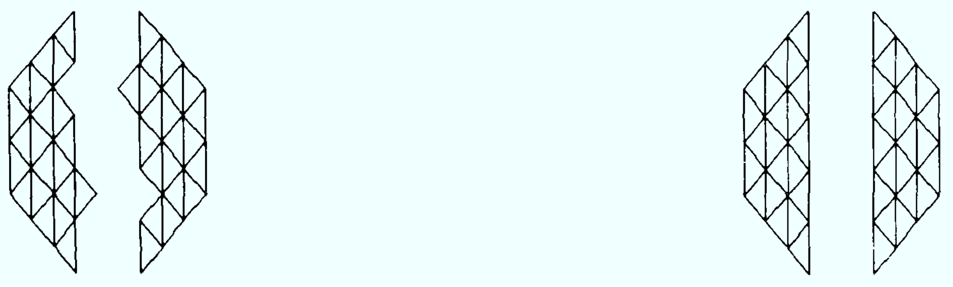


Figure 3.8: An example of the local cost function used in the recursive clustering method

3.5 The Algorithm

The recursive clustering algorithm proceeds as follows :

1. Arbitrarily assign each element to one of two clusters A and B, such that there is an approximately equal number of element on each.

2. Evaluate the communication cost of this partition and find which pairs of elements when swapped give the maximum reduction in costs. (This is done as described for the Kerninghan-Lin method).

3. Temporarily removing the previously swapped pair, find the next best pair and continue until no more pairs remain.

4. From the set of all swaps, find the subset which minimises the communication costs. Provided this reduces the cost, make the swap.

The procedure is repeated in a recursive manner, so that we obtain 2^n partitions.

When applied to the simple problem shown in Figure 3.9(a), the division into four domains is straightforward and can be seen in Figure 3.9(b). Using another simple example shown in Figure 3.10(a), the division into four and eight can be seen in Figure 3.10(b) and Figure 3.10(c).

It appears that the recursive clustering method works well on a variety of simple meshes and that the partitions obtained are well-clustered.

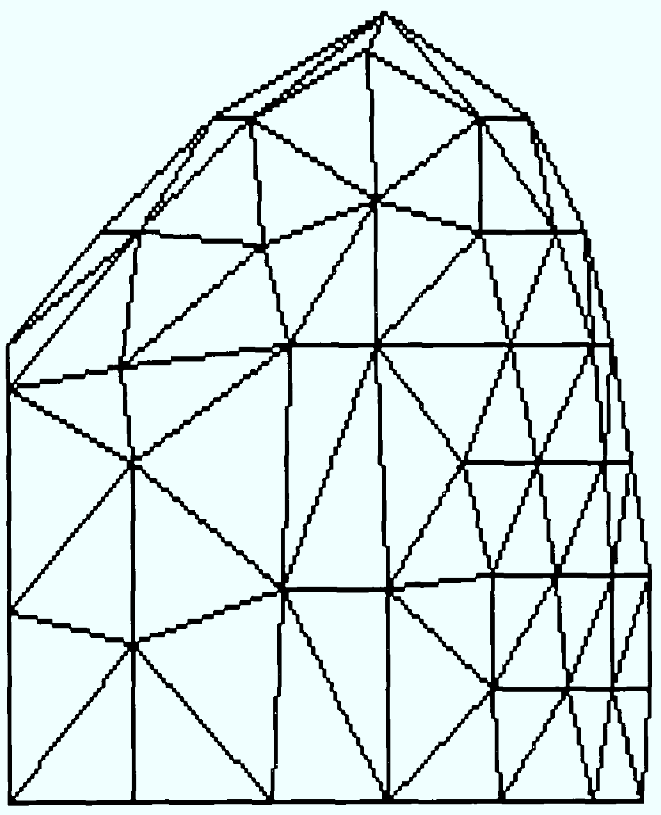


Figure 3.9(a): Unpartitioned mesh

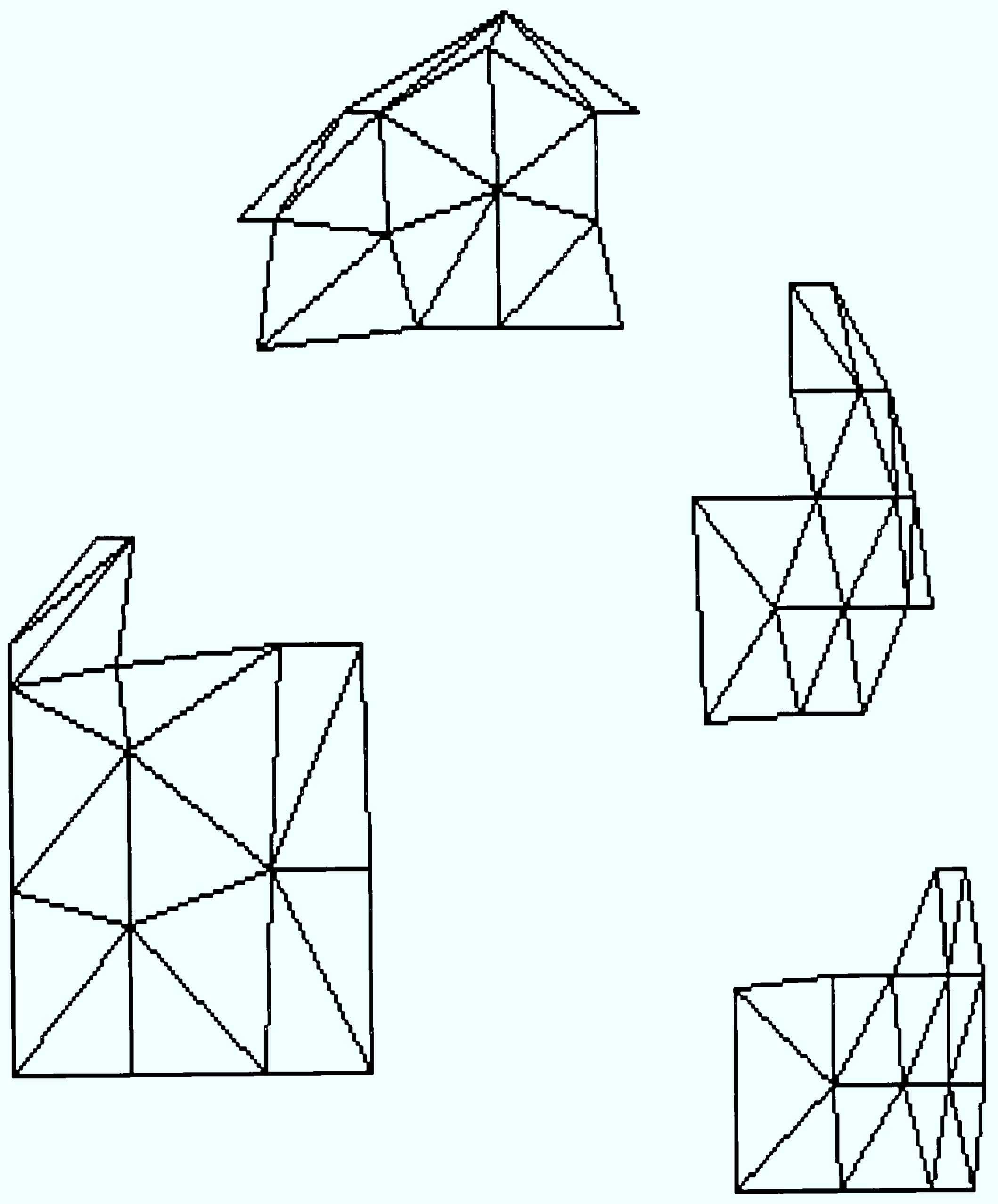


Figure 3.9(b): Decomposition into four

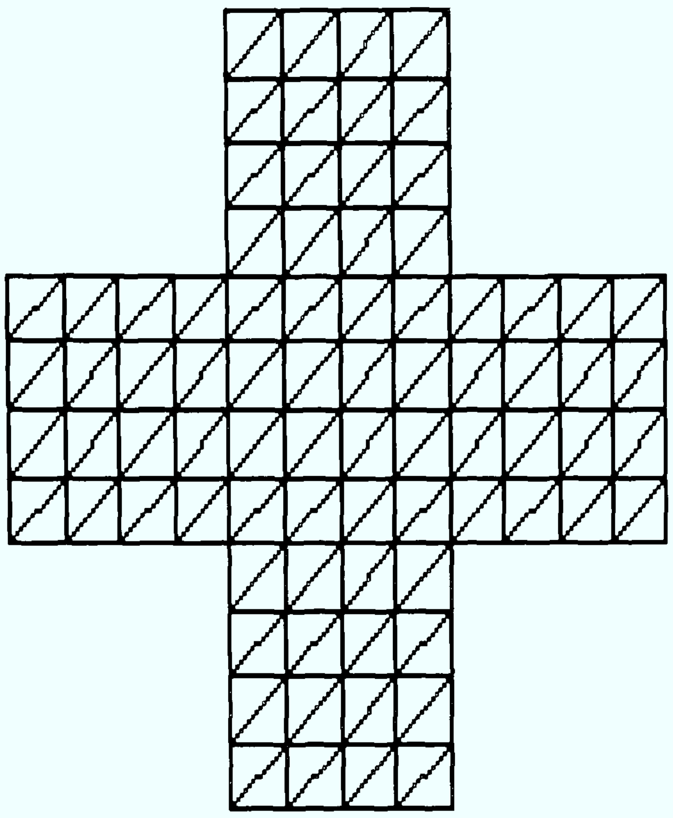


Figure 3.10(a): Unpartitioned mesh

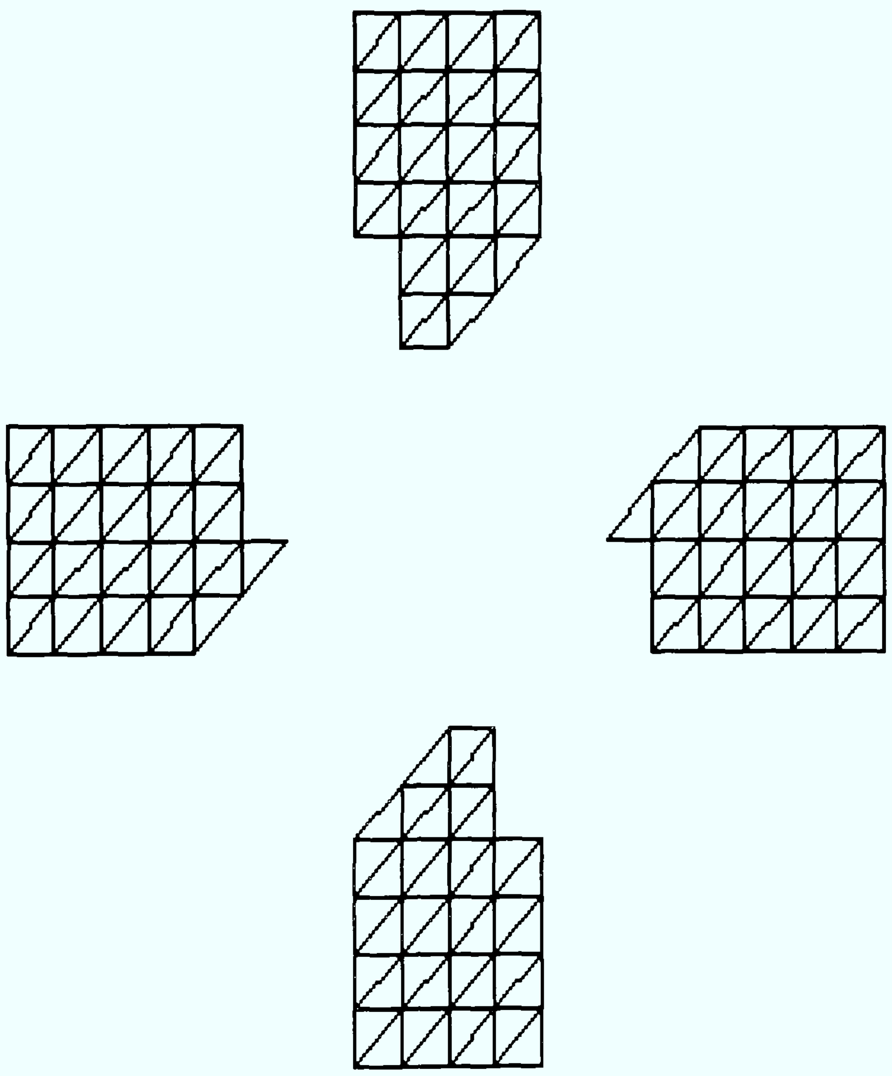


Figure 3.10(b): Decomposition into four

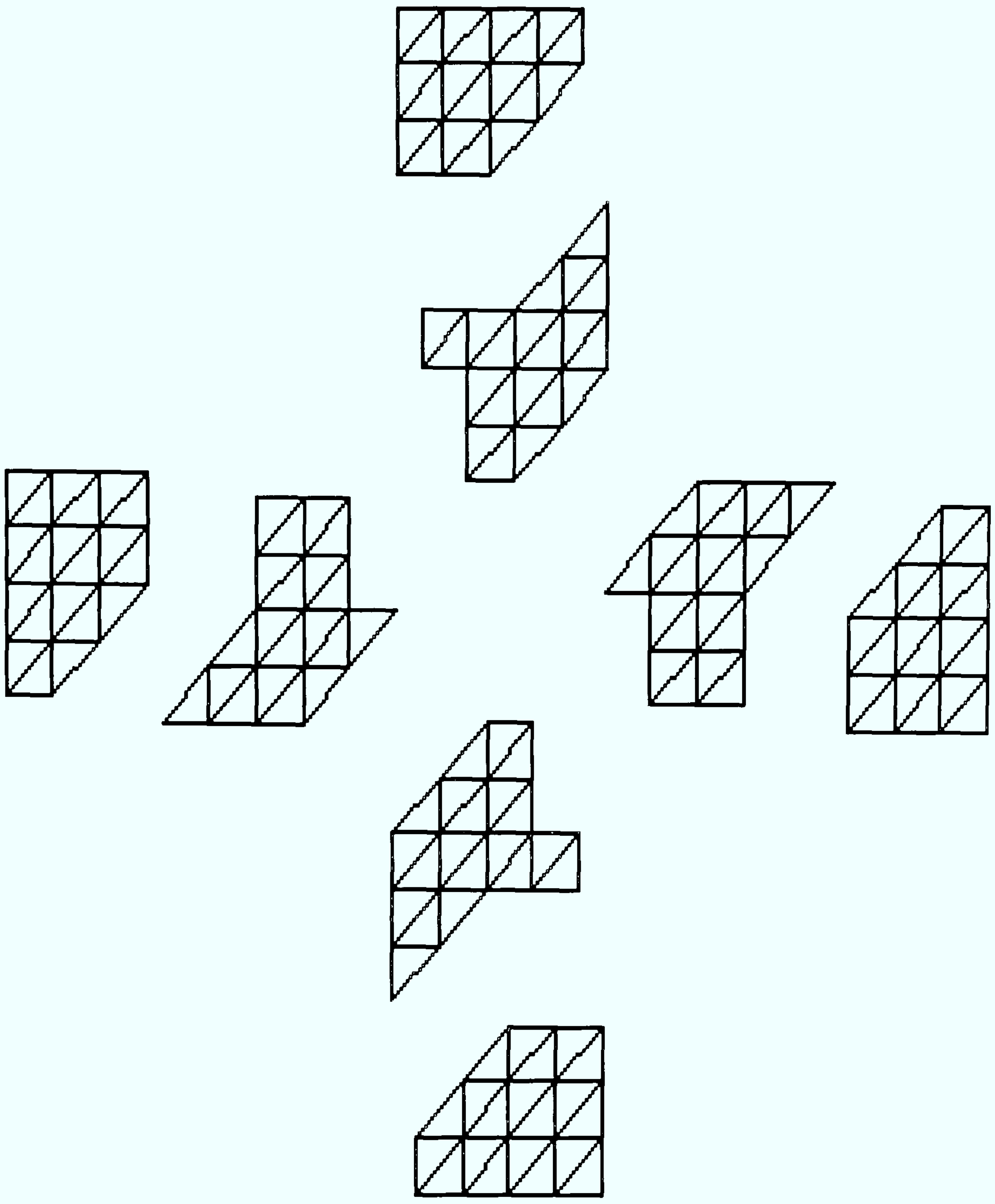


Figure 3.10(c): Decomposition into eight

However, the recursive clustering method does have its limitations.

1. The mesh can only be partitioned into 2^n sub-meshes.

2. Each split, even if perfect (i.e. optimal) does not imply an overall optimal solution.

3. The processor topology is not taken into account.

4. The optimisation procedure tends to get caught in local minima.

We have attempted to overcome some of these limitations and Chapter 4 describes the modifications that we have made to the standard recursive clustering.

Chapter 4
Extension of the Recursive Clustering Algorithm

4.1 Introduction

The examples demonstrated in Chapter 3 show that the recursive clustering method can perform well on a variety of simple meshes and that the partitions obtained are adequately clustered. Despite its limitations (some of which are shared by other approaches anyway) it seems to have the potential to be very effective at mesh decomposition with minimal inter-processor communication. In this chapter, we describe modifications to the standard recursive clustering algorithm to provide a more flexible and robust mesh decomposition software tool.

4.2 Eliminating Constraint of 2^n sub-meshes.

As mentioned in the previous chapter, the recursive clustering algorithm, because of its recursive nature, is limited to splits of 2^n clusters. Since we only obtain 2^n sub-meshes, we can only map onto a multi-processor system with 2^n processors, but for our purposes this is a severe limitation, since we want to be able to map onto any number of processors. However, it is a straightforward matter to eliminate this constraint of splitting into 2^n sub-meshes and obtaining *k* sub-meshes.. The solution to this limitation is to use an iterative technique for any number of processors. The essential idea is to start with an arbitrary split into *k* sub-meshes with equal load. Then, every pair of sub-meshes is operated on to minimise the communication costs between the pair as in the conventional recursive clustering algorithm. We shall now call this algorithm the iterative clustering algorithm.

4.3 Local Minima Trap

Unfortunately, as with the recursive clustering algorithm, this method taking any number of processors into account is still susceptible to local minima trap.
There are two types of local minima traps which are described below:

4.3.1 Type 1

This is the standard local minima trap i.e the method fails to find a minima of the given objective function and gets stuck in local optima.

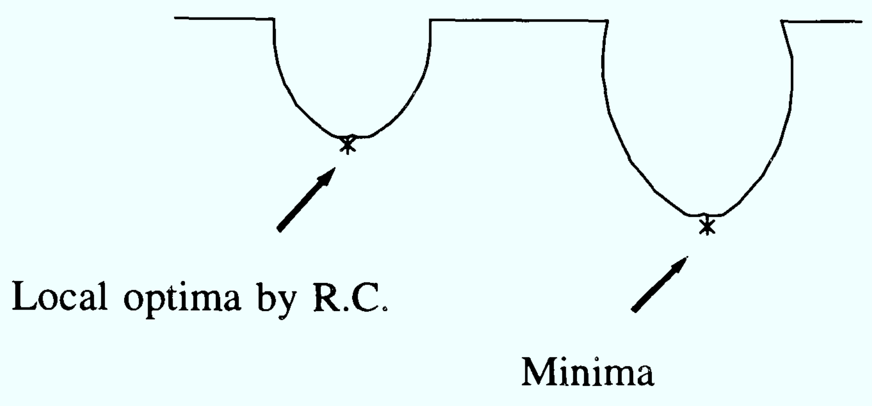


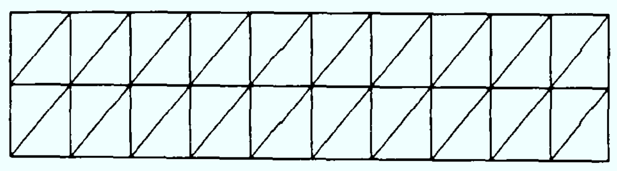
Figure 4.1: Local minima trap

4.3.2 Type 2

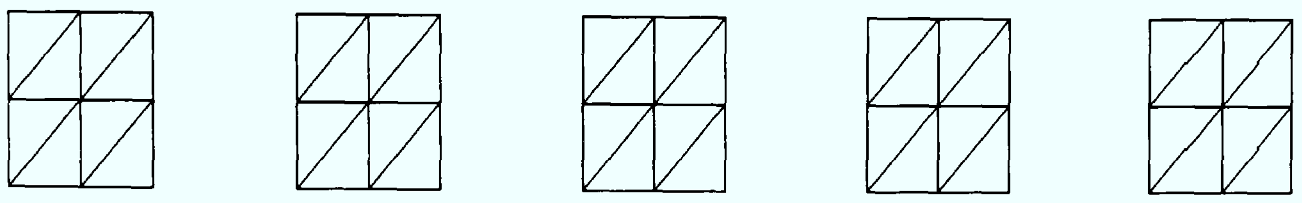
The other type of local minima trap is that the method uses a pairwise optimisation technique. Even if the method finds a global minimum at each stage, the solution found will not be a global minimum.
For example, looking at the simple mesh shown in Figure 4.2(a), if we wanted to partition this mesh into five, the minimum cost solution would be 12 and an optimum result can be seen in Figure 4.2(b). However, when using the method above, the result of partitioning into five can be seen in Figure 4.2(c). At this point the sub-meshes are allocated onto any processor. The cost of this split is 15. By looking at this split, we can

see that the elements that have been assigned to processor 3 are not all connected (even though sometimes a disconnected sub-domain is more desirable, in this example it is not). It finds the local minima between the pair but does not find an overall global minimum. Unfortunately, it is the optimisation of the local cost measure using the swapping strategy that leads to the undesirable disconnected domains in Figure 4.2(c). For example, the number of nodes shared by processor 3 and 4 is 3, and hence the cost between these two is 3. An optimum cost between two processors for this particular mesh is 3. When the algorithm tries to optimise the cost between processors 3 and 4 it does not take into account the fact that processor 3 also shares nodes with processor 5. Since the cost between processors 3 and 4 is already 3, the algorithm will not be able to overcome the problem of the disconnected sub-domain because it has no way of knowing how the other sub-meshes are connected.

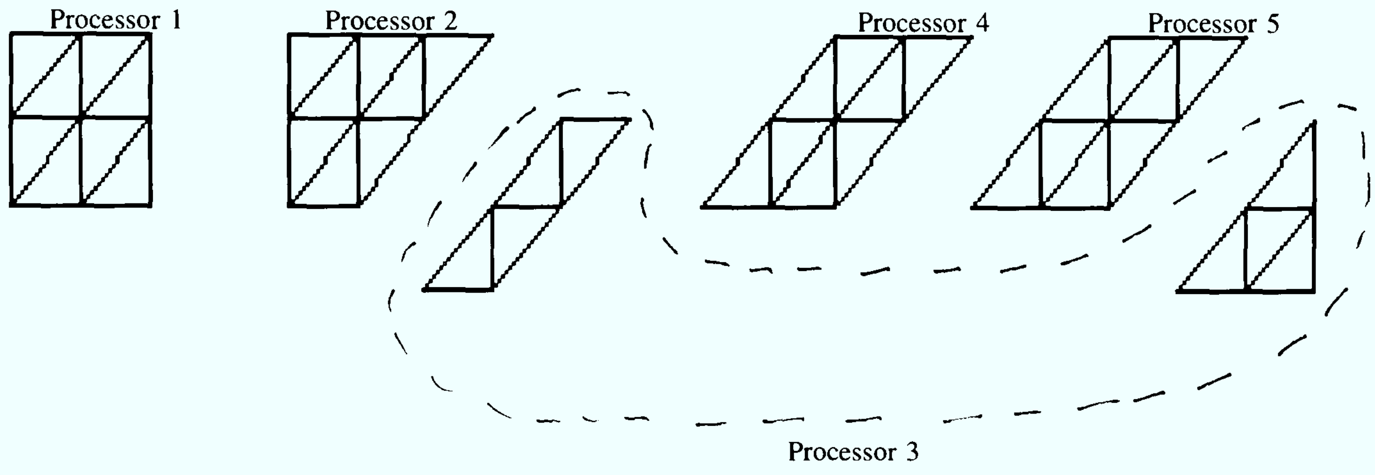
Since the initial partition into *k* sub-meshes is arbitrary, the resulting sub-meshes obtained obviously depends on the starting sub-meshes. By choosing good starting sub-meshes, it may make the probability of an optimal solution higher, although this tendency is very difficult to evaluate. Another reason for choosing good starting sub-meshes is that is can reduce the amount of work required to make the system pairwise optimal.



a) Simple mesh



b) Optimal solution cost=12



c) Solution using Iterative Clustering with random initial partition, cost=15

Figure 4.2: Limitation of the iterative clustering algorithm

4.3.3 Renumbering Elements

One technique that can be used to create good starting sub-meshes is to renumber the elements. With the iterative clustering algorithm, the initial split is arbitrary and if we have a mesh with n elements and we want to split into k sub-meshes, then elements numbered *1* to *k/n* are set aside for one sub-mesh, elements numbered *k/n+1* to *2k/n* are set aside for another sub-mesh, and so on.

Figure 4.3(a) shows a simple example with only 16 elements and the figure shows the elements numbered arbitrarily. Say, for example, we wish to partition this mesh into two, elements numbered 1 to 8 are in one sub-mesh and elements numbered 9 to 16 in the other. Figure 4.3(b) shows the initial split into two.

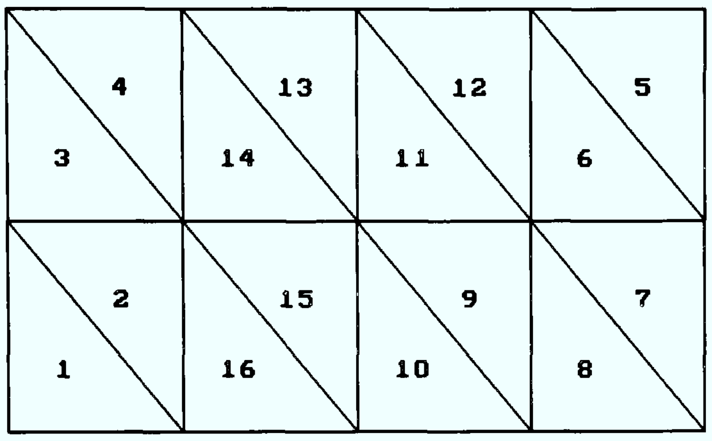


Figure 4.3(a): Simple mesh with random element numbering

Proceeding with the algorithm, we identify two elements a[i] and b[j], one from the sub-mesh A and the other from sub-mesh B. These two elements when interchanged produce the largest possible reduction in cost. The elements a[i] and b[j] are temporarily set aside, and the algorithm proceeds to find the next two elements from the remaining sub-meshes that again reduce the cost the most. This is continued until all the elements are exhausted. Table 4.2 shows the elements that have been chosen from each sub-mesh for each step.

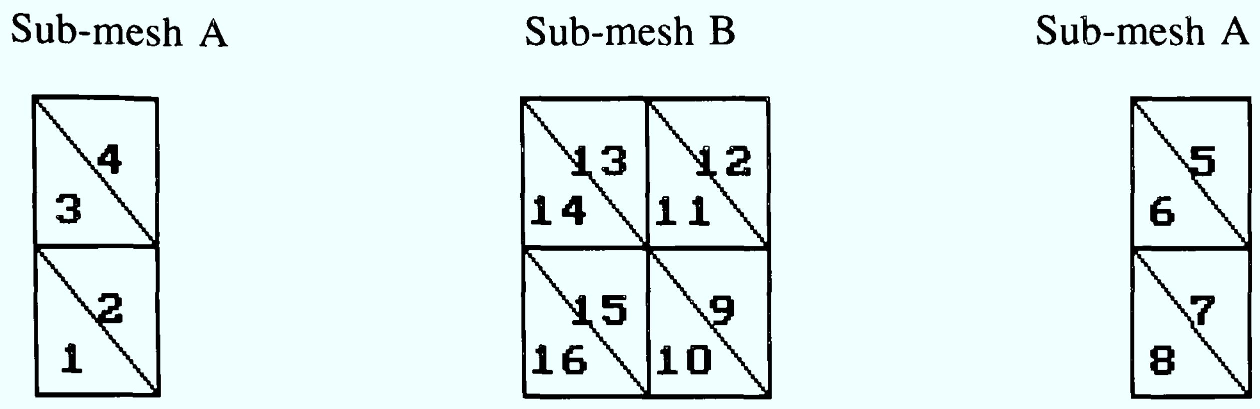


Figure 4.3(b): Simple mesh split into two sub-domains

a[i]	b[j]	MaxG	SumG	BigG	kbest
4	16	-2	-2	-2	1
3	12	-2	-4	-2	1
2	9	-2	-6	-2	1
1	11	6	0	0	4
8	10	-2	-2	0	4
7	15	1	-1	0	4
5	13	-2	-3	0	4
6	14	6	3	3	8

Table 4.1: Table to show the best pairs to be swapped

KEY:

a[i] and b[j] : Elements that when swapped give the largest reduction in cost.

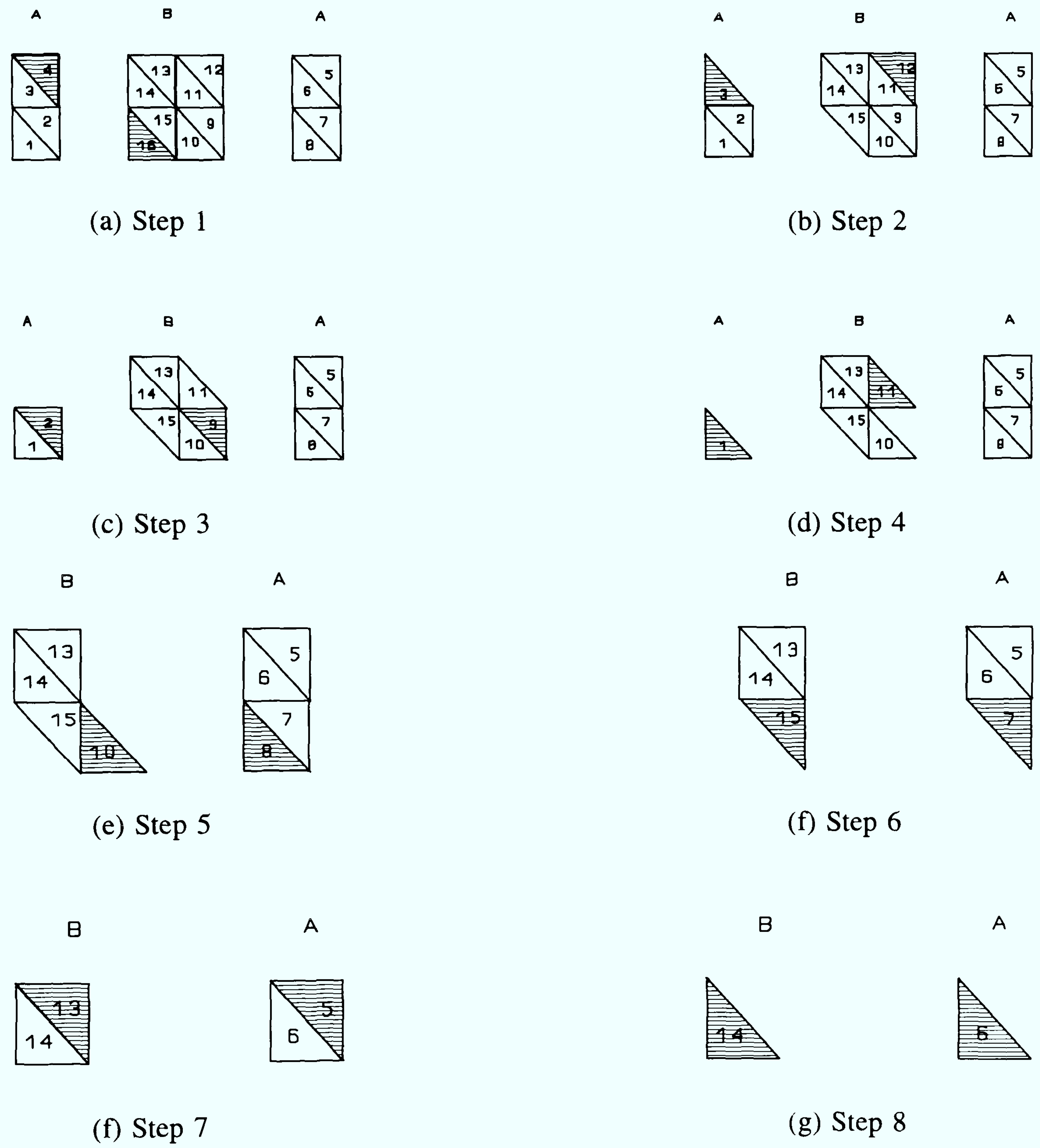
MaxG : The reduction (or increase) in cost when elements a[i] and b[j] are swapped.

SumG : The reduction in cost so far when k elements have been swapped.

BigG : The best reduction in cost so far.

Kbest : The number of pairs of elements in the sub-meshes that need to be swapped.

We can see by looking at Table 4.1 that the number of pairs of elements that need to be swapped is 8. Since we only have eight pairs of elements that are considered for swapping, it is obvious that all pairs of elements have been swapped. So all the elements that were in sub-mesh A are now in sub-mesh B, and all the elements that were in sub-mesh B are now in sub-mesh A. Figure 4.4 illustrates the elements that were removed for each step.



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

(e) Step 5

(f) Step 6

(f) Step 7

(g) Step 8

Figure 4.4: Illustration of elements that were removed at each step.

Figure 4.5 shows the resultant sub-meshes after the swapping has been made. Despite the fact that eight swaps have been made, it is clear that the algorithm has made no difference whatsoever to the cost between the sub-meshes.

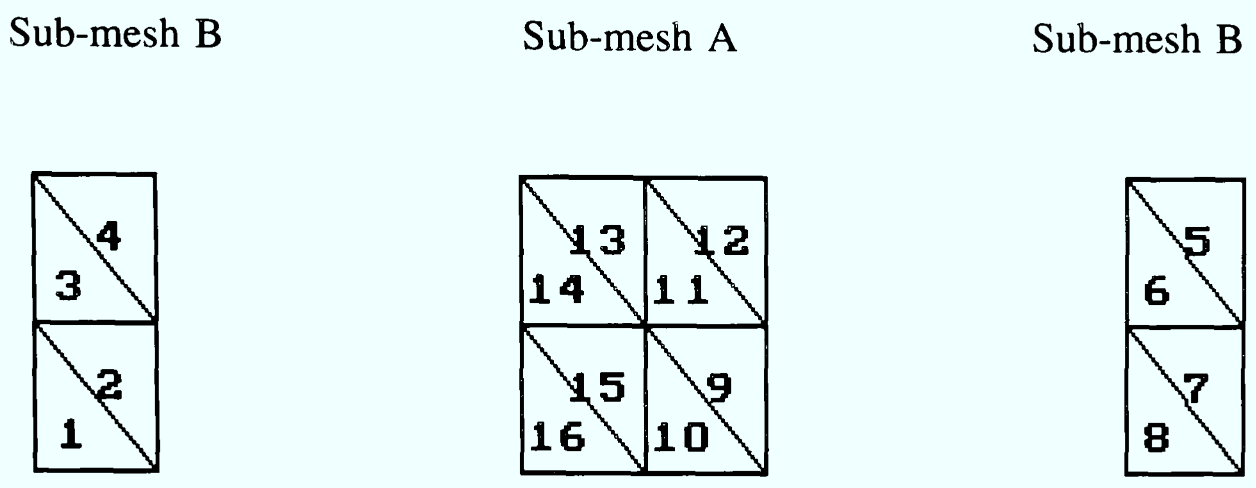


Figure 4.5: Sub-meshes after swapping

Figure 4.3 shows one example with the elements numbered randomly. Suppose we take the same mesh and number the elements differently to the way they are numbered in Figure 4.3. The element numbers can be seen in Figure 4.6(a). The initial arbitrary partition can be seen in figure 4.6(b), where the shaded elements are in sub-mesh A and the unshaded elements in sub-mesh B. We then continue with the algorithm until all pairs of elements are considered for swapping. The results can be seen in Figure 4.6(c) and we can see that an optimum cost of 3 between the two-submeshes has been found, with no disconnected sub-domains.

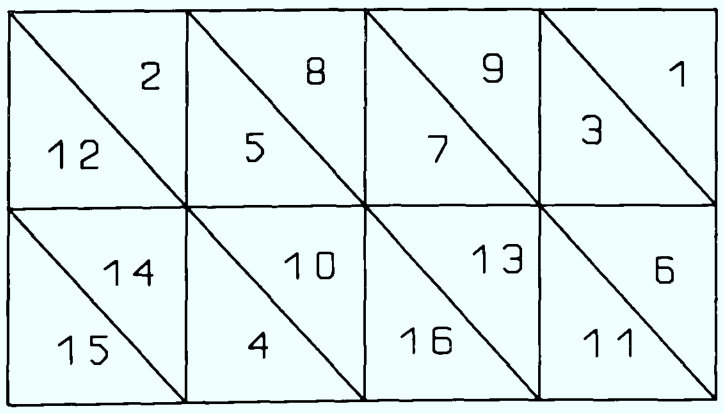


Fig 4.6(a): Simple mesh with numbered elements

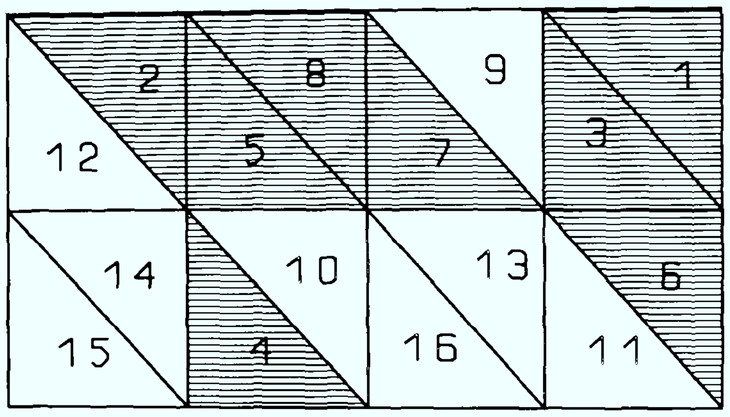


Figure 4.6(b): Initial arbitrary split

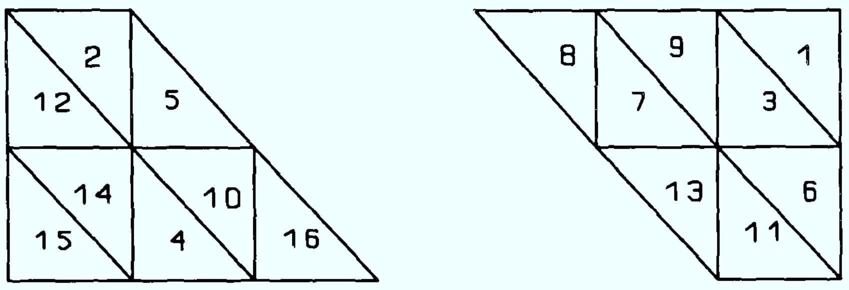


Figure 4.6(c): Final split, cost=3

Using the simple mesh shown in Figures 4.4(a) and 4.6(a) with two different sets of element numbers, it is obvious that the element numbers can be important. If we number the elements in such a way so that the starting partitions are fairly well connected, then we might be able to overcome the problem of obtaining final sub-meshes which are disconnected as shown in Figure 4.4(c). We looked at various techniques that can be used to renumber the elements and the one that we found to be most satisfactory is the Cuthill-McKee algorithm [CM69].

4.3.4 Cuthill-Mckee Algorithm

The Cuthill-McKee method provides a simple scheme for renumbering elements [CM69]. This algorithm was developed to number nodes in order to reduce the bandwidth of sparse symmetric matrices [AM65]. Our investigations show that it is also reliable for renumbering elements since we achieve good starting partitions when the elements are renumbered in this way.

The renumbering scheme is given below :

1. Choose an element to be relabelled 1. This element should be located at the extremity of the mesh and should, if possible, have few connections with other elements.
2. The elements which share a node (i.e those which are connected) with this element number 1 are relabelled 2,3, etc,. in the order of their increasing degree. (The degree of an element is the number of elements which share a node with this element).
3. The procedure is repeated by relabelling elements which are connected to element number 2 and which have not previously been relabelled.
4. The above procedure is repeated for each of the new element numbered 3,4 etc., until the renumbering is complete.

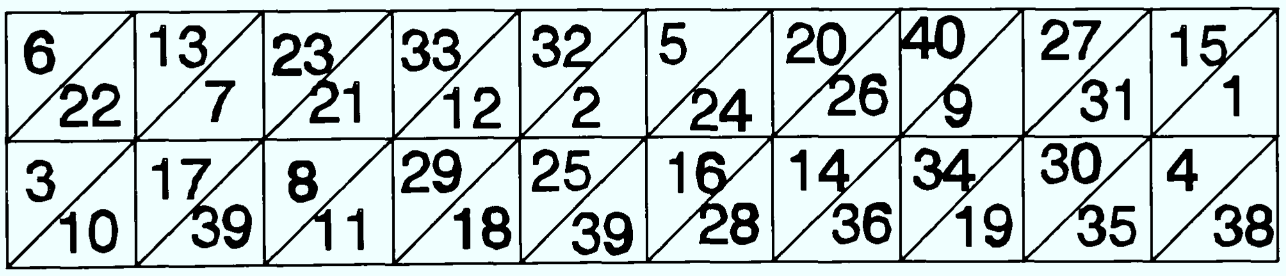


Figure 4.7(a): Mesh with original element numbers

The algorithm is applied to the simple mesh given in Figure 4.7(a), which is the same as the mesh shown in Figure 4.2(a). The element numbers that were used to obtain the sub-domains shown in Figure 4.2(c) can also be seen in Figure 4.7(a). The renumbering will yield the mesh with numbered elements shown in Figure 4.7(b). The element which has been relabelled 1 was previously element number 6 since it is one element which has fewer connections. Table 4.2 shows an element connection lists which can easily be constructed. The Cuthill-McKee algorithm may be more easily implemented by referring to such an element connection list. Consider the renumbering of elements using the information presented in Table 4.2 and starting with element number 6 since this element has fewer connections. By referring to row 6 of the connection list it is apparent that elements 3, 13 and 22 need to be renumbered 2, 3 and 4. Element numbers 3 and 22 should be renumbered first since both these elements have fewer connections than element number 13. Since it makes no difference which one of these two elements is renumbered first, the choice is arbitrary, hence we may renumber element number 3 before element number 22 . An examination of row 3 show that elements numbered 7,10 and 17 should be renumbered 4,5 and 6. According to the number of connections to element number 3, the first element that should be renumbered is element number 10.

Element	No of Connections	Connection List
1	6	4,15,30,31,35,38
2	9	5,12,16,18,24,25,29,32,39
3	6	6,7,10,13,17,22
4	6	1,15,30,31,35,38
5	7	2,16,20,24,25,32,39
6	3	3,13,22
7	9	3,8,10,13,17,21,22,23,37
8	9	7,11,12,17,21,23,29,33,37
9	9	14,19,26,27,30,31,34,36,40
10	6	3,7,13,17,22,37
11	7	8,12,18,21,29,33,37
12	9	2,8,11,18,21,25,29,32,33
13	7	3,6,7,10,17,22,23
14	9	14,19,26,27,30,31,34,36,40
-	-	-
etc.		

Table 4.2: Connectivity list for mesh shown in Figure 5.7(a)

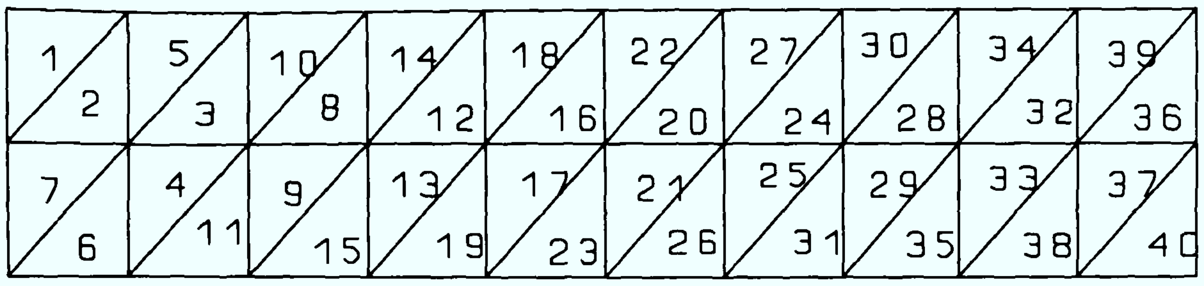


Figure 4.7(b): Cuthill-McKee renumbering

We now apply the iterative clustering algorithm to the mesh shown in Figure 4.7(a). If we choose to partition this mesh into five then the result can be seen in Figure 4.7(c). We can see by looking at Figure 4.7(c) that by renumbering the elements before applying the iterative clustering algorithm, we generate an optimum decomposition where a minimum cost of 12 is achieved as opposed to a cost of 15 with the decomposition of the same mesh with random element numbering as shown in Figure 4.2(c)

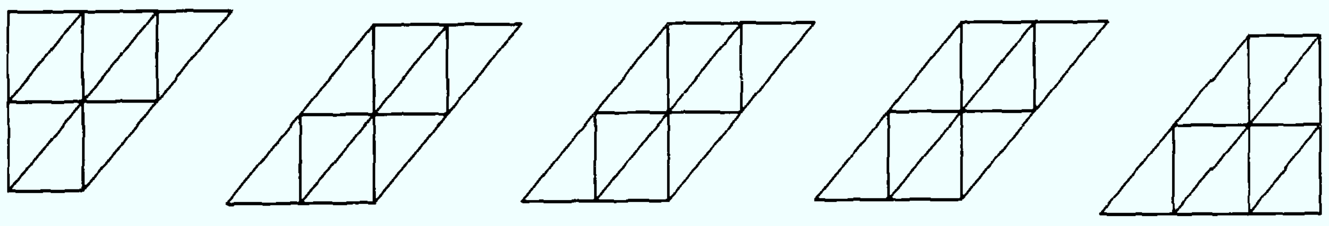


Figure 4.7(c): Split into five using renumbered elements, cost=12

Obviously, this is a very simple mesh to demonstrate the effectiveness of renumbering elements. Due to its rectangular shape, the renumbering scheme will work very well for any mesh of this shape therefore it is important to demonstrate the effectiveness of the algorithm on meshes without a rectangular-based shape. Figure 4.8(a) shows a Y-shaped mesh which has been partitioned into three sub-domains and the decomposed mesh is shown in Figure 4.8(b). Another example is shown in Figure 4.9(a) and this cross-shaped mesh has been decomposed into four sub-domains which is shown in Figure 4.9(b). It can be seen in both Figures 4.8(b) and 4.9(b) that the sub-domains achieved are well-clustered sub-domains with fairly low costs between each pair of sub-domains.

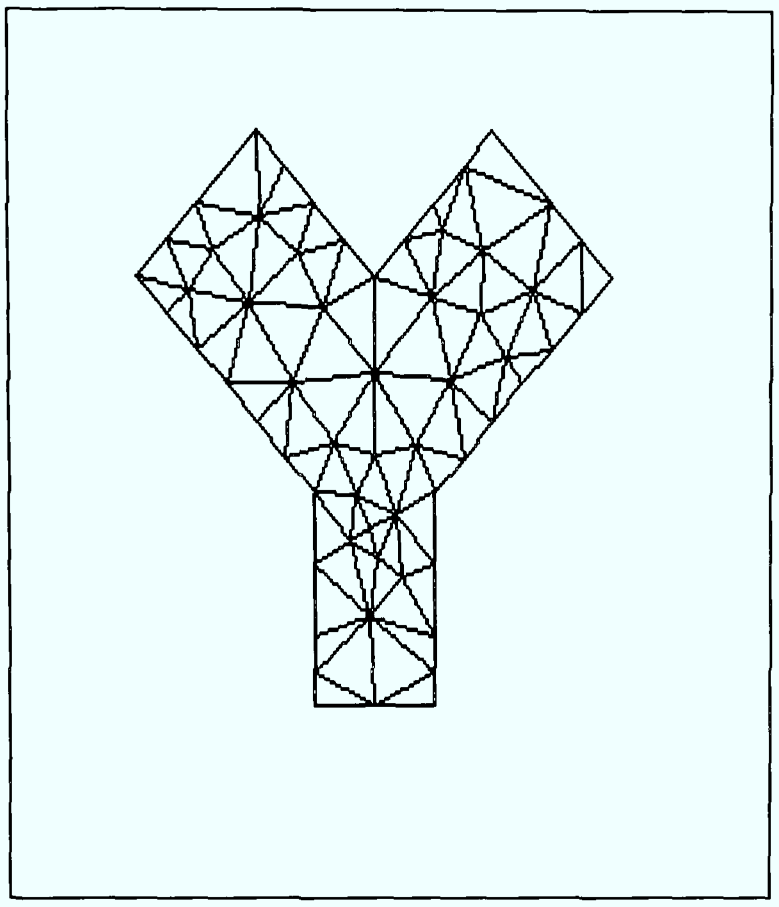


Figure 4.8(a): Y-shaped mesh.

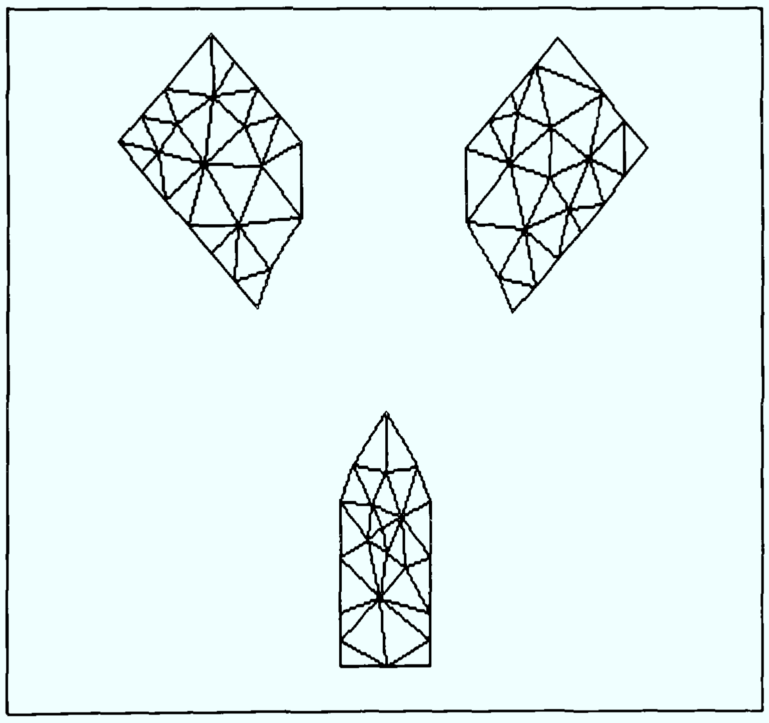


Figure 4.8(b): Y-shaped mesh split into three.

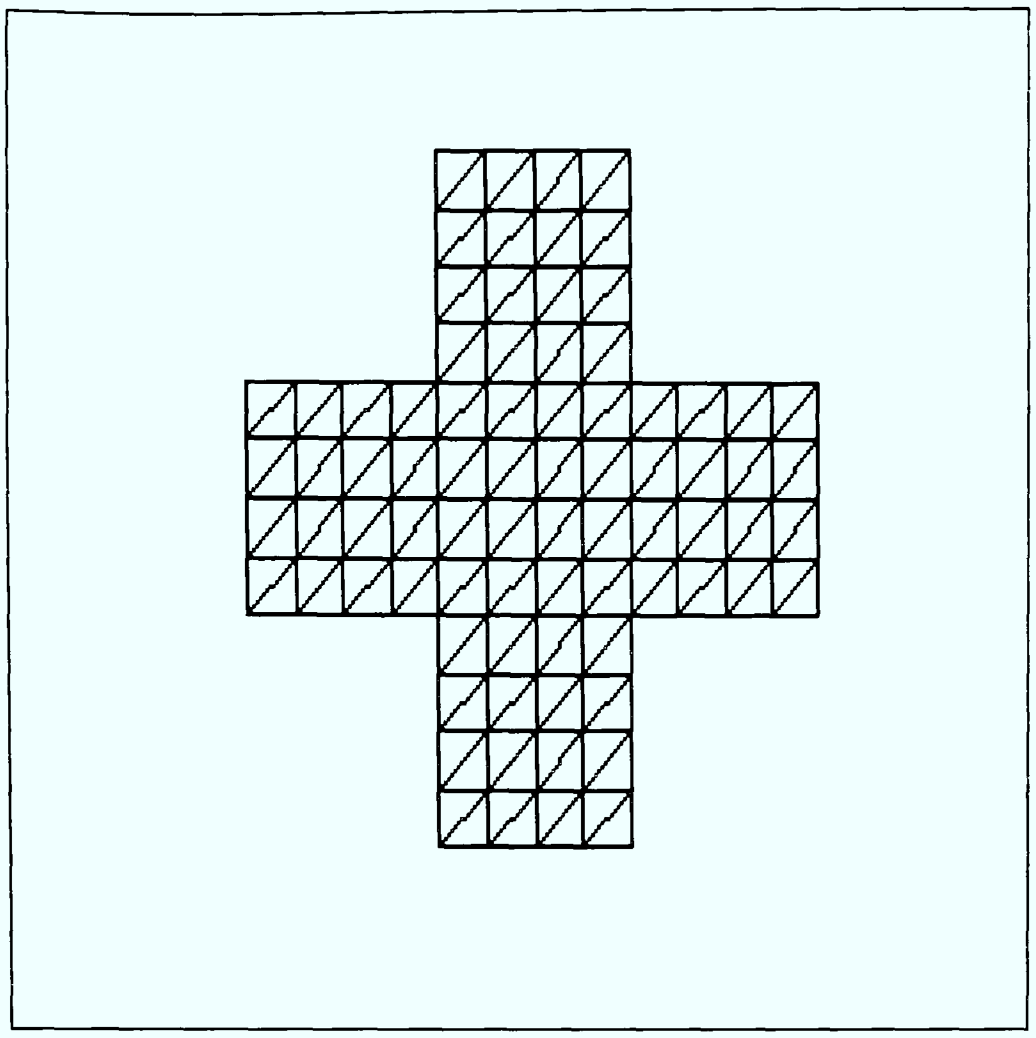


Figure 4.9(a): Cross-shaped mesh.

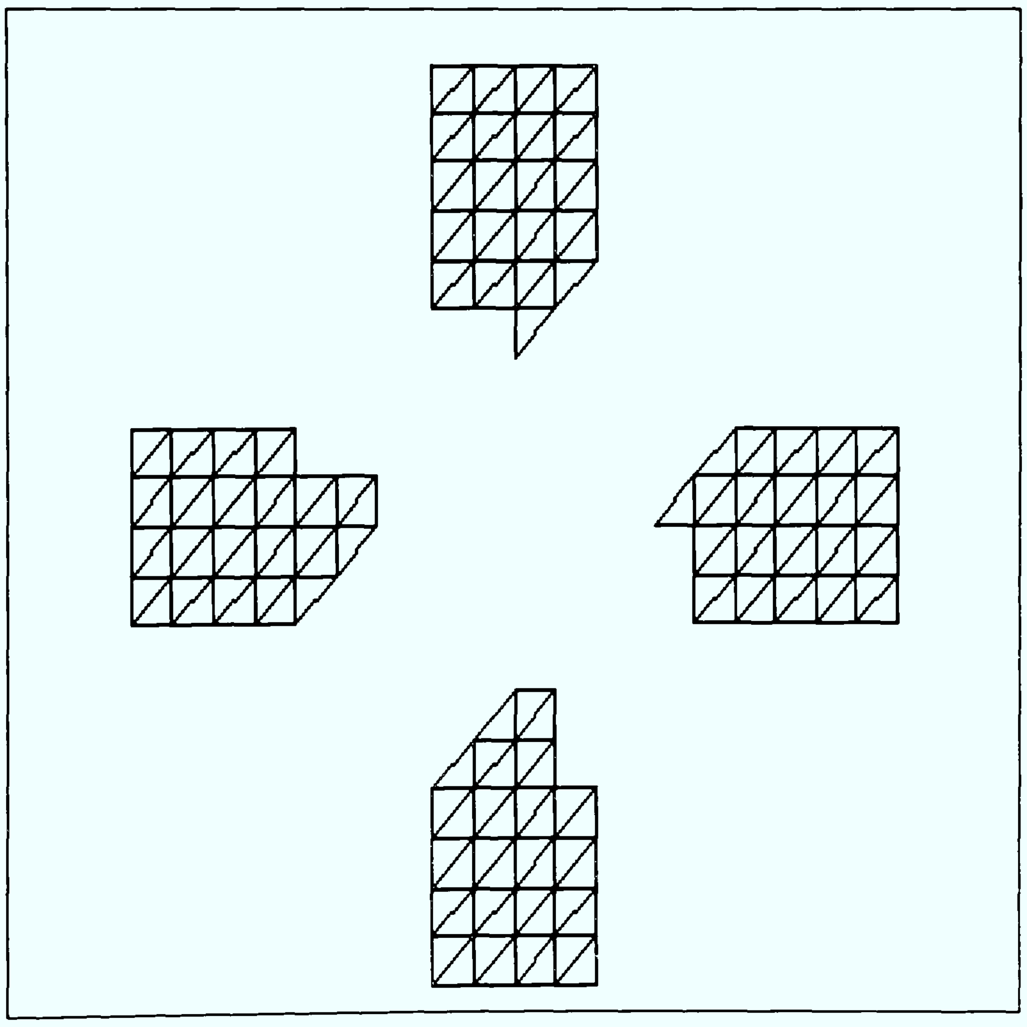


Figure 4.9(b): Cross-shaped mesh split into four.

4.4 Specifying Processor Topology

Another feature required in the iterative clustering method is the influence of the processor topology on the partition calculation (this is true for other methods too). For the recursive clustering method, the problem is not so acute when mapping onto a hypercube topology due to the high connectivity of the processor network, but when we have modified this to account for any number of processors, it doesn't take into account the processor topology at all. Therefore, a version is required which will account explicity for a given processor topology.

One way of modifying the iterative clustering method to account for the processor topology is to exploit the flexibility of the method by changing the cost function in the optimisation procedure. As we recall from Chapter 3, the cost between two sub-meshes is the number of nodes they shared.

The simplest function to minimise is the total inter processor distance travelled over the topology which enables all relevant communicaton to take place. Let's take the simple mesh illustrated in Figure 4.10(a) and suppose the processor topology that we wanted to map this mesh onto is configured as a simple chain of five processors as in Figure 4.10(b). Obviously we would not want an element on processor 1 to have its neighbour on processor 5 as they would have to communicate via 4 other processors. The mesh illustrated in Figure 4.10(a) shows a simple 40 element mesh together with an assignement of element numbers. The element numbers have been randomly generated and we assign elements numbered 1 to 8 to the first processor, elements numbered 9-16 to the second, etc. The unmodified cost function is based upon the number of shared nodes and hence takes no account of the location of neighbouring elements. In contrast the modified cost function contains a distance measure of communication cost which becomes part of the optimisation procedure.

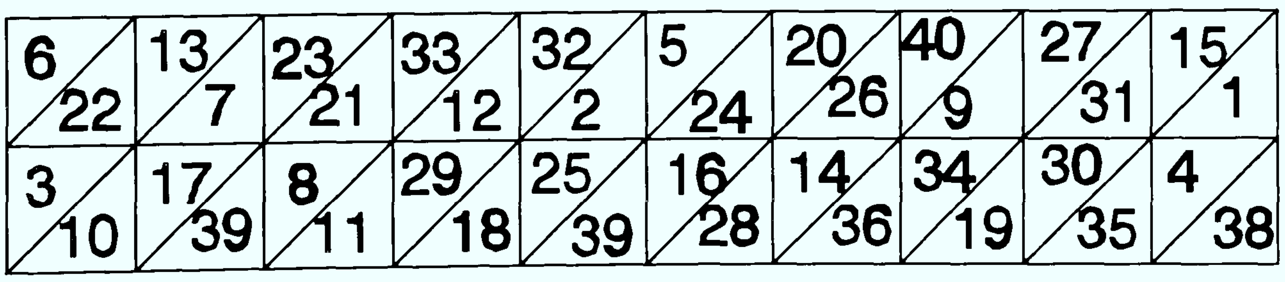


Figure 4.10(a): Simple 40 element mesh.

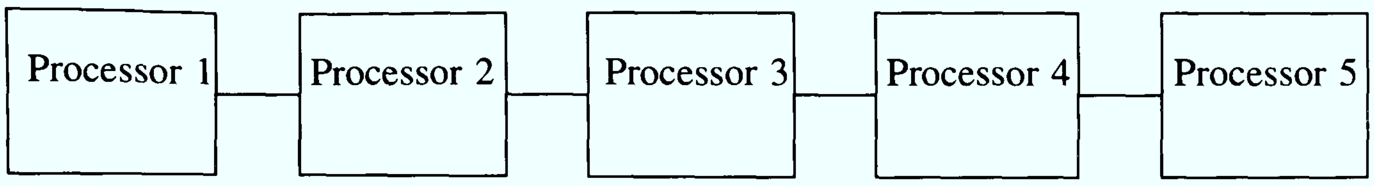
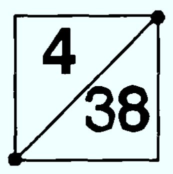


Figure 4.10(b): Processor pipeline.

Suppose an element on processor 1 shares a node with an element on processor 5, then the cost of this node is now 4 (not 1) since it has to be communicated via 4 other processors. An example of such a pair of elements is shown in Figure 4.10(c) which are two elements taken from the mesh shown in Figure 4.10(a). Since element 4 is on processor 1 and element 38 is on processor 5, they have to communicate via 4 processors and they share 2 nodes; we therefore assign each node a cost of 4. The cost of each node in the mesh is calculated and summed to give what we shall call the global cost.



Cost between element 4 and element 38 = 2 x 4 = 8

Figure 4.10(c): Two elements taken from mesh shown in Figure 4.10(a).

The basis of this global cost is to force elements to be on the same or neighbouring processors. If two neighbouring elements are on different processors which are some distance away, then it obviously makes the value of the global cost much higher. However, if one of the elements was moved to the same processor as its neighbour, then the global cost is greatly reduced. The global cost now reflects the processor topology and we proceed with a standard minimisation procedure. If we use the mesh given in Figure 4.10(a), then we can see by looking at Figure 4.10(d) that by using this global cost method, we generate an optimum decomposition (without renumbering the elements first) where all neighbouring elements are on the same or neighbouring processors and the minimum cost of 12 is achieved.

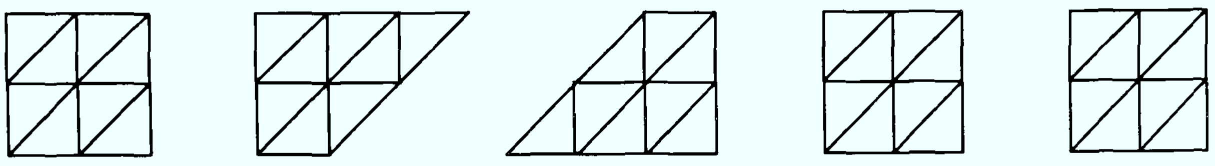


Figure 4.10(d): Split into 5 using the global cost method, cost=12.

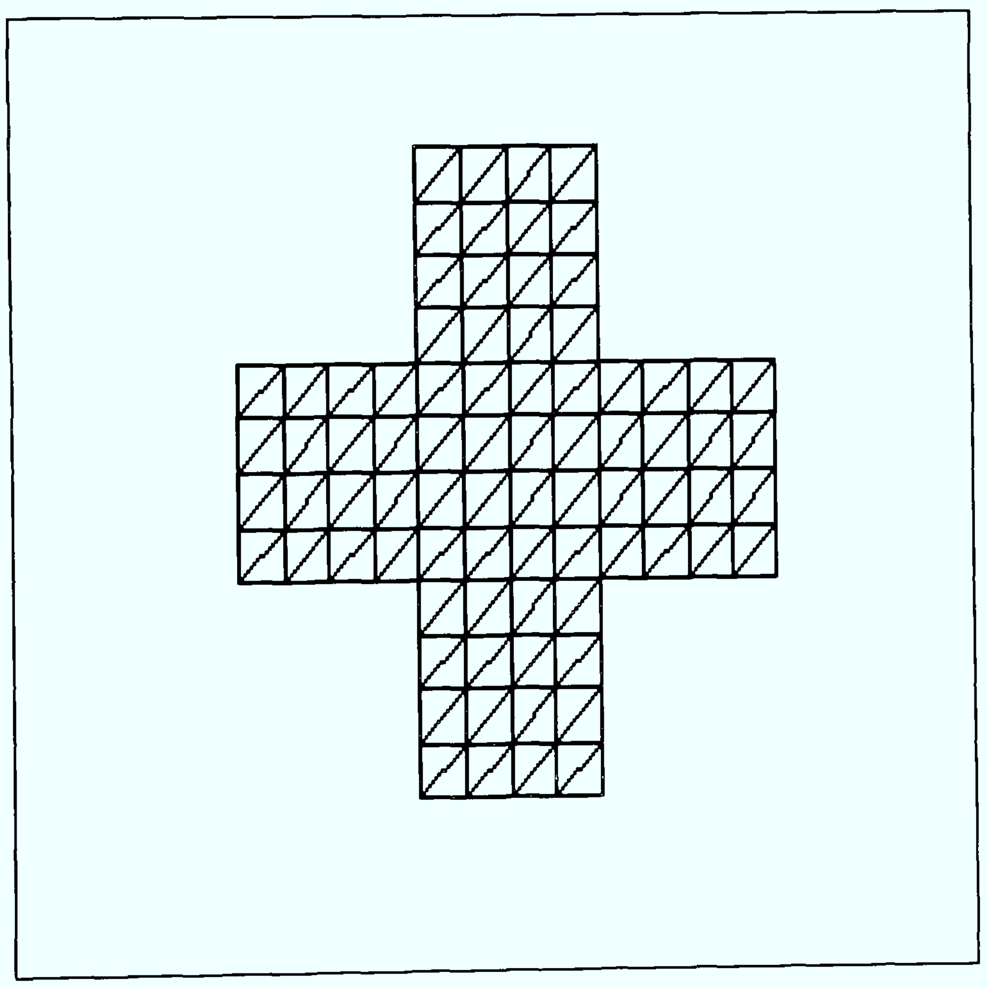


Figure 4.11(a): Original mesh.

Figure 4.11(a) shows a simple example of a geometry that has a key feature of internal flows (i.e. the external boundary is much more complex than in external flows). If we split this mesh into five using the iterative clustering technique, a total communication cost of 29 (see Table 4.4) is achieved. The sub-domains can be seen in Figure 4.11(c) together with a processor topology that would be required for nearest neighbour communications. Obviously, this is a very simple mesh to use as an example and the processor connections required is not too bad, but if larger meshes were partitioned using the iterative clustering method, then the processor topologies required would be much more complex. This would be a severe limitation since the cheaper multi-processor systems don't have such flexibility and are limited to the number of connections they can have.

Say, for example, we have a simple processor topology with a pipeline of 5 processors. If we partition the same mesh into five using the global cost method and taking this processor topology into account, then the sub-meshes obtained can be seen in Figure 4.11(b). Looking at this figure, we can see that all neighbouring elements are on the same or neighbouring processors. Table 4.3 gives a breakdown of the communication costs between every pair of processors and we can see that the total communication cost is now 34. Even though this cost is slightly higher than with the iterative clustering technique, it has used a simple processor topology.

If we now take the sub-domains obtained when using the iterative clustering technique and map them onto the chain of 5 processors, then by looking at Table 4.5, we see that the cost has now increased to 40. This is fairly high compared to the total cost achieved using the global cost method, and there is even a node being shared by processor 1 and 5, which means that two processors have to communicate via four other processors, and hence adds a cost of 4 to the total communication cost.

Pairs of Processors	Cost
1 and 2	5
2 and 3	12
3 and 4	10
4 and 5	7
Total	34

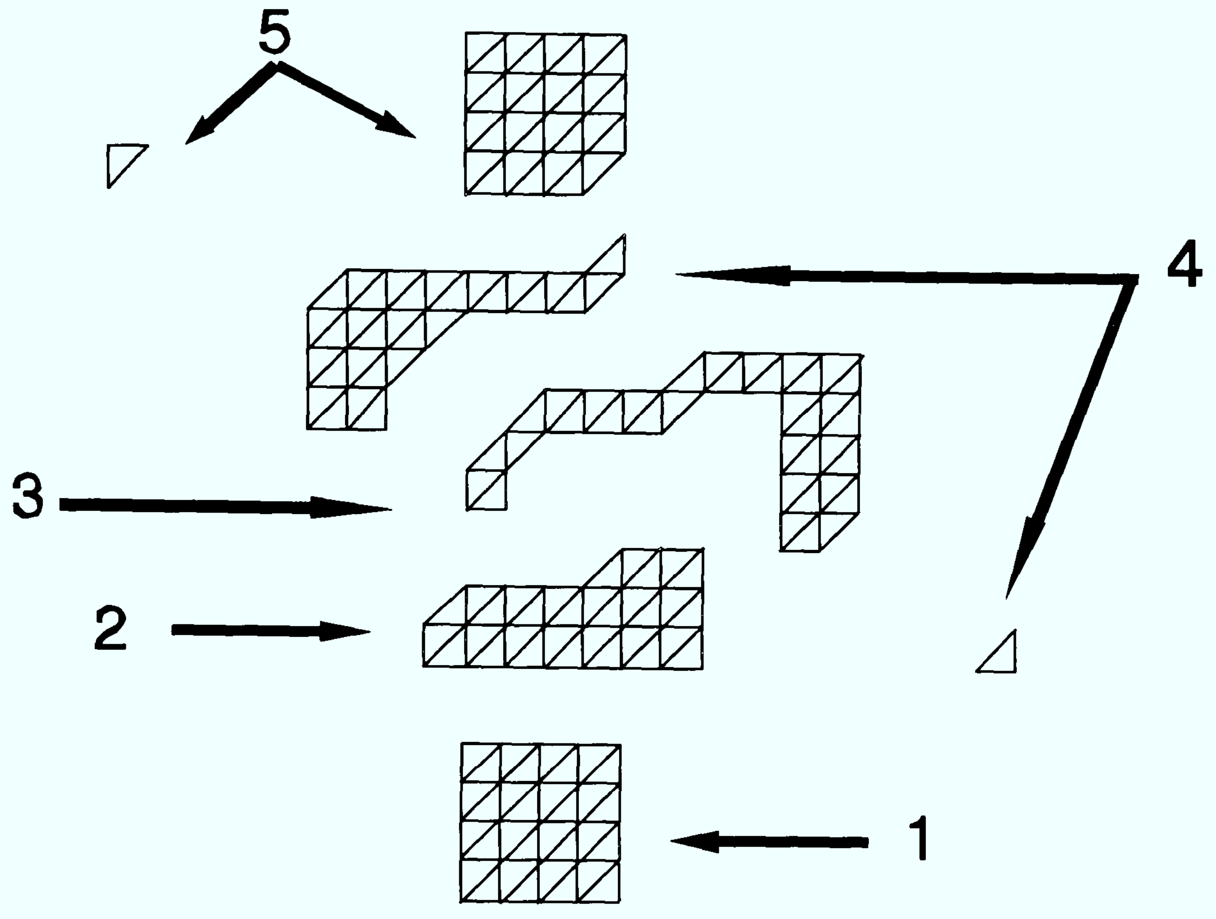
Table 4.3: Communication costs between processors when using global cost method.

Pairs of Processors	Cost
1 and 2	5
1 and 5	1
2 and 3	9
2 and 5	2
3 and 4	5
3 and 5	4
4 and 5	3
Total	29

Table 4.4: Communication costs between processors not taking processor topology into account.

Pairs of Processors	Shared nodes X Distance travelled = Cost
1 and 2	5 X 1 = 5
1 and 5	1 X 4 = 4
2 and 3	9 X 1 = 9
2 and 5	2 X 3 = 6
3 and 4	5 X 1 = 5
3 and 5	4 X 2 = 8
4 and 5	3 X 1 = 3
Total	40

Table 4.5: Costs achieved using the iterative clustering method and mapping onto a pipeline of processors. (Having not taken processor topology into account).



Processor Topology :

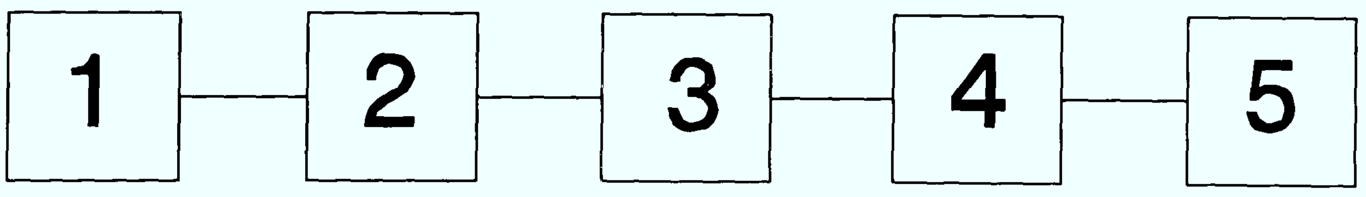
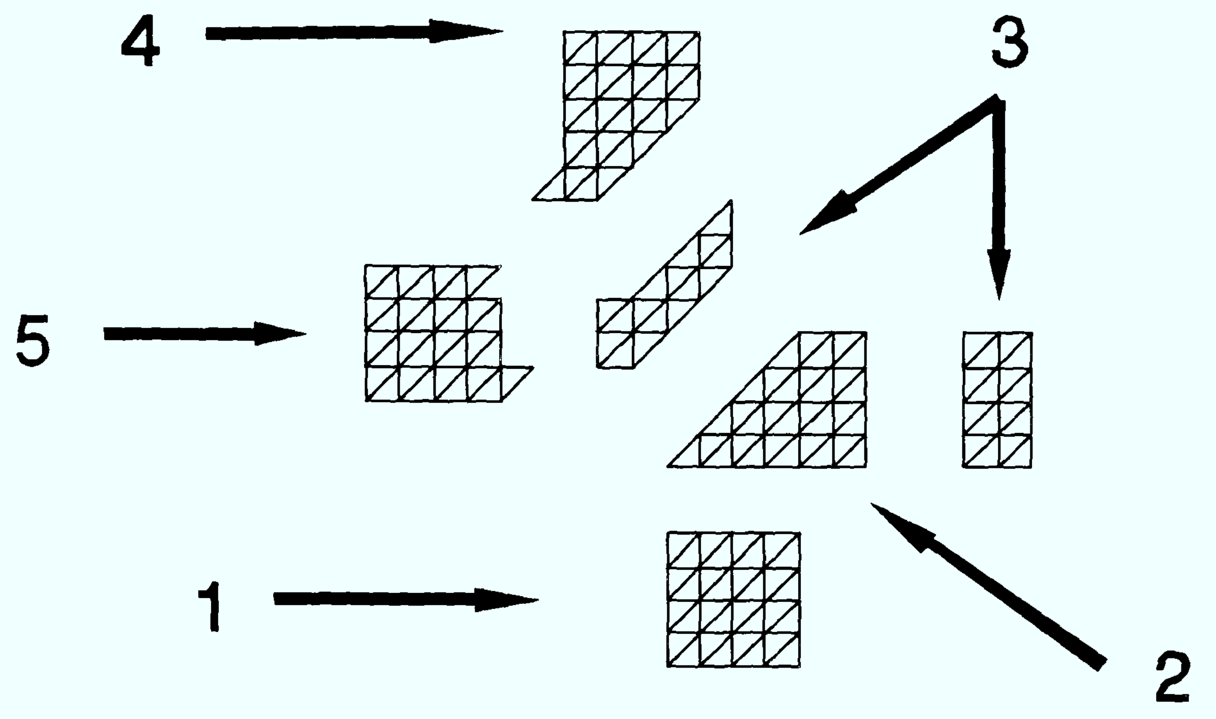


Figure 4.11(b): Mapping the cross-shaped mesh onto a chain of 5 processors.



Processor Topology :

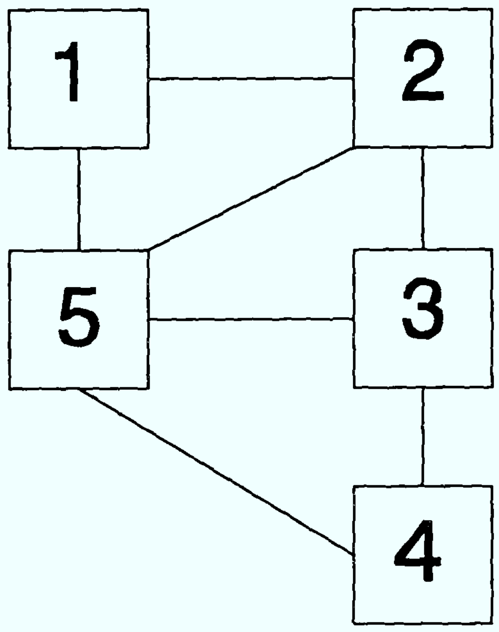


Figure 4.11(c): Partition of cross-shaped mesh into five not taking processor topology into account.

4.5 The Algorithm

In this chapter, we have shown how the iterative clustering method has been modified to cater for our needs, and have also shown how we have overcome some of the problems that we encountered. In this section, the algorithm used is described. We have been working with an unoptimised proto-type code which is not intended for practical use and was just a development tool. Figure 4.12 shows the flow chart of the routines called within this algorithm. An explanation of each routine called is given in sections 4.5.1 to 4.5.4.

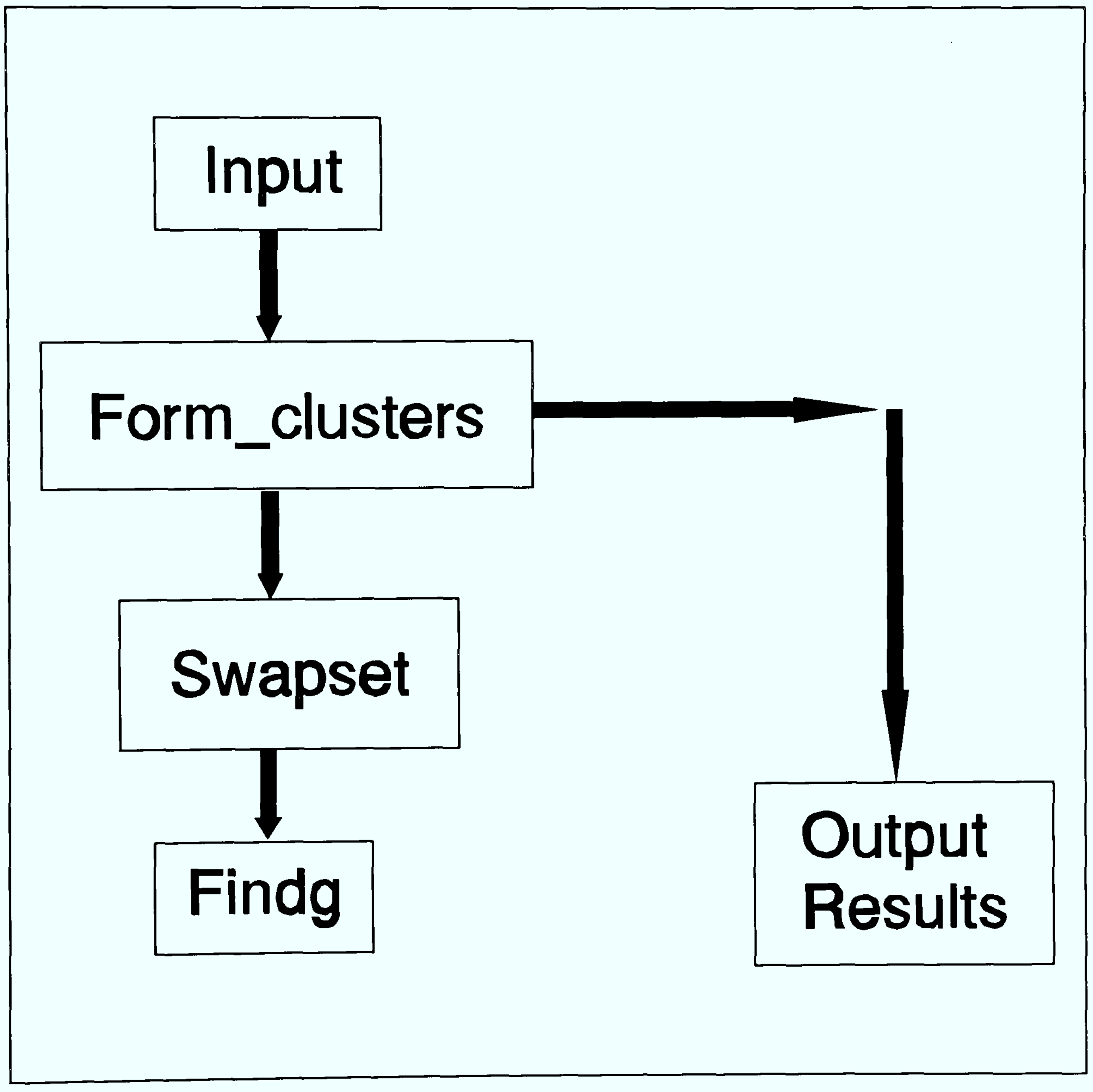


Figure 4.12: Flow chart showing routines called within the algorithm.

4.5.1 Routine "Input"

The purpose of this routine is to enter all the relevant data about the mesh.

Reads the data from <filename>.dat

<filename>.dat contains information about the nodes of each element. The first line gives the total number of elements and each line then on gives the node numbers for each element.
e.g. for the simple mesh shown in Figure 4.13, <filename>.dat would contain the following data:

```
4
1 2 4
2 4 5
2 3 5
5 6 3
```

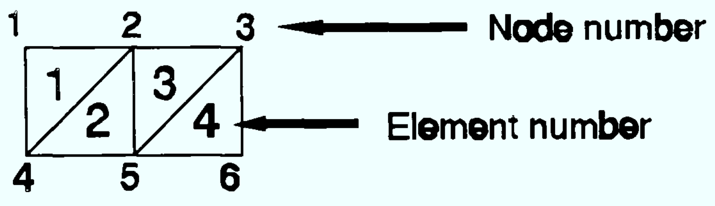


Figure 4.13: Simple 4 element mesh.

The first line tells us that there are 4 elements in the mesh. The second line corresponds to element number 1, so we can see that the node numbers for this element is 1 2 4. The third line corresponds to element number 2 and so on.

The nodal data is put into a one dimensional array *mesh*, where *maxnode* (the maximum number of nodes per element) columns are assigned to each element. The number of nodes per element should not be equal to or greater than maxnode, and the rest of the columns are filled with zeros. For the simple mesh given in Figure 4.13, the array *mesh* would hold the following data (where *maxnode*=5):

[1,2,4,0,0,2,4,5,0,0,2,3,5,0,0,5,6,3,0,0......]

Columns 1 to 3 represent the nodes of element number 1, columns 6 to 8 represent the nodes of element number 2 and so on.

4.5.2 Routine "Form_Clusters"

In this routine the number of processors is entered and the value is assigned to the variable *ntransp*.

It then allocates sets of elements to processors and the data is held in records *sett[i].elemts[j]*, where *i* is the processor number and *j* is the *jth* element in processor *i*. For example using the mesh in Figure 4.13 again,

sett[1].elemts[1]=1;

sett[1].elemts[2]=2;

sett[1].elemts[3]=3;

sett[2].elemts[1]=4;

sett[2].elemts[2]=5;

sett[1].elemts[3]=6;

... and so on

Loop over processors for all pairs of processors and count the node uses for all processors and store in the record *inter[k].incl[j]*, where *k* is the processor number and *j* is the node number. For example, if elements numbers 1 and 4 from the mesh in Figure 4.13 were assigned to processor 1 and elements numbers 2 and 3 were assigned to processor 2 then

inter[1].incl[1]=1;	Node number 1 is used once in processor 1
inter[1].incl[2]=1;	
inter[1].incl[3]=1;	
...	
inter[2].incl[1]=0;	Node number 1 is not used in processor 1
inter[2].incl[2]=2;	
inter[2].incl[3]=1;	
...	

and so on.

4.5.3 Routine "Swapset"

This routine finds the best pairs of elements that can be swapped in order to reduce the communication costs between a pair of processors.

Reads the shortest distance matrix which gives the shortest path between processors and this is held in the variable *d[i,j]*, where *i* and *j* are processor numbers. For example if we have a processor topology as shown in Figure 4.14, then the shortest path matrix is :

$$\begin{bmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

where column *i* and row *j* correspond to the shortest path between processor *i* and processor *j*. With the example given, the shortest path between processor 1 and processor 4 is 2.

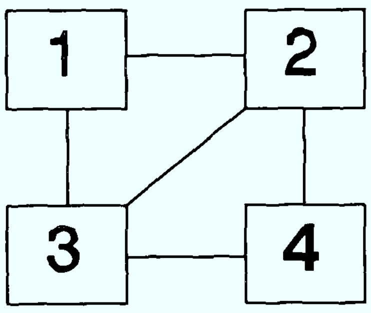


Figure 4.14: A simple processor topology.

Loop with every pair of processors and for each loop do:

Find all possible swaps in order of cost reduction by looping *n/ntransp* (the number of elements in each partition) times:

Calculate *glocost* which is the total communication cost. This is calculated

by looping over all pairs of processors, checking to see if a particular node is shared.If it is shared between processor *achip* and processor *bchip*, find the shortest path by looking at *d[achip,bchip]*, square this value and add it onto *glocost.*
Call routine "findg", which finds the pair of elements that minimise the global cost the most when swapped. (see section 4.5.4)

These two elements, say *a* and *b* are passed back to swapset. The variable *maxG* which is the reduction in cost by swapping these two elements, is also passed back.

The node usage of these elements are changed. For example, the nodes of element *a* are removed from the node usages of processor *achip* and added onto the node usage of processor *bchip*. The same is done for element *b*.

The elements are now swapped by finding the end of the list of elements for processor *achip*. The chosen element *a* is moved from the list and the list is shortened. The other chosen element *b* is also removed from the list of elements of processor *bchip* and the list is shortened. For example,

Before moving elements :

Processor *achip*	[1,2,3,4,0,0,...]
Processor *bchip*	[5,6,7,8,0,0,...]

If elements number 3 is chosen from processor *achip* and element number 5 is chosen from processor *bchip* then the lists will now look like this :

Processor *achip*	[1,2,4,0,5,0,...]
Processor *bchip*	[6,7,8,0,3,0,...]

We now calculate *sumG* which is the summation of all *maxG* so far.
If this summation is the first or the maximum so far, then *bigG* = *sumG*

and *kbest* becomes the number of elements that have been considered for swapping so far.

We now swap the first *kbest* pairs of chosen elements and returning all others to their original partition sets.

Pointers are calculated for processors *achip* and *bchip* to check to see where the *kbest* pairs of elements are in the list. The elements that were last found, which are the unrequired swapped elements are returned to their original processor lists. The node usages are adjusted as the elements are replaced. The swapped elements which we want to remain swapped are now connected fully to the other elements in the list.

For example, if our original lists contained eight elements each and originally looked liked this

Processor *achip*	[1,2,3,4,5,6,7,8,0]
Processor *bchip*	[9,10,11,12,13,14,15,16,0]

and after finding eight pairs of elements to swap the lists now look like :

Processor *achip*	[0,12,11,16,9,14,10,13,15]
Processor *bchip*	[0,1,4,6,3,5,2,7,8]

The pointers are calculated to find the position in the list where elements are to remain swapped. In this case, since *kbest* = 2, the pointer will be at position 7, so all elements from position 1 to 6 are to be returned to their original lists. This is done by moving the element in position 2 of the first list is moved to position 1 of the other list and so on. With the above example, the lists now look like

Processor *achip*	[1,4,6,3,5,2,10,13,15]
Processor *bchip*	[12,11,16,9,14,10,2,7,8]

As the elements are swapped, the node usages are also adjusted.
Note that elements numbers 2 and 10 are contained in both lists, but this is taken care of when the elements that are to remain swapped are connected to the list by being moved up. The lists now look like :

Processor *achip*	[1,4,6,3,5,2,15,13]
Processor *bchip*	[12,11,16,9,14,10,8,8]

If on the other hand, *bigG* is negative, i.e. there is no reduction in cost when pairs of elements are swapped, then all the elements that have been swapped are returned to their original lists.

4.5.4 Routine "Findg"

This routine finds a pair of elements that reduces the overall communication costs the most.

Loop over elements in partition *achip* and for each loop do:

Loop over elements in partition *bchip* and for each loop do:

Reduce the global cost assuming that element *achip[p]* is removed.
Reduce the global cost assuming that element *bchip[r]* is also removed.

Assuming element *achip[p]* is added to list *bchip*, the global cost is increased, looking at each node of the element. The same is done assuming element *bchip[r]* is added onto list *achip*.

The minimum global cost found is saved as *minglob* together with the relevant elements.

MaxG which is the reduction in cost so far is also calculated.

4.6 Test Cases

In this section, we will show some examples of some simple meshes decomposed with the iterative clustering method.

Figure 4.15(a) was partitioned into four onto a chain of four processors and the resulting split can be seen in Figure 4.15(b). We can see that the algorithm has succeeded to map neighbouring elements onto the same or neighbouring processors. None of the elements need to communicate via more than one processor.

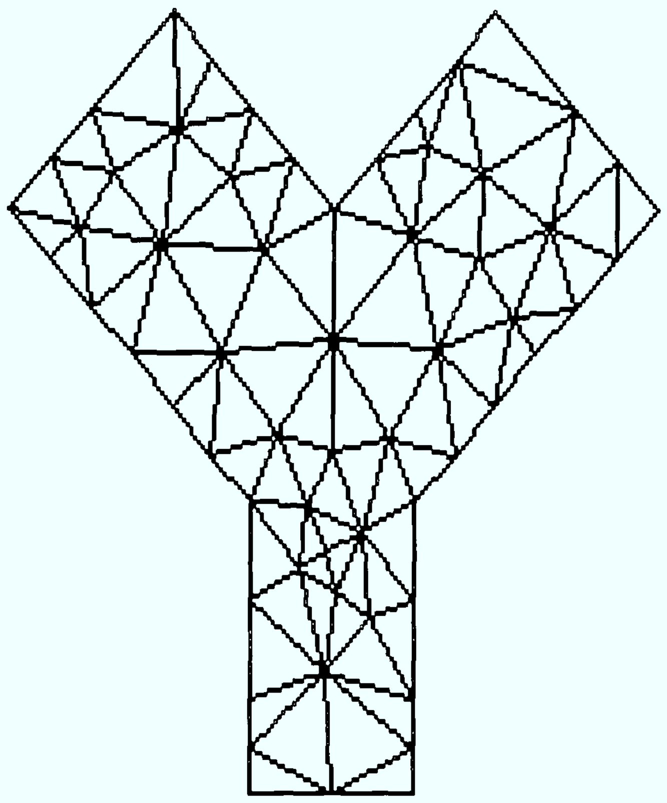


Figure 4.15(a): Y-shaped mesh.

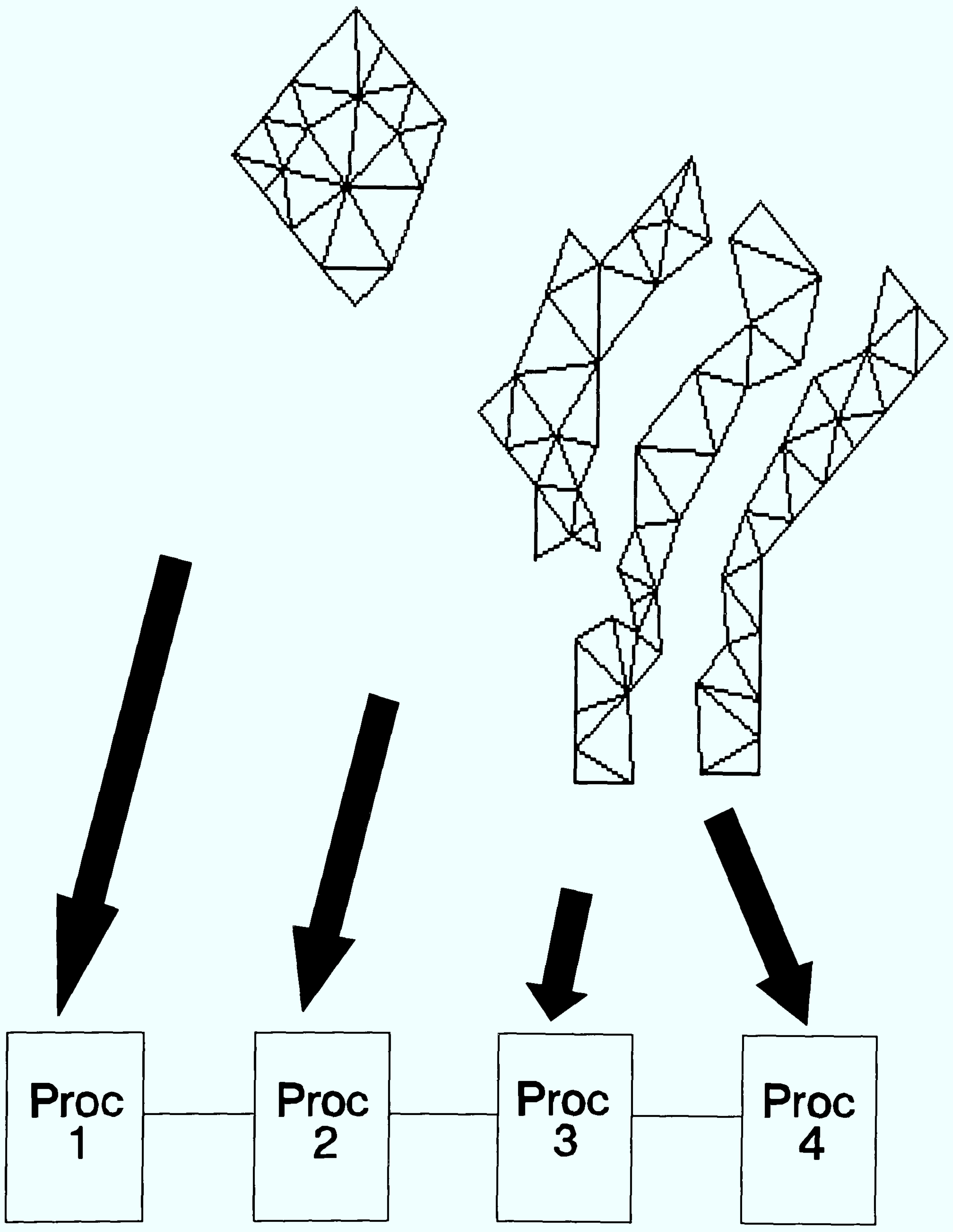


Figure 4.15(b): Y-shaped mesh mapped onto a chain of four processors.

So far, only examples of meshes mapped onto a chain of processors have been given. We now give an example of a mesh split into five with the processor topology specified in Figure 4.16(a).

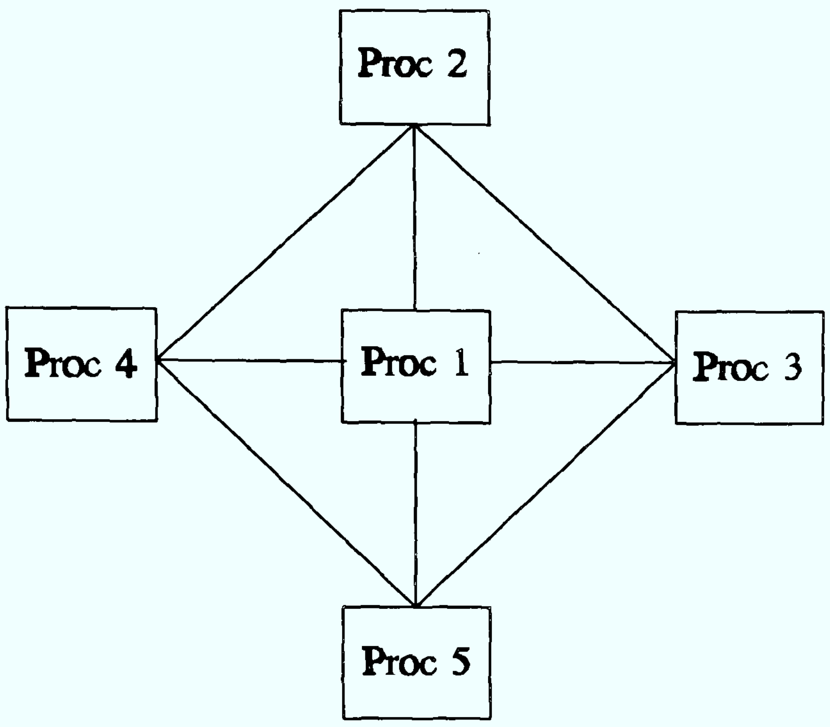


Figure 4.16(a): Processor topology.

The mesh (which is illustrated in Figure 4.16(b)) was split into five with the above processor topology in mind. The sub-meshes obtained can be seen in Figure 4.16(c). From the processor topology above, we can see that there are no connections between processors 2 and 5, and processors 4 and 3. Looking at the sub-meshes obtained, there are no communications between sub-meshes allocated to processors 2 and 5, and processors 3 and 4. Again, we see that only neighbouring elements are mapped onto the same or neighbouring processors.

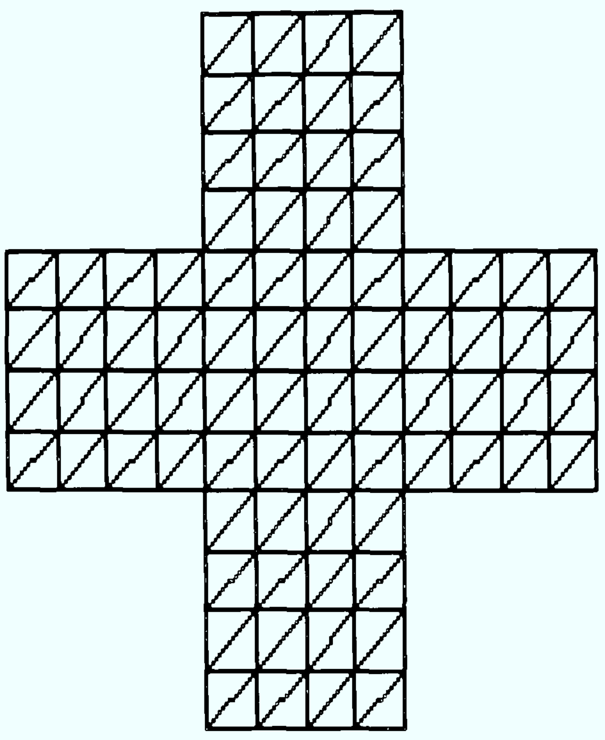


Figure 4.16(b): Cross shaped mesh.

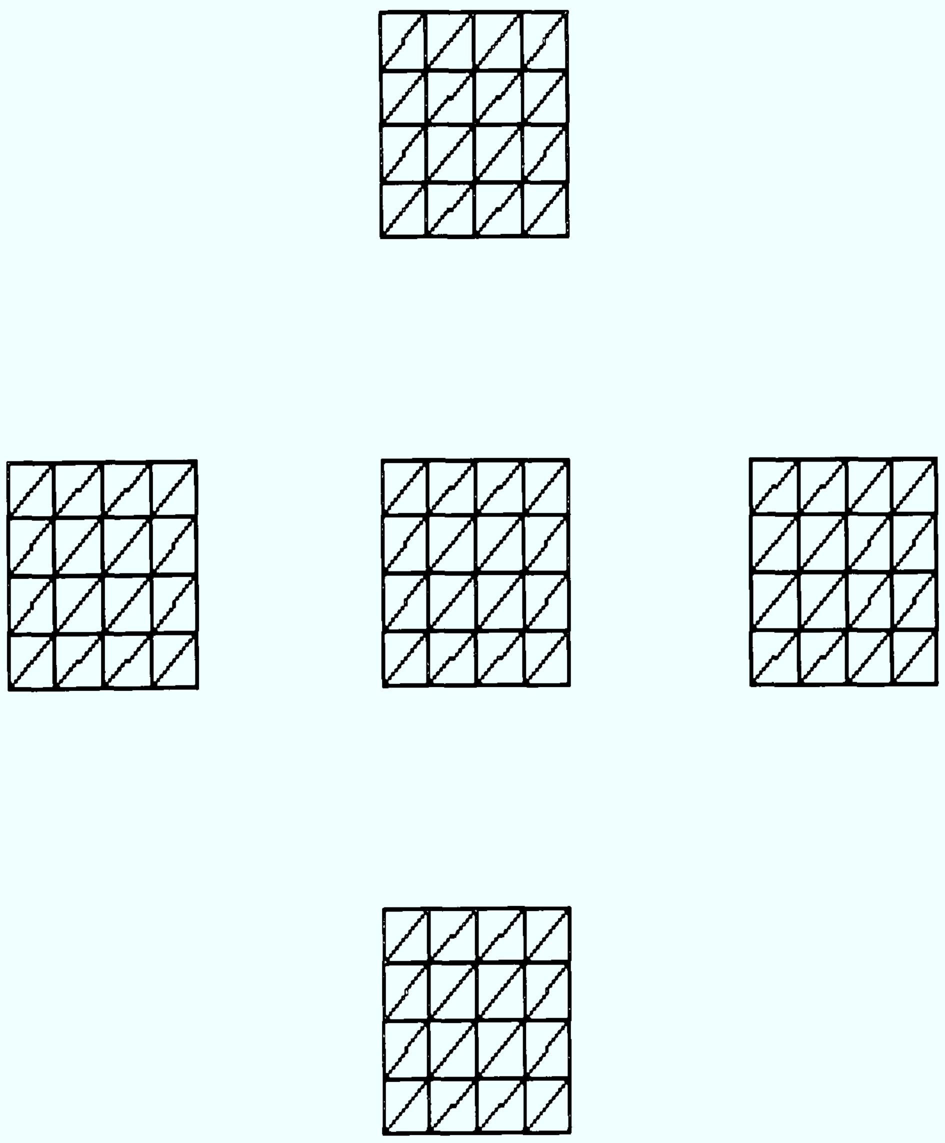


Figure 4.16(c): Cross shaped mesh split into five and mapped onto processor toplogy shown in Figure 4.16(a).

4.7 Larger Meshes

So far in this chapter we have shown that the algorithm can work successfully but most of the examples used were very simple. In this section the algorithm is applied to real life problems.

The mesh shown in Figure 4.17 has 516 elements which is used to simulate the resulting convection currents due to a moving lid as shown in Figure 4.18.

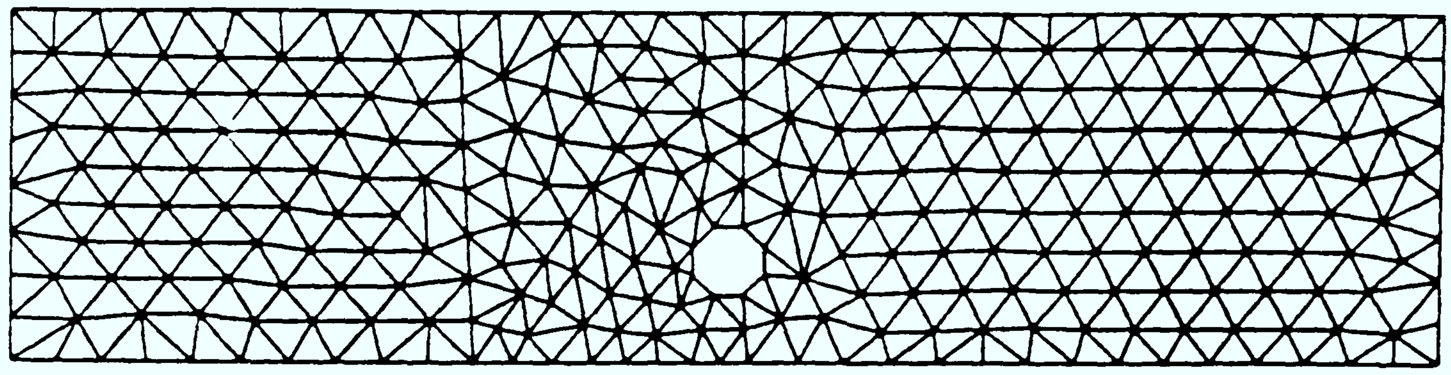


Figure 4.17: Mesh with 516 elements.

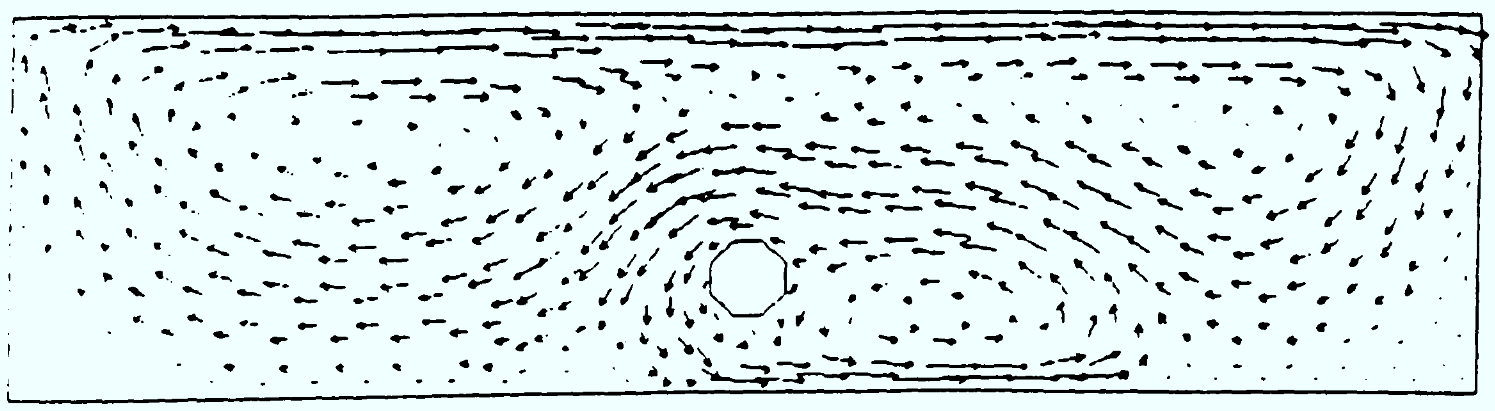


Figure 4.18: Flow of vectors due to moving lid.

The mesh shown in Figure 4.17 was partitioned into 3, 4, 5 and 6 sub-meshes, all being mapped onto a chain of processors. The results can be seen in Figure 4.19 and it is clear that by using this modified cost function, a reliable decomposition is generated where all neighbouring elements are on the same or neighbouring processors.

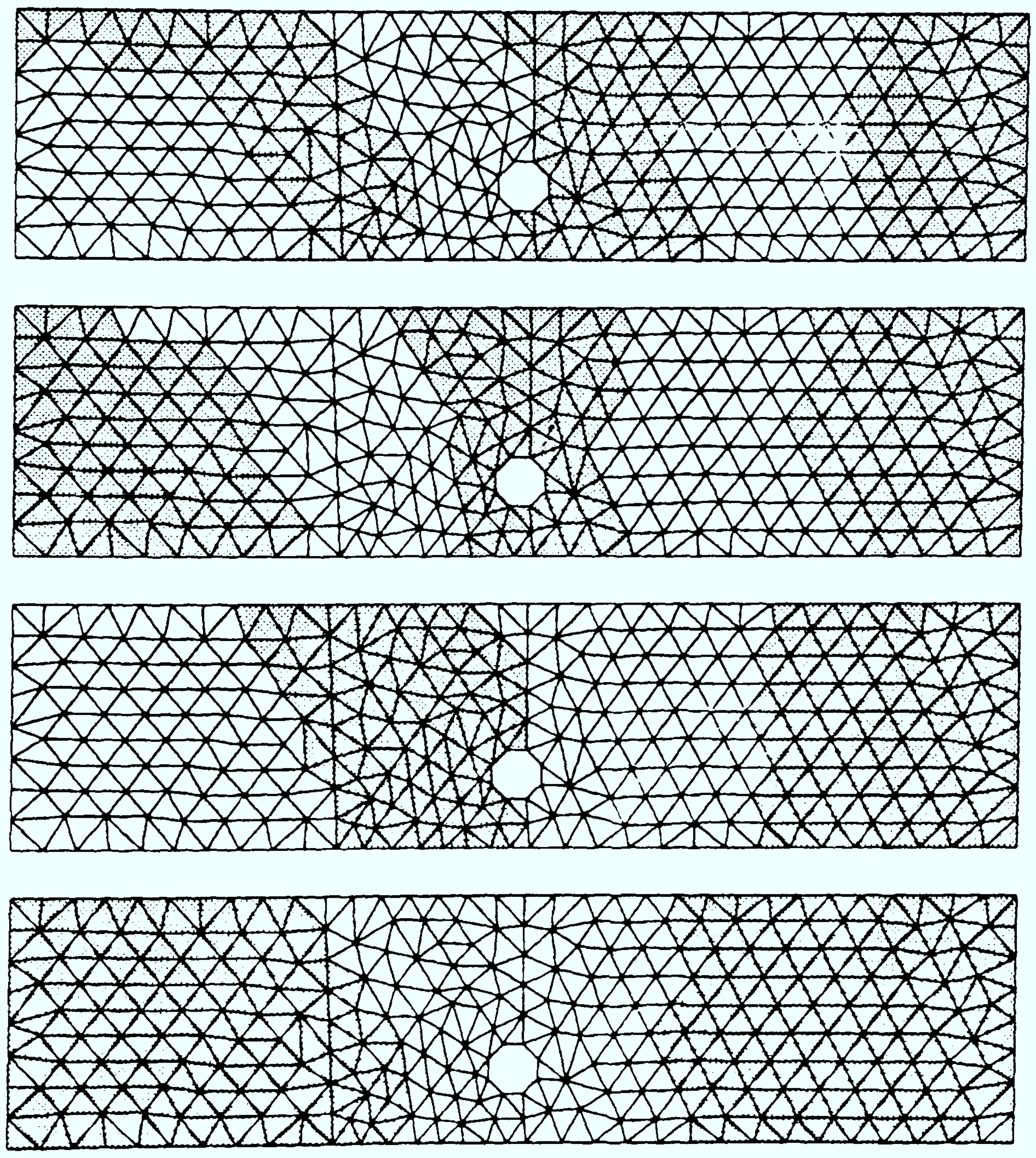


Figure 4.19: Mesh partitions for 516 element mesh.

A mesh containing 3034 elements as shown in Figure 4.20(a) was partitioned into 5 taking into account a processor topology of a chain of 5 processors. The decomposition achieved can be seen in Figure 4.20(b) and again, we can see that this decomposition is a reliable one where all neighbouring elements are on the same or neighbouring processors.

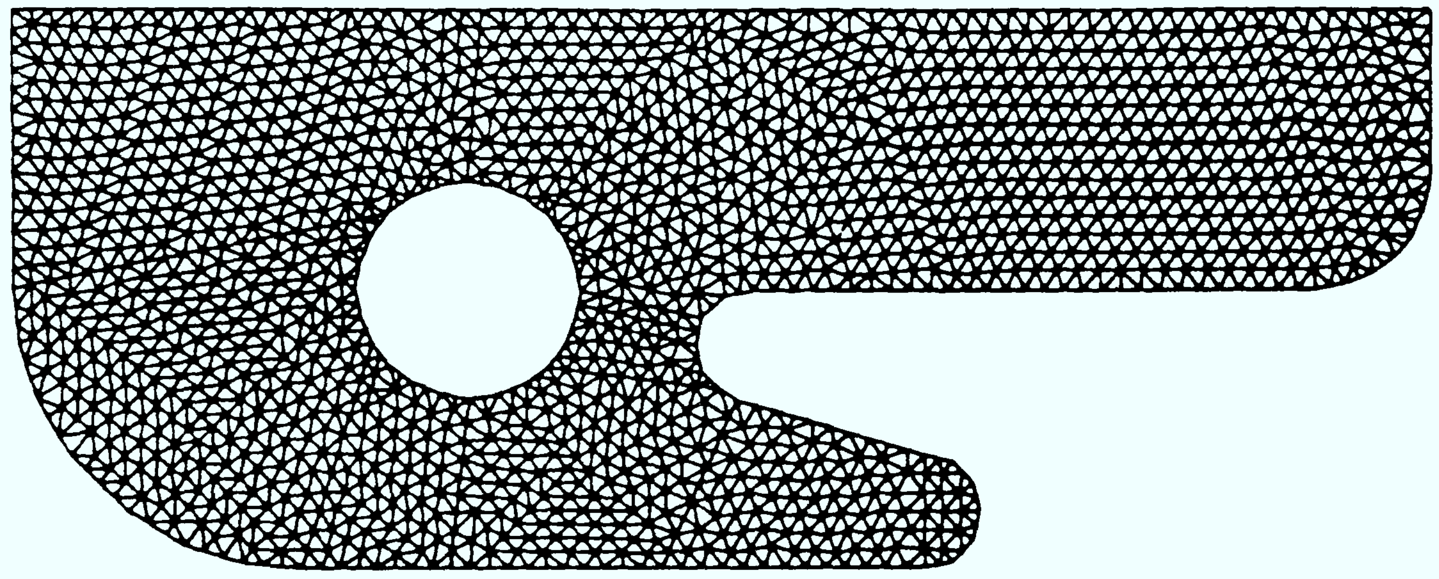


Figure 4.20(a): Mesh containing 3034 elements.

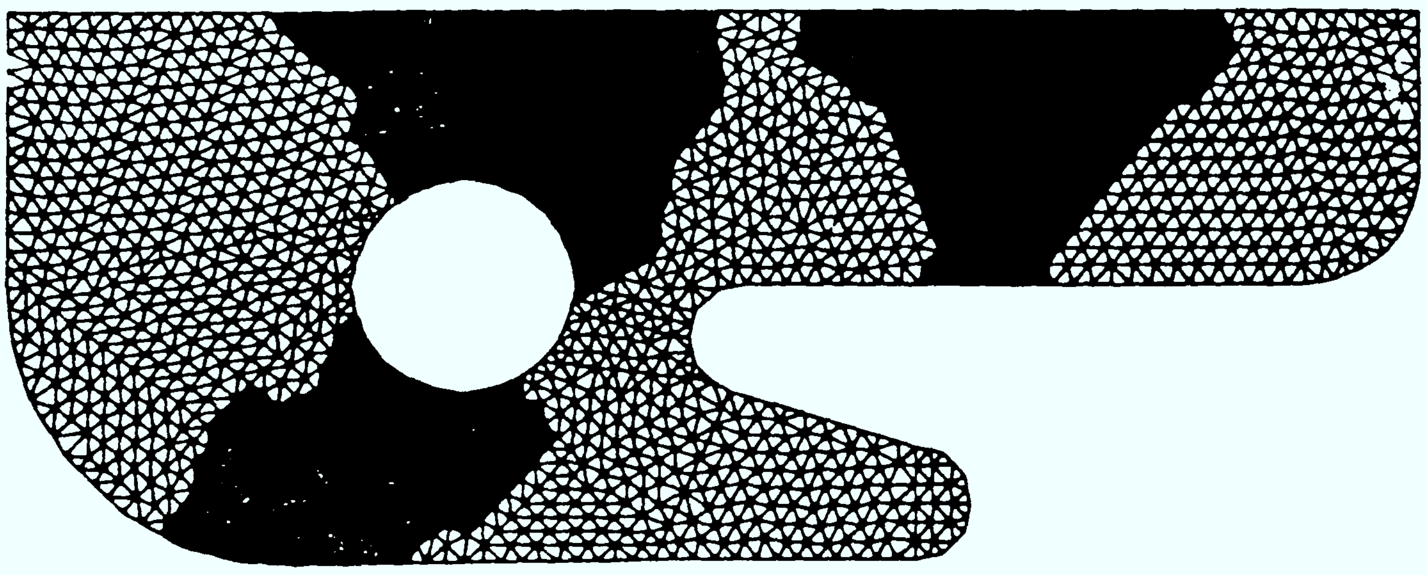


Figure 4.20(b): Decomposition of 3034 element mesh into five.

Another example which is shown in Figure 4.21 is the graph of Great Britain with over 5000 elements. This mesh was split into four taking into account a processor topology of a chain of four processors. We can see that neighbouring elements are placed on the same or neighbouring processors. Again, we can see that we have a disconnected sub-domain (red area). It seems that the cost along the boundaries of the red and green areas are lower than a cost along the boundary if the red area was not disconnected.

However, despite obtaining a reliable decomposition, the time taken to decompose the mesh was approximately 15 minutes. It is vitally important that the time taken to decompose a mesh is a small fraction of the overall solution time. Although the time taken to decompose the 3034 element mesh is not very large, it would increase greatly if we were dealing with a mesh containing millions of elements.
One way to overcome this problem is to reduce the number of elements in the mesh. This can be done by grouping together clusters of elements to create what we call 'super-elements' and then applying the decomposition algorithm onto this mesh of super-elements.
Chapter 5 describes how these super-elements are created and some of our results will be demonstrated.

Chapter 5
Dealing with Large Meshes

5.1 Introduction

As we have seen in Chapter 4, the clustering method is not suitable for dealing with large scale meshes since the calculations often require prodigious amounts of computer power. It is important that the time taken to decompose these large meshes must be a small proportion of the overall solution time.

This problem can be overcome by creating clusters of the original elements and using these to create a reduced network which is homomorphic to the original mesh. These clusters will be known as 'super-elements'. By creating these super-elements, we are reducing the number of elements in the mesh and hence reducing the computational time to decompose the mesh. For example, the mesh in Figure 5.1(a) has 516 elements, but after we have clustered the elements to create the super-elements, the mesh now has 64 super-elements and the mesh can be seen in Figure 5.1(b).

In this chapter, we describe how these super-elements are created and what impact it has on the time taken and the decomposition achieved.

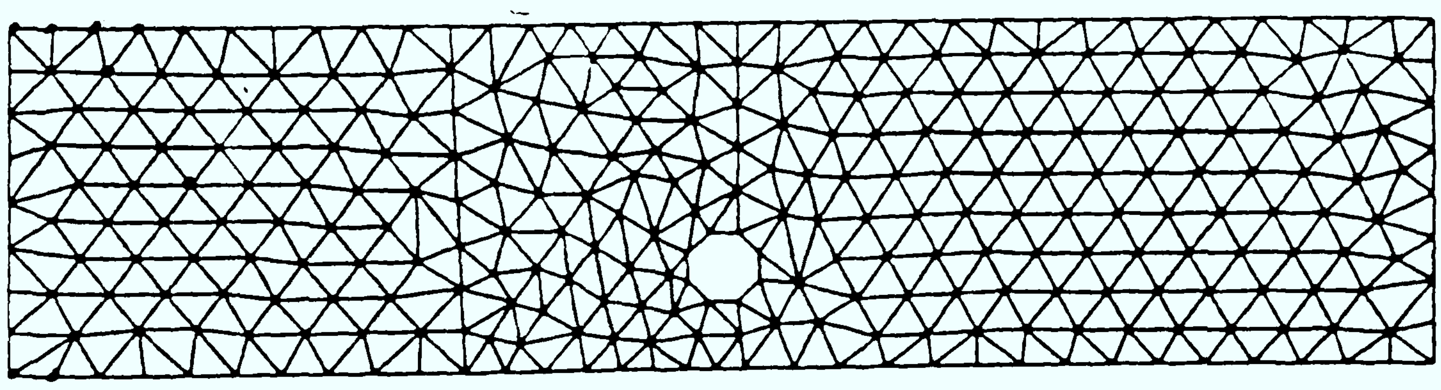


Figure 5.1(a): Original mesh with 516 elements

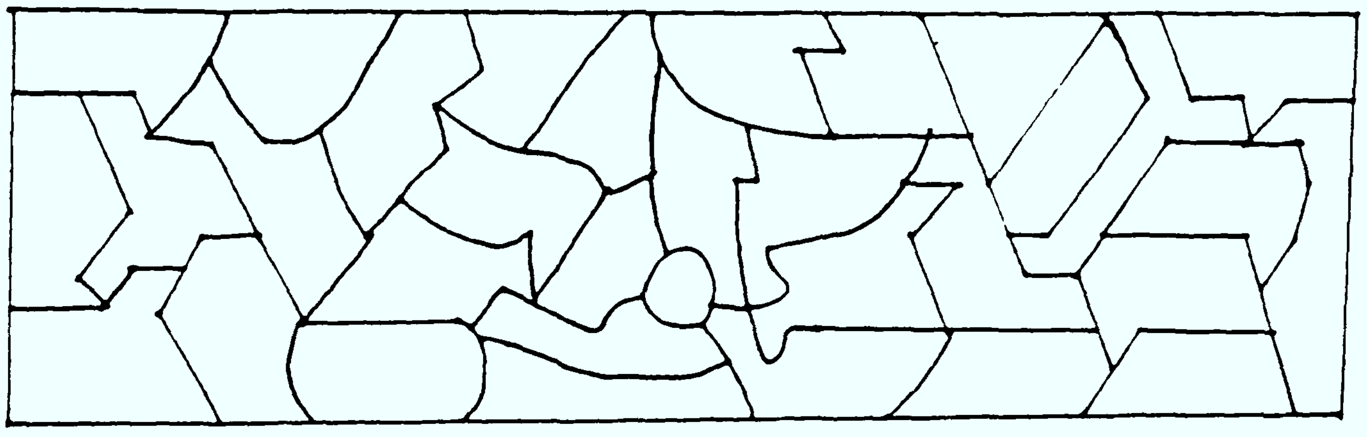


Figure 5.1(b): The same mesh with 64 super-elements

5.2 Creating Super-Elements

Since the super-elements are created in order to speed up the decomposition, it is vitally important that the time taken to create these super-elements is very quick. A method that we have used to create the super-elements is the graph bisection method which is described in section 5.2.1. Obviously, a number of alternative methods can be used to create the super-elements. Ideally, they should be connected and compact i.e. have small diameters but this is not essential.

5.2.1 Recursive Graph Bisection

The recursive graph bisection method works as follows:

1. Find two elements (from a set of n elements) which are a maximal or near maximal distance away.

2. Assign one of these elements to a set.

3. Find an elements which shares an edge with this element and assign these to the same set.

4. Repeat the procedure by finding elements adjacent to the newly assigned elements.

5. Stop when the set contains n/2 elements. The other unassigned elements are assigned to the other set.

This algorithm is repeated recursively so as to achieve the required number of super-elements. By using the above algorithm, we can illustrate how the super-elements are created. Figure 5.1 shows a mesh containing 160 elements and we shall create 32 super-elements each containing 5 of the original elements.

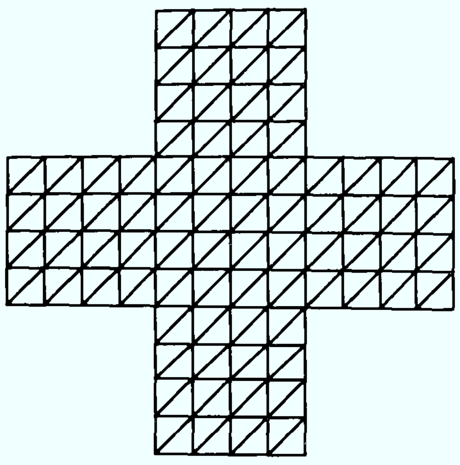


Figure 5.1: Original mesh

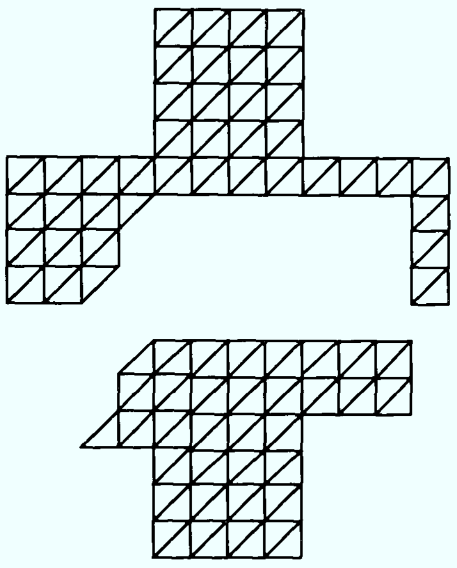


Figure 5.2(a): 2 super-elements

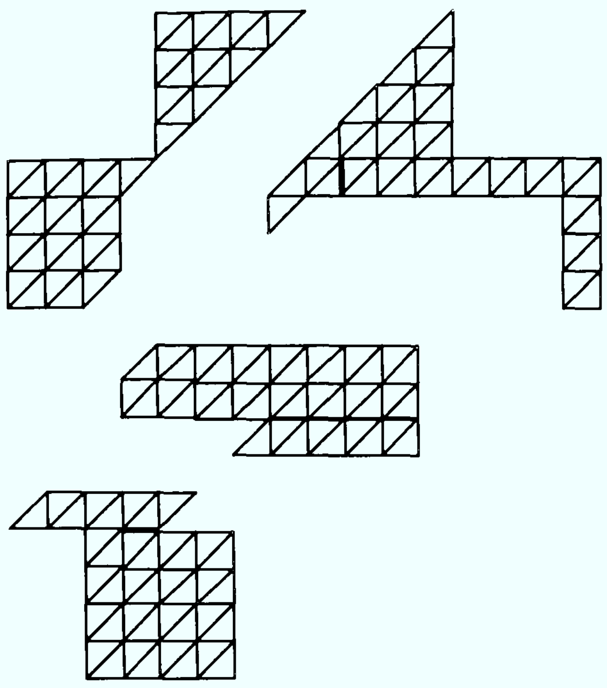


Figure 5.2(b): 4 super-elements

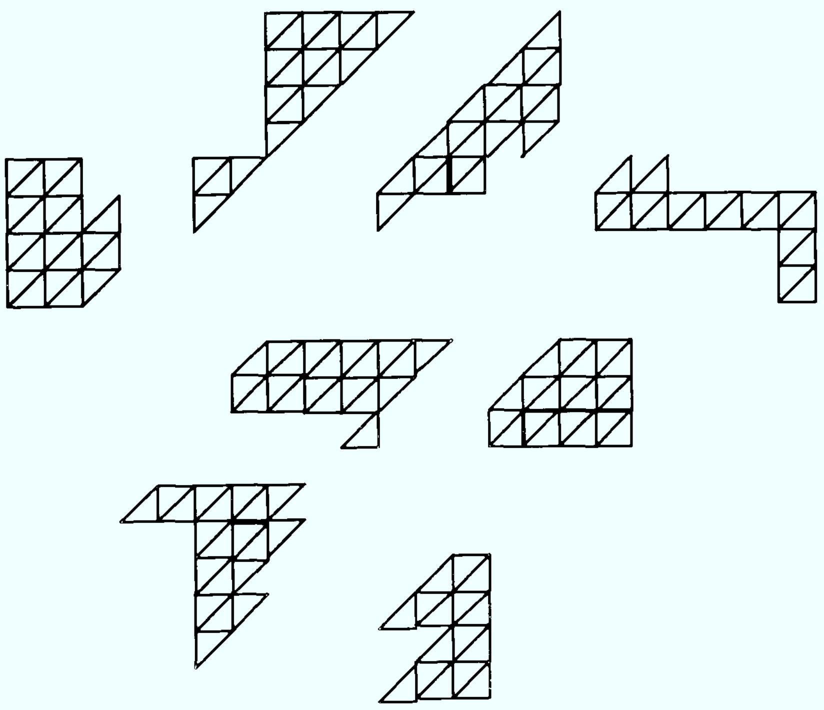


Figure 5.2(c): 8 super-elements

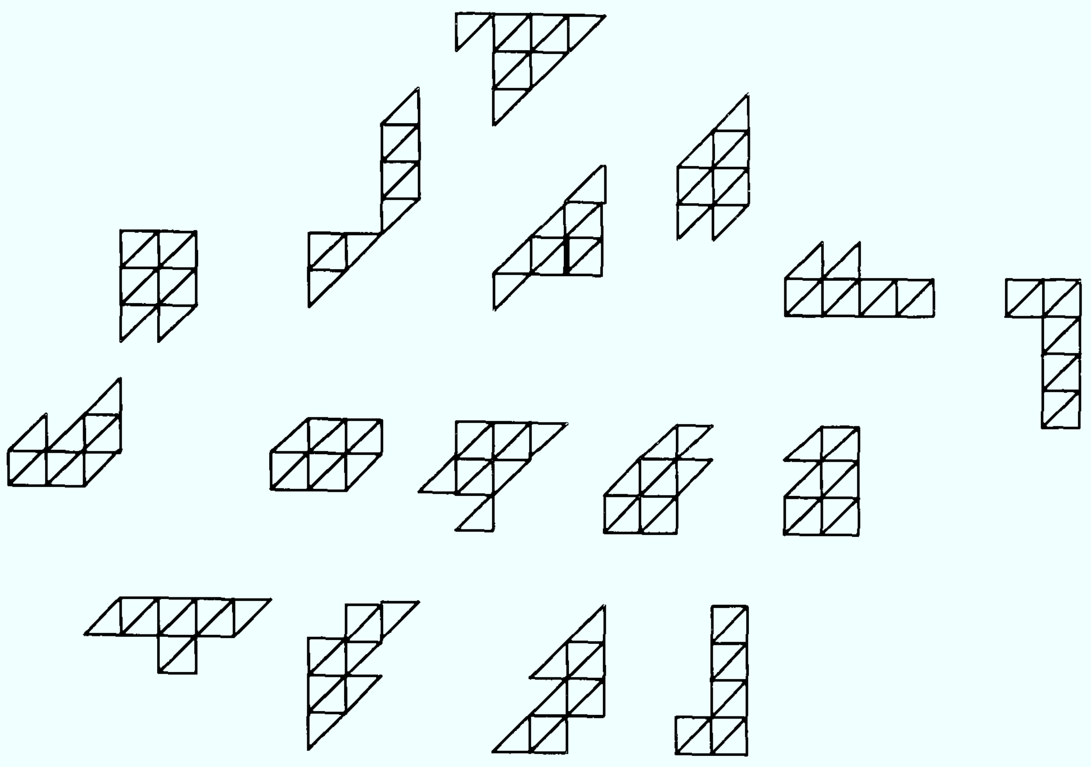


Figure 5.2(d): 16 super-elements

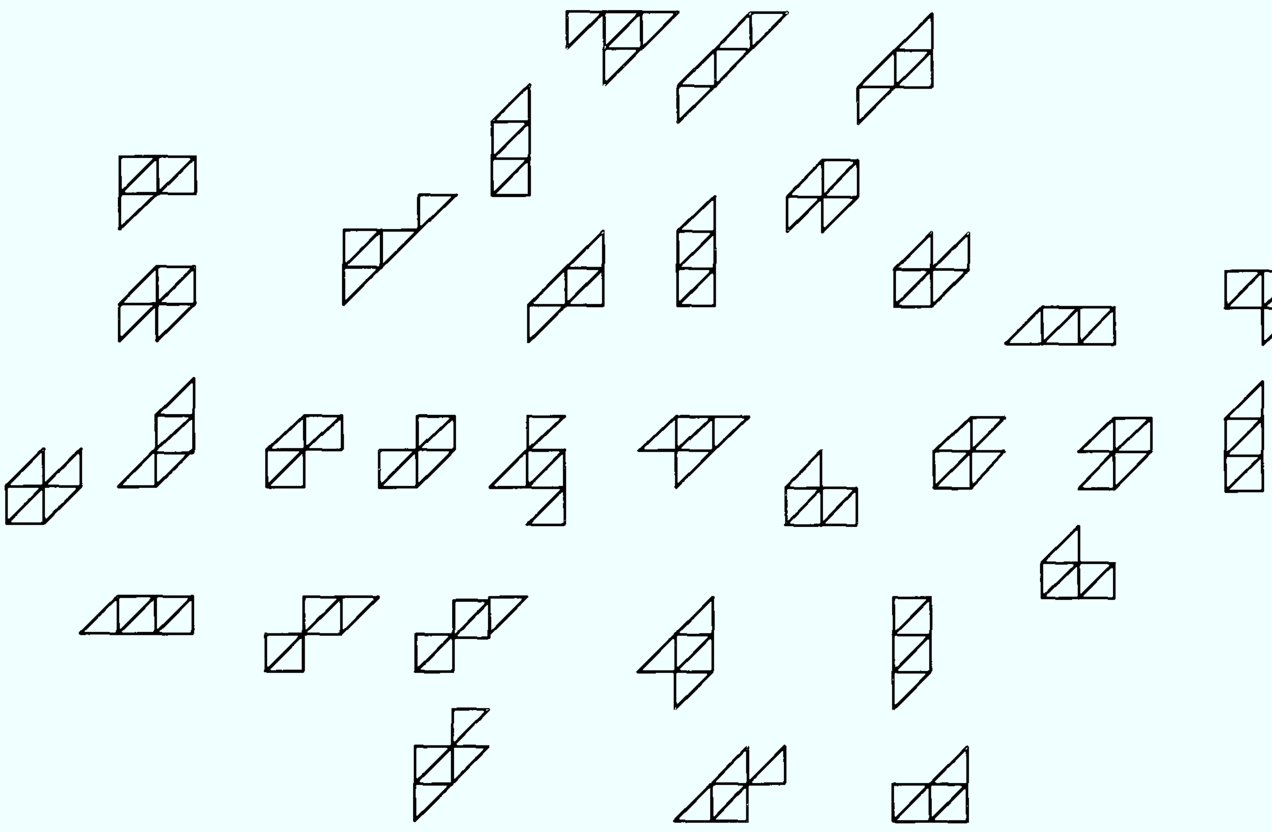


Figure 5.2(e): 32 super-elements

Figure 5.2(f) shows the mesh now with only 32 super elements.

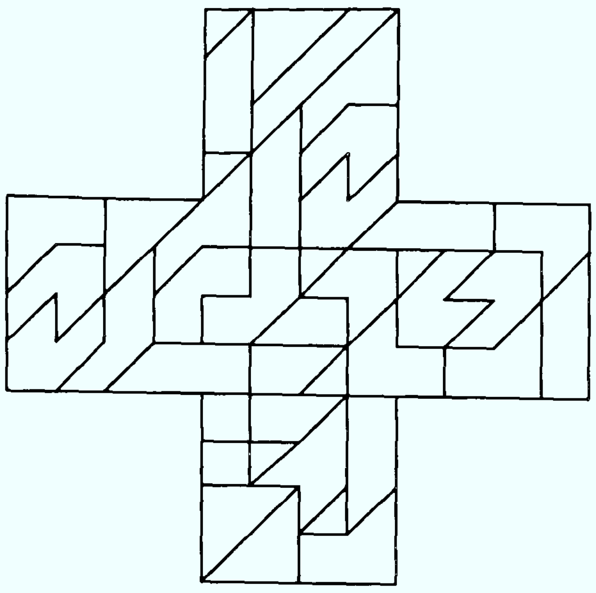


Figure 5.2(f): Mesh shown with 32 super-elements

5.2.2 Image Network

We now need to form an image network which will hold the details of the cost between the super-elements. Obviously, the cost between each super-element will be greater than one since we need to take into account the shared nodes of the original elements since these are the nodes that will eventually be used.

To illustrate the image network, we shall use a simpler mesh than the cross given above. We only have to consider the nodes on the boundaries of the super-elements.

Take the mesh shown in Figure 5.3. The same mesh can be seen with its super-elements in Figure 5.4 with the original nodes on the boundaries clearly visible.

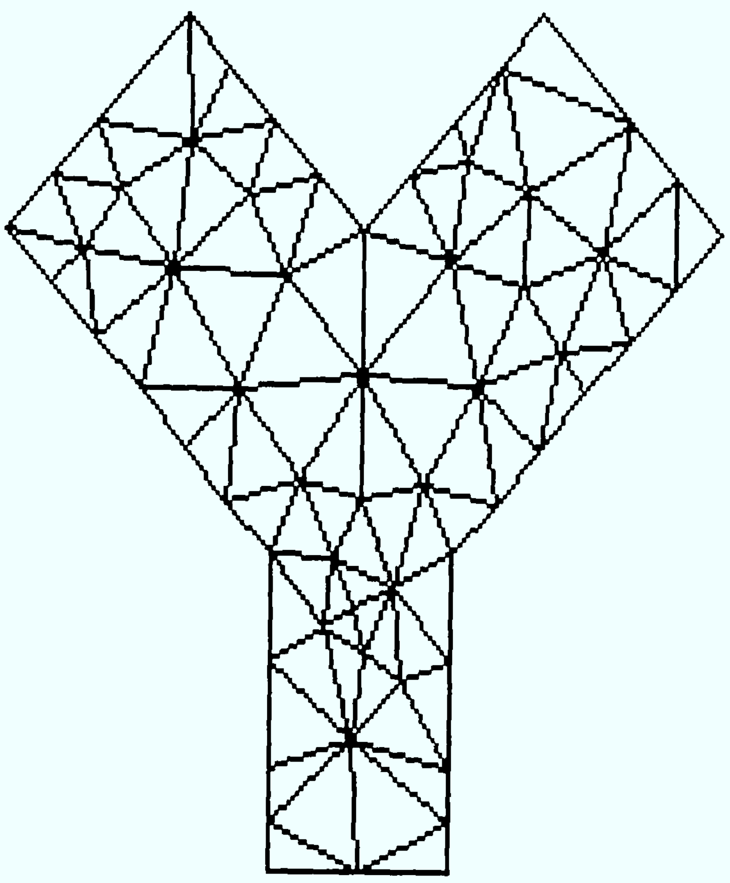


Figure 5.3: Y shape with original elements

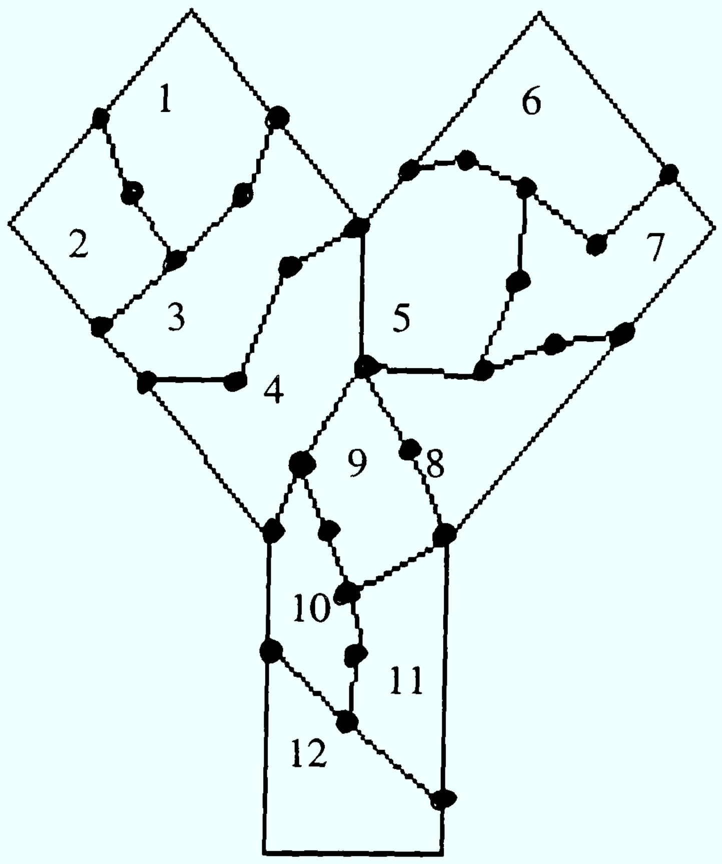


Figure 5.4: Y shape with super-elements shown.

The nodes of the original elements can be seen in Figure 5.4. Note that only nodes that are on the boundaries of the super-elements are shown. There is no need to use the internal nodes since they are not included in the cost function. We are only attempting to minimise the communication between the boundaries of the super-elements for the time being.

From Figure 5.4, we can see that super-elements 1 and 2 share 3 nodes, hence there is a communication cost of 3 between these two. Figure 5.5 shows the image network of the mesh and each node corresponds to the cost between the two super-elements. Hence node 1 corresponds to super-element 1, node 2 corresponds to super-element 2 and the edge between the two which has a weight of 3 corresponds to a communication cost of 3 between the two super-elements.

Figure 5.5 shows the complete image network for the mesh shown in Figure 5.4.

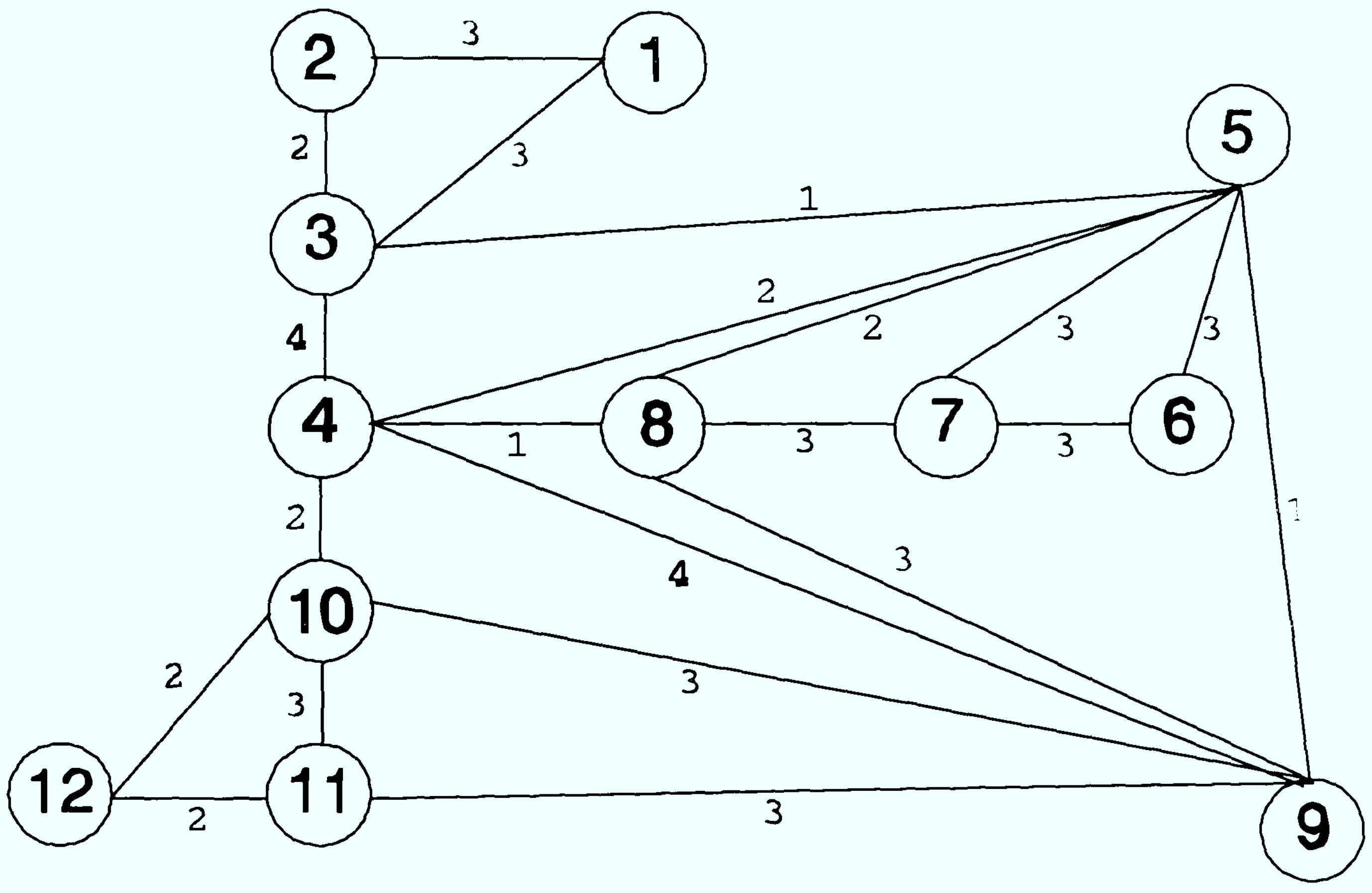


Figure 5.5: Image network for mesh shown in Figure 5.4

The bisection algorithm is then applied to the mesh and the cost between super-elements can be found by referring to the image network. Using the example of the cross shown in Figure 5.1(a) with 32 super-elements (illustrated in Figure 5.1(f)), the algorithm is applied taking into account a processor topology of a chain of 5 as shown in Figure 5.6.

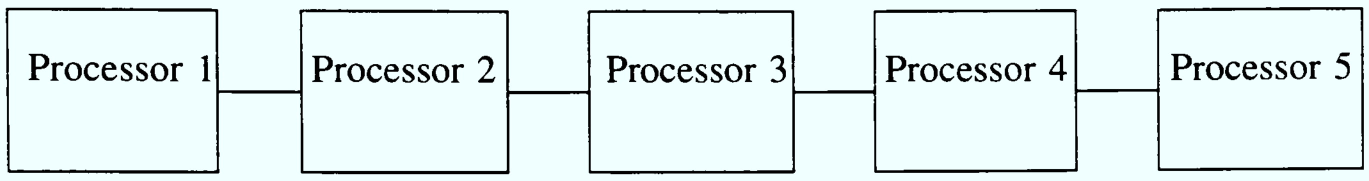


Figure 5.6: Processor pipeline

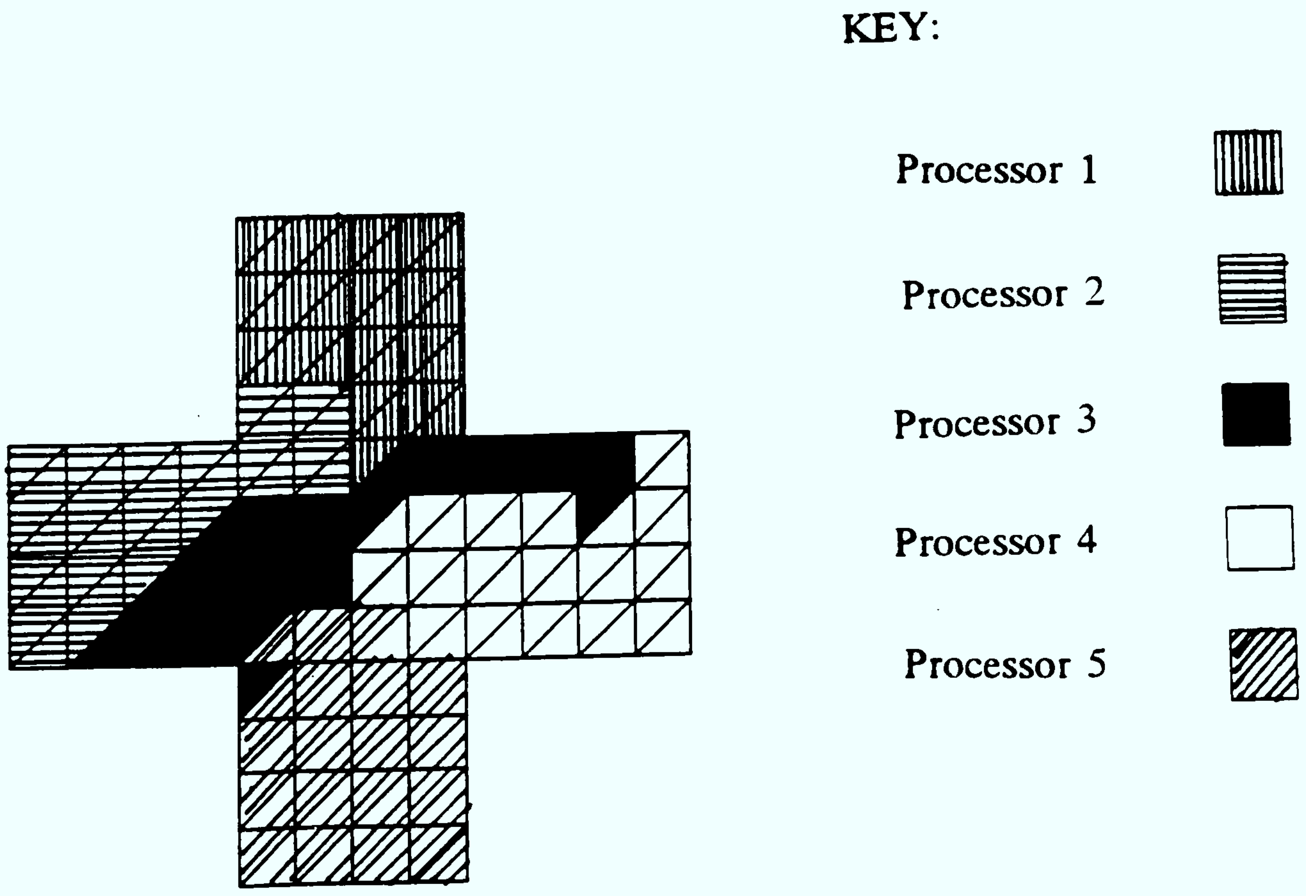


Figure 5.7(a): Decomposition of mesh using super-elements

Figure 5.7(a) shows the decomposed mesh after the algorithm has been applied. It can be seen that the algorithm has made some attempt to obtain a decomposition to map onto a chain of 5 processors. Table 5.1 shows the number of nodes that are being shared by each pair of processors. From the table we can see that non-neighbouring processors have to communicate. For example, processors 1 and 3 have four shared nodes and processors 3 and 5 have five shared nodes. This communication between non-neighbouring processors will incur a large overall communication cost. The reason that we have shared nodes between non-neighbouring processors is because of the shape of the super-elements. Obviously, they are not always smooth around the edges, and since this is a very simple example with not many super-elements, this type of problem cannot be avoided at this stage. The boundaries need to be tidied up so that we have less nodes being communicated. This is done by looking at the original elements that are on the boundaries of the processors and applying the algorithm to these boundary elements.

Pairs of processors	Number of shared nodes
1 and 2	6
1 and 3	4
1 and 4	0
1 and 5	0
2 and 3	5
2 and 4	0
2 and 5	0
3 and 4	9
3 and 5	5
4 and 5	5

Table 5.1: Number of shared nodes of every pair of processors for decomposition shown in Figure 5.7(a)

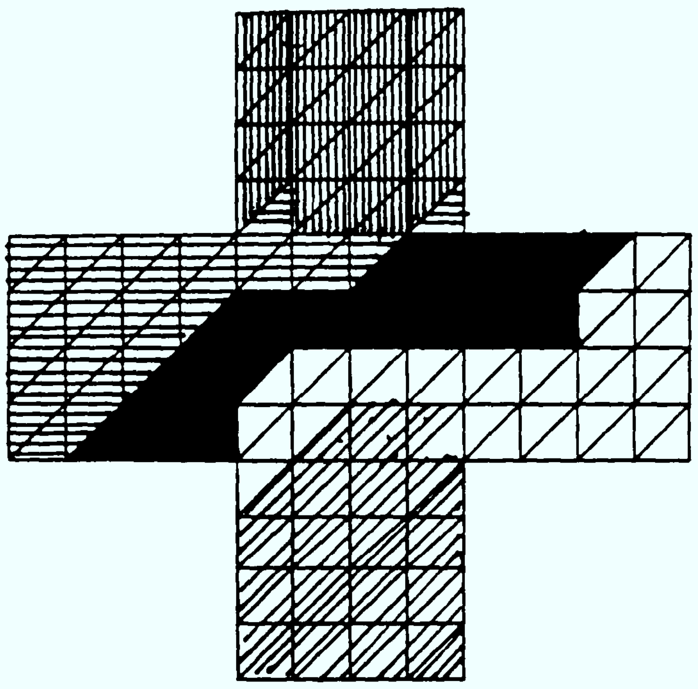


Figure 5.7(b): Decomposition after tidying up boundary once

Pairs of processors	Number of shared nodes
1 and 2	6
1 and 3	1
1 and 4	0
1 and 5	0
2 and 3	8
2 and 4	0
2 and 5	0
3 and 4	10
3 and 5	0
4 and 5	7

Table 5.2: Number of shared nodes of every pair of processors for decomposition shown in Figure 5.7(b)

Having done this to the above example, we can see the results in Figure 5.7(b). It can be seen that the decomposition looks much better with very smooth boundaries and that nearly all neighbouring elements have been placed on the same or neighbouring processors. Table 5.2 shows which processors have to communicate. However, we still have one node being shared by non-neighbouring processors, namely processors 1 and 3. The reason for this is that when we were tidying up the boundaries we were only looking at one layer of original elements.

To overcome this problem, we need to iterate the application of the algorithm to the original elements on the boundaries until no further changes are made. For this example, it only had to be done once again and the decomposition can be seen in Figure 5.7(c). This might not be such a problem with very large meshes with small super-elements. Looking at Figure 5.7(c) we can now see that all neighbouring elements are mapped onto the same or neighbouring processors and this would be a good mapping onto a chain of 5 processors.
Table 5.3 shows the number of shared nodes for every pair of processors. We can see from this table that only neighbouring processors have to communicate.

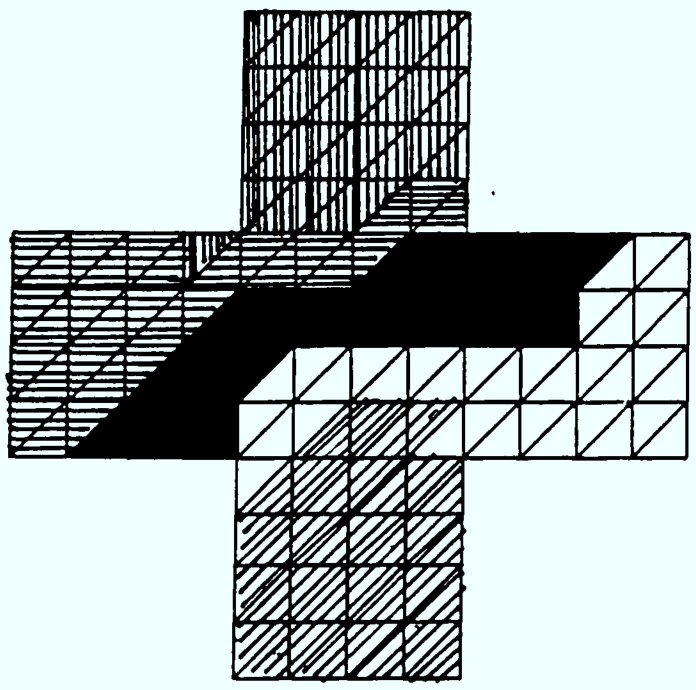


Figure 5.7(c): Decomposition after tidying up boundary iteratively

Pairs of processors	Number of shared nodes
1 and 2	7
1 and 3	0
1 and 4	0
1 and 5	0
2 and 3	8
2 and 4	0
2 and 5	0
3 and 4	10
3 and 5	0
4 and 5	7

Table 5.3: Number of shared nodes of every pair of processors for decomposition shown in Figure 5.7(c)

However, for this particular example, using the graph bisection method in a recursive way may not be the best way to create the super elements. If we examine the results obtained in Figure 5.7(c), we see that processors 1, 2 and 3 have all got 30 of the original elements, whereas processors 4 and 5 have 35 original elements. This has happened because we were mapping 32 super-elements (each containing five of the original elements) onto 5 processors. Six of the super-elements were assigned to processors 1, 2 and 3 with seven super-elements being assigned to processors 4 and 5. Therefore, load balancing has not been achieved. Even though this is only a small problem and the difference in the number of elements is only 5, the problem of load balancing would worsen considerably when using larger meshes.

This problem can easily be overcome by applying the graph bisection in a non-recursive manner. The method used is described below:

1. From a set of n elements, find 2 elements which are a maximal or near maximal distance away.

2. Create a list where one of these elements is assigned to the top of the list.

3. Repeat the procedure by finding elements adjacent to the newly assigned elements adding these elements to the list.

4. Stop when the list contains n elements.

5. The first n/p (p is the number of processors) is assigned to one set, the second n/p elements is assigned to another set, and so on, until we obtain p sets each containing n/p elements.

6. Repeat for each set until the required number of super-elements is obtained.

This method can be used to create super-elements for any number of processors.

Therefore to summarise, Figure 5.8 shows a flow diagram to illustrate the steps involved when creating the super-elements.

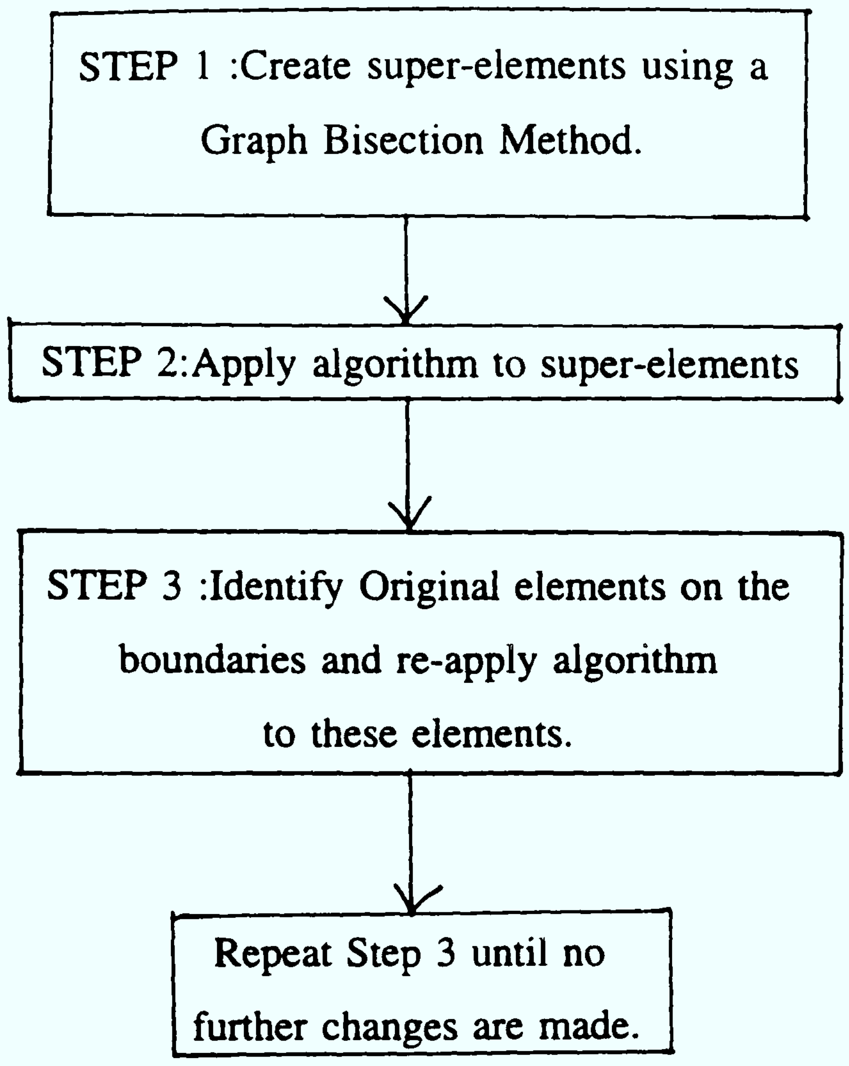

STEP 1 :Create super-elements using a Graph Bisection Method.
STEP 2:Apply algorithm to super-elements
STEP 3 :Identify Original elements on the boundaries and re-apply algorithm to these elements.
Repeat Step 3 until no further changes are made.

Figure 5.8: Flow diagram illustrating creation of super-elements

5.3 Level of Granularity

Now that super-elements can be used to reduce the time taken to decompose the meshes, one question that needs to be answered is 'What level of granualarity should be taken ?' i.e. How many super-elements should be created and what impact various numbers of super-elements have on the time taken and the overall communication costs of the decomposition achieved.

The mesh shown in Figure 5.9 will be used as an example and this mesh has 3034 elements.

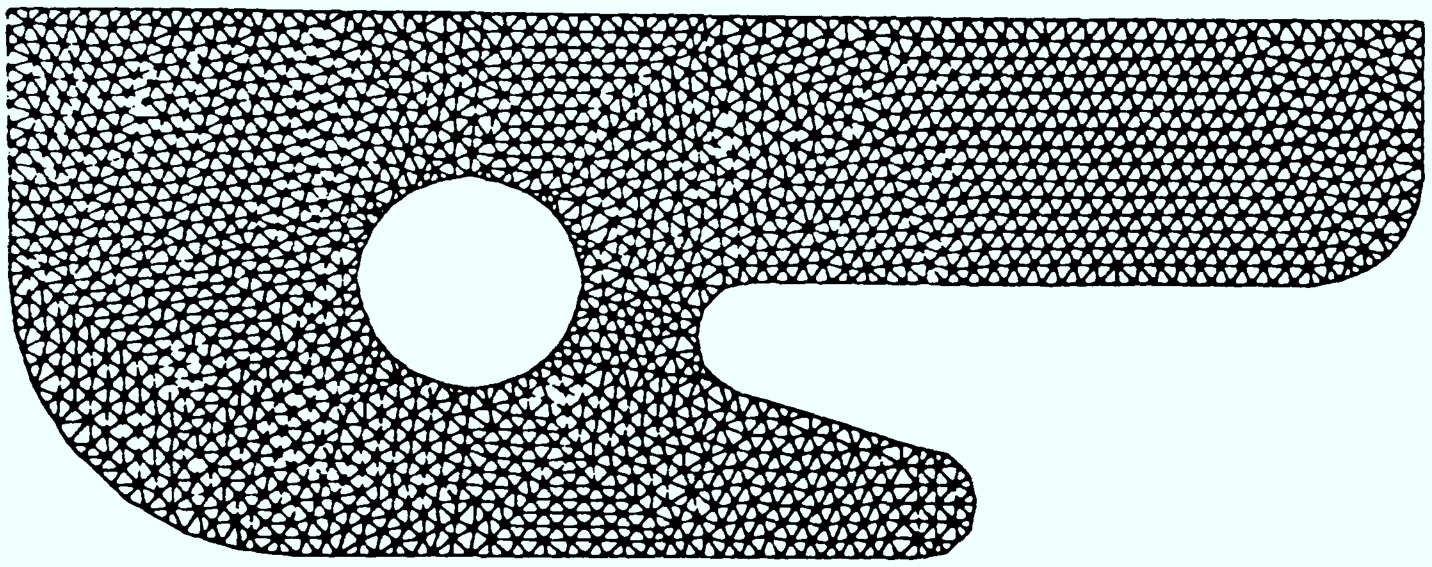


Figure 5.9: Mesh with 3034 elements.

This mesh was reduced to meshes containing 32, 64, 128, 256 and 512 super-elements respectively. These meshes were then decomposed taking into account a processor topology of a chain of 4 processors. The decomposition achieved can be seen in Figures 5.10(a) - 5.10(e).

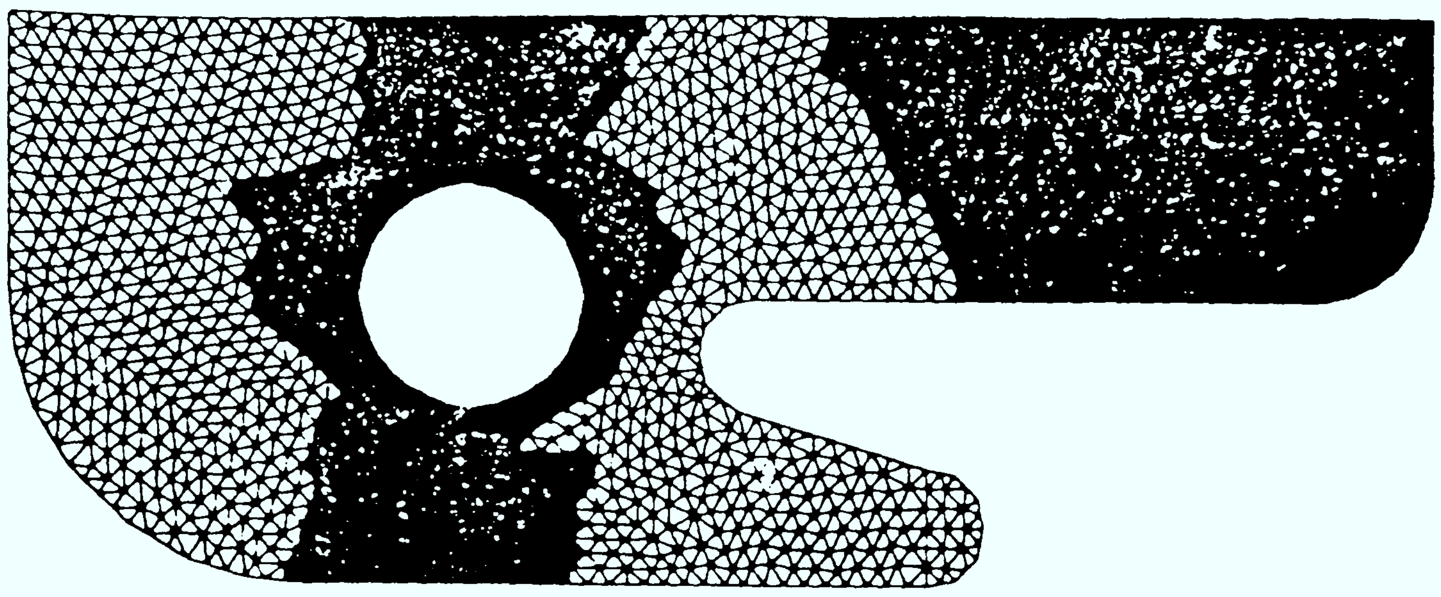


Figure 5.10(a): Decomposition of 32 super-element mesh

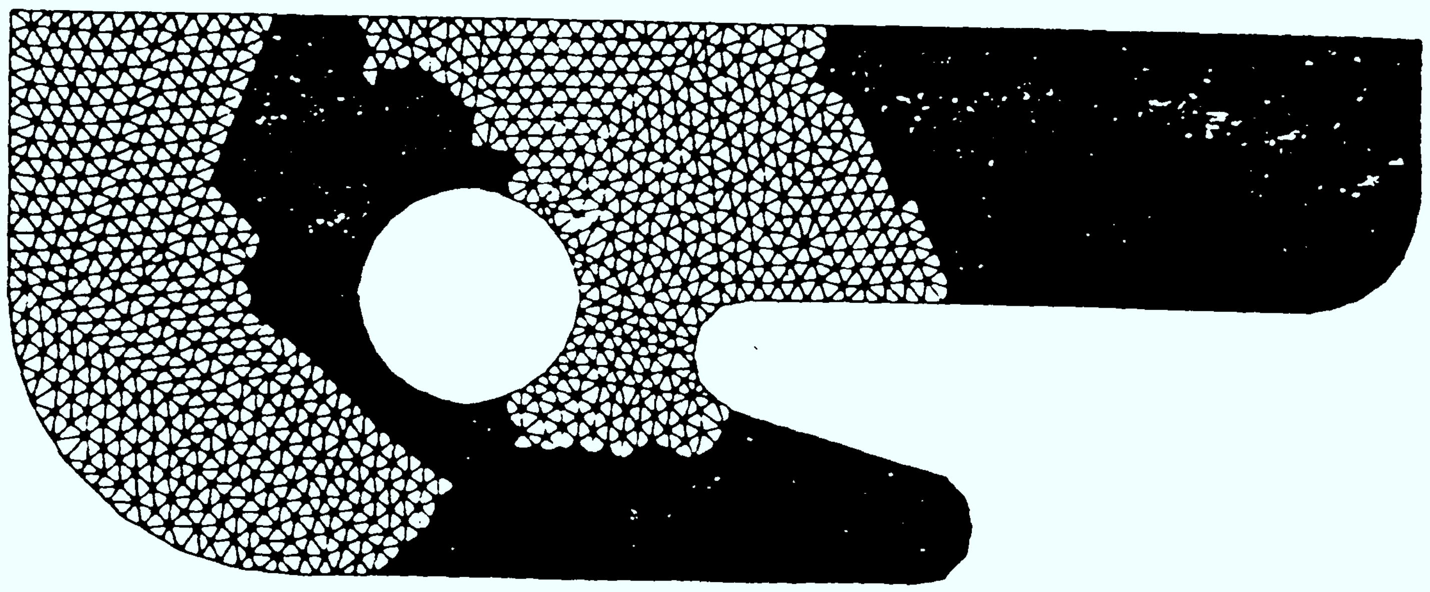


Figure 5.10(b): Decomposition of 64 super-element mesh

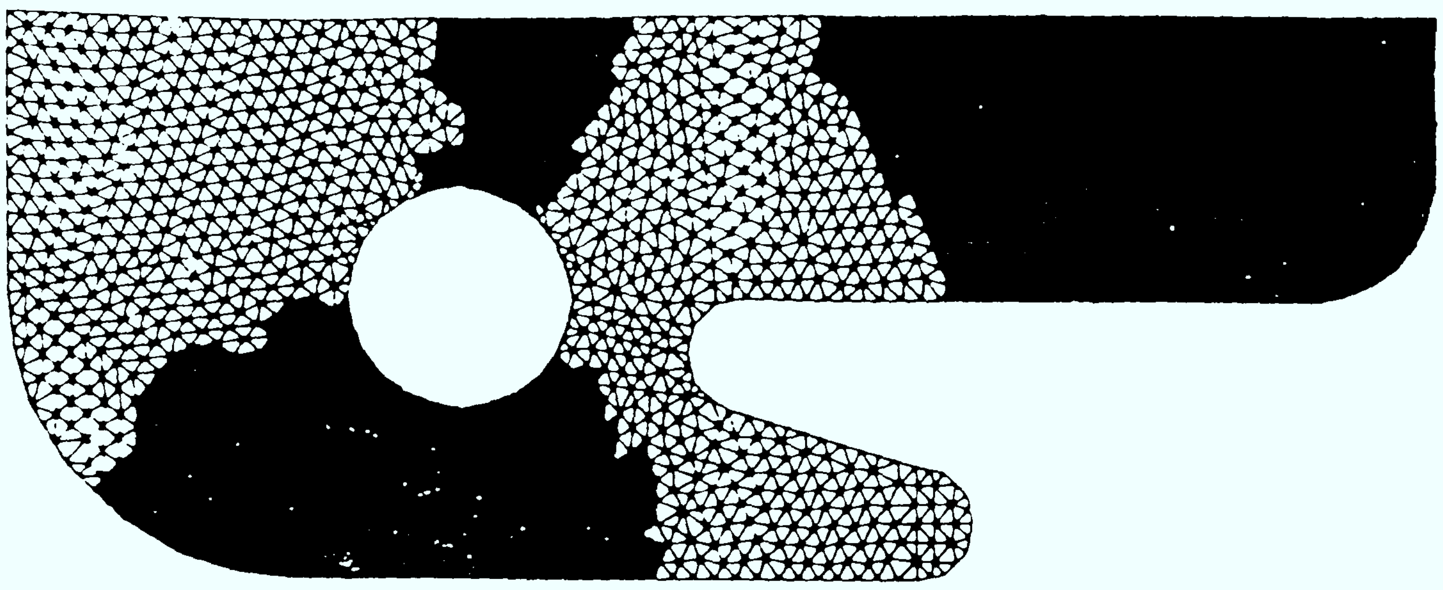


Figure 5.10(c): Decomposition of 126 super-element mesh

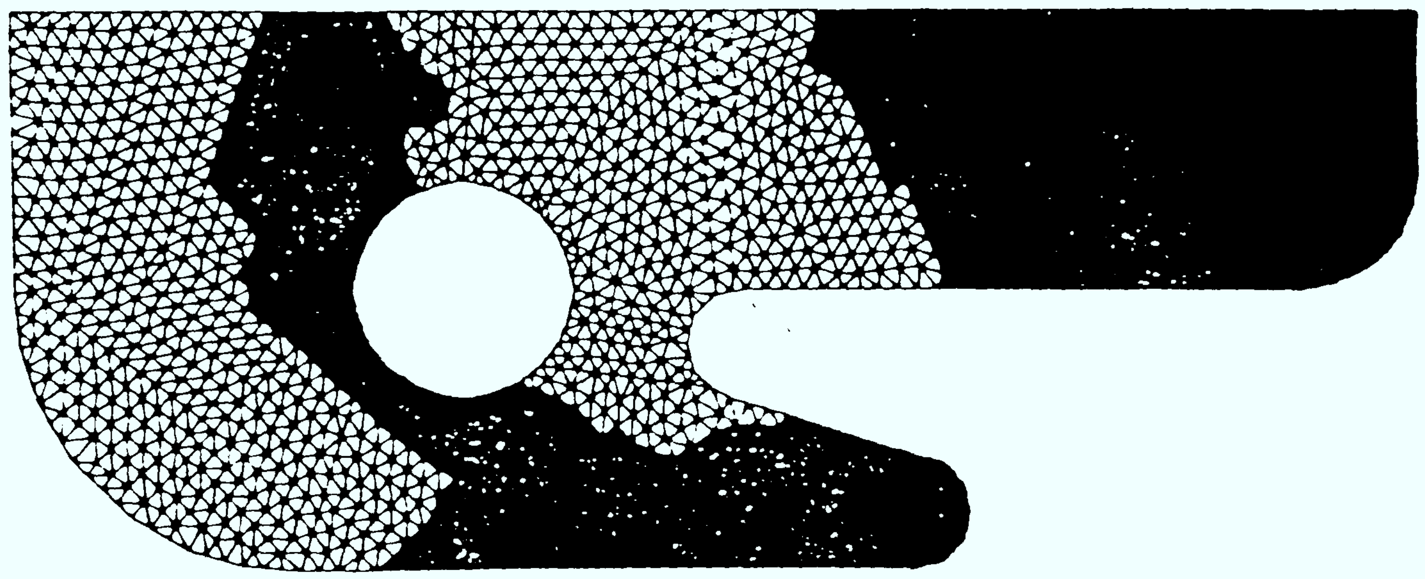


Figure 5.10(d): Decomposition of 256 super-element mesh

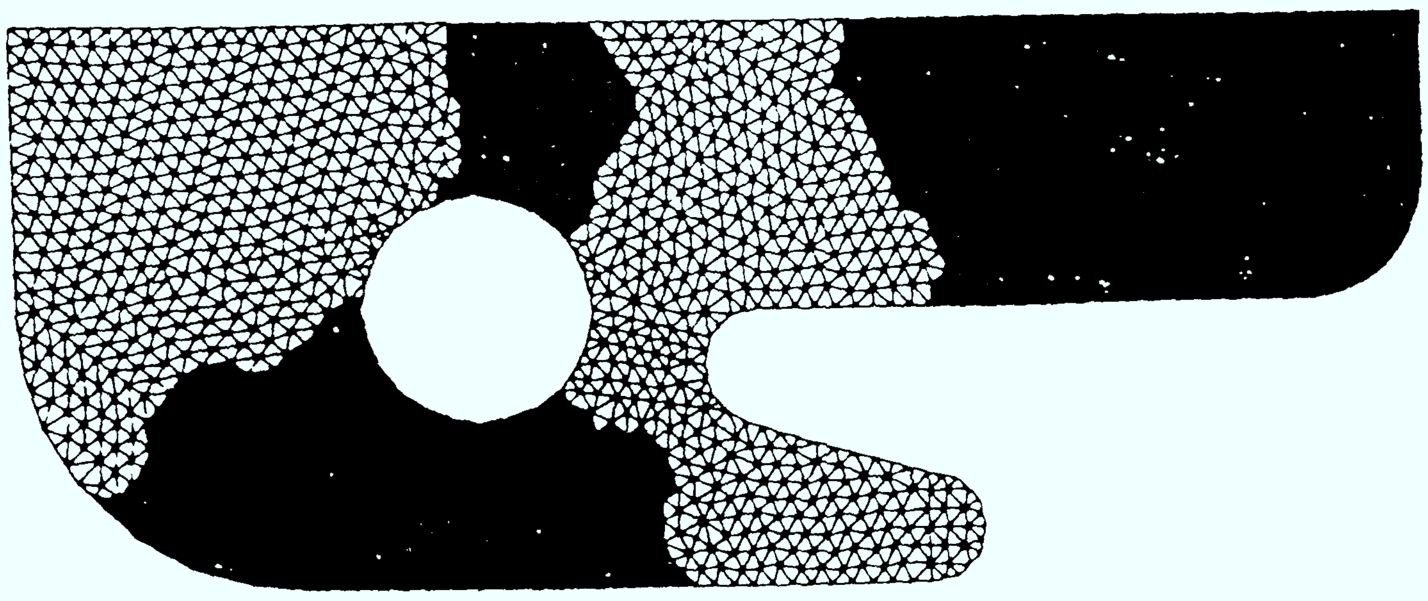


Figure 5.10(e): Decomposition of 512 super-element mesh

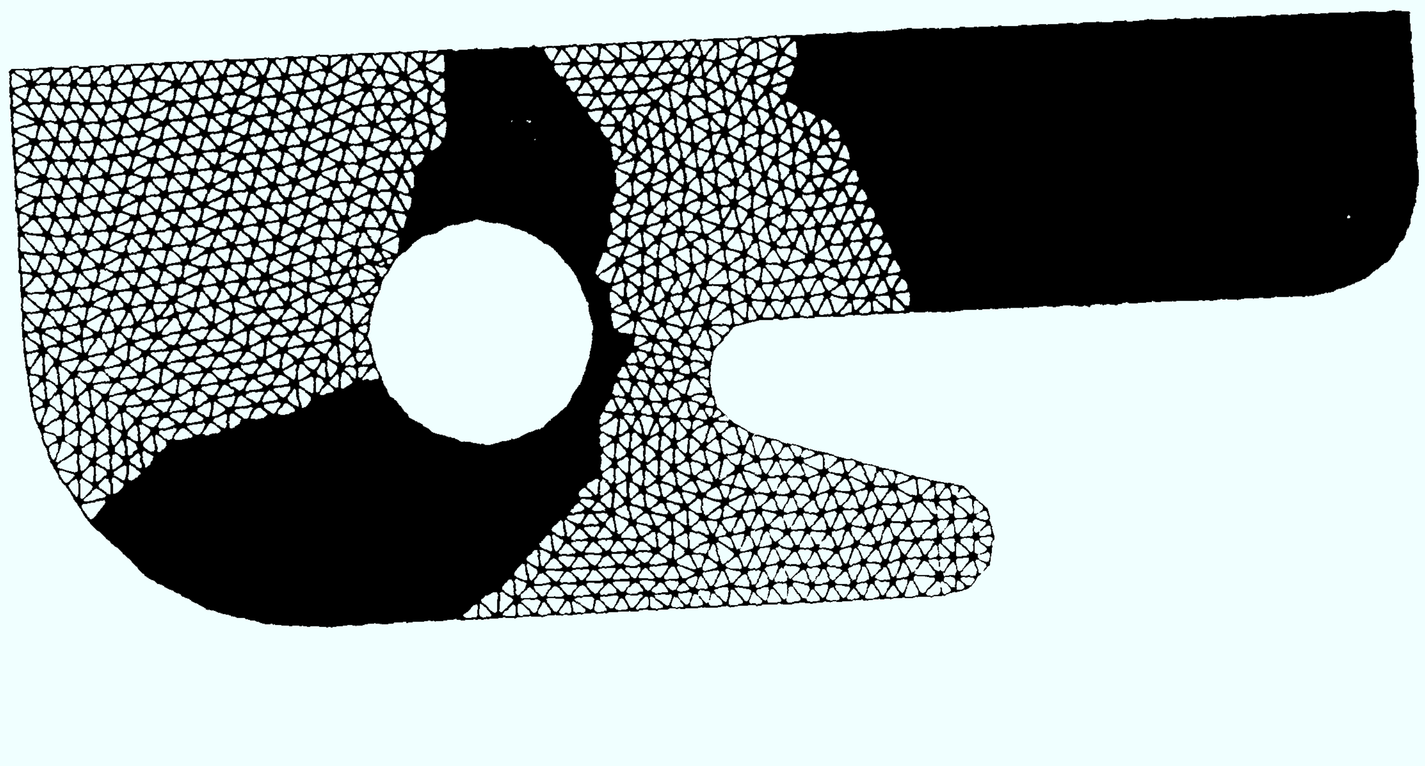


Figure 5.10(f): Decomposition with no super elements

Looking at the decompositions achieved we can see that they look very similar but that there seems to be two different patterns. For example, Figures 5.10 (a), (c) and (e) are simliar and Figures 5.19(b) and (d) are simliar to each other. It is possible that the problem converges into two different minimas and we can speculate that the reason for this will be different starting partitions. However, the size and shape of the super elements could also be the cause of this.

However, it is important that we look at the actual communication costs and the time taken to decompose the mesh.

Table 5.4 shows the number of super-elements used and the communication costs between each processor and the total communication costs. This can also be seen graphically in Figure 5.11. We can see that the communication costs do decrease when we use a larger number of super-elements.

However, Table 5.5 shows the time taken to create the super-elements and the time taken to decompose the mesh into four using various sizes of super-elements. This can also be seen graphically in Figure 5.12 and from this graph we can see that the time taken to decompose the mesh increases exponentially with a larger number of super-elements.

Communication between pairs of processors

No of super-elements	1 and 2	2 and 3	3 and 4	Total
32	38	43	19	100
64	38	36	19	93
128	35	36	19	90
256	38	29	19	86
512	30	30	17	77
No super elements	27	30	17	77

Table 5.4: Communication costs between processors for different sizes of super-elements

No of super-elements	Time taken to partition (s) *
32	0.8
64	1.4
128	11.2
256	32.3
512	59.0
No super elements	> 600

Run on a Sun SPARC station

* Time taken to partition includes time taken to create the super-elements.

Table 5.5: Time taken to partition different sizes of super-elements

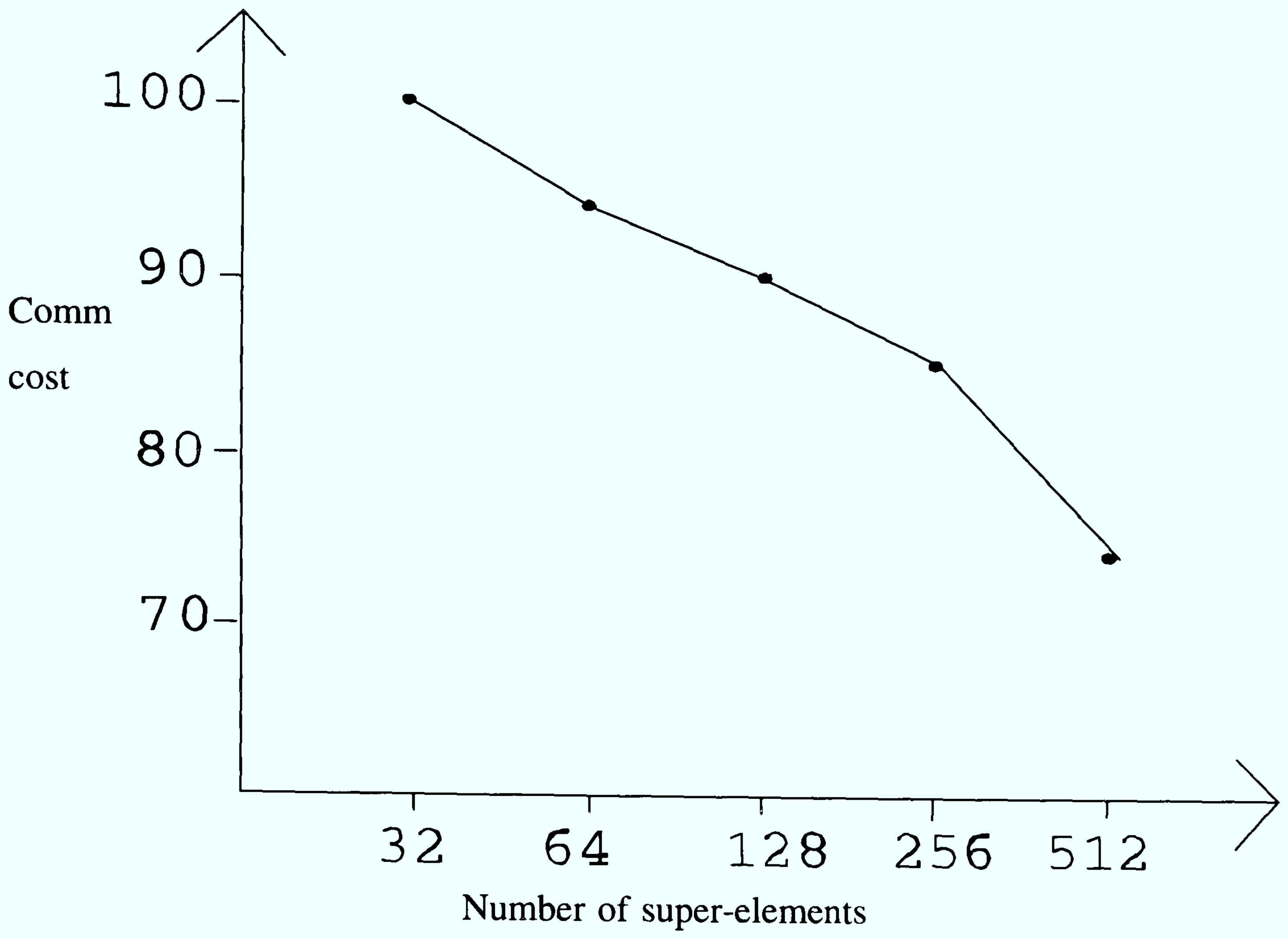


Figure 5.11: Graph showing communication costs against no. of super-elements

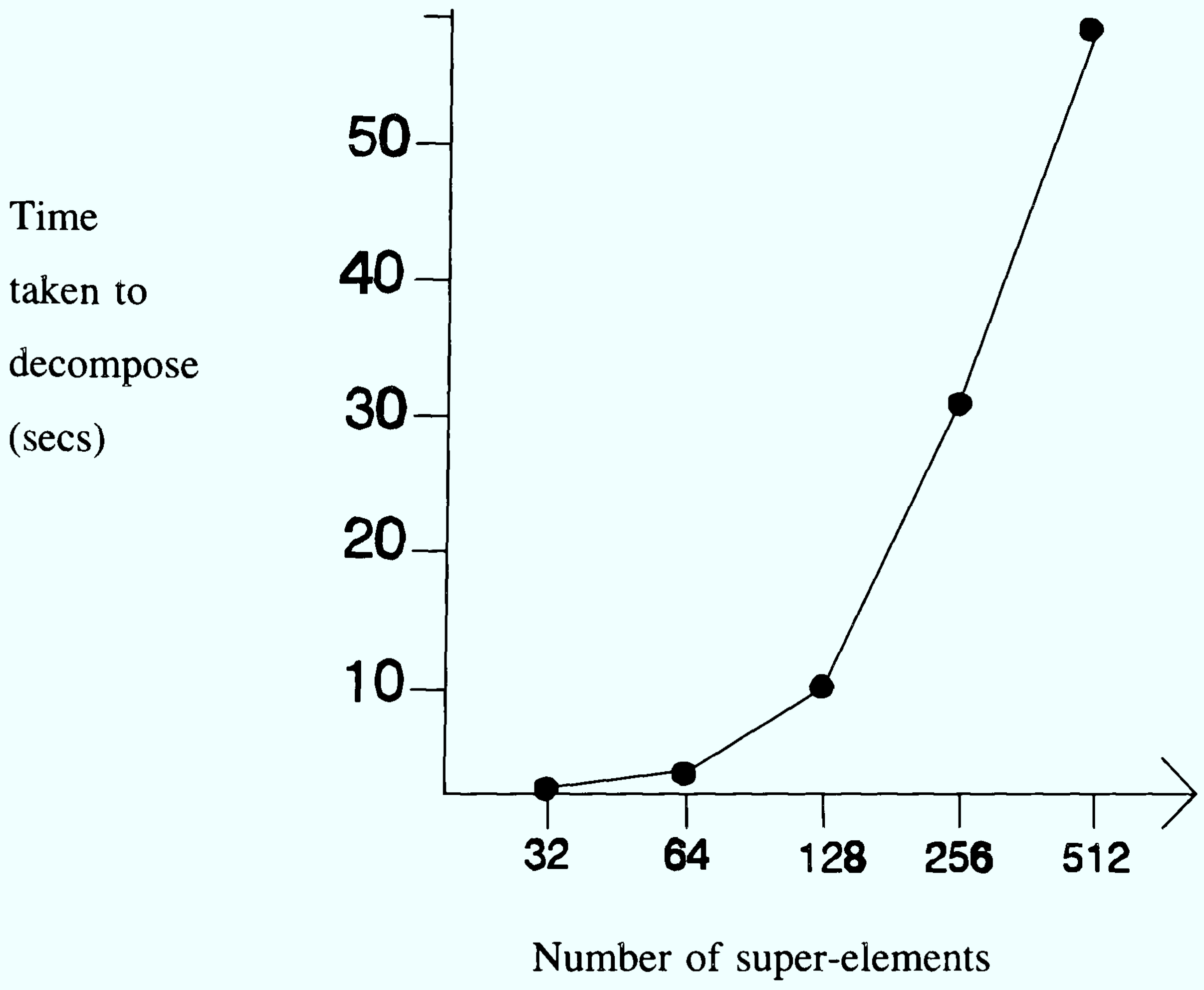


Figure 5.12: Graph showing time taken against no. of super-elements.

From these two graphs we can conclude that it would be best to use a smaller number of super-elements, even though the communication costs might be slightly higher. However, there will be a huge saving in time taken to achieve the decomposition. Whilst saying this, there must be a minimum number of super-elements that can be used. Obviously, if the super-elements are too large and there are only a few of them, then it will not be possible to take the processor topology into account. Of course, many factors need to be taken into account. For example, the shape and size of the mesh used.
Using the cross shape shown in Figure 5.1, it would be very difficult to map onto the topology shown in Figure 5.13 without using at least 32 super-elements.If any less is used, then it will not be possible to map onto such a topology without having non-neighbouring processors communicating.

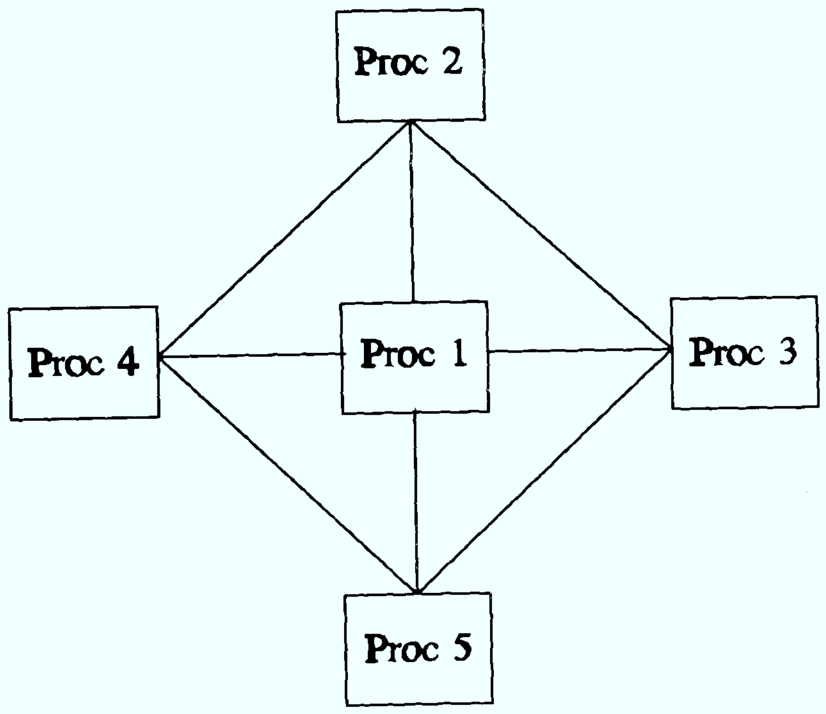


Figure 5.13: Processor Topology

The number of super-elements to be used must be decided by the user but care must be taken in his choice. The processor topology, along with the size and shape of the mesh must be taken into consideration.

5.4 Conclusion

This chapter has dealt with the creation of super-elements in order to minimise the time taken to decompose the meshes. We have shown that by using this method, we can generate reliable decompositions with a huge reduction in time.
However, we have not yet addressed the effectiveness of the parallelisation of the code. Chapter 6 shows some more test cases and some results obtained of the parallel efficiency using the sub-domain achieved using our decomposition technique.

Chapter 6

Computational Results and Conclusions

6.1 Introduction

In chapters 4 and 5 we have demonstrated the success of obtaining reliable mesh decompositions using the iterative clustering algorithm and also by creating super-elements to obtain time-effective sub-meshes. We have not yet demonstrated the effectiveness of the parallelisation using the sub-meshes obtained with our algorithm. In this chapter some examples showing parallel efficiency will be shown.

6.2 Parallelisation of UIFS

UIFS is a control volume unstructured mesh flow and unstructured stress analysis code developed at the University of Greenwich with the intention of modelling metal casting and other processes [CHOW93], [CCP93], [CROSS92]. This is a fully unstructured mesh 2D and 3D code which uses a cell centred Rhie and Chow interpolation [RC82] with a pressure correction solution procedure and false time stepping. The procedure for solving the discrete equations is iterative using the SIMPLE algorithm. Iterations are repeated until changes are small enough to satisfy convergence criteria. Three types of solvers are available; Jacobi, Gauss Seidel SOR and the Conjugate Gradient method.

This code has been parallelised using domain decomposition [MKCJ94], [JMCEJ93] with explicit message passing in Fortran to fulfil the following objectives:

i) Minimise changes to the original serial code. Ideally the parallel code should produce identical results to the serial. This is a necessary requirement for user acceptability.

ii) Minimise visibility of parallel code. The code is under continual development so the parallel code should be hidden from the serial code developers and the serial code users.

iii) Maximise parallel efficiency to take full advantage of DM parallel hardware.
The motivation for parallelisation is to reduce the run time.

iv) Be portable to a variety of DM MIMD platforms.

v) It should be automatable. Human intervention should be minimised in the parallelisation process as in the Computer Aided Parallelisation Tools [JCIL93] used at the University of Greenwich.

6.3 Mesh Division

The mesh is split into the required number of domains using the method described in Chapter 5, where each node or element is allocated to only one region. Each processor works on these core nodes or elements in its own domain. At the edge of a domain, there will be a mixed element where the nodes belong to a different domain. So that each region has a complete mesh discretisation, halo nodes/elements are added to the domain. These are copies of nodes/elements from neighbouring domains as illustrated in Figure 6.1.

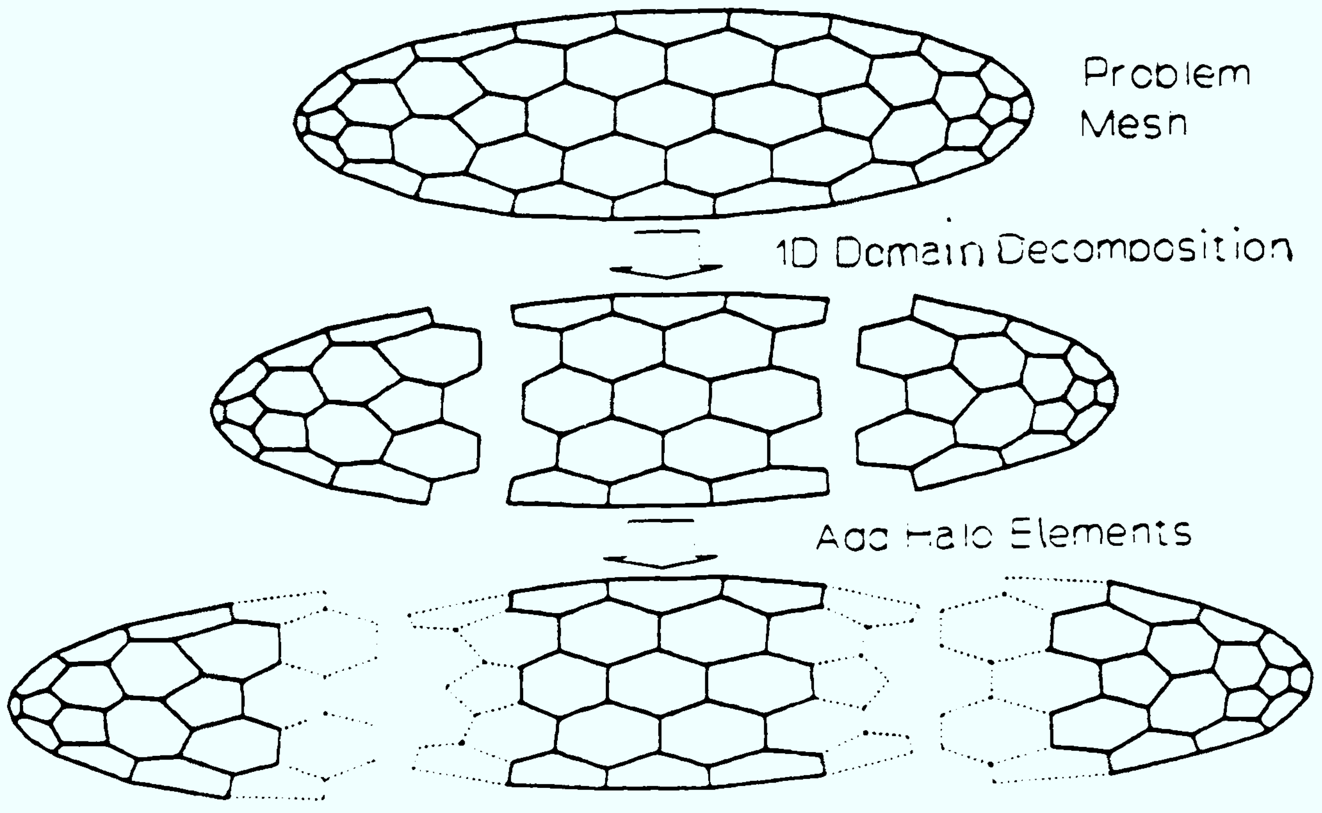


Figure 6.1 : Halo Elements

Each processor calculates only the values of variables for points and elements inside its own domain, no computation is performed on the haloes. Variable values are swapped into the halo from the processors on which the variables are calculated, as shown in Figure 6.2. There is an obvious exception however, where data operations are so trivial that it is faster to perform the operation locally on the halo than to import the new values from a neighbour. Halo values are exchanged between processors as soon as practically possible, for example, at each iteration of the solver. This exchange of data between processors is synchronised on an odd-even alternate pair basis which allows the exchange to be carried out as a parallel process.

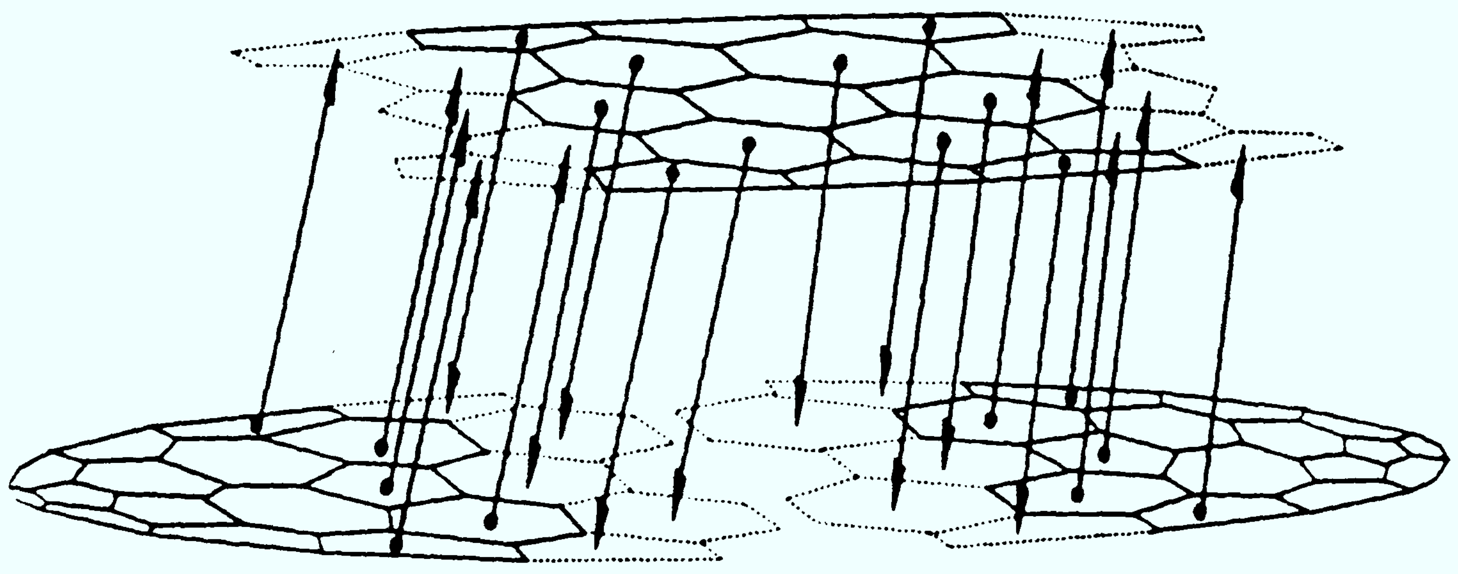


Figure 6.2 : Halo swapping scheme.

Variation between serial and parallel code is sometimes inevitable. Like many CFD codes, UIFS builds a system matrix which is solved using a variety of iterative schemes. The main change to the serial algorithm is the order of coefficient evaluation within the solvers. Using a Jacobi type solver the parallel solution variables remain identical to those of the serial code at each step of the solution procedure. It is however impractical, if not impossible, to identically parallelise a Gauss-Seidel iterative solver. Such algorithms are dependent on the order of evaluation of the coefficients and must be modified to achieve a parallel scheme. The resulting parallel algorithm becomes a near Gauss-Seidel hybrid of Gauss-Seidel and Jacobi. The results so far have shown that variations in the serial and parallel variables and differences in the number of iterations required to converge are both minimal.

6.3 Efficiency of Parallel Solution

6.4.1 Simple 2D Problem

The effectiveness of the parallelisation approach is first demonstrated on the simple 2D problem. The problem is a simple modification to the moving lid problem which produces a flow filled with a number of recirculation zones, as shown in Figure 6.3.

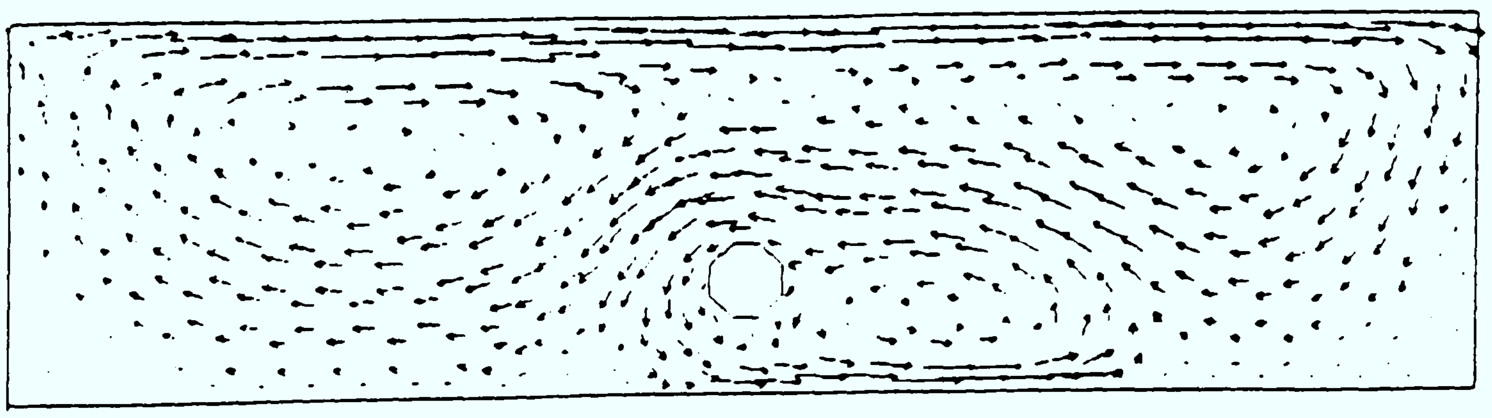


Figure 6.3: Flow vectors due to moving lid

The mesh contains 516 elements and has been split into 3, 4, 5 and 6 sub-meshes as shown in Figure 6.4.

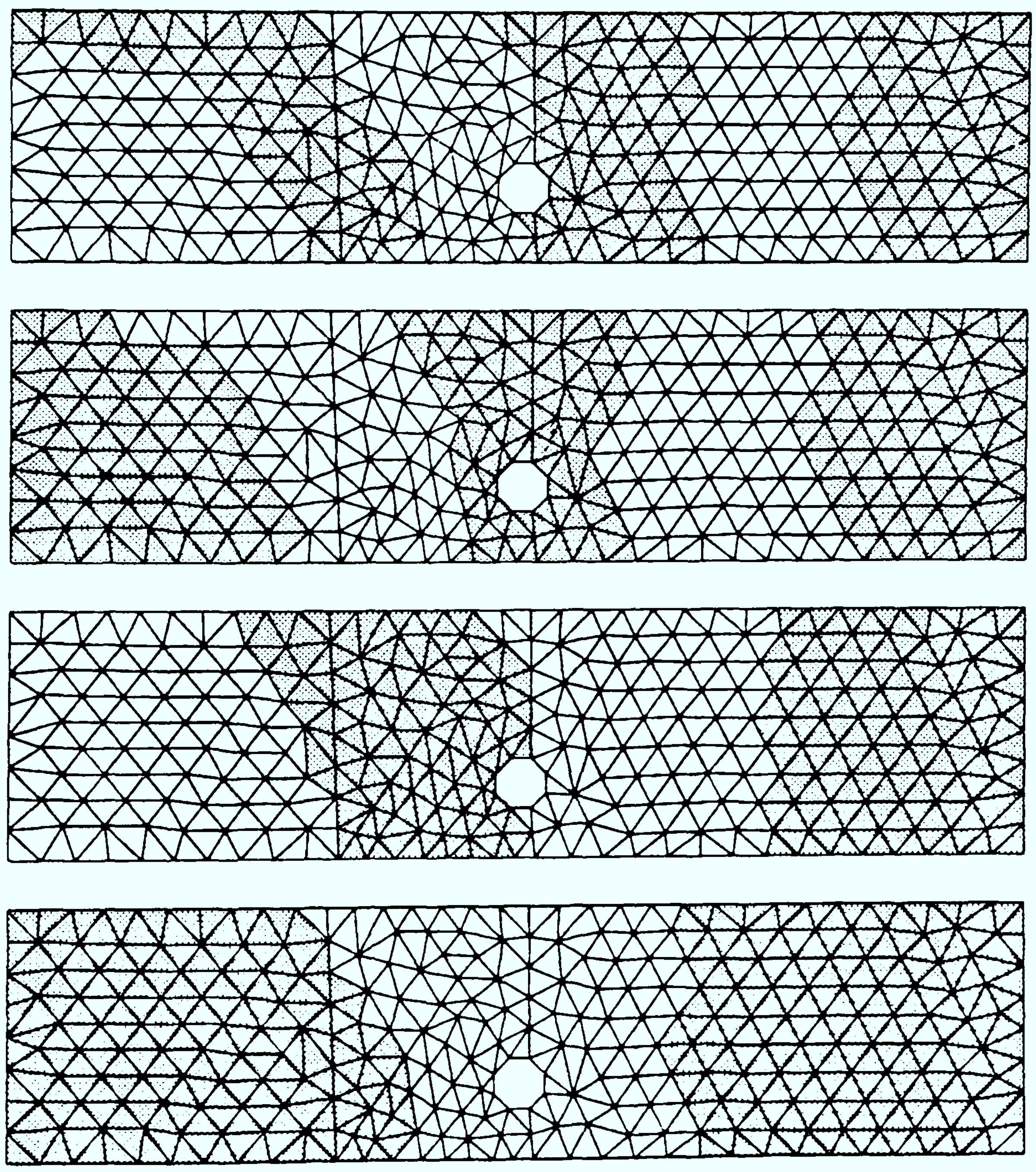


Figure 6.4: Mesh containing 516 elements split into 3, 4, 5 and 6 sub-meshes

The parallel UIFS code was run on the above problem on an array of T800-20 transputers with a 3L FORTRAN compiler, and on a TRANSTECH parallel system where each node uses a T800 for communications and an i860 processor for processing. Here the Portland FORTRAN compiler is used with the CTOOLSET to handle interprocessor communications. Results for parallel efficiencies are shown on 1-6 processors on both the transputer and i860 systems in Table 6.1 and Figure 6.5.

The proportion of the mesh decomposition time of the simulation time is approximately 4% for this particular problem. However, this can vary with different problems sizes.

Parallel Efficiency (%)

No of processors	T800	i860
1	100	100
2	98	92
3	97	81
4	96	76
5	94	70
6	90	66

Table 6.1: Parallel Efficiency for 516 element mesh

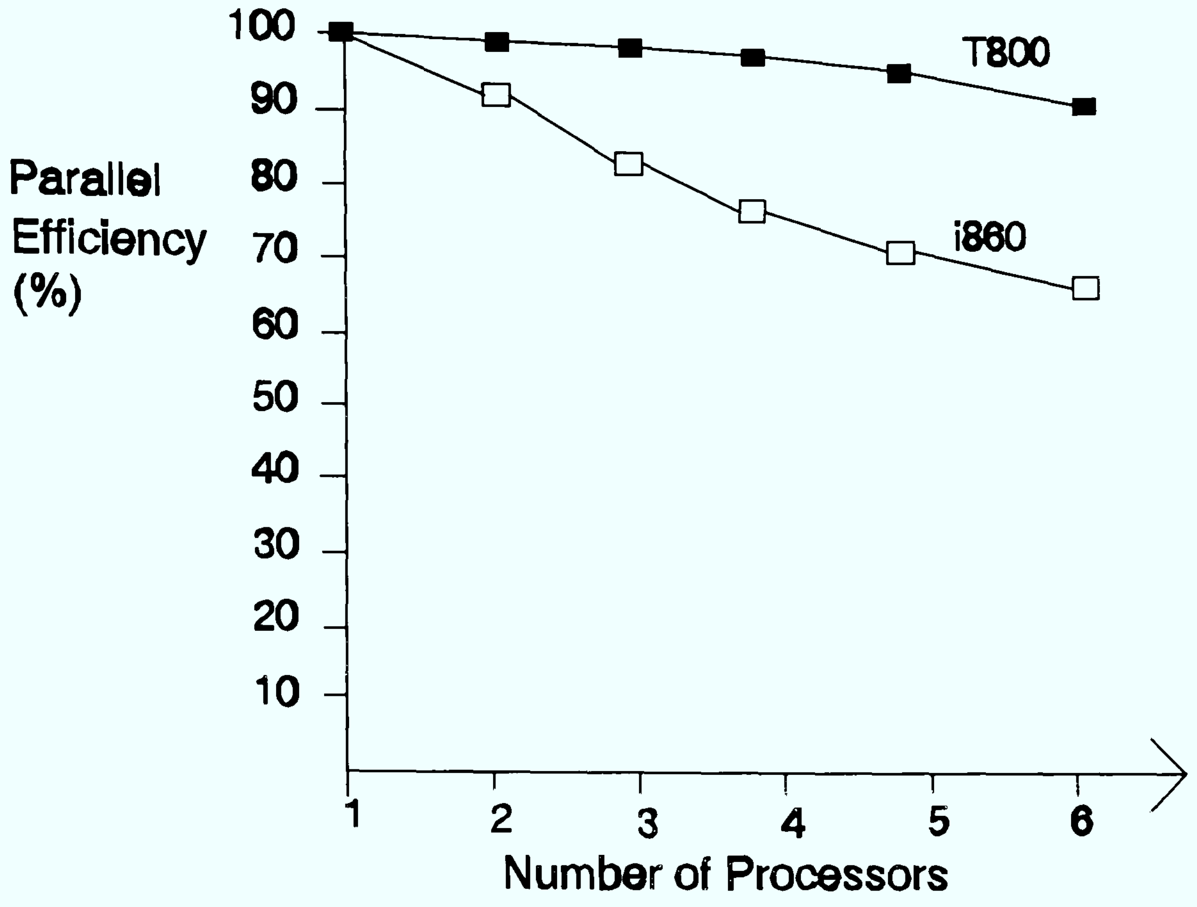


Figure 6.5: Parallel efficiency

In reality the problem is a small one and the efficiencies on the transputer system are reasonably consistent with what would be expected from a structured mesh code employing otherwise similar solution procedures. It is anticipated [JC91] that as the problem size increases, then the efficiency will rise and remain above 90% on transputer systems beyond 20-30 processors. The efficiency results for the i860 system is much worse - at the 65% level for 6 processors compared with 90% on the transputer system. The reason for this degradation in performance is simply a function of the parallel system characteristics [GCCHI92]. Although an i860 processor is about 10-15 times faster than a T800-20 transputer, the latency of the T800-i860 node is around 10 times that of a T800-20 processor. This is due to the interrupt overhead between the i860 and T800 processors. As such, the interprocessor communication time between the i860 nodes is relatively much larger than between T800 nodes and leads to the degradation in parallel efficiency. The efficiencies are still consistent with structured mesh codes with otherwise similar solution procedures.

6.4.2 Larger meshes

Figure 6.6 shows a solidifying metal problem which has been meshed into 3034 and 10000 elements as shown in Figure 6.7 and Figure 6.8. Both meshes were partitioned using the method described in Chapter 5.

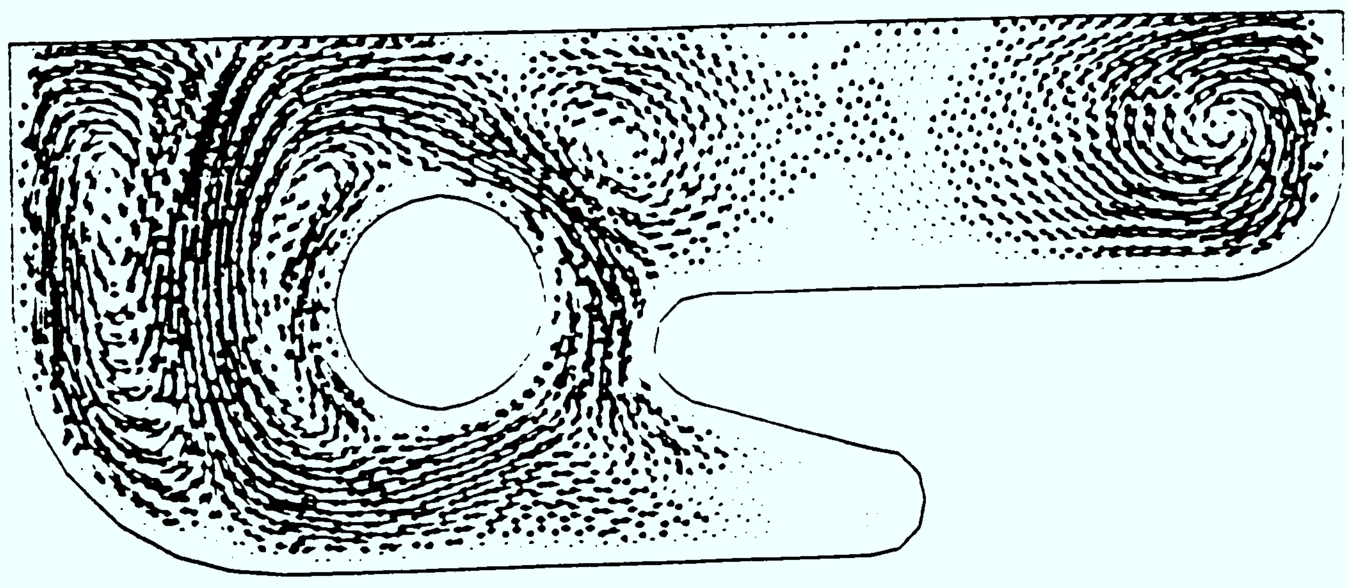


Figure 6.6: Solidifying metal

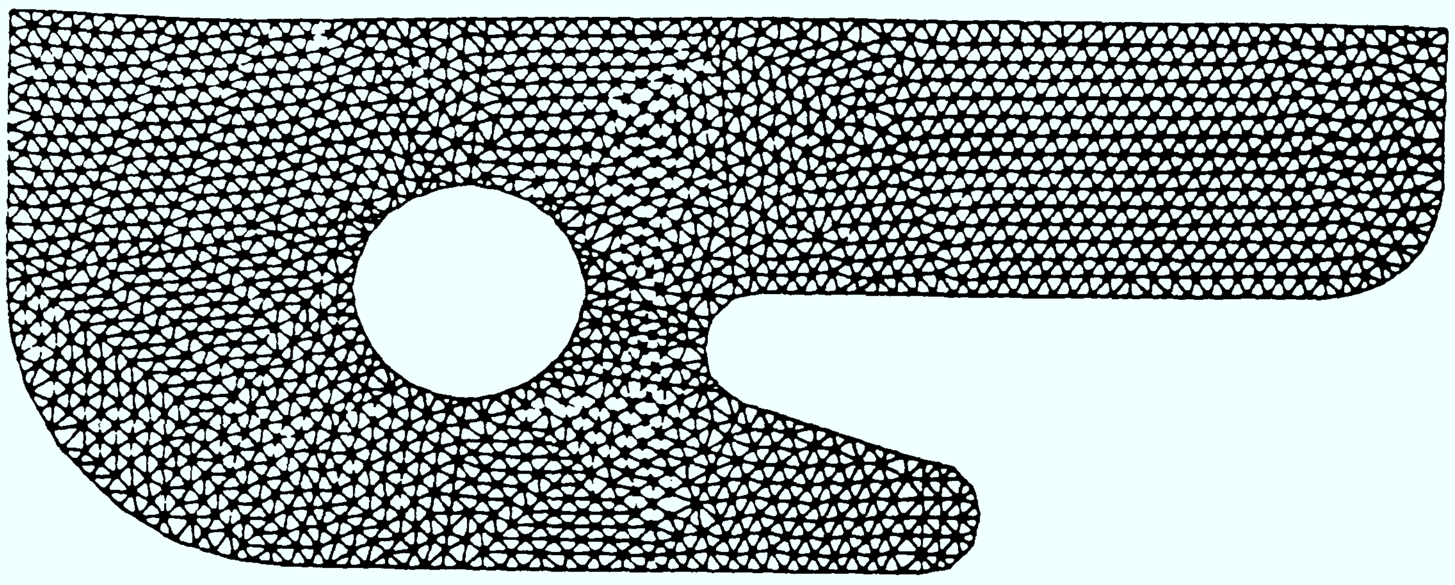


Figure 6.7: Mesh containing 3034 elements

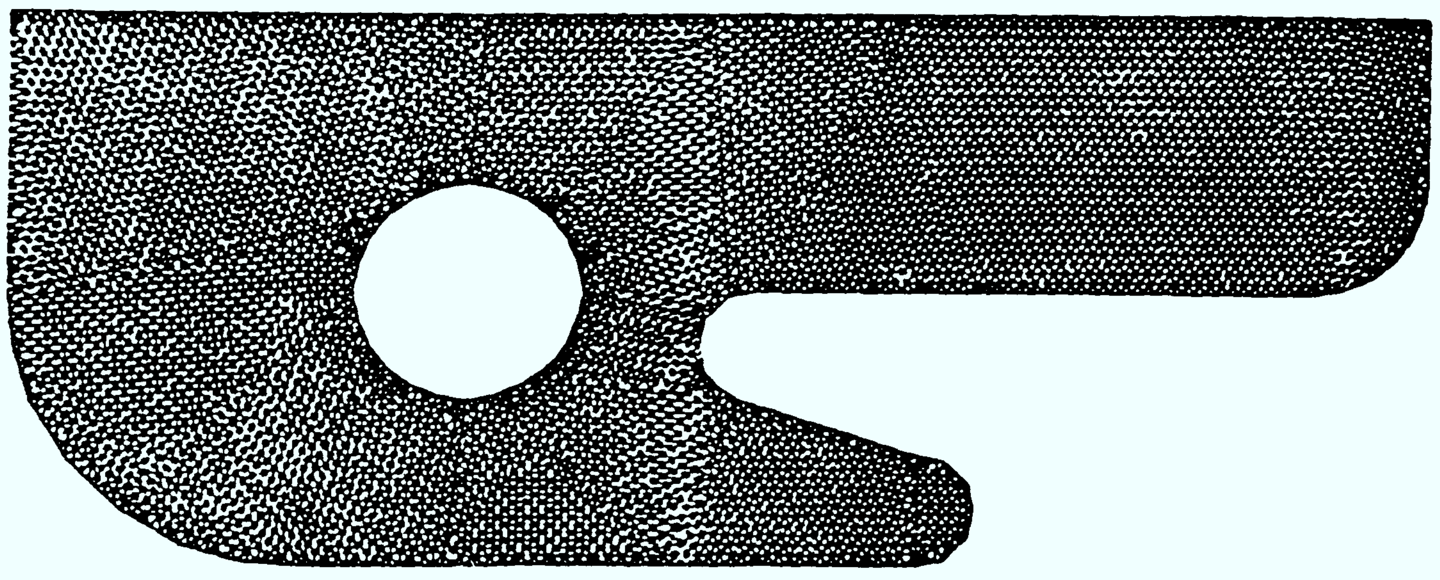


Figure 6.8: Mesh containing 10000 elements

Figure 6.9 is a graph showing parallel efficiency and speed-up using between one and twelve processors for both the meshes shown in Figures 6.7 and 6.8. We can see that parallel efficiency decreases to 60% if twelve processors are used for the smaller size mesh but only decreases to 83% for the larger mesh. Speed up is also very good for the larger mesh and increases if more processors are used.

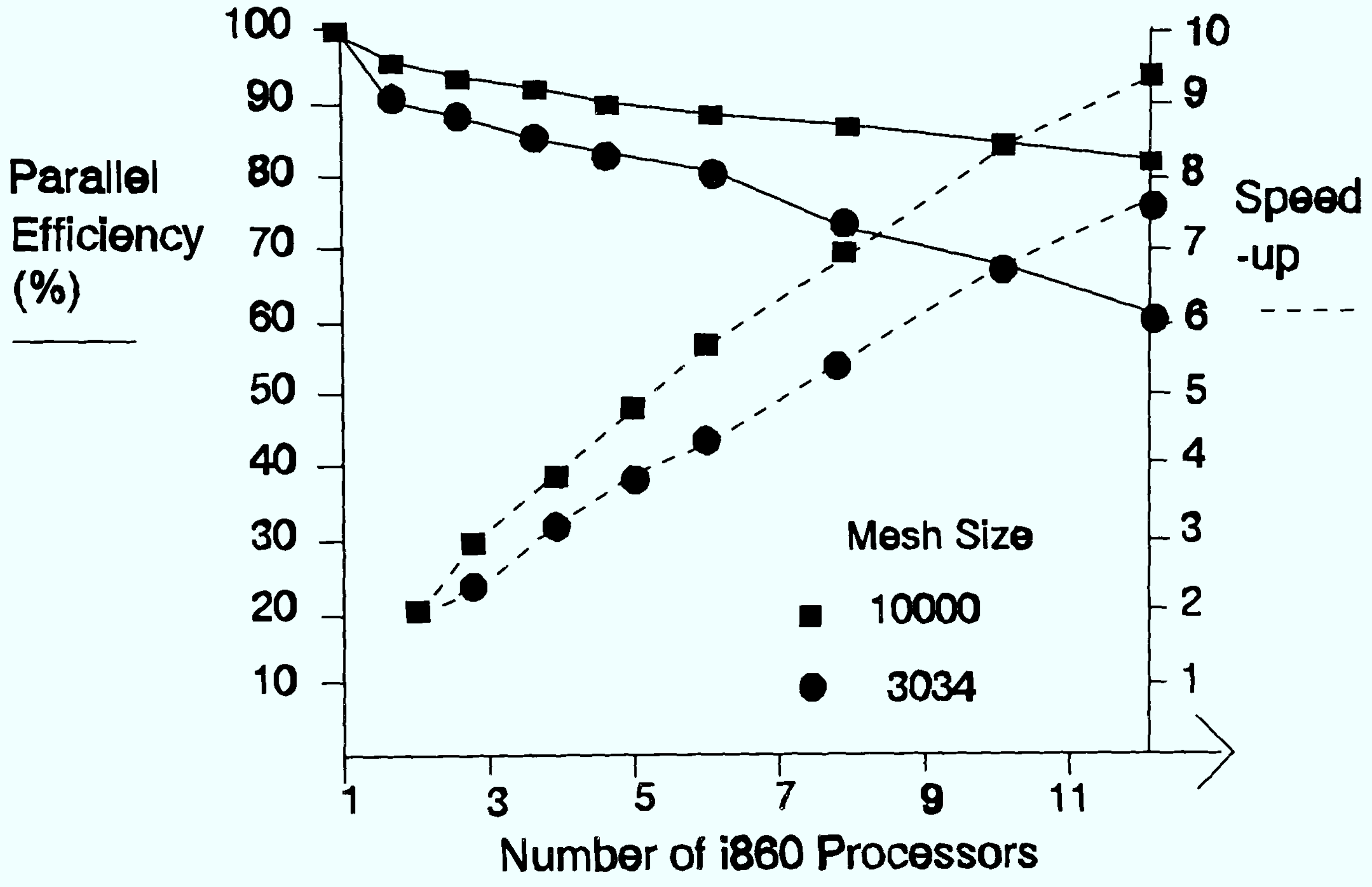


Figure 6.9: Parallel efficiency and speed-up

6.5 Conclusions and Further Work

In this thesis we have investigated mapping unstructured meshes onto distributed memory parallel architectures. We have discussed various techniques used to solve the mapping problem and we conclude that an extension of the recursive clustering method was most suitable for our needs. This method was extended so that any number of processors can be mapped onto and that the processor topology is also taken into account whilst decomposing the meshes. Results have shown that this method can work successfully.

Our preliminary results shown in Figure 6.9 is an early indication that the method is scalable but further experiments need to be performed since our experiments have not dealt with a parallel machine containing more than twelve processors.
We need to investigate further the efficiencies and speed-up if very large meshes and a large number of processors is used.

Decomposing very large meshes can be very time consuming, so we have developed a method to reduce this time. This is done by clustering elements together to form super-elements and we have demonstrated that by using this method,the time taken to decompose the meshes can be reduced dramatically. Hence, super-elements provide a good method of 'speeding-up' domain decomposition algorithms. The unstructured CFD codes which have been decomposed using the above methods can be effectively mapped onto distributed memory parallel architectures with efficiencies that are equivalent to structured mesh codes.

There are many extensions that can be made to the work presented here. The graph bisection method may not necessarily be the best method used to create the super-elements, therefore other methods will need to be examined. For example, Chris Walshaw et al [WCJE94] uses a variant of the Greedy algorithm [FA88] which he uses recursively. In addition, the current method we use for boundary refinement may not be the best therefore other methods can be investigated.

The shape, design and properties of the super-elements will be investigated. There are many issues that need to be addressed. For example, should the super-elements be connected and have small diameters? Should the size of the super-elements be determined with respect to the size of the mesh and should their shape be similar to the shape of the mesh?

Another important issue is the overall computation time. We have seen that by creating super-elements we can reduce the time taken for mesh decomposition but how much time should we spend on creating super-elements? Further investigations need to be carried out in order to answer these questions. We need to ask ourselves to what extent is mesh decomposition important? When solving a real problem on a parallel system, we need to know what the difference is in decomposing using a random technique, using a quick and dirty method, a fairly good but long method and an exact one. Comparisons of solutions obtained and time taken to obtain a solution using different techniques can be made.

The current implementation of our method assumes that all node weights in the task graph are equal and that the edge weights are equal. Additionally, the iterative clustering method assumes that the computer is homogeneous. Our method can be extended to operate on heterogeneous task and processor graphs. With many problems this is not always the case. However, we could overcome this problem by taking into account the weights of the task at the initial partition stage and also during the optimisation stage.

Currently, our mesh decomposition method used has only been implemented sequentially. However, if the mesh decomposition could be done in parallel, this would increase the speed-up. The initial partition, which need to be quick and cheap, would have to be done sequentially and the optimisation could then be done in parallel. However, only elements on the boundaries could be swapped between the processors, hence the final partition will not deviate too far from the initial one. Again, we would attempt to ensure load-balancing in the initial partition and only swapping of elements would be allowed. Chris Walshaw et al [WCJE94] are developing a parallelisable algorithm for partitioning unstructured meshes. Their method, encapsulated in a software tool, JOSTLE, uses a combination of techniques including the Greedy algorithm to give an initial partition, together with some

optimisation heuristics, including a localised version of the Kerninghan-Lin algorithm [KL70]. However, as mesh and machine sizes grow, the need for parallel mesh partitioning becomes increasingly acute, since an O(N) overhead is simply not scalable. Developing techniques such as parallel mesh generation should also be taken into account. These methods result in meshes which are already distributed among the processors of a parallel machine. If this is the case, then it can be very expensive to transfer the whole mesh back to a single processor for sequential load-balancing. However, if the parallel mesh generation could be done so that load balancing and nearest neighbour communication is achieved, then parallel mesh decomposition could be applied locally.
The creation of super-elements could also be done in parallel. If the initial partition could be done using the original elements of the mesh, then the super-elements could be created in parallel and optimisation could be done locally in parallel using the super-elements. The tidying up of the boundaries could also be done locally in parallel.

Another idea for creating super-elements is to make use of recursive mesh generation. The coarser mesh could be used for the creation of the super-elements and tidying up could be done on the more refined mesh. The parallelisation of CFD codes using adaptively refined grids also needs to be explored and these require automatic load balancing at run-time. After grid-refinement, re-partitioning and re-mapping of the grid can be necessary to obtain again a satisfactory load balance. Therefore, this requires fast (parallel) mesh partitioning and mapping algorithms acting on the already distributed grid. Ideally, these algorithms should take into account the existing mapping of the grid in order to avoid excessive data transport during mapping.

The order of the methods used for decomposition is also very important. As mentioned previously, the time taken to decompose a mesh should be a fraction of the overall solution time. From our experience, the computation time is very large so we would suggest that the decomposition algorithm should be one order of magnitude less to be sufficient. The decomposition achieved could also be stored and used more than once if many runs need to be done.

Investigations also need to be done on whether the overall solution time is dependent on the decompositions. Figures 5.20(a)-(f) shows that the problem can converge into two different minimas (these figures have used a different number of super-elements). If we had the same number of super-elements and used a different initial partition, then it is probable that we would obtain different decompositions. These two decompositions may have the same cost or may have different costs therefore experimental work needs to be done to see whether or not two decompositions would produce similar characteristics. It is likely that the solution would be different.

Finally, the domain decomposition method that has been developed in this thesis is for mapping onto message-passing multiprocessors. A network of computers would also be appropriate. For this type of machines, the communication of shared data is achieved via messages exchanged directly between processors. This requires mapping neighbouring elements onto the same or neighbouring processors so as to avoid large latency. However, the future of parallel architectures include mesh-connected machines such as a Cray T3D or Intel Paragon, which use wormhole routing. This means that a processor can pass a message on without interrupting the work it is doing or slowing down the message too much. The implication is that a message travelling between two processors far apart has hardly any more latency than a message passing between two adjacent processors. If such a machine is used, then we wouldn't have to worry too much about mapping neighbouring elements onto the same or neighbouring processors. It seems that this type of machine is the future of parallel architectures but such machines are extremely expensive to purchase. Because of their high costs, not everyone will be able to afford such machines and will be using the cheaper message-passing multiprocessors or computer networks.

The experience of computing technology has shown that high performance machines that were once used by certain specialist only. However, many of these machines are now being used by a more general group and are therefore becoming more affordable. It would also be a reasonable assumption that this pattern will be repeated for parallel machines and there are many computation-intensive applications today for which parallel processing makes or will make a significant difference.

For these two reasons, there will always be a need for domain decomposition algorithms which insist on achieving nearest neighbour communication.

Appendix A
Further Results

The mesh shown in Figure 6.8 contains 10000 elements and the PARC mesh (supplied by H.Simon) contains 4320 elements. Both these meshes were partitioned (on a Sun SPARC station) into 5, 7, 9 and 12 sub-meshes and the sub-meshes were to be mapped onto a chain of processors.

The 10000 was partitioned using 512 and 256 super-elements and the results can be seen in Tables 1A and 2A.

The PARC mesh was partitioned using 256 and 128 super-elements and the results can be seen in Tables 3A and 4A.

The tables show the communication between the processors i.e. the total number of nodes shared by all processors. The tables also show the time taken to partition the meshes and the times given include the time taken to produce the super-elements.

No of Processors	Communication between processors	Time taken to partition (s)
5	208	66.2
7	332	89.3
9	448	106.7
12	621	132.1

Table 1A: Results for 10000 element mesh using 512 super-elements

No of Processors	Communication between processors	Time taken to partition (s)
5	219	39.0
7	337	66.2
9	450	83.5
12	633	112.7

Table 2A: Results for 10000 element mesh using 256 super-elements

No of Processors	Communication between processors	Time taken to partition (s)
5	71	37.6
7	97	52.9
9	122	66.8
12	161	88.4

Table 3A: Results for PARC mesh using 256 super-elements

No of Processors	Communication between processors	Time taken to partition (s)
5	75	14.2
7	100	22.8
9	132	31.1
12	171	45.9

Table 4A: Results for PARC mesh using 128 super-elements

Looking at the above tables, we can see that there is a difference in the total number of nodes shared for the same mesh using a different number of super-elements and we can see that this varies from 10% to a factor of 2. This difference is relatively small and a saving in the time taken to partition the mesh is made compared to partitioning the meshes without using super-elements (which were measured in hours!).

As mentioned in Chapter 6, the graph bisection method may not necessarily be the best method used to create the super-elements and other methods need to be investigated. This could make a significant difference in the variation of communication costs using different number of super-elements.

REFERENCES

[AM65] G. G. Alway and D. W. Martin. An algorithm for reducing the bandwidth of a matrix of symmetrical configuration. *Computer Journal*, 8:264-272. 1965

[BB87] M. J. Berger and S. H. Bokhari. A partitioning strategy for non-uniform problems on multi-processors. *IEEE Trans. Comp.*, 36(5):570-580. 1987.

[BM76] J. A. Bandy and U. S. R. Murty. *Graph Theory with Applications*. North Holland. 1976.

[Bok81] S. H. Bokhari. On the mapping problem. *IEEE Trans. Comput.* 30(3):207-214 1981.

[BP83] B. R. Baliga and S. V. Patankar. A control-volume finite-element method for two-dimensional fluid flow and heat transfer. *Numerical Heat Transfer* (6):245-261. 1983.

[BP88] B. R. Baliga and S. V. Patankar. Elliptic system: Finite Element Method I. *Handbook of Numerical Heat Transfer*. Edited by W. J. Minkowycz et al, pub Wiley. 1988.

[BS93] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *R. F. Sinovec et al ed., Parallel Processing for Scientific Computing, Siam*:711-718. 1993.

[Car88] A. Carling. *The Transputer and Occam*. Sigma, Wilmslow. 1988.

[CC92] P. Chow and M. Cross. An enthalpy control volume unstructured mesh algorithm for solidification by conduction only. *International Jnl for Numerical Methods in Engineering* (35):1849-1870. 1992.

[CCP93] P. Chow, M. Cross and K. Pericleous. A natural extension of the classical control volume method into unstructured meshes for CFD application.
(In press).

[Chow93] P. Chow. Control volume unstructured mesh procedure for convetion diffusion solidification processes. *PhD Thesis*. University of Greenwich. 1993.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. *Proc 24th ACM National Conference*. Brandon Systems Press, New York:157-172. 1969.

[Cri88] J. M. Crichlow. *An Introduction to Distributed and Parallel Computing*. Prentice Hall, London. 1988.

[Cross92] M. Cross. Towards an integrated control volume unstructured mesh code for the simulation of all the macroscopic processes involved in shape casting. In *Numerical Methods in Industrial Forming Processes (NUMIFORM 92)*:787-792. Edited by J. Chenon et al, pub A. A. Balkema. 1992.

[Duc86] P. G. Ducksbury. *Parallel Array Processing*. Ellis Harwood, Chichester. 1986.

[EK72] J. Edmonds and R. M. Karp. Theoretical improvements in algorithm efficiency for network flow problems. *J. Ass. Comput Mach.*19:248-264. 1972.

[ELJC93] M. G. Everett, P. Lawrence, B. W. Jones and M. Cross. Software tools for aspects of computational modelling codes for materials processing. *Mathematical Modelling for materials Processing*:529-538. edited by M. Cross, J. F. T Pittman and R. D. Wood. Oxford U. Press. 1993.

[ERS88] F. Ercal, J. Ramanujam and P. Sadayappan. Task allocaton onto a hypercube by recursive mincut bipartitioning. In *Proceedings of the 3rd Hypercube Concurrent Computers and Applications Conference*. Pasadena, Ca. Jan 1988.

[Far88] C. farhat. A simple and efficient FEM domain decomposer. *Comp. and Struct.*, 28:579-602. 1988.

[Fei73] M. Fiedler. Algebraic connectivity of graphs. *Czech. Maths. Journal.* 23:298-305. 1973.

[Fei75] M. Feidler. A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czech. maths. Journal.* 25:619-633.

1975.

[FF62] L. R. Ford Jr and D. R. Fulkerson. *Flows in Networks.* Princeton, N.J. Princeton Univ. Press. 1962.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. *IEEE Design Automation Conference,* Las Vegas, Nevada, IEEE Press:175-181. 1982.

[GCCHI92] E.R. Galea, A. Chan, M. Cross, N. Hoffman, C. Ierotheou, S. Johnson and K. Pericleous. Application of aparallel CFD code to large scale practical problems. *Parallel CFD 92*:147-159. 1992.

[Glo89] F. Glover. Tabu search - Part 1. *ORSA J. Comput.* 1:190-206. 1989.

[Glo90] F. Glover. Tabu search - Part 2. *ORSA J. Comput.* 2:4-32. 1990.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* (Freeman, San Francisco). 1979.

[GMG82] Z. Galil, S. Micali and H. Gabow. Priority queues with variable priority and an O(evlogv) algorithm for finding a maximal weighted matching in general graphs. In *Proc 23rd IEEE Symp. on Foundations of Computer Science*:255-261. 1982.

[GZ87] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *Int Journal Parallel processing.* 16(6):427-449. 1987.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Institute of Physics Publishing, Bristol. 1988.

[JAMS89] D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Scheron. Optimisation by simulated annealing; An experimental evaluation; Part 1, Graph partitioning. *Operational Res.*37:865-892. 1989.

[JC92] S. P. Johnson and M. Cross. Mapping structured grid 3D CFD codes onto parallel architectures. *Appl. Math. Modelling*. 15:394-404. 1992

[JCIL93] S. P. Johnson, M. Cross, C.S. Ierotheou. CAPTools- Computer Aided Parallelisation Tools. *Technical Description.* University of Greenwich. 1993.

[JH89] S. L. Johnson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comp.* 38:1249-1268. 1989.

[JMCEJ93] B. W. Jones, K. McManus, M. Cross, M. G. Everett and S. Johnson. Parallel unstructured mesh CFD codes: A role for recursive clustering techniques in mesh decomposition. *Parallel CFD 93*, North Holland (1993).

[KGV83] S. Kirkpatrick, C. D. Gelatt Jr and M. P Vecchi. Optimisation by simulated annealing. *Science*. 220:671-680. 1983.

[KL70] B. W. Kerninghan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. Journal.* 49:291-307. 1970.

[LA87] S-Y. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. Comp.* C-36(4):433-442. 1987.

[Law76] E. L. Lawlor. *Combinatorial optimisation.* Holt, Rinehart and Winston, New York. 1976.

[LC91] A. Lim and T-M. Chee. graph partitioning using tabu search. *IEEE International Symposium on Circuits and Systems.* 2:1164-1167. 1991.

[Lo88] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comp.* 37(11):1384-1397. 1988.

[LW89] R. D. Lonsdale and R. Webster. The application of finite volume methods for modelling three dimensional imcompressible flow on an unstructured mesh. *Methods in Laminar and and Turbulent Flow.* Edited by C. Taylor et al, pub Pineridge Press: 1615. 1989.

[MCJ94] K. McManus, M. Cross and S. Johnson. Integrated flow and stress on an unstructured mesh. *Parallel CFD 94*. (In press).

[Moh88] Eigenvalues, diameter and mean distance in graphs. *preprint series Dept math No 259,* University E. K. of Ljubljana, Yugoslavia. 1988.

[PSL90] A. Pothen, H. D. Simon and K-P Liou. Partitioning sparse matrices eith eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.* 11(3):430-452. 1990.

[RSEHP92] K.G. Reinsch, W.schmidt, A. Ecer, J. Hauser and J. Periaux. *Parallel CFD 91*.North Holland, Amsterdam. 1992.

[RC82] C. M. Rhie and W. L. Chow. A numerical study of the turbulent flow past an isolated airfoil with trailing edge seperation. *AIAA Journal* 21:1525-1532. 1982.

[Ros68] R. Rosen. Matrix bandwidth minimization. *Proceedings of 23rd National Conference ACM*, ACM Publication P-68, Brandon/Systems Press, Princeton, N.J.:585-595. 1968.

[Sch88] G. E. Schneider. Elliptic System: Finite element method II. *Handbook of Numerical Heat Transfer*. Edited by W. J. Minkowycz et al, pub Wiley. 1989.

[Sim91] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*. 2(2/3):135-148. 1991.

[SE87] P. Sadayappan and F. Ercal. Nearest neighbour mapping of finite element graphs onto processor meshes. *IEEE Trans. Comput.* 36(12):1408-1424. 1987.

[SER90] P. Sadayappan, F. Ercal and J. Ramanujam. Cluster partitioning approaches a mapping parallel programs onto a hypercube. *Parallel Computing*. 13:1-16.1990.

[Sin87] J. B. Sinclair. Efficient computation of optimal assignment for distributed tasks. *J. Parallel and Dist. Comput.* 4(4):342-362. 1987.

[ST85] C-C. Shen and W-H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Trans. Comp.* 34(3):197-203. 1985.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Soft. Eng.* SE-3(1):85-93. 1977.

[TZTS92] L. Tao, Y. C. Zhao, K. Thulasiraman and M. N. Swamy. Simulated annealing and tabu search algorithms for multiway graph partition. *Jnl of Circuits, Systems and Computers . 2(2):159-185. 1992.*

[VSB92] V. Venkatakrishnan, H. D. Simon and T. J. Barth. A MIMD implementation of a parallel Euler solver for unstructured grids. *Journal of Supercomputing* 6(2):117-127. 1992.

[WCJE94] C. Walshaw, M. Cross, S. Johnson and M. Everett. A parallelisable algorithm for partitioning unstructured meshes. In *Proc. Irregular '94: Parallel Algorithms for Irregularly Structured Problems.* 1994.

[Wil85] Robin J. Wilson. *An introduction to graph theory.*Longman, London. 1985.

[Wil91] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency:Practice and Experience.* 3(5):457-481. 1991.

[Zie77] O. C. Zienkiewic. *The Finite Element Method.* Mc Graw-Hill UK. 1977.