Parallel solution of power system linear equations

Grey, David John

Parallel Solution of Power System Linear Equations

David John Grey
B.Eng. (York)

School of Engineering and Computer Science
University of Durham

A thesis submitted in partial fulfilment of the requirements of the Council of the University of Durham for the Degree of Doctor of Philosophy (Ph.D.).

February 1995

Abstract

At the heart of many power system computations lies the solution of a large sparse set of linear equations. These equations arise from the modelling of the network and are the cause of a computational bottleneck in power system analysis applications. Efficient sequential techniques have been developed to solve these equations but the solution is still too slow for applications such as real-time dynamic simulation and on-line security analysis. Parallel computing techniques have been explored in the attempt to find faster solutions but the methods developed to date have not efficiently exploited the full power of parallel processing.

This thesis considers the solution of the linear network equations encountered in power system computations. Based on the insight provided by the elimination tree, it is proposed that a novel matrix structure is adopted to allow the exploitation of parallelism which exists within the cutset of a typical parallel solution. Using this matrix structure it is possible to reduce the size of the sequential part of the problem and to increase the speed and efficiency of typical LU-based parallel solution. A method for transforming the admittance matrix into the required form is presented along with network partitioning and load balancing techniques.

Sequential solution techniques are considered and existing parallel methods are surveyed to determine their strengths and weaknesses. Combining the benefits of existing solutions with the new matrix structure allows an improved LU-based parallel solution to be derived. A simulation of the improved LU solution is used to show the improvements in performance over a standard LU-based solution that result from the adoption of the new techniques. The results of a multiprocessor implementation of the method are presented and the new method is shown to have a better performance than existing methods for distributed memory multiprocessors.

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree, and that all sources of information have been duly acknowledged.

This thesis is dedicated to Anna, my wife and best friend.

Acknowledgments

The following people have been vital to the production of this work; either in their direct advice and input or just in putting up with me whilst I was going crazy writing it.

- To my wife, Anna - for her love and support.
- To my supervisor, Doctor Janusz Bialek of the University of Durham – for his direction and advice.
- To Kelvey Marden of the University of Durham – for friendship.
- To Alan, Alex, John, Raghu, Jeremy, Juliette, Sue, Matt, Chris, Phil, Howard and Hayley – for companionship and a good laugh when in need.
- To Alan - with grateful thanks for all the proof reading

List of Abbreviations

BBDF	Bordered Block Diagonal Form
CEGB	Central Electricity Generating Board (now the National Grid Company)
CSP	Communicating Sequential Processes
FPU	Floating Point Unit
IBM	International Business Machines
IEEE	Institute of Electrical & Electronic Engineers
I/O	Input and Output
MD	Minimum Degree
MDML	Minimum Degree Minimum Length
MDMLLRU	Minimum Degree Minimum Length Least Recently Used
MFLOPS	Million Floating point Operations per Second
MIMD	Multiple Instruction stream Multiple Data stream
MISD	Multiple Instruction stream Single Data stream
ML	Minimum Length
MLMD	Minimum Length Minimum Degree
RAM	Random Access Memory
RBBDF	Recursive Bordered Block Diagonal Form
RISC	Reduced Instruction Set Computer
RP	Recursively Parallel
SIMD	Single Instruction stream Multiple Data stream
SISD	Single Instruction stream Single Data stream
TRAM	Transputer Application Module

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern electrical power systems represent some of the largest non-linear systems in existence. Today's power systems are complex interconnected networks consisting of many thousands of nodes and the analysis of these systems and planning their expansion and performance is no simple task. The modern power engineer is forced to rely upon powerful software tools to enable him to get the best out of the system. As systems continue to grow more complex and engineers wish to gain deeper insight into their workings there is a continually growing demand for faster and more powerful software analysis tools. Many such tools have been developed over the past four decades but there is always more that can be done. This thesis focuses on the solution of one of the problems which currently limits the performance of many power system analysis applications. Solving this problem surmounts an important obstacle in the creation of more efficient analysis tools.

1.1 The Components of a Power System

Gross [1] defines an electrical power system as

"a network of interconnected components designed to convert non-electrical energy continuously into the electrical form; transport the energy over potentially large distances; transform the electrical energy into a specific form subject to close tolerances; and convert the electrical energy into a usable non-electrical form."

This definition shows that the power system consists of three basic components

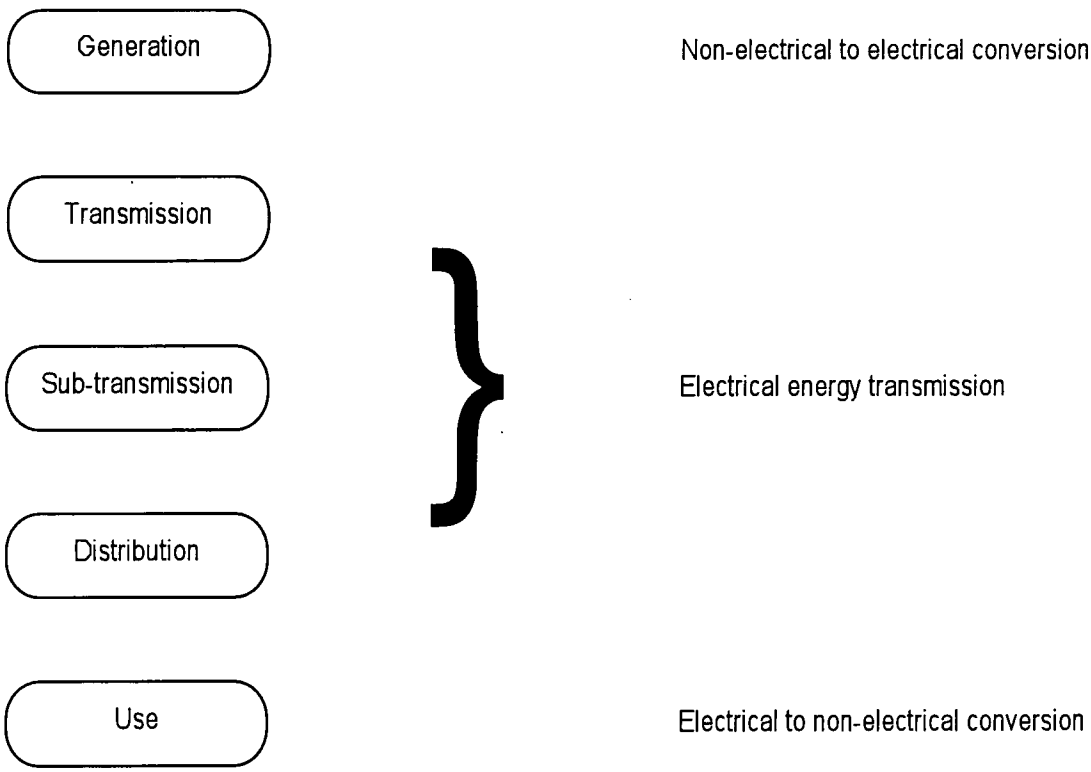


Figure 1.1: The components of a modern power system

- Non-electrical to electrical energy conversion
- Electrical energy transmission
- Electrical to non-electrical energy conversion

In fact this can be subdivided further and the modern power system thus has the five facets shown in Figure 1.1

The generation process turns non-electrical kinetic energy of a rotating shaft into electrical energy. Kinetic energy is imparted to the shaft by some form of turbine. Steam, gas and hydro turbines are the most common prime movers, although wind turbines are making an increasing contribution to electricity generation. The generator itself makes use of the principle of electromagnetic induction to generate three sinusoidal alternating voltages which differ in phase by 120°. This is the most efficient form for electrical energy transmission as it makes best use of the transmission cables and give constant power in balanced loads. The aim of generation is simply to pump enough electrical energy into the system to satisfy the demand of the end users and to account for the losses which occur in transmission.

Strictly speaking the transmission network takes power from substations at the generating sites and delivers it to substations in the load centres. The generation sites are often remote from areas of load for many reasons. Originally generators were located where there is a ready supply of fuel *e.g* near coal fields, but such areas are seldom areas of electricity demand. As it is cheaper and easier to transport the electrical energy to areas of demand, power stations have tended to be sited near the fuel supplies. In some cases this is essential as the 'fuel' cannot be moved *e.g.* hydropower and wind power. Government policy in the UK has also affected the location of power generation plant with current legislation dictating that all fossil fuel burning power plant have to be sited outside urban areas. Consequently there is a need for transmitting electrical energy from generators to loads in the most efficient way possible. The UK national grid is a typical transmission system consisting of transmission lines with a total length exceeding 7,500 kilometres. This network of lines provides a high degree of interconnectivity between the sites of generation and the major load centres. It also ties the UK system into other continental systems to increase the reliability of the system and the availability of power. The transmission system has to be able to deliver large amounts of power to the loads and to ensure reliability it has to be able to deliver this power by a number of alternative routes. The process of transmitting electrical energy is not 100% efficient and the losses are proportional to the square of the current, making it more efficient to transmit power at high voltage and low current. The UK grid developed in three distinct phases and it now operates at three different voltage levels of 132 kV, 275 kV and 400 kV. Transmission voltage levels vary throughout the world but they all lie in the range 100 kV to 1MV.

The distinction between subtransmission and distribution is not altogether clear but is dependent upon voltage levels and geographical extent. Subtransmission systems take electrical power from the point of arrival in the load area and deliver it to a number of distribution substations throughout this area. Subtransmission operates at much lower voltages than transmission, with 11kV and 33kV being the typical voltages used in the UK. As well as the lower voltages, subtransmission systems also have a much smaller geographical extent. Transmission systems extend to cover countries and whole continents whereas subtransmission systems are usually limited to the extent of a given urban area.

The distribution network is the final link in the chain between the generator and the user. Distribution is distinguished from subtransmission by its lower voltage levels with 12kV down to 2.4kV being common. The distribution system takes power from the distribution

substation and forwards it to individual users who are located within a short distance (2km) of the substation.

The purpose of transmitting electrical power through the various stages of the network is to supply the end user with electrical energy. Upon receipt of this energy the user converts it back into some more usable non-electrical form using any one of the electrical appliances available to him. To ensure that his appliances work correctly the user expects his electricity supply to have constant voltage magnitude, constant voltage frequency and an ideal sinusoidal waveform. To meet these criteria the power companies specify three performance measures to which they must adhere

- Voltage regulation - the deviation of voltage magnitude as load varies. Typically this is around 5%.
- Frequency regulation - the deviation of system frequency from the nominal value. For a 50Hz system this is typically ± 0.1Hz.
- Harmonic content - The ideal supply has only a single sinusoidal component.

There is a complex control system at the heart of every power system which allows the system operators to control these three parameters so that the user's demands on the quality of his electricity are satisfied. Controlling such a large and complicated system is no simple problem and to do so efficiently requires a detailed understanding of the system and how its constituent parts interact with one another.

1.2 Power System Analysis

Modern power systems are extremely large and complex entities. Planning, maintaining and operating such a system would be difficult if it were not for the wide range of analytical methods available to help the power engineer. Today there is a wealth of software analysis tools to help in understanding any conceivable aspect of the power system. Prior to 1940 there were very few interconnected systems of any complexity and analysis methods existed only for dealing with generators, transformers and transmission lines. With the advent of interconnected systems came the development of techniques which enable the engineer to determine the electrical state of the network and how it would respond to given disturbances (*i.e* load flow, stability analysis). Further work has led to the development of power flow programs, contingency analysis tools and economic analysis packages. Requirements for

low operating and construction costs and the ever increasing complexity of systems have led to a constant demand for powerful new automated analysis methods. Software tools are available to aid analysis in the areas of operations planning, systems planning, contingency and security analysis, reliability and economics. Several of these areas are worth considering in more detail but first the power flow problem must be examined.

1.2.1 The Power Flow Problem

The applications outlined in the previous section are all limited in their performance by the time taken to solve the *power flow* equations for the system under consideration. At the heart of the power flow solution lies a large sparse set of linear network equations. It is the solution of these equations which is so time consuming and over the last 35 years much research effort has been expended on improving and accelerating this solution. The set of algebraic equations is of the form

$$\mathbf{Ax} = \mathbf{b} \tag{1.1}$$

where $\mathbf{A}$ is the matrix of coefficients of the set of equations. $\mathbf{b}$ is a vector of known values whilst the elements of vector $\mathbf{x}$ are the unknowns in the equations. One of the earliest and most successful improvements came through the application of sparse matrix techniques. The matrix, $\mathbf{A}$, has few non-zero elements and many zero elements; often more than 95% of the matrix elements are zero. It is wasteful both of memory and solution time to store the zero elements of this sparsely populated matrix. Sparse matrix techniques are used to store and process only the non-zero coefficients and this gives a significant increase in the speed of solution [2, 3].

1.2.2 Power System Simulation

Like many other simulations, power system simulation involves an iterative process of solving the equations which model the system. On each iteration the simulation time is incremented and the system equations must be solved again for this new time step. If the time for one iteration is less than the time constant of the fast dynamics of the system then the simulation can be considered real-time. The integration time step of a typical power system simulation is of the order of one second. This is sufficient to allow all but the very fast (sub-transient) dynamics to be modeled. If the simulation is to operate in real-time it is necessary to solve all the mathematical equations which model the system more than once a second if real-time

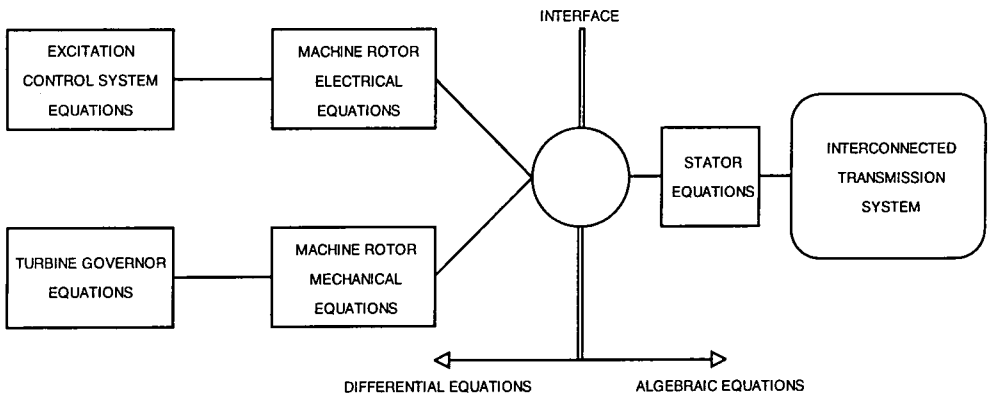


Figure 1.2: A model of a synchronous generator connected to the transmission network

simulation is to be achieved.

A power system simulation consists of two distinct sets of equations relating to the different parts of the system. Firstly there is the network model which describes the transmission network of the system and the stators of all the machines connected to the system. The network model consists entirely of algebraic equations. Secondly there is a set of non-linear machine models which may be broken down into first order non-linear differential equations that describe all the generators and loads connected to the network. On each iteration the differential equations must be solved to determine the currents which are injected into the transmission network. The linear equations must then be solved to determine the power flows around the system and the voltage at each system bus. Figure 1.2 depicts a single machine connected to the transmission network and it shows how the model is divided into a set of differential equations and a set of algebraic equations.

The differential equations are of the form

$$\dot{y} = f(y, x) \tag{1.2}$$

and this set of equations contains the differential equations of every machine connected to the network. Each machine in the system is only coupled to other machines through the transmission network and, as the network is treated separately, the differential equations are a collection of uncoupled sets of equations, one set for each machine. The equations may be represented as [4]

$$\dot{y} = f(y, u) = \mathbf{A}y + \mathbf{B}u \tag{1.3}$$

where $\mathbf{A}$ is a sparse, square, block diagonal matrix and $\mathbf{B}$ is a rectangular sparse matrix

with a block structure. When the effects of saturation are neglected **A** and **B** become constant in most models. Chapter 2 and Appendix B discuss the mathematical models of machines in detail.

The algebraic equations are of the form

$$\mathbf{y} = g(\mathbf{x}, \mathbf{u}) \tag{1.4}$$

where **u** is the vector of stator voltages and is the input to the equations whilst **y** is a vector of currents. The equations may be separated into two parts

$$I(\mathbf{E}, \mathbf{V}) = \mathbf{YV} \tag{1.5}$$

and

$$u = u(\mathbf{E}, \mathbf{V}) \tag{1.6}$$

where **Y** is the bus admittance matrix, **V** is the vector of terminal voltages at load busses and **E** is the vector of stator voltages at generation busses. **I** is the vector of bus injection currents. Injection current at a generator bus is a function of stator voltage whilst the injection current at a load bus is a function of the terminal voltage. It is important to note that coefficients of **Y** depend on the topology of the network and these coefficients may vary. If the topology of the transmission network changes due to equipment outages then the values in the matrix **Y** are changed. When auto-tap changing transformers are represented **Y** can change frequently.

For power system simulation the problem is to solve the differential and algebraic equations simultaneously. The conventional approach is to solve (1.3) separately by integration to yield values for y. Equation (1.5) is then solved and the solutions are alternated in some manner. Another approach is to solve (1.3) and (1.5) simultaneously using the implicit trapezoidal rule. The solution of (1.5) is the bottleneck in the solution process and three methods of solution exist.

1. *Gauss-Seidel* - This method is simple to program and easily accommodates changes in **Y** as the method operates directly on admittance matrix values [4]. The method is iterative and convergence to acceptable accuracy varies from problem to problem. Simple problems may converge in 2 or 3 iterations whereas difficult problems may require hundreds of iterations

2. *Factorisation of* **Y** - Triangular factorisation of **Y** is a direct method for solving the algebraic equations. LU decomposition is used to yield the factored form of **Y** and

$$\mathbf{I} = \mathbf{L}.\mathbf{U}.\mathbf{V} \tag{1.7}$$

where **L** is a lower triangular matrix and **U** is an upper triangular matrix. Forward and backward substitution with **I** is used to provide a solution for **V**. If **Y** does not change then **L** and **U** remain constant and **V** may be obtained by repeat solution of (1.7). Any change in **Y** requires a complete refactorisation to yield new values of **L** and **U**, before (1.7) can be solved for **V**. It is observed that factorisation takes up to six times as long as forward/backward substitution [4] but substitution only takes about 1.5 times as long as a single iteration of the Gauss-Seidel method and it is difficult for Gauss-Seidel to be competitive.

3. *Newton's Method* - Newton's Method is also an iterative method for solving systems of non-linear equations. Equation (1.5) becomes non-linear when non-impedance loads are connected to the network. Under these conditions the relationship between node voltage and bus current is non-linear and is defined by the impedance characteristics of the load. Newton's method linearises the equations using a truncated Taylor series approximation and forms the Jacobian matrix **J** [1]. Equation (1.5) can be rewritten as

$$F = \mathbf{I} - \mathbf{Y}.\mathbf{V} \tag{1.8}$$

and $F = 0$ when the correct solution of **V** is obtained. The solution is obtained by iterative correction and each iteration requires the solution of the Jacobian matrix equation

$$F = -\mathbf{J}.\Delta\mathbf{V} \tag{1.9}$$

LU factorisation and triangular substitution may be used to give a direct solution to this equation on each iteration. A strict implementation of Newton's method requires **J** to be updated and factorised for each iteration but this is computationally intensive. Faster solutions are achieved by allowing the triangular factors of **J** to be used for several consecutive iterations.

The choice of the models used in the simulation has implications for the method chosen to solve the network equations. If there is no saliency, saturation or non-impedance loads

in the model then the injection currents, **I**, are a function of stator voltage, **E**, only. An exact solution for the node voltages, **V**, can rapidly be obtained using (1.7). Introducing machine saliency leads to a constant term being introduced into the admittance matrix **Y**. To compensate for this a corrective term is added to **V** and this term is a function of **I**. It is necessary to iterate repeat solutions of (1.7) until **V** achieves convergence. The representation of non-impedance loads similarly leads to a portion of each load being represented by a shunt admittance in **Y**. The remainder of the load appears as a non-linear function of **I** in **V**. Again it is necessary to iterate repeat solutions of (1.7) until **V** achieves convergence.

1.2.3 Power System Security

In power system engineering terms security is defined as the

'ability of the system to withstand any one of a predefined list of possible contingencies without serious consequences'

where a contingency is an interruption in the normal functioning of the network caused by objects in the environment. The objective of power system operators is to maintain system voltages and power flows within defined limits regardless of changes in generation and load. Operating interconnected systems requires strict control over synchronisation; any loss of synchronisation may be catastrophic. Equipment outages themselves seldom cause much damage but the readjustment of voltages and power flows throughout the system may lead to a dangerous cascade of overloads causing large sections of the system to be switched out or damaged. In the past system security has been assured through the construction of robust systems. Rising costs and environmental concerns have meant that it is no longer economically or environmentally feasible to build extremely robust systems. As a result systems are being operated closer to their limits with smaller safety margins leading to a greater exposure to unsatisfactory recovery following disturbances. System security is no longer seen as a systems planning concern but as an exercise in risk aversion which is controlled by the system operators. There is therefore a pressing need for operators to keep a close eye on the security of the system.

Assessment of system security is performed by determining the probability that the system will move from its normal operating conditions into an abnormal, or emergency state. These calculations are based upon a knowledge of the current state of the system, the conditions at the time and a forecast of load demand. In order to determine the response

of the system it is necessary to apply these data to a model of the system and this is achieved using computer simulation. Contingency analysis packages attempt to determine what the response of the system will be to any of a list of possible contingencies, based upon a knowledge of current system state. A full analysis requires a simulation of the system for each combination of the contingencies in the list.

Many utilities operate their control systems in conjunction with on-line security monitoring and contingency analysis tools. Operators can continually monitor the state of system security and how that might be changed by future events (the contingencies). Using contingency analysis on this basis allows operators to determine what is likely to happen to the state of the system and take the appropriate preventative, or corrective, action as necessary. The many thousands of cases that have to be considered for a full contingency analysis make it prohibitively slow for on-line usage. Many techniques have been developed which reduce the list of contingencies to include only those which are most likely to occur. This reduces the time needed to perform an analysis and makes on-line contingency analysis possible. Such packages are still slow requiring between 5 and 15 minutes [5] to generate their results. If the contingency analysis package takes more than 15 minutes to complete its calculations then it is not worth using it as its predictions are based upon a model of the system which was updated too long ago for it to be accurate. Due to the high volume of calculations and the speed required, contingency analysis tools usually have to be executed on expensive, dedicated, high performance computers.

Many of the current contingency analysis tools only consider the steady state system and ignore the dynamic response of the system to transients. Dynamic analysis is possible but this requires many more calculations than the steady state analysis and as yet is too slow for on-line operation. Future developments would ideally make real - time dynamic analysis packages available to operators giving them much faster response to contingencies and improving standards of system security. With the increasing speed and capacity of modern computers it should soon be possible to develop such tools using the techniques of parallel and distributed computing in conjunction with the latest generation of high performance processors.

1.2.4 System Planning

System planners are continually seeking ways to improve and expand the current system. Analysis tools can be useful in this work by allowing the designer to perform 'what-if' studies

on various designs. Such studies are performed using computer models of the systems to simulate what happens to the systems when various changes are made. As system planning is not an on-line application there is no pressing need for real - time simulation but continued improvements in the speed of the tools used by system planners will manifest themselves in a reduction in the design to construction time of system improvements. As a final stage in the design assessment it may be necessary to undertake a real - time dynamic simulation to determine the performance of the new design.

1.2.5 Operator Training

The flow of power in the power system, or part of the system, is controlled from a central control room by trained operators known as dispatchers. The dispatchers meet the demand from users by controlling the flow of power around the network and have to deal with emergencies such as sudden changes in load demanded or equipment outages. New dispatchers must be trained to operate the control room mechanisms and to handle the various emergency situations that may arise. Trained dispatchers need to continually improve their skills, especially those relating to potentially catastrophic emergencies which seldom occur. It would be foolhardy to let trainees practice on the real system as an error in their responses could place the system into a potentially unstable state. Recent advances in reliability have meant that it is difficult to acquire the skills needed to cope with all conceivable operating conditions within the actual operating environment [6]. Some form of simulated environment is the most effective method of providing the necessary training. Training with a simulator gives the dispatcher greater confidence to take the correct actions when faced with the same situation in the actual system. To be most effective it is necessary for the simulator to provide the same interfaces to the system as the control room provides. This requires the development of a control room mock-up in which all interaction with the simulator is made through the same computers, screens and other hardware used in the real control room. Due to their effectiveness dispatcher training simulators are now an integral part of the energy management systems of many power utilities [7].

All dispatcher training simulators have two main components; a control centre model and a power system model. The control centre model provides all the main functions of a real dispatch control centre including data acquisition, supervisory control, system monitoring and man-machine interfaces. The system model provides an equivalent model of the system being controlled which must be sophisticated enough to realistically reproduce the responses

of the actual system. This requires the model to accurately simulate the dynamics of the system. Two distinct components of the system model can be identified in all dispatcher training simulator designs [6, 7, 8] - the static model and the dynamic model.

- Dynamic model - Models the system components to provide dynamic simulation of generators, loads, prime mover systems, protective relays, substation controls *etc.*
- Static model - Models the power system network and provides the power flow solution, network topology analysis, system frequency deviation and transient stability calculations.

Biglari [7] and other researchers observe that it is the solution of the power flow equations which is the most time consuming stage in the power system model. Furthermore, the solution time of the power flow equations directly determines the iteration cycle time of the power system model component of the simulator. Although fast decoupled solution techniques are employed the time taken to solve for power flow equations is significant. The long solution times mean that it is not possible to accurately represent the responses of the system's fast dynamics.

1.3 Power Systems Analysis and Computer Architectures

Digital computer analysis of power systems developed in the 1960's when the availability of large digital computers made it feasible to use such technology as a fast and flexible tool for modeling power system behaviour. Early research into computer analysis techniques quickly highlighted the disparity between problem size and the capabilities of the computer technology available at that time [9]. Researchers concentrated on developing highly efficient algorithms which would extract the maximum performance from machines which were primitive and limited in their performance. As computers have become more powerful, engineers have exploited this increased power by creating analysis tools to run on these machines. With the development of extremely high performance supercomputers the software designers have naturally looked toward these machines to see what benefits they have in store for power engineering applications. Whilst they are extremely efficient and offer incredible performance these machines are expensive and beyond the budget of most power utilities.

As the sequential computer nears the limits of its performance much attention has been

focused on the development of parallel computers which offer higher performance from existing technology than conventional sequential machines. These computers are built from tried and tested processors and achieve their performance by allowing many operations to be performed simultaneously, thereby reducing the total amount of time taken to execute a given algorithm. Parallel computers are not new; they have been in existence since the digital computer was first developed. Given their performance benefits it may seem surprising that they have not achieved widespread usage in commercial, scientific and engineering applications. This apparent unpopularity is due to the lack of commercial software and the difficulties of developing applications software. Whilst it may be argued that concurrency is a more natural and logical way of solving problems, a different approach is required for the development of parallel programs than is used in developing sequential programs. Many programmers used to sequential machines are used to the von Neumann model and find it difficult to work with the different architectural models and programming paradigms associated with parallel machines

In recent years a number of smaller 'turnkey' parallel computing systems have arrived in the marketplace (*e.g.* Intel iPSC, Meiko Computing Surfaces and INMOS Transputer based systems). These systems are built from readily available microprocessors which makes them cheap and accessible to a wide user base. Even in 1988 for as little as £15,000, the price of a mid-range workstation, it was possible to buy an off-the-shelf parallel system with more than 8 processors and a performance approaching 100 MFLOPS [10], which exceeded the performance of the similarly priced workstation. With the introduction of such systems the software developers have seen a market niche opening up. Software manufacturers have expanded to fill this niche and there are now a variety of operating systems available for these entry level parallel systems, including the ever-popular Unix. The development of parallel versions of traditional sequential languages (*e.g.* C and FORTRAN) along with the introduction of new parallel languages (*e.g.* Occam) has eased the process of developing software applications. In the past one criticism that has been levelled at parallel computers is that algorithm development and hardware architecture are too deeply intertwined to be treated independently. The advent of high-level 'parallel' languages has seen a move toward abstracting the logical development of programs away from the architectural details of the machine. As these systems fall in price and more software is developed, more details of the hardware become hidden from the programmer making the machines easier to program.

It is a generally held belief that parallel computers will become significant in many

scientific and engineering applications. It is already proven that using readily available technology simple parallel computers can exceed the performance of high performance sequential machines. Trew [11] and Sabot [12] observe that parallel computers tend to be much more cost effective than serial machines with the same level of performance. With the advent of cheap off-the-shelf parallel systems it seems that power engineers may have machines which can satisfy the performance requirements of their software applications at reasonable cost. Many of the developments in uniprocessor computers that have occurred in the last 20 years (*e.g.* pipelining, coprocessors) have occurred as a result of the application of techniques used in parallel computer systems [10, 13, 14, 15]. If research into parallel machines continues to increase their performance it is likely that they will satisfy the performance requirements of scientific and engineering applications for some time into the future. Whilst parallel machines are today primarily the domain of researchers, many experts predict that by 1998 they will be serious contenders to the industrial/commercial domination of the conventional von Neumann machine.

1.4 Parallel Processing and Parallel Architectures

Almasi and Gottlieb [13] define parallel processing as

'A large collection of processing elements that can communicate and cooperate to solve large problems fast'

Parallel computers achieves high performance by using multiple processors to solve the independent parts of a problem concurrently. If correct results are to be produced then the individual processors must cooperate with one another by exchanging data and synchronizing their operations. An important feature of every parallel machine is the ability of processors to communicate with each other other and this is provided by an interconnection network which links the processors. Many different type of interconnection network exist but all involve some trade-off between cost and performance.

1.4.1 Classification of Computer Architectures

Before discussing parallel architectures it is worth reviewing the von Neumann model [11] of sequential computation as this is the model upon which many of today's computer systems are based. In the von Neumann model program instructions and program data are both stored in a common memory connected to a single processor, giving the von Neumann

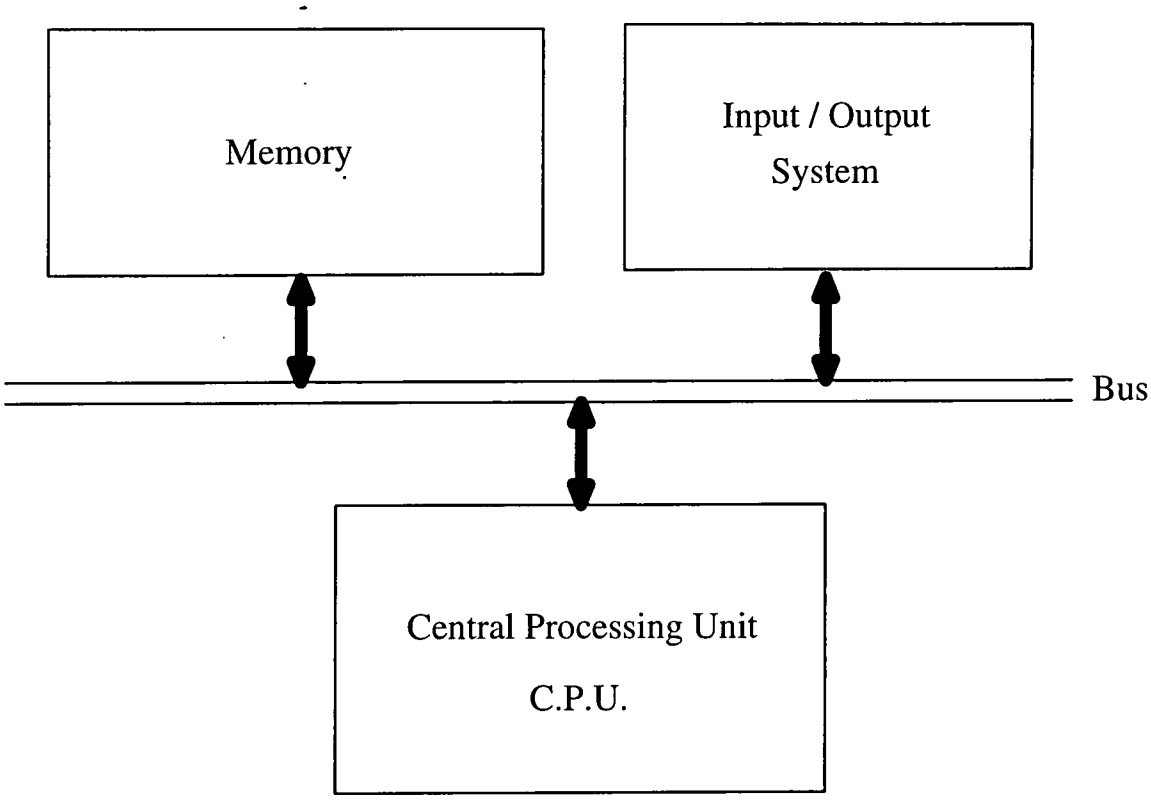


Figure 1.3: The basic von Neumann machine

machine the architecture of Figure 1.3. The flow of instructions and data is shown in Figure 1.4. Instructions are executed sequentially with instruction operands and data being fetched from memory, operated upon by the Arithmetic Logic Unit (ALU) and data returned back to the memory. The model employs a single stream of instructions and a single stream of data.

In 1966 Flynn [16] proposed his scheme for classifying computer architectures which has become known as Flynn's Taxonomy. Flynn classifies computer architectures according to the number of instruction (I) and data (D) streams that they have and this yields the four classes of computer shown in Table 1.1.

	Single data stream	*Multiple data stream*
Single instruction stream	SISD (von Neumann)	SIMD (Array/Vector processor)
Multiple instruction stream	MISD	MIMD (*true* multiprocessor)

Table 1.1: Flynn's Taxonomy.

SISD computers are conventional von Neumann, sequential machines such as PC clones based on Intel 80x86 processors. This class also includes uniprocessor supercomputers such

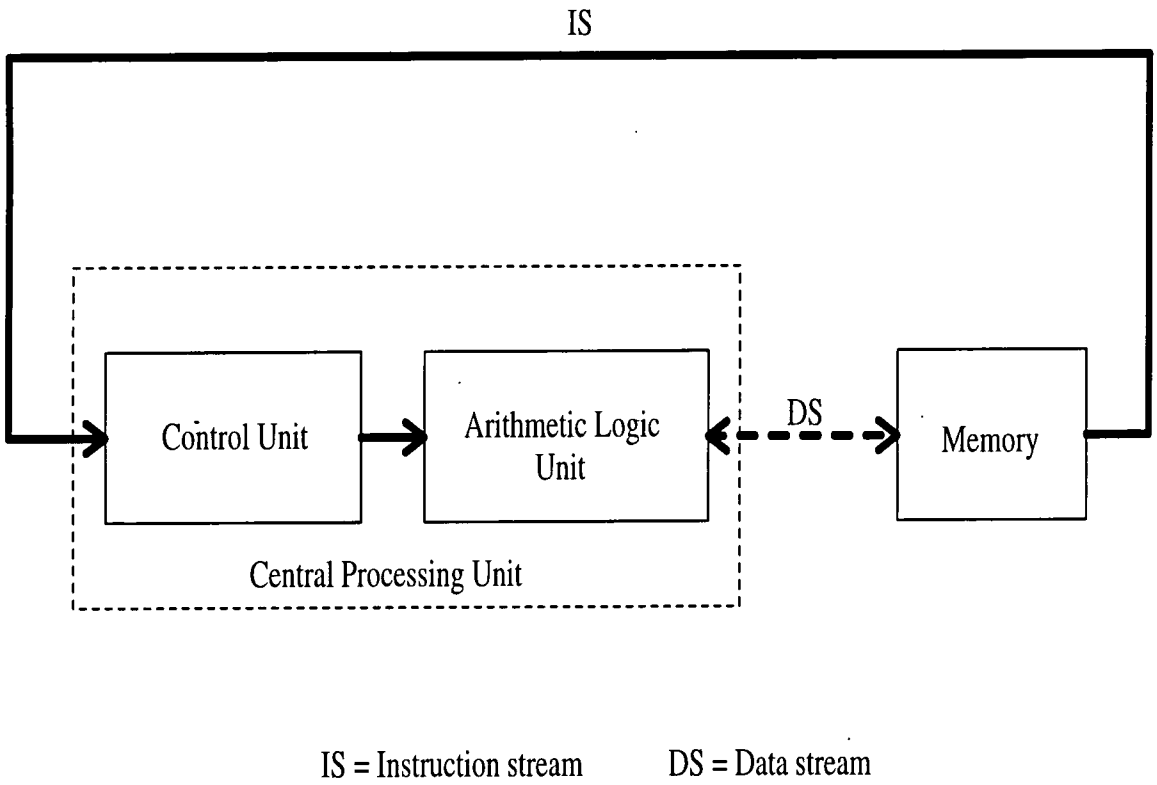


Figure 1.4: Instruction and data streams in the von Neumann machine

as the Cray 1 which achieves high performance through pipelining of instructions. The remaining three categories refer to different types of parallel computer.

SIMD computers are parallel computers which consist of multiple processors controlled by the same control unit. Each receives the same instruction from the controller and executes it on a different data stream in synchronous lockstep. The vector processors used by vector supercomputers are considered as SIMD devices as the processing of vector quantities is performed by multiple ALU's connected in an array processing manner. SIMD machines built from multiple CPU's are in existence - the CPU's are interconnected by a data routing network in the form of a regular array. Such machines are useful for problems which exhibit a high degree of parallelism although they are difficult to program and are application specific.

Flynn's classification of a MISD machine sees autonomous processors executing different instructions on the same stream of data. The data flows between the processors in a pipelined fashion. No computer has yet been identified as falling into this category and many researchers claim that a MISD machine is purely conceptual as it is difficult to visualise an application for which it would be useful.

The MIMD category encapsulates a wide variety of multiprocessor and multicomputer systems. Multiple processing elements autonomously execute different instructions on different streams of data. The MIMD classification is an extremely wide one covering asynchronous arrays of microprocessors through to distributed multicomputer systems. The off-the-shelf Transputer systems supplied by companies such as Meiko are prime examples of the sort of MIMD machines which are widely used by scientific researchers.

1.4.2 SIMD Architectures

The SIMD paradigm [17, 18, 13] consists of interconnected processors which receive their instructions from a central control unit. The interconnection network allows for communication between individual processors and between processors and local memory. All SIMD machines are variants of *array processors.*

Vector processors are considered array processors due to the arrays of ALU's used to process individual elements of vector operations in parallel. Some degree of pipelining is often used in vector processors and this incurs a significant start-up overhead. Due to the presence of these pipelines vector processors are only efficient if their pipelines are always full.

Systolic array architectures are processor arrays in which the processing elements are extremely simple and perform an invariant sequence of primitive operations. Data is pumped through the network from the memory and returns to the memory after processing. Flow of data through the network is synchronised by a global clock and data appears to pulse through the network in a similar manner to blood flowing through the heart. Systolic arrays are well suited to intensive computations on regular data. Suitable applications are invariably algorithm specific.

Despite their lack of generality SIMD architectures have some advantages over more flexible MIMD architectures. The synchronous nature of SIMD machines eliminates the delays associated with synchronisation and the need to wait for the slowest processor. The single instruction stream allows the use of a common instruction memory and does not require local replication of parts of the program, as in a MIMD architecture. This gives SIMD a much higher memory efficiency compared to MIMD. The single instruction scheme also fixes the interleaving of operations, unlike the MIMD paradigm which guarantees *some* interleaving of operations although it is not possible to determine which of the possible interleavings will occur. The guaranteed order of instruction execution makes SIMD programs

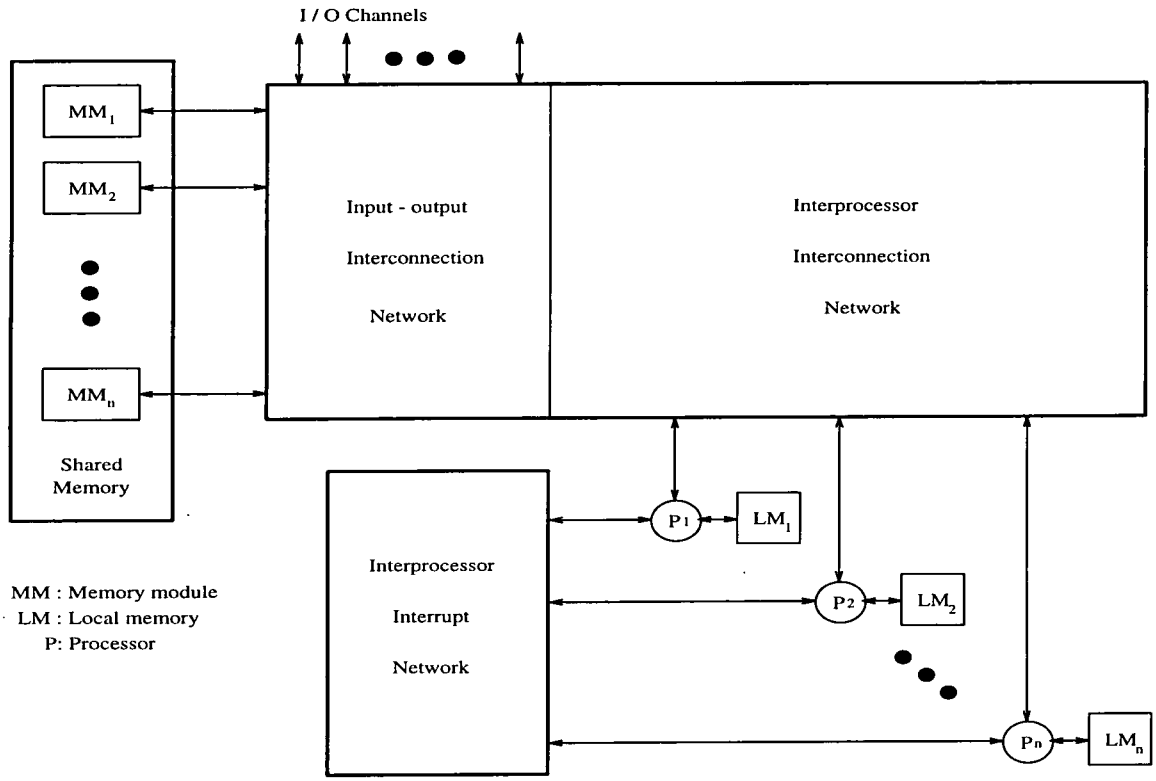


Figure 1.5: Functional design of a MIMD computer, after Hwang and Briggs

much easier to create, debug and maintain.

1.4.3 MIMD Architectures

MIMD architectures come in two distinct flavours, *shared memory* and *distributed memory*. A third hybrid flavour *distributed shared memory* is also possible but less popular. Figure 1.5 shows the functional design of a MIMD computer and how the different elements combine to yield the three flavours.

Shared memory MIMD architectures [17, 18] use some form of bus interconnection network to connect all the processors to a common memory bank. Synchronisation and communication between processes is performed via the common memory. In order to increase the efficiency of the bus, shared memory machines often utilise some form of local cache at each processing node to reduce the amount of traffic passing across the bus. There then arises the problem of cache coherency [18]; each cache can contain a copy of the same memory data. Correct operation requires some mechanism that ensures that all caches contain the same values for a given data item. Cache coherency is a major problem in the design of shared memory systems. Another problem arises in trying to extend the system to use a larger

number of processors. Whilst the bandwidth of modern memory systems is sufficiently large to allow the connection of multiple processors, bus contention begins to become more significant as more processors are added. Bus-based shared memory systems seem to have a limit of around 20 processors. Other forms of memory interconnection networks have been devised including crossbar switching and various multistage networks. These allow larger numbers of processors to be used but impose some penalty on system performance. Even with these interconnection systems it is not possible to achieve the easy scalability that is possible with distributed memory architectures.

Distributed memory MIMD architectures [17, 18, 11] have no global memory but each processor has its own private memory. Both program code and data are partitioned into the local memories of the processors in the system. Processors which wish to synchronise their operation or exchange data must do so by passing explicit messages across the communications network interconnecting the processors. Unlike shared memory architectures, distributed memory systems can be scaled up to use any number of processors and commercial systems are in existence which use hundreds of processing nodes [11]. Distributed memory systems are also easier to design and cheaper to build as there is no complex hardware required to provide access to a global memory. The major disadvantage of distributed memory architectures is that delays are associated with the communication of messages, especially if messages have to be routed via intermediate processors. These delays can seriously reduce the performance of the system. Another disadvantage of distributed memory machines is that their lack of global memory makes it harder to write and debug programs. Writing programs for distributed memory architectures requires the programmer to think in a distributed manner. The global memory of a shared memory architecture allows the programmer to utilise the conventional von Neumann programming paradigm to a certain extent. Despite these disadvantages the cheapness and ease with which distributed memory machines can be built makes the distributed memory architecture ideal for entry level parallel computing systems. Indeed many of the entry level parallel systems available today are distributed memory MIMD machines.

Distributed shared memory architectures [17] are a compromise between the distributed and shared memory approaches which attempts to solve the problems associated with both of these models. Distributed shared memory machines have both local private memory at each processing node and access to a common shared memory. Two interconnection networks are used, one to connect all the processors to allow distributed message passing and

one to connect each processor to the common memory. Data exchange and synchronisation can now be performed either via shared memory or via explicit message passing. These architectures are more scaleable than shared memory architectures and give better performance than distributed memory architectures. They are however complicated to build and few are commercially available.

1.4.4 Interconnection Networks for MIMD Architectures

The preceding discussions have made reference to the interconnection network between processors in the case of distributed memory machines, and between processors and memory in the case of shared memory machines. The choice of interconnection network can make or break the performance of the parallel machine and is therefore of critical importance. Choosing an inappropriate interconnection can severely reduce speed-up due to the extra communication overheads involved. Often a particular configuration is suitable for one particular algorithm but is inappropriate for a different algorithm, making it difficult to find an efficient general purpose interconnection strategy. The most efficient general topology to date is the hypercube [15, 17, 18, 13, 19, 11] shown in Figure 1.6. Some of the other common interconnection strategies are also shown in Figure 1.6. These interconnections are referred to as static topologies as they are determined by a physical interwiring of the processors. Dynamic interconnection networks are also possible [17, 18, 13] in which interprocessor communications are made via a multiple stage switching network. The switching network automatically routes message to the destination processor by configuring the switches according to address information contained in the message and operation is similar to that of a modern packet-switched telephone exchange. Dynamic multistage networks are highly efficient in that they allow an arbitrary input to be connected to an arbitrary output with a constant communication delay. Such networks are expensive to implement due to the dynamically configurable switching hardware required and are beyond the scope of this thesis. Instead this thesis is based upon work performed on a reconfigurable statically connected multiprocessor system. In this system physical processors interconnections are made via crosspoint switch mechanisms which have to be set up before the system can be used. Once set the network retains that topology until it is reset and a new interconnection topology defined. The network cannot be reconfigured during the course of a program's execution and hence the topology is essentially static.

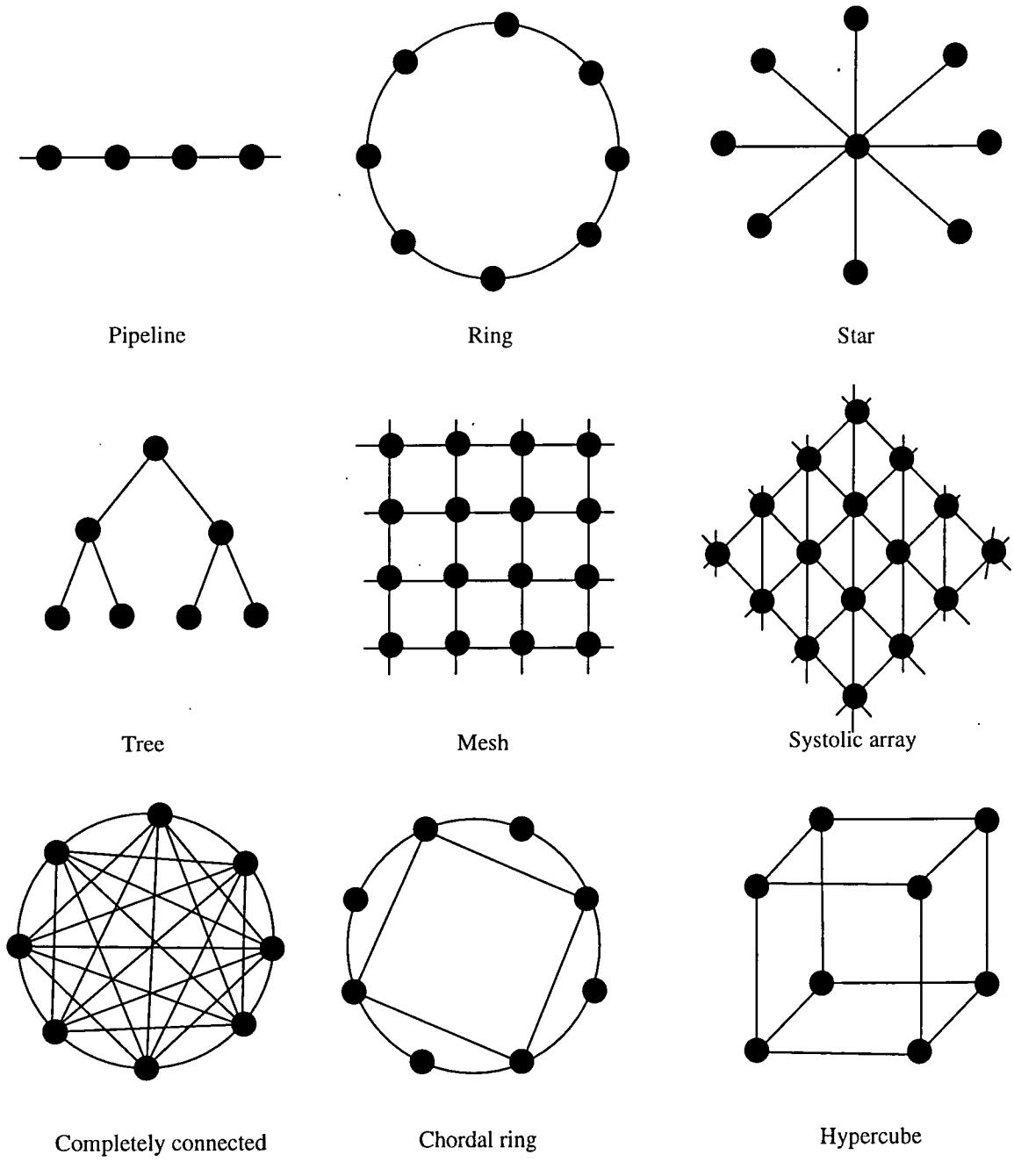


Figure 1.6: Static interconnection network topologies, after Hwang

1.4.5 The INMOS Transputer

The INMOS Transputer is a general purpose reduced instruction set (RISC) processor designed specifically for use in parallel computers [20, 21, 22]. Each Transputer consists of a fast microprocessor, four serial communication links, fast cache memory, external memory interfacing, floating point coprocessor, real-time clocks and a hardware implemented multitasking scheduler. An array of Transputers may be created by interconnecting the serial links with those of other Transputers in a point to point fashion. Although it is possible for an array of Transputers to access a global shared memory the usual configuration of Transputer systems is as a distributed memory machine. To enable the easy building of scaleable parallel systems INMOS have created a modular system for building Transputer based machines. This standard is based around the use of Transputer Applications Modules (TRAM's) which are small circuit boards measuring 3.6 inches by 1.1 inches. Each TRAM hosts a single Transputer, RAM and all necessary interfacing logic and is a complete computer in its own right. TRAM's plug into a motherboard which resides in a host PC or workstation and the motherboard provides all the power and control signals to each TRAM. Two of the serial communication links of each TRAM on the motherboard are hardwired into a pipeline configuration. The remaining two links from each Transputer may be connected in any desired fashion using the reconfigurable crosspoint switch. In addition the interface between the motherboard and the host allows the Transputers to access the disk, screen and keyboard I/O systems of the host computer. Figure 1.7 provides a conceptual view of the entire parallel machine configuration. Further details about the INMOS Transputer are presented in Appendix A. Graham and King [21] provide an excellent overview of the Transputer and Transputer-based systems.

The parallel computing system used throughout the duration of this research project consisted of 16 INMOS T805 30MHz Transputers and one INMOS T805 20MHz Transputer. Fifteen of the 30 MHz processors were supplied with 1 MB of fast RAM whilst one 30 MHz processor was equipped with 4 MB of RAM. The 20 MHz processor was supplied with 16 MB of fast RAM and was used as the root processor in the Transputer network. All of the Transputers were mounted on two INMOS B008 compatible motherboards and hosted by an IBM PC AT clone. Each motherboard could accommodate up to 10 Transputers and was equipped with an electronic crosspoint switch which allowed the Transputer interconnection network to be reconfigured from software.

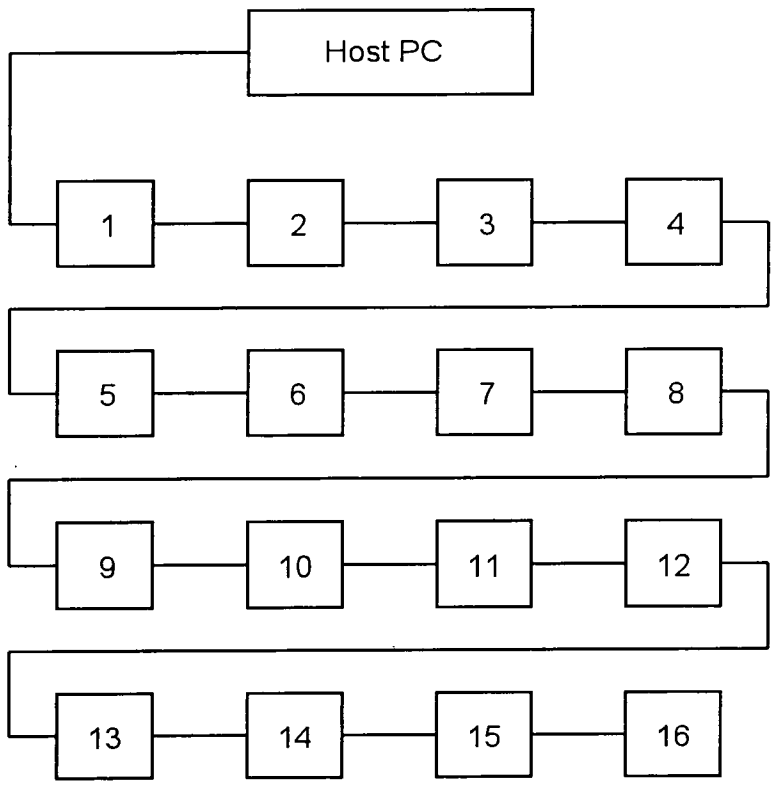


Figure 1.7: Conceptual overview of the configuration of the Transputer based parallel machine

1.4.6 Bounds on Multiprocessor Performance

The *speed-up*, $S(n)$, that is obtained in using n processors to solve a problem defines how much quicker that problem is solved by n processors than by one processor. If t_s is the time taken to execute the algorithm on one processor and t_p is the time taken to execute the *same* algorithm on n processors then speed-up is defined as

$$S(n) = \frac{t_s}{t_p} \tag{1.10}$$

This is the strict definition of speed-up and it describes the improvement over a uniprocessor implementation of an identical algorithm. The most efficient sequential algorithm is not always the best parallel algorithm and a less efficient sequential algorithm will often produce a better parallel implementation. Due to these possible differences in algorithms the user is ultimately interested in the speed-up relative to the best sequential algorithm. Equation (1.10) still defines the speed-up but now t_s is the time to execute the best sequential algorithm and t_p is the execution time of the parallel algorithm. Both algorithms must be executed on the same type of processor running at the same clock speed in order to make

the analysis valid. An algorithm which achieves high speed-up but requires a large number of processors to operate is obviously inefficient. The *efficiency* of implementing a parallel algorithm is expressed as the ratio of speed-up to the number of processors required to yield that speed-up. Hence the parallel efficiency, $E(n)$, is defined as

$$E(n) = \frac{S(n)}{n} = \frac{speed - up}{number\ of\ processors} \tag{1.11}$$

The maximum speed-up that can be achieved with n processors working simultaneously is n. This ideal case is known as *linear* speed-up. The speed-up achieved in practice is often much less than this due to the inability of the algorithm to exploit all the concurrency in the problem, communication overheads and the time processes spend idling waiting for synchronisation and/or communication. Minsky's conjecture [23] gives a lower bound on the performance that can be expected of the n processor system of $S(n) = \log_2 n$ but this is a rather pessimistic estimate of performance. Hwang [15] gives a more optimistic estimate of a practical upper bound on speed-up which is based upon statistical analyses of the performance of real programs and thus takes account of communications overheads *etc.* Hwang's calculations show that the upper bound on realistically achievable speed-up is asymptotic to $\frac{n}{\ln n}$. These calculations were based upon experiments performed in the early 1980's and since that time advances in parallel technology have meant that it is now possible to achieve speed-ups in excess of those predicted. Figure 1.8 summarises the relationship between these various predictions of performance and it is obvious from this figure that actual speed-ups often fall short of the theoretical ideal.

Amdahl [24] gives a quantitative analysis of expected speed-up based upon the amount of parallelism in the problem. In any problem there is a certain amount, W_p, which can be solved in parallel but Amdahl argues that there is always a sequential part of the problem, W_s, which cannot be parallelised. If we define $C_s(n)$ to be the cost of performing a single sequential operation on an n processor machine and $C_p(n)$ to be the cost of performing n parallel operations , we can define the uniprocessor, and multiprocessor execution times, t_s and t_p, as

$$t_s = W_s C_s(1) + W_p C_p(1) \qquad t_p = \frac{W_s C_s(n)}{n} + \frac{W_p C_p(n)}{n} \tag{1.12}$$

where $W_s + W_p = 1$ as W_s and W_p are normalised. If the cost of a single sequential or

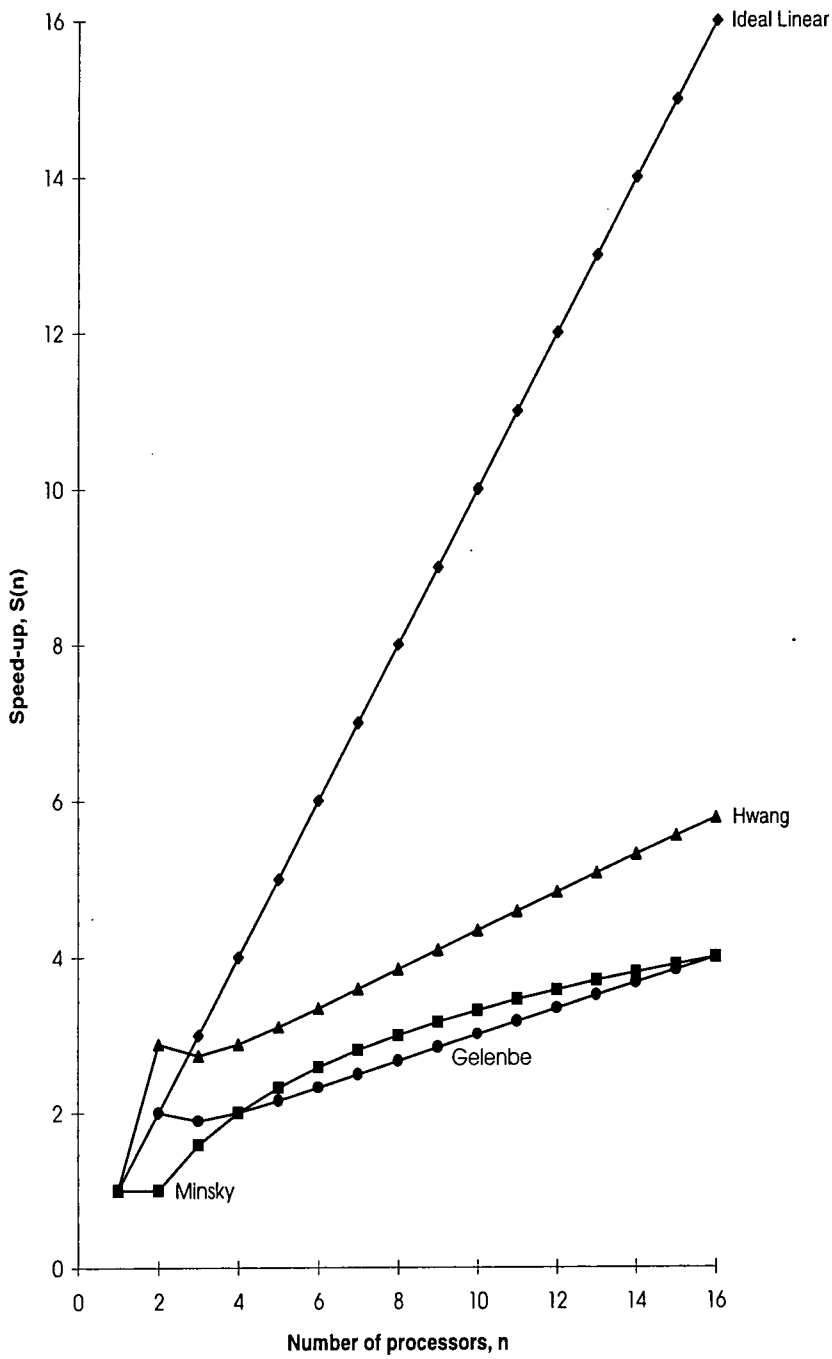


Figure 1.8: Various bounds on parallel performance

parallel operation takes one unit of processing time then

$$t_s = W_s + W_p \tag{1.13}$$

Considering the n processor machine; if $C_s(n) = nC_s(1)$

$$t_p = \frac{W_s n C_s(1)}{n} + \frac{W_p C_p}{n} = W_s + \frac{W_p C_p(n)}{n} \tag{1.14}$$

Similarly, if we assume that $C_p(1) = C_p(n) = 1$ then t_p becomes

$$t_p = W_s + \frac{W_p}{n} \tag{1.15}$$

Speed-up $S(n)$ is defined as the ratio of uniprocessor execution time to multiprocessor execution time. Hence

$$S(n) = \frac{t_s}{t_p} = \frac{W_s + W_p}{W_s + \frac{W_p}{n}} \tag{1.16}$$

Equation (1.16) is Amdahl's Law and the speed-up it predicts for various numbers of processors, n, and various values of W_s, is shown in Figure 1.9. This graph vividly shows the effect that sequential operations in the parallel algorithm have on speed-up. The greater the sequential part of the problem, the lower the speed-up and the greater the rate of speed-up saturation. For example, consider the case with 16 processors. If only 10% of the problem must be solved sequentially, the speed-up that can be achieved is only half that of the ideal linear speed-up.

Amdahl's Law provides an asymptotic upper bound on speed-up of

$$\lim_{n \to \infty} S(n) = \frac{1}{1 - W_p} \tag{1.17}$$

It should be noted that Amdahl's Law focuses only on the computations involved and does not take account of other aspects of multiprocessor performance such as communication overheads and cache misses. Amdahl's Law implies that a greater speed-up could be achieved by partitioning a problem into more parallel parts and executing them on more processors. However this would require much more interprocessor communication and the measured speed-up is likely to be significantly less than Amdahl's law predicts. Gelenbe [25] has proposed a number of extensions to Amdahl's Law which take interprocessor communication into account. He notes that as n increases so does the fraction of the execution

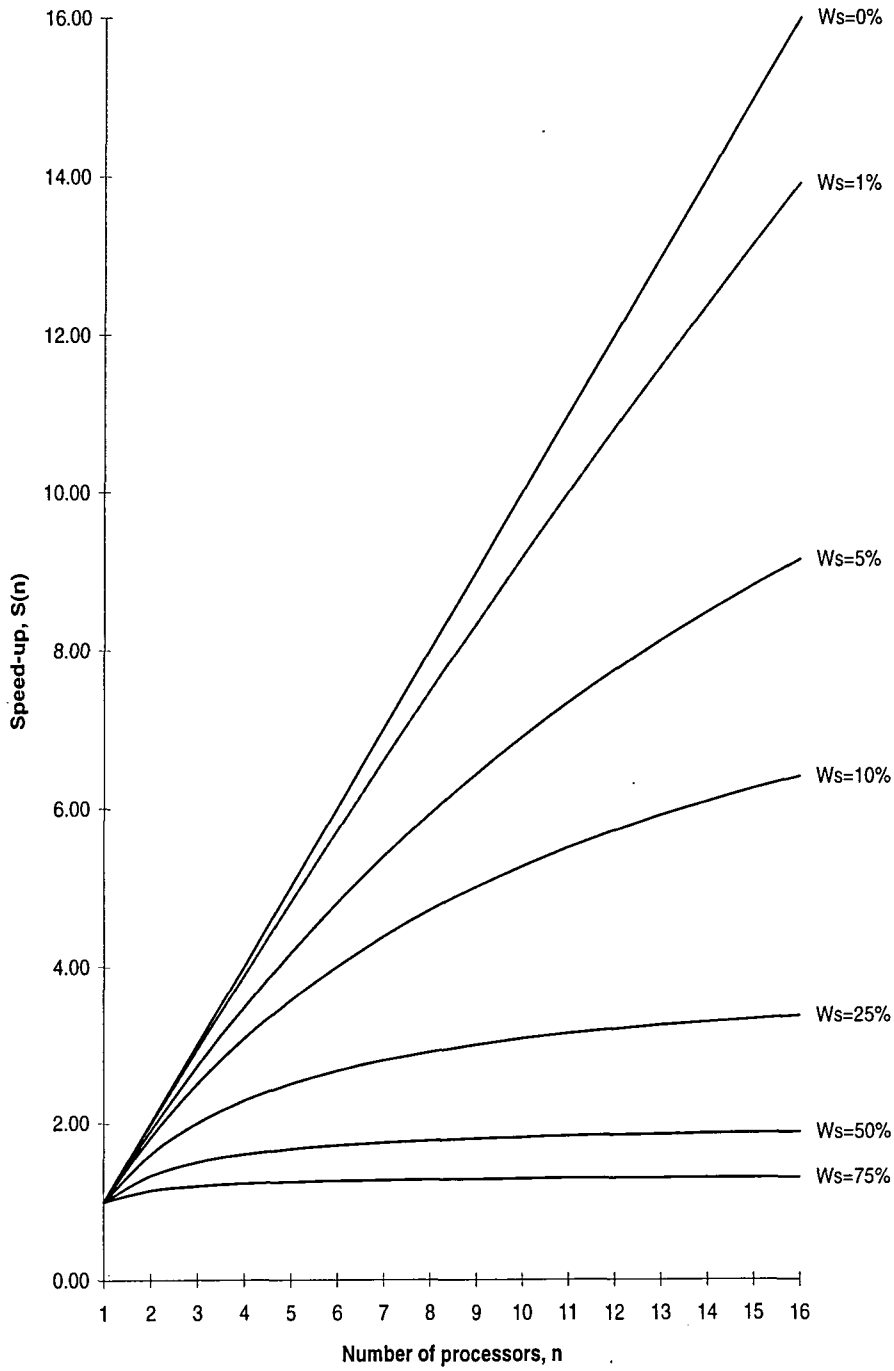


Figure 1.9: Speed-up predicted by Amdahl's Law

time spent in communication, $c(n)$. This makes the communication time the limiting factor to speed-up for large numbers of processors and the upper bound on speed-up thus becomes

$$S \approx \frac{1}{c(n)} \tag{1.18}$$

A second extension considers the fact that the parallel program does not make full use of the n processors and this gives an upper bound on speed-up of

$$S \leq \frac{n}{\log_2 n} \tag{1.19}$$

Gelenbe's bounds on speed-up are plotted in Figure 1.8.

1.5 Parallel Processing in Power System Analysis Problems

Earlier in this chapter the subject of power systems analysis was introduced and standard problems such as power flow, dynamic simulation, security analysis and operator training were considered. Many of these, and other power system analysis problems, require the solution of a set of linear equations. This set of equations is often large and in real-time analysis software the equations must be solved as quickly as possible. Despite the use of techniques such as sparse matrix storage it is often not possible to solve these equations as fast as desired [26, 27, 28, 29, 30, 31, 32, 33, 34, 35]

One of the most promising approaches is to use parallel computers to give a fast solution. Parallel computers have more than one processor and their high performance results from their use of these multiple processing units. The basic premise of parallel computing is that a problem can be solved more quickly if it is split into independent parts which can be solved simultaneously. With regard to the equations for the power system network, this involves the use of diakoptical techniques to partition the problem for solution by individual processors. Several successes have been achieved [36, 37] with three- or fourfold decreases in solution times recorded by a number of researchers.

The use of parallel computers to solve power system analysis problems is not new and numerous researchers have developed many different parallel approaches to problems such as transient stability analysis [38, 39, 40], short circuit analysis [41], state estimation [42] and simulation of electromagnetic transients [43]. A number of researchers [26, 36, 27, 29, 33, 44, 45, 46, 35] have concentrated solely on the parallel solution of the linear network

equations. Most of the methods developed to date are based on triangular factorisation and solution although other methods have been attempted, such as the Multiple Factoring method [46, 29] and an approach based on the use of the Conjugate Gradient method [28]. The existing methods will be considered more fully in Chapter 3 but the basis of the triangular decomposition methods is to partition the set of equations into subsets which may be solved independently. Unfortunately it is not easy to split the equations into independent subsets and a successful partitioning strategy requires detailed analysis of the numerical algorithms [29]. Kron's method of diakoptics [47, 48] is often used to decompose the system into subsets which may be solved concurrently using multiple processors. A coordination phase is introduced into the solution algorithm to combine and modify the individual solutions to give the overall solution of the set of equations.

None of the methods developed to date has successfully exploited the full performance of parallel computers due to the nature of the problem. Speed-up seems to be limited to about 3 or 4 and the coordination phase is recognized as being a bottleneck in the solution [49]. Some methods do achieve higher speed-ups [26, 46] but they require a large number of processors which makes them expensive and inefficient. The authors of [50] note that whilst algorithm development has produced good theoretical results, little software has actually been developed for parallel machines.

This thesis discusses the development of a parallel solution which attempts to circumvent the limitations of existing parallel methods. Whilst the thesis only discusses the technique in relation to the solution of power system network equations it is also valid when applied to other similar networks (*e.g.* telecommunications networks, electronic circuit analysis, gas and water networks *etc*). Indeed the technique can be applied to any network which can be represented by a set of sparse symmetric diagonally dominant linear equations.

1.6 Summary

This chapter has introduced power systems and parallel computation. The power system and its components have been discussed and a number of power system analysis applications have been examined. The linear network equations have been clearly identified as having a central role to play in these applications. Some of these applications, such as dynamic simulation and on-line dynamic security assessment require real-time or faster operation and in these applications it is essential that the linear equations be solved as fast as possible.

Sequential computers are limited in their ability to perform these computations within the required time frame and other computer architectures must be considered if the solutions are to be accelerated.

Parallel computers have been introduced as a different type of computer from the conventional sequential machine. The operation of parallel computers has been considered and a number of parallel computer architectures have been examined. Using a parallel computer it is possible to solve a problem faster than on a sequential machine by dividing up the problem and solving independent parts concurrently on the multiple processors of the parallel machine. Parallel processing seems to be an ideal technique for accelerating the solution of the power system network equations and indeed this approach has already been used. The equations are solved by partitioning them into independent subsets which may be solved concurrently by the multiple processors. Unfortunately it is not easy to split the equations into independent subsets and a coordinating phase must be introduced into the parallel solution to combine and modify the results for each of the subsets. This combination phase is a bottleneck in the solution process and limits the speed-up of the parallel solution to about 3 or 4 regardless of the number of processors used. Some of the existing methods achieve reasonable speed-ups but require many processors to achieve their performance [46, 26] and they are therefore rather inefficient.

During the early developments in computer-based power system analysis tools significant performance enhancements were made through improvements in the algorithms used. Through these improvements the algorithms for the sequential solution of linear equations have evolved into the highly efficient state we see today. Whilst developments in computer hardware will produce faster machines capable of solving the network equations more quickly than they can be solved at present there is still much that can be done to improve the algorithms used in parallel solutions. Rather than buying a bigger hammer to crack the nut it is more profitable to redesign the smaller hammer so that it cracks nuts more efficiently.

1.7 Outline of Thesis

This aim of this thesis is to explore the derivation of a technique for efficiently solving power system linear equations on a distributed memory multiprocessor. This introductory chapter has discussed what these equations are and where they arise. The concepts of

parallel computing have been introduced and the need for parallel methods of solving linear equations has been demonstrated.

Chapter 2 will survey sequential methods for solving the linear equations as all parallel solutions are based around these sequential methods. LU based triangular decomposition is introduced as the standard technique whilst sparse matrix methods and optimal reordering are introduced as ways of minimizing computation and processing time. These techniques will provide a good platform from which to explore parallel solution methodologies. The elimination tree will be shown to be a powerful tool for providing insight into the solution process and its introduction is intended to highlight areas of independence and potential parallelism in the solution of the equations.

Chapter 3 moves on to discuss existing methods for the parallel solution of the equations. The two flavours of solution, iterative and direct, will be considered with the aim of determining which approach is most suitable for power system computations. Typical parallel methods will be considered in some detail for both direct and iterative solutions. It is intended that this chapter will highlight the beneficial features of these methods so that they may be used later either to form the basis of a new method or to suggest improvements to existing techniques.

Chapter 4 returns to the subject of the elimination tree and will show how it can be used in partitioning the problem for parallel solution. It will also be shown that the elimination tree is useful in optimizing the assignment of computations to individual processing elements in a multiprocessor. The insight provided by the elimination tree will be of paramount importance in improving the amount of parallelism that can be exploited when solving the equations concurrently.

Chapter 5 will take the insight provided by the elimination tree and combine it with the beneficial features of existing parallel solutions to produce an improved parallel solution method. It will be shown that this is not a radical new algorithm but a restructuring of the problem which allows more of the inherent parallelism to be exploited. The benefits of the method and improvements in performance will be illustrated with the results of simulations which solve several systems of equations using both the standard technique and the improved method.

Chapter 6 considers how the improved method may be implemented on a multiprocessor array. Some of the techniques for improving the sequential solution will be revisited and applied to the parallel solution whilst other techniques that can be used to further improve

the implementation of the new parallel solution will also be presented. The performance of a parallel implementation of the new method will be compared with that of existing methods and the theoretical predictions of Chapter 5. This chapter aims to show that the new method gives faster and more efficient solutions than those obtained from existing parallel solutions.

Chapter 7 will present suggestions for further work on the methods discussed in this thesis. Having demonstrated the effectiveness of the new approach, Chapter 8 will conclude the thesis by assessing what has been achieved. These achievements will be compared to the initial aims and objectives.

Chapter 2

Solving the Network Equations

2.1 Modeling the Power System

Modern power systems are complex entities consisting of thousands of interconnected nodes. To analyse such a system it must be described by an equivalent formal mathematical model. This requires the use of a mathematical model for each different type of system component and the interrelationship between all these different models yields the set of equations which form the basic framework of the analysis. Suitable models for the main system components described in the previous chapter are now discussed.

2.1.1 The Generator Model

The synchronous generator has two main components; the rotor and the stator. The stator is a hollow cylindrical structure which provides a housing for the rotor. Wound into slots along the length of the stator casing are coils which are connected together to form three separate phase windings. The rotor is a solid cylindrical structure which can rotate freely about its axis within the stator structure. A coil wound on to the rotor is excited from a DC source. This winding produces an intense magnetic field which sweeps the stator as the rotor rotates, inducing a sinusoidal voltage into each of the stator windings. The voltages are identical in amplitude and frequency but are 120^o separated in phase. Mechanical rotation of the rotor is provided by some form of turbine connected to the rotor shaft, with steam, hydro, gas and wind turbines being the four main types of rotor prime mover.

The synchronous generator can be described in terms of the real and reactive power it

delivers.

$$P = \Re[\bar{S}] = \frac{VE_f}{X_d} \sin \delta \tag{2.1}$$

$$Q = \Im[\bar{S}] = \frac{VE_f}{X_d} \cos \delta - \frac{V^2}{X_d} \tag{2.2}$$

where

$$\bar{E}_f = jX_d\bar{I} + \bar{V} \tag{2.3}$$

$\bar{S}$ = complex power delivered

E_f = stator internal voltage

V = stator terminal voltage

P = real power delivered by generator

Q = reactive power delivered by generator

δ = power angle

X_d = direct axis synchronous reactance

The equivalent circuit model of the synchronous generator and the derivation of these equations is presented in Appendix B.1. Taken together (2.1) to (2.2) provide a model of the synchronous generator suitable for use in a real-time dynamic simulation.

2.1.2 The Load Model

In analysing an electrical power system it is necessary to consider the loads connected to the system. Loads at each bus are treated as composite loads and may be modeled as either constant current sinks or, more usually, as a simple impedance. Appendix B.4 considers the characteristics of system loads and methods of modeling them,

2.1.3 The Transmission Line Model

It is possible to derive a set of complex equations which give a complete mathematical model of a transmission line. This model has an analogous electrical equivalent circuit model, shown in Figure 2.1 as the equivalent π circuit model. The model comprises a series impedance term, Z, which accounts for the resistive and inductive losses on the line. Similarly a shunt admittance term, Y, is included to account for the shunt displacement currents arising from the electric fields between the conductors. It is usual to place half of this shunt admittance at either end of the line. Appendix B.2 derives the values of the series

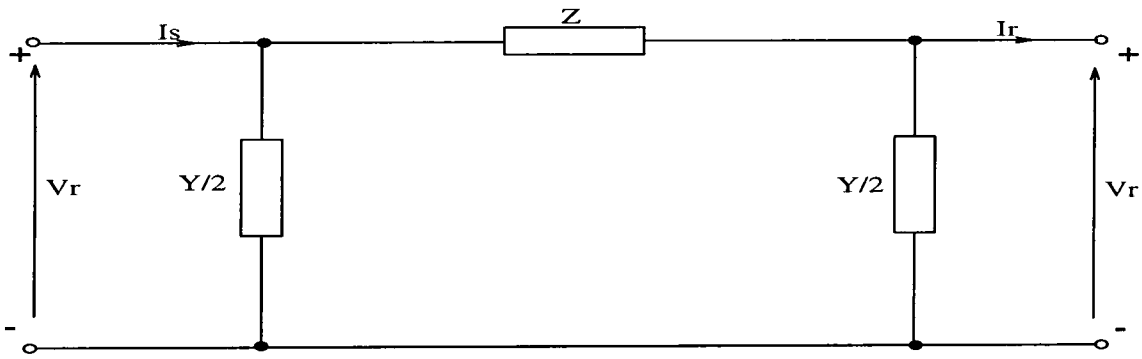


Figure 2.1: Transmission line equivalent π circuit model

impedance and shunt admittance and discusses transmission line modeling in more detail. In the problem formulation which follows it is assumed that the values of the parameters Z and Y are given for each line.

2.1.4 The Transformer Model

The transformer is a constant power device comprised of two or more coils used in electrical power systems to transform voltage and current levels. The coil connected to the power source is known as the primary winding and the coil connected to the load is known as the secondary winding. Assuming the transformer to be ideal, the power input to the primary winding is equal to the power delivered by the secondary winding. If there are N_1 turns on the primary winding and N_2 turns on the secondary winding then the terminal voltages and currents are related by

$$\frac{V_1}{V_2} = \frac{N_1}{N_2} \qquad \frac{I_1}{I_2} = \frac{N_2}{N_1} \tag{2.4}$$

A practical single phase equivalent circuit model of a two winding transformer is given in Figure 2.2. The model accounts for finite core permeability, winding resistance, imperfect flux linkage and eddy current and hysteresis losses. The parameters of the model are expressed in terms of the series resistance and flux leakage of the primary and secondary windings. x_1 and x_2 account for flux leakage in the primary and secondary windings respectively. Similarly r_1 and r_2 are the series resistances of the primary and secondary windings. Appendix B.3 provides a full derivation of the model.

2.2 Formalizing the Problem

Given a power system we would like to address issues such as

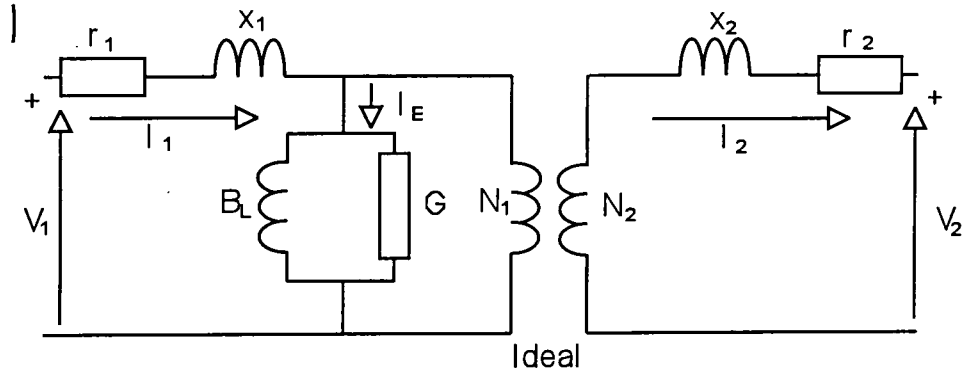


Figure 2.2: Equivalent circuit of a single phase transformer

- What are the loads on transformers, lines and generators in the system ?
- What are the voltages and currents at each point in the system ?

This is the power flow problem and it is concerned with calculating the voltage magnitude and phase at each bus in the system.

A system bus is defined as a point of physical interconnection of system components and a power system consists of many buses interconnected by a transmission network. At each bus there will be three components contributing to the total power delivered at that point; generation, load and transmission although either generation or load may be missing. Generation delivers power into the bus whilst transmission and load extract power from the bus, *i.e*

$$\bar{S}_g = \bar{S}_l + \bar{S}_t \tag{2.5}$$

where

$\bar{S}_g$ = complex power delivered into the bus by the generator

$\bar{S}_l$ = complex power absorbed by the load

$\bar{S}_t$ = complex power extracted / delivered by the transmission network

The distribution of power is achieved by the transmission network and it is this network which must be analysed to determine the voltage characteristics at each system bus.

We can consider the transmission network as an n-port network to which generators and loads are connected to form the power system, as depicted in Figure 2.3 As generators and loads are external to the n-port transmission network they need not be considered further. Should the generator voltages and currents be required they can be calculated using the equations of Section 2.1.1. For the purposes of this thesis it is assumed that at each system

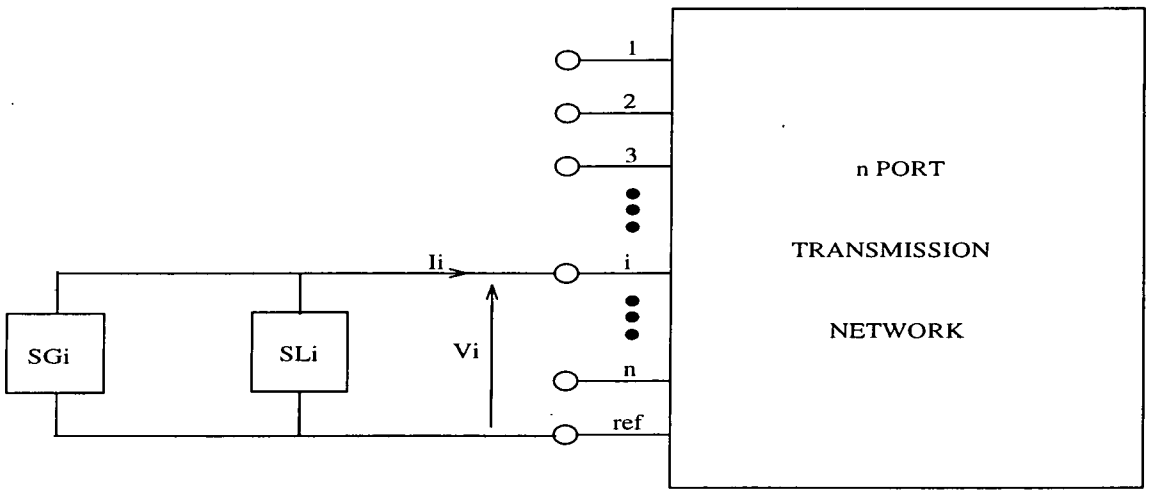


Figure 2.3: General n-port representation of a power system

bus the current $I_{i_{i=1,\ldots,n}}$ is given, where I_i is a current due to generation less load at bus i. The transmission network is simply a network of impedances as each transmission line in the network can be replaced by its equivalent π circuit, as described in Appendix B. Given that the current I_i is known at each bus and that the impedances comprising the network are known, the Ohmic equation

$$\mathbf{V} = \mathbf{IZ} \tag{2.6}$$

can be used to obtain the voltage characteristics of each bus. It is more usual to consider the transmission network in terms of its admittance as the impedance matrix, $\mathbf{Z}$, is dense whereas the admittance matrix, $\mathbf{Y}$ is sparse. This has important consequences for computational efficiency, as later sections will show. With the network described in terms of its admittance, the currents injected at each bus become the inputs to the system whilst the unknown bus voltages become the output of the system. Hence (2.6) becomes

$$\mathbf{YV} = \mathbf{I} \tag{2.7}$$

where $\mathbf{V}$ is the vector of bus voltages and $\mathbf{I}$ is the vector of bus currents. $\mathbf{Y}$ is known as the *bus admittance matrix* and this characterizes the admittance of the transmission network. The solution of (2.7) is the problem on which the work in this thesis is based. The solution for the unknown voltage vector is given by

$$\mathbf{V} = \mathbf{Y}^{-1}\mathbf{I} \tag{2.8}$$

2.3 Linear Equations, Matrices and Sparsity

Many real world systems can be modeled by a set of simultaneous linear equations of the form

$$\begin{aligned} a_1x_1 + a_2x_2 + a_3x_3 &= b_1 \\ a_4x_1 + a_5x_2 + a_6x_3 &= b_2 \\ a_7x_1 + a_8x_2 + a_9x_3 &= b_3 \end{aligned} \tag{2.9}$$

This set of equations can be represented in matrix notation as $\mathbf{A}.\mathbf{x} = \mathbf{b}$, which is expressed in full as

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.10}$$

Most real systems have many zero entries in the **A** matrix and these systems are known as *sparse* systems. The matrix of coefficients, **A**, is known as a *sparse* matrix. Such systems are of special significance in the design of a computer program for solving linear equations and will be considered shortly.

The values of **x** are unknown and a solution for the elements of this vector is required. The usual method for solving a set of simultaneous equations involves some form of Gaussian elimination of the set of equations followed by substitution to yield the unknown values. Given the set of equations it can be seen that the problem of finding a solution to **x** requires determination of the matrix inverse $\mathbf{A}^{-1}$. Once the inverse has been determined the vector **x** is obtained by the simple matrix multiplication

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{2.11}$$

In power systems analysis it is necessary to solve the voltage at each node in the system using the equation

$$\mathbf{Y}.\mathbf{V} = \mathbf{I} \tag{2.12}$$

where

Y= System nodal admittance matrix

V = Nodal voltage vector

I = Nodal current vector

The matrix **Y** is derived using nodal admittance analysis of the given power network, as described in Appendix C, and for systems of any reasonable size this matrix exhibits some degree of sparsity *i.e* many of the elements in this matrix are zero. This admittance matrix is symmetrical by virtue of the structure of the network and it is diagonally dominant. This allows computer solution techniques to achieve greater accuracy through a reduction in relative numerical error.

As the admittance matrix is sparse only non - zero matrix elements contribute to a solution and hence only those elements need enter into calculations. This has strong implications for the design of computer algorithms which store and operate on these matrices.

2.4 Direct Solution of the Linear Equations

The determination of the inverse of **A** is an inefficient, computationally expensive procedure. The key to solving the network equations is to produce the *effect* of the inverse without actually calculating the full matrix inverse. The basis of the method is to decompose the coefficient matrix into a number of factor matrices which are multiplicatively combined to produce the effect of the inverse. This approach is significantly more efficient, requiring fewer calculations than the determination of the full inverse and a solution obtained using this method is termed a *direct* solution.

Three of the more common methods are now introduced. The method used exclusively throughout the research work described in this thesis, Zollenkopf's Bifactorisation method, is described in more detail.

2.4.1 Gaussian Elimination and Fill-Ins

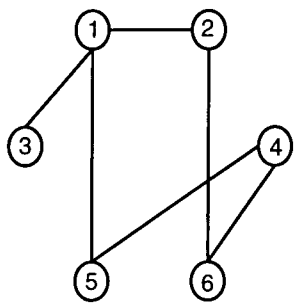
The set of linear equations $\mathbf{Ax} = \mathbf{b}$ arises directly from the structure of the network they model. The known vector **b** can be considered as the input to the system and the unknown vector **x** corresponds to the system output. The structure of the matrix of coefficients, **A**, is directly determined by the topology of the network. Graph theory [51] tells us that there is a duality between graphs and matrices and that a matrix may be used to describe a graph. The *adjacency matrix* associated with a graph is square and has as many rows/columns as there are nodes in the graph. If nodes i and j of the graph are directly connected then

non-zero entries are inserted into the matrix at elements (i,i), (j,j), (i,j) and (j,i) [1]. The values added to these elements usually represent some parameter of the network and in the case of power systems the values represent the admittance connected between nodes i and j. The bus admittance matrix (Appendix C) is thus the adjacency matrix of the power system network.

Given **A** and **b** the problem is to solve the equations to yield **x**. This is usually achieved through some form of Gaussian elimination in which the aim is to modify matrix **A** using successive column eliminations to reduce it to upper triangular form. Back substitution with **b** then yields the vector **x**. Given the graph-matrix duality one would expect operations on the matrix to manifest themselves in the associated graph. Elimination of columns from the matrix is equivalent to the elimination of nodes from the graph.

Consider the example graph and adjacency matrix shown below

$$\begin{bmatrix} 15 & 5 & 5 & & 5 & \\ 5 & 10 & & & & 5 \\ 5 & & 5 & & & \\ & & & 10 & 5 & 5 \\ 5 & & & 5 & 10 & \\ & 5 & & 5 & & 10 \end{bmatrix}$$



The first stage in the Gaussian elimination of this system is to eliminate elements in column 1 of rows 2 to 6. This is achieved by subtracting multiples of row 1 from rows 2-6 so as to make the first element of each row equal to zero. For example, column 1 is eliminated from row 2 by subtracting $\frac{1}{3}$ of row 1 from row 2. This modifies the elements of the second row and the matrix becomes

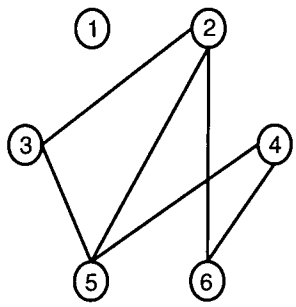
$$\begin{bmatrix} 15 & 5 & 5 & & 5 & \\ & \frac{25}{3} & -\frac{5}{3} & & -\frac{5}{3} & 5 \\ 5 & -\frac{5}{3} & 5 & & & \\ & & & 10 & 5 & 5 \\ 5 & -\frac{5}{3} & & 5 & 10 & \\ & 5 & & 5 & & 10 \end{bmatrix}$$

Notice that two new elements have appeared in columns 3 and 5 of row 2. The signifi-

[1] This assumes an undirected connection between i and j. If the connection is directed from i to j then non-zero values will only be inserted in elements (i,i), (j,j), (i,j)

cance of these elements can be seen by examining the connectivity of the adjacency graph. Node 2 is indirectly connected to nodes 3 and 5 via node 1. When node 1 is eliminated from the graph new connections must be inserted between 2 and 3 and 5 and 3 to preserve the connectivity of the network. These new connections, known as *fill-ins* appear in the matrix as the new elements $(2,3)$ and $(2,5)$. Once column 1 has been eliminated from rows 2-6 the matrix and the graph have been modified to give

$$\begin{bmatrix} 15 & 5 & 5 & & 5 & \\ & \frac{25}{3} & -\frac{5}{3} & & -\frac{5}{3} & 5 \\ 5 & -5 & 5 & & & \\ & & & 10 & -\frac{10}{3} & 5 \\ 5 & -\frac{5}{3} & -\frac{5}{3} & 5 & \frac{5}{3} & \\ & 5 & & 5 & & 10 \end{bmatrix}$$



The elimination of node 1 has produced fill-in connections in the graph between 2 & 3, 2 & 5 and 3 & 5. New fill-in elements have been created in the adjacency matrix at elements $(2,3)$, $(2,5)$, $(3,2)$, $(3,5)$, $(5,2)$ and $(5,3)$. Following the elimination of each node in the network at successive steps of the Gaussian elimination algorithm, it may be necessary to introduce fill-ins to preserve the connectivity of the network. Note that even if fill-ins are not created, the values of existing matrix elements may be modified.

The significance of fill-ins becomes apparent when the number of operations required to eliminate all nodes is considered. The original coefficient matrix is sparse. In eliminating columns from the matrix it is only necessary to operate on non-zero elements. The introduction of fill-ins increases the number of non-zeros and thus increases the number of operations required to eliminate all the nodes. The more fill-ins there are, the longer it will take to complete the elimination. It is possible to reorder the matrix in such a way that the amount of fill-in is reduced. Reordering the matrix to give the minimum fill-in also gives rise to the minimum solution time. Section 2.5 and Section 2.7 consider matrix ordering in more detail.

Fill-ins are also important when the storage requirements for a solution are considered. With a sparse matrix it is only necessary to store the non-zero elements. Introducing fill-ins increases the storage required and this could be a problem if memory is limited. Minimum fill reorderings are useful in that they also minimize the amount of memory required.

2.4.2 LU Decomposition

The LU factorisation technique is one of the more widely used triangular factorisation techniques. The coefficient matrix is considered to be the product of two triangular factor matrices

$$\mathbf{A} = \mathbf{LU} \tag{2.13}$$

where

L is a lower triangular matrix in which the leading diagonal elements are unity

U is an upper triangular matrix

Hence

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 & & \\ l_{21} & 1 & \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ & u_{22} & u_{23} \\ & & u_{33} \end{bmatrix} \tag{2.14}$$

The LU factorisation is used as the first stage in the three stage solution of linear equations

Stage 1	Factorise	$\mathbf{A} = \mathbf{LU}$
Stage 2	Forward substitute to solve for $\mathbf{y}$	$\mathbf{Ly} = \mathbf{b}$
Stage 3	Backward substitute to solve for $\mathbf{x}$	$\mathbf{Ux} = \mathbf{y}$

The vector, $\mathbf{y}$, is a vector of intermediate results. The advantage of this approach is that operations on the right hand side (stages 2 and 3) may be performed independently of the factorisation stage. This allows the same system to be solved with multiple right hand side vectors.

The formulae for generating the elements of **L** and **U** are

$$l_{i,j} = \frac{a_{i,j} - \sum_{k=1}^{j-1} l_{i,k}u_{k,j}}{u_{jj}} \qquad j = k+1 \ldots n \tag{2.15}$$

$$u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k}u_{k,j} \qquad j = k+1 \ldots n \tag{2.16}$$

The coefficients of **L** and **U** may be merged and stored in a single $(n \times n)$ matrix, $\mathbf{A}_F$. $\mathbf{A}_F$ is created by overwriting the elements of **A** as the factorisation progresses. Note that if **A** is symmetric, the LU factorisation destroys the symmetry as $\mathbf{A}_F$ is not symmetric *i.e.* $l_{i,j} \neq u_{j,i}$

By way of example, consider the LU factorisation of the matrix

$$\begin{bmatrix} 16 & 4 & 8 \\ 4 & 5 & -4 \\ 8 & -4 & 22 \end{bmatrix}$$

This results in the factor matrix

$$\mathbf{A_F} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{bmatrix} = \begin{bmatrix} 16 & 4 & 8 \\ 0.25 & 4 & -6 \\ 0.5 & -1.5 & 9 \end{bmatrix}$$

2.4.3 LDU Decomposition

Another triangular factorisation method often used in power system computations is the LDU factorisation. The coefficient matrix is considered to be the product of three factor matrices, **L'**,**D** and **U**.

Here

L' is a lower triangular matrix with unity elements on the leading diagonal

U is an upper triangular matrix in which the leading diagonal elements are unity

D is a diagonal matrix

The method is similar to LU factorisation and the diagonal elements of the **U** matrix in LU factorisation appear as the the diagonal elements of the **D** matrix in LDU factorisation. The lower triangular matrix of LDU factorisation, **L'**, is obtained from the lower triangular matrix of LU factorisation, **L**, by dividing each column of **L** by the diagonal element of that column.

Consider the following matrix by way of an example

$$\mathbf{A} = \begin{bmatrix} 3 & -1 & -1 & \\ -1 & 2 & & \\ -1 & & 2 & 1 \\ & & -1 & 1 \end{bmatrix}$$

LU factorisation of this matrix gives $\mathbf{L} = \begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ -1 & -\frac{1}{3} & 1 & \\ & & -1 & 1 \end{bmatrix}$ $\mathbf{U} = \begin{bmatrix} 3 & -\frac{1}{3} & -\frac{1}{3} & \\ & \frac{5}{3} & -\frac{1}{5} & \\ & & \frac{8}{5} & -\frac{5}{8} \\ & & & \frac{3}{8} \end{bmatrix}$

LDU factorisation of $\mathbf{A}$ yields

$$\mathbf{L}' = \begin{bmatrix} 1 & & & \\ -\frac{1}{3} & 1 & & \\ -\frac{1}{3} & -\frac{1}{5} & 1 & \\ & & -\frac{3}{8} & 1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 3 & & & \\ & \frac{5}{3} & & \\ & & \frac{8}{5} & \\ & & & \frac{3}{8} \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 1 & -\frac{1}{3} & -\frac{1}{3} & \\ & 1 & -\frac{1}{5} & \\ & & 1 & -\frac{5}{8} \\ & & & 1 \end{bmatrix}$$

Notice that if $\mathbf{A}$ is symmetric then

$$\mathbf{U} = \mathbf{L}'^T$$

Hence

$$\mathbf{A} = \mathbf{L}'\mathbf{D}\mathbf{L}'^T$$

This has important consequences for storage of the matrix factors as it is only necessary to derive and store $\mathbf{L}'$ and $\mathbf{D}$. As power system network matrices are often symmetric this factorisation method finds widespread usage in power system computations. The LDU factorisation again forms the first stage of a three stage solution process.

Stage 1	Factorise	$\mathbf{A} = \mathbf{LDU} = \mathbf{L}'\mathbf{D}\mathbf{L}'^T$
Stage 2	Forward substitute to solve for $\mathbf{y}$	$\mathbf{L}'\mathbf{D}\mathbf{y} = \mathbf{b}$
Stage 3	Backward substitute to solve for $\mathbf{x}$	$\mathbf{U}\mathbf{x} = \mathbf{y} \quad (\mathbf{L}'^T\mathbf{x} = \mathbf{y})$

2.4.4 Bifactorisation

Bifactorisation is another factorisation technique similar to LU and LDU decomposition. Given an $(n \times n)$ matrix this method splits it into $2n$ factor matrices, each of order $(n \times n)$. The method produces n left hand factor matrices $\mathbf{L}^{(1)}, \mathbf{L}^{(2)}, \ldots, \mathbf{L}^{(n)}$ and n right hand factor matrices $\mathbf{R}^{(1)}, \mathbf{R}^{(2)}, \ldots, \mathbf{R}^{(n)}$. The factor matrices satisfy the constraint that

$$\mathbf{L}^{(n)}.\mathbf{L}^{(n-1)} \ldots \mathbf{L}^{(2)}.\mathbf{L}^{(1)}.\mathbf{A}.\mathbf{R}^{(1)}.\mathbf{R}^{(2)} \ldots \mathbf{R}^{(n-1)}.\mathbf{R}^{(n)} = \mathbf{I} \tag{2.17}$$

where **I** is the unit matrix of the same order as **A**. Note that

$$\mathbf{A}^{-1} = \mathbf{R}^{(1)}.\mathbf{R}^{(2)}\ldots\mathbf{R}^{(n-1)}.\mathbf{R}^{(n)}.\mathbf{L}^{(n)}.\mathbf{L}^{(n-1)}\ldots\mathbf{L}^{(2)}.\mathbf{L}^{(1)} \tag{2.18}$$

One factor matrix exists for each row and column of the coefficient matrix and each factor matrix differs from the unit matrix by only one row, if it is a right hand factor, or one column if it is a left hand factor. Once again it is possible to merge the $2n$ factor matrices to create a single $(n \times n)$ factored matrix $\mathbf{A_F}$.

$$\mathbf{A_F} = \begin{bmatrix} L_{1,1} & R_{1,2} & R_{1,3} & R_{1,4} \\ L_{2,1} & L_{2,2} & R_{2,3} & R_{2,4} \\ L_{3,1} & L_{3,2} & L_{3,3} & R_{3,4} \\ L_{4,1} & L_{4,2} & L_{4,3} & L_{4,4} \end{bmatrix} \tag{2.19}$$

For a symmetric matrix $L_{i,j} = R_{j,i}$ and hence only the left hand factor elements need to be derived. A further saving on memory can be made by storing only the left hand factor elements so that the factored matrix becomes

$$\mathbf{A_F} = \begin{bmatrix} L_{1,1} & & & \\ L_{2,1} & L_{2,2} & & \\ L_{3,1} & L_{3,2} & L_{3,3} & \\ L_{4,1} & L_{4,2} & L_{4,3}\cdots & L_{4,4} \end{bmatrix} \tag{2.20}$$

Consider again equation (2.18). As $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ it is possible to write

$$\mathbf{x} = \mathbf{R}^{(1)}.\mathbf{R}^{(2)}\cdots\mathbf{R}^{(n-1)}.\mathbf{R}^{(n)}.\mathbf{L}^{(n)}.\mathbf{L}^{(n-1)}\cdots\mathbf{L}^{(2)}.\mathbf{L}^{(1)}.\mathbf{b} \tag{2.21}$$

Commencing from the left hand end the first operation involves post multiplying $\mathbf{R}^{(1)}$ by $\mathbf{R}^{(2)}$ yielding

$$\mathbf{x} = \mathbf{R}^{(1)}\mathbf{R}^{(2)} = \begin{bmatrix} 1 & a & b & c \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} . \begin{bmatrix} 1 & & & \\ & 1 & d & e \\ & & 1 & \\ & & & 1 \end{bmatrix} = \begin{bmatrix} 1 & a & ad+b & ae+c \\ & 1 & d & e \\ & & 1 & \\ & & & 1 \end{bmatrix} \tag{2.22}$$

where a, b, c, d, e are values produced by the factorisation of the coefficient matrix and

may, or may not, be zero. The result of post multiplying any $(n \times n)$ matrix by any other $(n \times n)$ matrix is itself an $(n \times n)$ matrix . By the time all the factor matrices have been multiplied $2n$ matrix multiplications will have occurred, each of which produces an $(n \times n)$ matrix result. The final operation to obtain the solution vector, $\mathbf{x}$, is the post multiplication of the product of the matrix factors (*i.e* the full matrix inverse) by the vector $\mathbf{b}$.

$$\mathbf{x} = \begin{bmatrix} k & l & m & n \\ o & p & q & r \\ s & t & u & v \\ w & x & y & z \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} kb_1 + lb_2 + mb_3 + nb_4 \\ ob_1 + pb_2 + qb_3 + rb_4 \\ sb_1 + tb_2 + ub_3 + vb_4 \\ wb_1 + xb_2 + yb_3 + zb_4 \end{bmatrix} \tag{2.23}$$

which produces an $(n \times 1)$ vector result.

Now consider performing the same factor multiplications but commencing from the right hand side. The final result is the same as in the previous case (2.23) but now the first operation is the post multiplication of $\mathbf{L}^{(1)}$ by b. *i.e.*

$$\mathbf{L}^{(1)}\mathbf{b} = \begin{bmatrix} k & & & \\ a & 1 & & \\ b & & 1 & \\ c & & & 1 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} kb_1 \\ (a+1)b_2 \\ (b+1)b_3 \\ (c+1)b_4 \end{bmatrix} \tag{2.24}$$

and the result is an $(n \times 1)$ vector. Subsequent operations involves post multiplying factor matrices by intermediate vector results to yield an $(n \times 1)$ vector result. Note that in multiplying from right to left no matrix inverse is generated but its effect is obtained.

Tinney [3] notes that this right to left multiplication, when applied to a dense matrix, requires n additions and $n^2 - n$ multiplication - additions to compute a solution whereas left to right multiplication requires $(n-1)(2n^3 + n)$ additions and $2n^2(n^2 + n)$ multiplications to yield the same solution. Right to left multiplication is therefore much more efficient and will yield faster solutions with the added advantage of requiring less intermediate storage.

As with LU and LDU factorisation the factor matrices are derived in n steps. At the k^{th} iteration the factor matrices $\mathbf{L}^{(k)}$ and $\mathbf{R}^{(k)}$ are determined and the coefficient matrix is updated to produce a new coefficient matrix $\mathbf{A}^k$.

The formulae for determining left and right factors at the k^{th} iteration step are

$$\mathbf{L}_{kk}^{(k)} = \frac{1}{a_{kk}^{(k-1)}} \quad \mathbf{R}_{kk}^{(k)} = 1 \tag{2.25}$$

$$\mathbf{L}_{ik}^{(k)} = \frac{-a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} \quad i = k+1, \ldots n \tag{2.26}$$

$$\mathbf{R}_{kj}^{(k)} = \frac{-a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}} \quad j = k+1, \ldots n \tag{2.27}$$

where $a_{kk}^{(k-1)}$ is referred to as the *pivot* or *pivotal element.*

It is clear that $L^{i,k} = R^{k,i}$ for a symmetric matrix and as only left hand factors need to be determined the rules for deriving the factors from a symmetric coefficient matrix become

$$\mathbf{L}_{k,k}^{(k)} = \frac{1}{a_{k,k}^{(k-1)}} \tag{2.28}$$

$$\mathbf{L}_{i,k}^{(k)} = \frac{-a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}} \tag{2.29}$$

where $\mathbf{L}_{i,k}^{(k)}$ is stored in $\mathbf{A}_{F_{k,i}}$, and $\mathbf{A}_F$ stores the compact factored matrix.

The coefficient matrix is updated according to

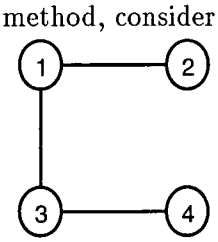
$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}.a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}} = a_{ij}^{(k-1)} + a_{ik}^{(k-1)}.\mathbf{A}_{F_{kj}}^{(k)} \quad \text{i,j=k+1, } \ldots \text{n} \tag{2.30}$$

Equations (2.29) and (2.30) imply that they must be applied to every matrix element $a_{i,j}$ for which $i, j = k+1, \ldots, n$. For a sparse matrix this is clearly inefficient as many of these elements will contain zeroes. Equations (2.29) and (2.30) only need to be applied to non-zero elements $a_{i,j}$ for which

$$k+1 \leq i, j \leq n \tag{2.31}$$

As an example of the bifactorisation method, consider the factorisation of the matrix

$$\mathbf{A}^{(0)} = \begin{bmatrix} 3 & -1 & -1 & \\ -1 & 2 & & \\ -1 & & 2 & -1 \\ & & -1 & 1 \end{bmatrix}$$



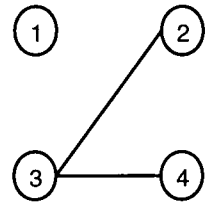
At the first step $k = 1$. Applying (2.29) to row 1 of the matrix produces the first left

hand factor matrix $\mathbf{L}^{(1)}$.

$$\mathbf{L}^{(1)} = \begin{bmatrix} \frac{1}{3} & & & \\ \frac{1}{3} & 1 & & \\ \frac{1}{3} & & 1 & \\ & & & 1 \end{bmatrix}$$

Applying (2.30) to row 1 of the matrix causes the remaining rows of the matrix to be modified. The result is

$$\mathbf{A}^{(1)} = \begin{bmatrix} 1 & & & \\ & \frac{5}{3} & -\frac{1}{3} & \\ & -\frac{1}{3} & \frac{5}{3} & -1 \\ & & -1 & 1 \end{bmatrix}$$



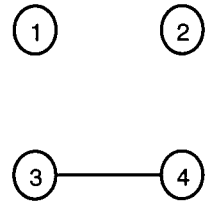
Notice that fill-ins occur at $(2,3)$ and $(3,2)$ and that rows 2 and 3 of the matrix have been updated as nodes 2 and 3 are the neighbours of node 1.

At the second factorisation step $k = 2$. Applying (2.29) gives

$$\mathbf{L}^{(2)} = \begin{bmatrix} 1 & & & \\ & \frac{3}{5} & & \\ & \frac{1}{5} & 1 & \\ & & & 1 \end{bmatrix}$$

and using (2.30) to modify rows 3 and 4 produces

$$\mathbf{A}^{(2)} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \frac{8}{5} & -1 \\ & & -1 & 1 \end{bmatrix}$$



No fill-ins occur during this step and row 3 is updated as node 3 is the neighbour of node 2.

Step 3 ($k = 3$) results in

$$\mathbf{L}^{(3)} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \frac{5}{8} & \\ & & \frac{5}{8} & 1 \end{bmatrix}$$

and

$$\mathbf{A}^{(3)} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & \frac{3}{8} \end{bmatrix} \begin{matrix} \textcircled{1} & \textcircled{2} \\ & \\ \textcircled{3} & \textcircled{4} \end{matrix}$$

Again no fill-ins occur and row 4 is modified as node 4 is the neighbour of node 3.

The final step ($k = 4$) simply creates $\mathbf{L}^{(4)}$. No updating of the coefficient matrix is required as all nodes have been eliminated.

$$\mathbf{L}^{(4)} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & \frac{8}{3} \end{bmatrix} \quad \mathbf{A}^{(4)} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

A further point to note about symmetric coefficient matrices and their factored counterparts is that all the information about the matrix is contained in both upper and lower triangles of the matrix. The rules for factorising and updating the coefficient matrix need only be applied to this triangular form, reducing the computation time as well as reducing the amount of memory needed to store the matrix. Given the matrix $\mathbf{A}$ in triangular form the bifactorisation method can be used to produce a triangular factored matrix. After the n^{th} iteration step the coefficient matrix, $\mathbf{A}$, has been reduced to a unit matrix and n factors have been produced and stored in the factored matrix. Note that the $\mathbf{R}^{(n)}$ factor is a unit matrix and can be disregarded as it has no effect in the subsequent multiplications.

Although the admittance matrix, $\mathbf{Y}$, is often symmetrical there are situations under which it is only incidence symmetric[2]. These situations arise from the presence of system components such as quadrature boosters, which are transformers with complex transformation ratios. Their effect is to introduce unequal admittances into symmetric locations of the admittance matrix and the techniques for storing and processing symmetric matrices can no longer be used. The square representation of the admittance matrix must be used and factorisation generates left and right hand factors for which $\mathbf{L}_{i,k}^{(k)} \neq \mathbf{R}_{k,i}^{(k)}$. A compact representation of the factors is no longer possible and both the left and right hand factors must now be stored explicitly. The formulae for generating factors from symmetric matrices (equations (2.28)- (2.29)) can no longer be used and equations (2.25)- (2.27) must be used instead.

[2]An incidence symmetric matrix is only symmetric in terms of element locations, not in terms of element values. When the element values are ignored the matrix is seen to be symmetric.

The test systems used in this thesis do not include devices which result in incidence symmetry and the admittance matrices of all the test systems are symmetric. This has allowed the more efficient symmetric storage and processing techniques to be used throughout.

2.5 Pivotal Ordering

When operating on a coefficient matrix to obtain the effect of its inverse it is not necessary to operate on rows or columns in the natural order in which they occur It is possible to process rows and columns in a different order so that a given diagonal element is selected as the pivot at a given iteration step.

There are three reasons for choosing to operate on a matrix in an order that is not necessarily the naturally determined order.

- increased numerical accuracy due to minimisation of round - off error.
- preservation of matrix sparsity.
- increase in computational efficiency.

These desirable properties result from minimizing the amount of fill-ins introduced during elimination. Minimizing the fill-ins reduces the amount of computation needed to yield a solution, thus increasing computational efficiency. Preserving the sparsity of the matrix also gives increased numerical accuracy as fewer round-off errors are introduced.

It has been observed that the matrices associated with power systems networks are diagonally dominant and by determining the elimination ordering based upon an examination of only the diagonal elements sufficient numerical accuracy is retained for most applications [2]. This allows the matrix ordering to be chosen so as to preserve sparsity and reduce memory requirements and computation time.

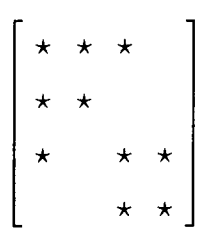
There are two main forms of ordering strategy used in matrix computations - *Pre - Ordering* and *Dynamic Ordering.*

2.5.1 Pre - Ordering

Pre - ordering strategies are used before processing the matrix and have short execution times. Such strategies cannot take account of changes in the coefficient matrix due to the factorisation process and are unlikely to produce the most optimal ordering. Despite this

pre - ordering strategies can be very useful for simple problems although they are often not very good at preserving sparsity.

As an example of a pre-ordering strategy consider the 'least number of connected branches' method. This orders matrix rows(columns) for elimination in ascending order of number of non - zero off diagonal elements. When two rows (columns) have the same number of non - zero elements the ordering becomes most efficient when these rows (columns) are taken in their naturally occurring order. For the matrix below the following ordering is obtained



Natural	*Ordered*
1	3
2	1
3	4
4	2

Note that the ordering strategy merely requires a knowledge of the location of non zero matrix elements. The actual values of these elements are irrelevant.

2.5.2 Dynamic Ordering

Dynamic ordering strategies differ from pre-ordering strategies in that they are used after each step of the factorisation algorithm to determine the most optimal order of elimination based upon an examination of the updated coefficient matrix. Such strategies slow down computation time, which is their main drawback, but they are very good at preserving sparsity. The simplest dynamic ordering strategy applies the 'least number of connected branches' algorithm after each iteration and this is the well known Minimum Degree algorithm [3]. Brameller et. al. [2] suggest the use of a semi - optimal ordering strategy. This has the advantage of off-line usage (like a pre-ordering) which does not slow down the actual computations yet still retains the sparsity preserving properties of a dynamic ordering strategy. The ordering is applied prior to processing of the matrix and uses the coefficient matrix sparsity pattern and a simulation of the factorisation process to determine the order which introduces least fill-in. This technique is very attractive if several solutions are to be obtained for matrices which have different numerical values but the same sparsity pattern as the same elimination order can be used for each matrix.

The semi-dynamic ordering algorithm operates by reading connection information for the required system from a datafile held on disk. The data in this file is used to establish

linked lists for each row which define the topology of the admittance matrix. These lists hold only column indices and no element values are stored, as depicted below.

$$\begin{bmatrix} \star & \star & \star & \\ \star & \star & & \\ \star & & \star & \star \\ & & \star & \star \end{bmatrix} \quad \begin{array}{l} 1 \rightarrow 2 \rightarrow 3 \rightarrow NULL \\ 1 \rightarrow 2 \rightarrow NULL \\ 1 \rightarrow 3 \rightarrow 4 \rightarrow NULL \\ 3 \rightarrow 4 \rightarrow NULL \end{array}$$

A suitable ordering strategy (*e.g.* Minimum Degree) is applied to determine the first row/column to be eliminated. The bifactorisation update rule (2.30) then determines where in the matrix fill-in will occur due to the elimination of this row and column. If a fill-in occurs in row k column i the topology list for row k is modified by the insertion of an entry for column i. After all fill-ins have been identified and the topology lists altered the list of the eliminated row is marked so that it will not be consulted further. Similarly all the other lists which make reference to the eliminated column have the entry for that column removed. The ordering strategy is again used to determine the next row and column to be eliminated and the necessary updating of lists is performed. This continues until all rows are marked as eliminated. The output of the ordering algorithm is a mapping array which specifies how the system nodes are to be renumbered so that when the nodes in the renumbered system are eliminated in ascending order, the optimal elimination ordering is being followed.

In the case of a symmetric matrix only the lower triangle needs to be stored to completely specify the matrix. Analysis of the ordering algorithms shows that if the linked lists store only the topological information for a triangular matrix a different ordering is produced than if the lists contain the topology information of the whole symmetric matrix. It is observed that the former case does not produce the optimum ordering of system nodes whilst the latter case does. Hence the ordering algorithm must use a full representation of the matrix whilst the actual solution for the algebraic network equations only makes use of a triangular matrix. Creating an ordering using the full matrix ensures that the triangular matrix used by the factorisation routine will be optimally ordered.

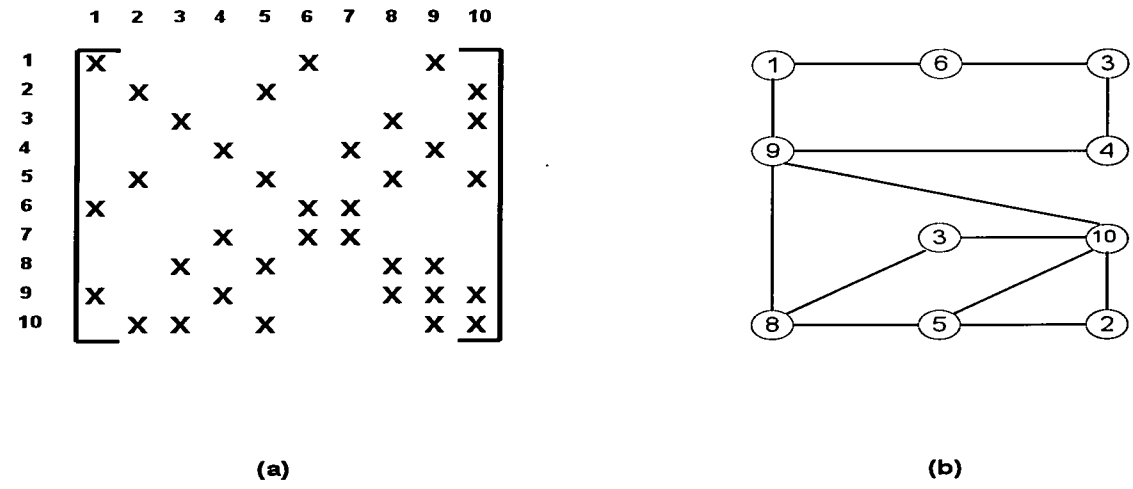


Figure 2.4: A simple 10 node graph (b) and its associated matrix (a)

2.6 Elimination Trees

The elimination tree is an extremely powerful tool that can be used to describe the precedence relationships which exist between nodes in the factorisation and substitution phases of LU decomposition. Liu [52] recognizes the elimination tree as a tool previously used in many different guises by a number of authors. First formalized by Schreiber [53], the elimination tree was applied to parallel processing by Jess [54] in the early eighties. Such trees were first introduced in a power systems context in the mid-eighties [55] and since that time they have been used extensively in examining the parallelism in the solution of large sparse sets of linear equations.

The duality between graphs and matrices is well known [51]. For a particular graph, $G(A) = V(A), E(A)$, with vertices $V(A)$ and edges $E(A)$ there exists an associated incidence matrix. The matrix contains numeric data when the edges $E(A)$ have weights associated with them. In an electrical network the edge weights are the admittances (or impedances) associated with each edge, or circuit branch. The matrix then becomes the admittance (impedance) matrix of the system. Figure 2.4 shows the duality of graphs and matrices with a simple 10 node network example.

If the matrix, **A**, is the coefficient matrix of a set of linear equations then the matrix may be factorised using one the Gauss-based L(D)U decomposition techniques. As elimination proceeds fill-ins are generated in the matrix **A** creating the filled matrix, **F**. The creation of **F** is equivalent to introducing extra connections in the graph, $G(A)$ to create the filled graph, $G(F)$. Figure 2.5 shows the filled graph, $G(F)$ for the simple 10 node example given

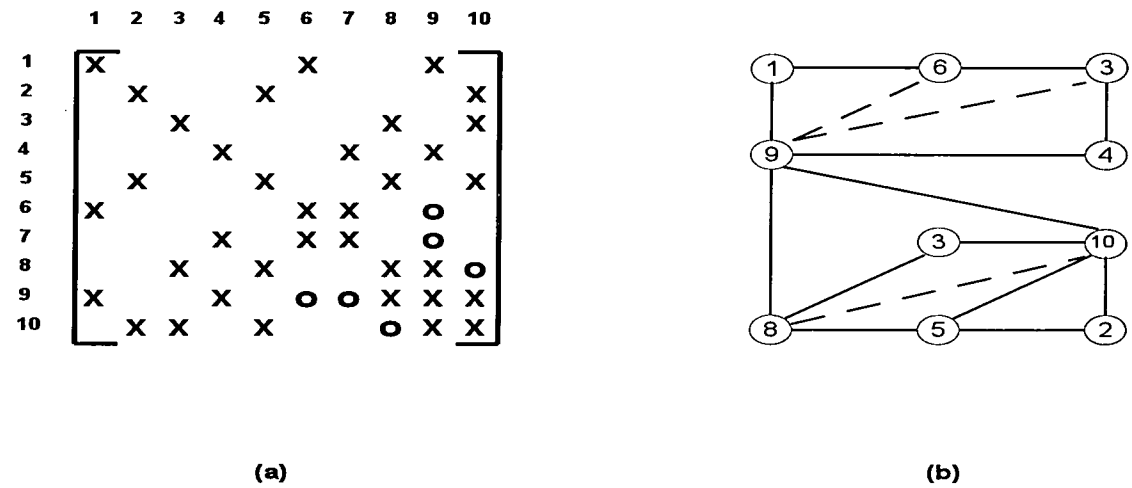


Figure 2.5: Filled graph (b) and associated matrix (a) for the simple 10 node example

above along with the corresponding filled matrix, **F**.

The elimination tree of a set of n linear equations is a rooted tree with n vertices labeled 1 to n. The labeling of the vertices in the tree corresponds to columns in the filled matrix, **F**. The *root* of the tree is the last column, n. Tracing the paths through the elimination tree gives the factorisation paths for given nodes. The *factorisation path* for a node, k, is an ordered list of nodes starting at k. The list contains the index of the first non-zero element below the diagonal in column k of the filled matrix **F**. This element is then taken as a column and the first non-zero element below the diagonal in this column of the filled matrix is added to the list. The process is repeated until no more unvisited nodes exist below the diagonal in column k. The elimination tree is generated by tracing the factorisation path of the last node, n. A node such as k is referred to as a *child* or *descendant* node and the node corresponding to the first non-zero below the diagonal is referred to as the *parent* or *ancestor* node. The root of the tree is unique in that it has no parent node and leaf nodes of the tree have no children. Every other node is both a child and a parent of another node in the tree. The elimination tree for the simple 10 node example system is shown in Figure 2.6. In this tree, node 10 is the root whilst nodes 1, 2, 3, 4 are leaf nodes. Node 6 is a child node and node 7 is its parent. Node 6 is also the parent of node 1. The factorisation path for node 6 is the list 6, 1.

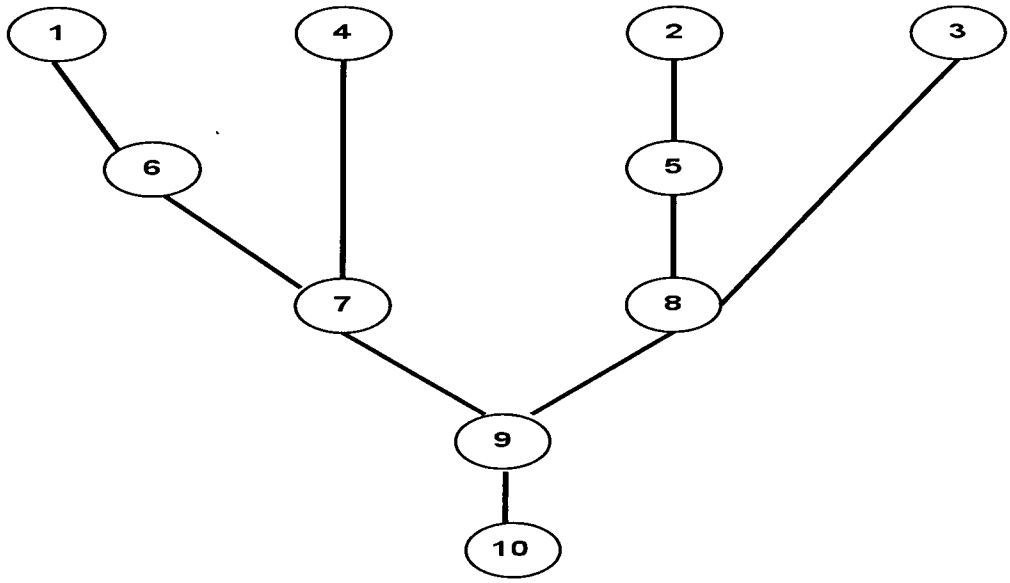


Figure 2.6: The elimination tree of the 10 node example

2.7 Near Optimal Ordering Strategies

Much research attention has been given to the subject of near optimal ordering in the last few decades and this section presents a brief summary of the most popular and effective methods developed to date. The various ordering schemes can be characterized by the amount of fill-in they introduce and the effect they have on the shape of the elimination tree.

2.7.1 The Minimum Degree Algorithm

The *degree* of a node in the graph $G(A)$ is the number of other nodes to which that node is connected *i.e.* the number of connections branching out from that node. For example, node 1 of Figure 2.4(b) has degree 2 whilst node 9 has degree 4. Comparing Figure 2.4(b) with Figure 2.4(a) shows the degree of a node to be equivalent to the number of non zero off diagonal entries in the row of the matrix corresponding to that node.

The Minimum Degree algorithm [3, 56, 2] determines the order of elimination based on an analysis of the degree of the nodes involved. The aim of the algorithm is to minimize the number of fill-ins introduced as a result of the elimination process as this reduces the amount of computation required to achieve a solution. At each stage of the elimination process the algorithm chooses the next node to be eliminated as the one which has the smallest degree. In the event of a two or more nodes having the same, smallest degree the choice of which

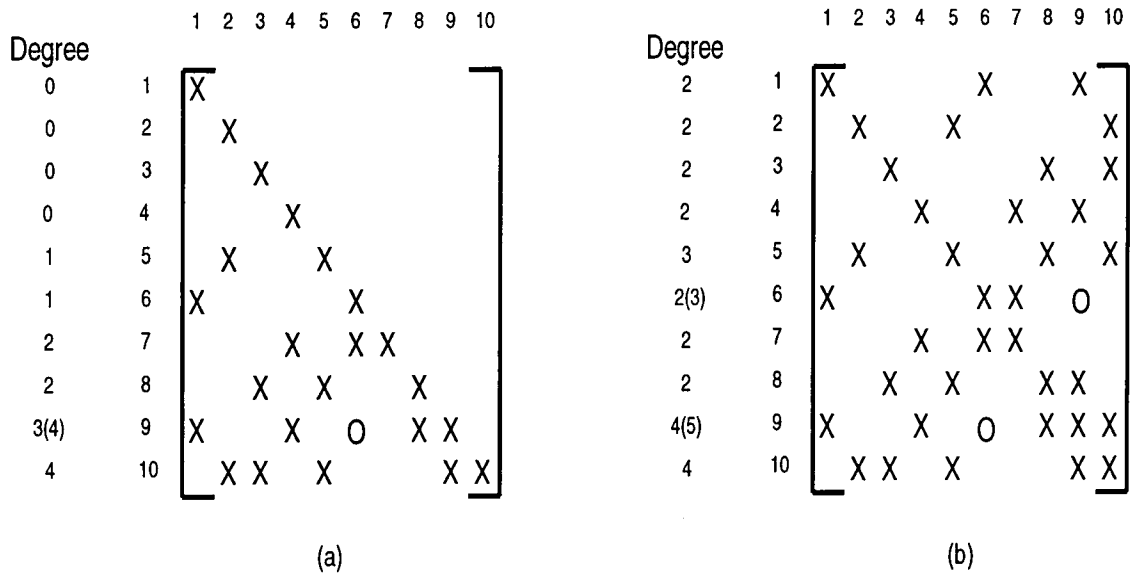


Figure 2.7: The effect of storage scheme on nodal degree a) triangular storage b) square storage

node to eliminate is an arbitrary choice from the set of nodes with smallest degree. The algorithm has to be applied dynamically or semi-dynamically as the degree of the nodes in $G(A)$ changes as elimination progresses. This is due to the removal of connections due to elimination and the introduction of new connections due to fill-in. Allowing the algorithm to take account of the changing degree of the nodes ensures that the minimum fill ordering is produced.

A description has already been given of how the sparse coefficient matrix, $\mathbf{A}$ (or $\mathbf{Y}$), maybe stored efficiently in memory. The scheme exploits the symmetry of the matrix by storing only the upper or lower triangle of the matrix as either of these contains all the numeric information required. When applying an ordering strategy to the coefficient matrix it is not sufficient to use a triangular representation as the resulting ordering differs from that generated using the full coefficient matrix. This is due to the observation that the degree of a node is equivalent to the number of off diagonal non zero elements in the corresponding row of the matrix. This observation holds true only for a complete matrix and does not hold for the triangular representation. Figure 2.7 shows a triangular and complete representation of the coefficient matrix for the 10 node example system of Figure 2.4.

It is easy to see that row 1 in Figure 2.7(b) has the correct degree of 2 whilst row 1 in Figure 2.7(a) has degree 0. Furthermore, if node 1 is eliminated from the graph $G(A)$

a fill-in is introduced at elements $(6,9)$ and $(9,6)$ in the complete matrix, increasing the degree of nodes 9 and 6 by 1. In the triangular matrix fill-in is only introduced at element $(9,1)$. The degree of node 9 is correctly increased by 1 whereas node 6 incorrectly retains its original degree.

In creating a computer program to determine the elimination ordering it is easier to use the square representation of the sparse matrix. It is possible to use the triangular matrix representation in the ordering algorithm but this requires greater programming effort as an extra data structure has to be used to store information about the degree of each node. The only situation where this might be useful would be in processing a very large system on a computer with a small amount of memory. Given the cheapness and availability of modern memory chips, and the use of fast, cache supported virtual memory, this situation is unlikely to occur. The type of storage used by the ordering routine does not create any problems for the solution algorithm as the ordering routine is applied off-line prior to solution.

2.7.2 The Minimum Length Algorithm

Like the Minimum Degree algorithm the Minimum Length algorithm orders the elimination of nodes from the network so as to minimize a given constraint, in this case the length of the elimination tree associated with the network. At each stage of the elimination process, the next node selected for elimination is the one which has the shortest path length. In the event of a tie the choice is arbitrary. Here path length is defined to be the length from the initial (root) node of the tree to the node currently under consideration. All nodes start with a path length of zero and, at each stage of the elimination the path length of nodes referenced by the elimination are updated using a simple formula [56]. Suppose node k has just been eliminated and that row k of the matrix has entries in columns i and j. Clearly an update or fill-in will be made to elements a_{ij} and a_{ji}. a_{ii} will be similarly updated and the path length of a_{ii}, written as d_{ii}, is modified according to

$$d_{ii} = max[d_{kk} + 2, d_{ii} + 1]$$

The path length of the last node to be eliminated is equal to the length of the critical path in the elimination tree.

Betancourt and Taylor [56, 57] both observe that the Minimum Length algorithm should result in a short critical path length for the elimination tree. As the algorithm does not

consider the degree of the nodes it cannot be expected to have good sparsity preserving properties. Whilst this algorithm is found to give shorter trees than the Minimum Degree algorithm, it is also found that it introduces significantly more fill-in.

2.7.3 The Minimum Degree Minimum Length Algorithm

Both of the algorithms described above operate by attempting to minimize a certain parameter. The decision as to which node to eliminate next is based purely on which node minimizes the desired parameter. When two equally likely nodes are encountered there is no protocol for breaking the tie and an arbitrary choice has to be made. Many implementations simply choose the first or last tied node encountered so as to ease the programming of the algorithm. It has been found [56, 57] that using a second criterion to resolve conflicts between tied nodes gives a significant improvement in the resultant ordering. The Minimum Degree Minimum Length (MDML) algorithm uses path length as the criterion used to resolve tie break situations. At each stage of the elimination, the next node to be eliminated is chosen to be the one with minimum degree. If more than one node has minimum degree then the path lengths of the tied nodes are examined. The node with the minimum path length is chosen as the next to be eliminated. If more than one node has minimum path length then the choice is again arbitrary. As the primary selection criteria is the degree of the nodes the MDML algorithm has the sparsity preserving properties of the MD approach but the secondary selection criteria reduces path lengths. MDML-ordered systems have similar amounts of fill-in to their MD counterparts but with shorter elimination trees.

2.7.4 The Minimum Length Minimum Degree Algorithm

It is also possible to use path length as the primary selection criterion with the degree of the nodes used as a secondary selection criterion in the event of a tie. This gives rise to the Minimum Length Minimum Degree (MLMD) algorithm. This algorithm results in short path lengths, as in the ML approach, but also reduces the amount of fill-in produced. However the fill-in is still significantly greater than that introduced by the Minimum Degree algorithm.

2.7.5 The Minimum Degree Minimum Length Least Recently Used Algorithm

Taylor [57] introduces a variant of the MDML strategy which he refers to as the Minimum Degree Minimum Length Least Recently Used (MDMLLRU) algorithm. A third selection criterion is used to resolve the conflict between nodes which remain tied after the application of the first two selection criteria. This criterion selects the next node for elimination as the tied node which has been least recently referenced by the elimination process. Suppose node k has just been eliminated and that row k of the matrix contains entries in columns i and j. Nodes i and j are thus referenced in updating the matrix $\mathbf{A}$ following the elimination of k. Associated with each node in the network is a timestamp which indicates when that node was last referenced. These timestamps are altered each time a node is referenced during an update or fill-in operation. When a tiebreak occurs the timestamps of the tied nodes are examined and the one with the smallest timestamp is chosen. If the tie still cannot be broken the choice is once again arbitrary. The timestamps used in this method do not have to be actual times but may be a simple integer value indicating at which step in the elimination the given node was last referenced.

The use of the Least Recently Used criterion ensures that the MDMLLRU algorithm does not continue following the same path through the elimination tree when a tie occurs but allows it to jump to other paths in the tree. Instead of focusing its attention on a particular part of the tree until that path has been eliminated, the MDMLLRU algorithm distributes its focus more evenly over the entire tree. This results in elimination trees which are both short and wide. The use of Minimum Degree as the primary selection criterion maintains good sparsity preserving properties.

2.7.6 Comparative Analysis of the Ordering Methods

A simple computer program was written to allow an analysis to be undertaken on the performance of each of the algorithms described above. The selected ordering technique was repeatedly applied to the given system, usually the CEGB 734 node system, for 1000 iterations. Before each iteration the nodes in the test network were randomly reordered to something other than their natural order. The ordering algorithm was applied and the path length and fill-in resulting from the chosen algorithm were recorded. At the end of the test run these figures were used to derive a set of statistics which characterized the performance

Ordering Scheme	Fill-in		Path length		
	Minimum	Mean	Minimum	Mean	Maximum
MD	616	636	28	38	45
MDML	616	628	23	31	37
MLMD	877	970	22	26	31
MDMLLRU	617	629	24	30	33

Table 2.1: Statistical performance of the ordering algorithms

of the chosen ordering scheme in terms of path length and fill-in. Applying this test to the same system for each ordering scheme allows the performance of these algorithms to be compared. Table 2.1 shows the results of this testing.

As Table 2.1 shows, the ordering algorithms behave much as expected. The Minimum Degree algorithm gives a small amount of fill-in but results in long elimination trees. Minimum Length reduces the length of the tree at the expense of the amount of fill-in. Minimum Degree Minimum Length improves on the situation by maintaining the low fill-in of Minimum Degree whilst reducing the length of the tree. Minimum Length Minimum Degree has good length qualities but poor fill-in performance. The best ordering algorithm is the MDMLLRU algorithm, which has short tree lengths and good fill-in performance. As Taylor predicts, the use of this algorithm gives rise to short, broad trees. Broad trees are desirable to facilitate the partitioning of the tree into subtrees for parallel processing whilst the short critical path length gives shorter execution times than those produced as a result of other ordering algorithms. Table 2.2 shows the effect of the different ordering strategies on the speed-up obtained by processing the CEGB network in parallel. The table clearly shows the beneficial effect of introducing tie-breaking criteria into the ordering strategy. Each result was obtained by partitioning the reordered system into the same number of independent parts which were solved using four processors. The actual parallel solution algorithm used to achieve the solution is unimportant as all the results were obtained using the same solution algorithm. The relative speed-ups show the effect of ordering on speed-up.

It is observed that in Table 2.1 there are certain shortest paths and minimum fills encountered which differ considerably from mean path length and mean fill-in. This phenomenon merits explanation. Although they are often referred to as optimal ordering strategies, the ordering schemes outlined above can best be thought of as approximate, near optimal techniques. Consider a graph with n nodes - there are $n!$ different ways in which the nodes in the

Ordering Scheme	Fill-in	Path Length	Speed-up
MD	655	35	3.07
MDML	631	31	3.17
MDMLLRU	634	24	4.52

Table 2.2: The effect of elimination ordering on speed-up

graph can be renumbered (reordered). To find the optimal elimination ordering the ordering algorithm must examine all $n!$ possible reorderings. When n is of the order of 1000 this becomes an intractable problem and it is in fact NP-complete [58, 59]. It is not possible to examine all possible orderings in a reasonable time so the near optimal ordering algorithms work by optimizing the elimination based upon the initial network ordering provided. Unfortunately these techniques are sensitive to the initial ordering and certain initial orderings result in better than average elimination orderings, as characterized by short path lengths and low fill-in. These solutions cannot in any way be considered optimal unless the whole solution space has been searched - it is always possible that a better reordering may be found in a different area of the solution space. The observed shortest paths and minimum fill-in of Table 2.1 are a direct result of randomly ordering the network before applying the chosen ordering algorithm. The initial random ordering causes the program to leap around the $n!$ search space and it occasionally happens upon a better than average resultant ordering. A similar argument explains why worse than average orderings are also encountered. Chapter 7 proposes a technique based on the use of genetic algorithms, which may help in rapidly locating the better than average orderings which exist within the solution space.

2.7.7 Deriving the Elimination Tree

There are a number of methods available for deriving the elimination tree from the filled graph of a given system. One technique eliminates all off-diagonal non zeroes from the filled graph except for the first non zero below the diagonal in each column. This matrix, referred to as $\mathbf{F}_t$ has a tree structured graph $G(\mathbf{F}_t)$ associated with it. This graph depends entirely on the structure of the original matrix $\mathbf{A}$ and its initial ordering and is the elimination tree, $T(\mathbf{A})$, for the system described by $\mathbf{A}$, $G(\mathbf{A})$ and $\mathbf{F}$, $G(\mathbf{F})$.

Another method of obtaining the elimination tree is to perform a depth first search using the data in F. Depth first search [58] is an efficient recursive algorithm for systematically visiting vertices in a tree. The search starts at the root of the tree and travels to unvisited

nodes along a path until it reaches the end (leaf node) of the path. The search then recurses back to previously visited nodes and from there travels to unvisited nodes on different paths until a leaf is reached again. Eventually all the nodes in the tree are visited and the search recurses back to the root. This approach is equivalent to starting with the last row of the matrix, $\mathbf{F}_t$, corresponding to node n, as the root of the tree. The tree is derived as the factorisation path of node n as follows. From row n, take the first non-zero to the left of the diagonal as the first entry in the ordered list. Take this as a row and place the index of its first entry to the left of the diagonal in the list. Repeat the process until a row is reached which has no non-zeroes to the left of the diagonal. This corresponds to a leaf node. Now trace back through the path of visited nodes to the first row which has unvisited nodes to the left of the diagonal. Place the index of the first unvisited entry to the left of the diagonal in the list and begin the search again, backtracking as necessary. The search continues until the list contains n entries. Sedgewick [58] provides a more detailed discussion of the depth first search algorithm. The algorithm is particularly suitable for computer implementation and the program which automatically plots the tree diagrams shown in this thesis is based on the use of a depth first search method.

2.8 Implementing a Sequential Solution of the Network Equations

A program which solves the algebraic network equations has two main components - an offline ordering routine to determine the optimal elimination order and the triangular factorisation (*eg* bifactorisation) routine which solves the equations for the unknown values. This section considers some of the practical aspects of implementing a sequential solution.

2.8.1 Storage of Sparse Matrices

The usual method of representing a matrix within a computer program is as a two dimensional array. Given a sparse matrix it is inefficient to store all the zero elements as these do not contribute to the solution. A better approach is to store only the non - zero matrix elements, [2, 60]. Under this scheme each row of the matrix is stored in memory as a linked list of the row elements, with only the non - zero elements being stored. Also stored alongside each element is its column index. The storage method is illustrated graphically in Figure 2.8. This data structure is implemented using a standard form of linked list created and

$$\begin{array}{lcccl} Value & -1 & & -1 & \\ Column & 2 & \longrightarrow & 3 & \longrightarrow Null \end{array}$$

$$\begin{array}{lcl} Value & -1 & \\ Column & 1 & \longrightarrow Null \end{array}$$

$$\begin{array}{lcccl} Value & -1 & & -1 & \\ Column & 1 & \longrightarrow & 4 & \longrightarrow Null \end{array}$$

$$\begin{array}{lcl} Value & -1 & \\ Column & 3 & \longrightarrow Null \end{array}$$

Figure 2.8: Storage Of Sparse Matrix Rows In Linked Lists

Diag	*Irap*	*Noze*
$3.0 + j4.0$	1140	1
$2.0 - j1.0$	1180	0
$2.0 + j5.0$	1200	1
$1.0 + j0$	1240	0

Figure 2.9: Storage Of Extra Information About The Matrix

managed by functions similar to those presented by Kelley & Pohl [61]. In the C language, each node of the linked list is a structure which has two fields

- *val* - A double floating point complex variable holding the value of the matrix element
- *col_no* - Integer variable holding column index of this element

Other information about the matrix also needs to be stored and this can be held in a two dimensional array with n rows, where n is the number of rows in the matrix. This arrangement, shown in Figure 2.9, allows diagonal elements to be accessed quickly and provides for easy searching of the row linked lists. The three fields of the array are

- *diag* - Double floating point complex value holding the diagonal matrix element value
- *irap* - Pointer to the memory address of the head of the linked list for this row
- *noze* - Integer variable holding number of non - zero entries in row

2.8.2 Determination of Elimination Ordering

The ordering algorithm operates by reading network connection data from the data file and is used prior to setting up the admittance matrix to determine the required order of

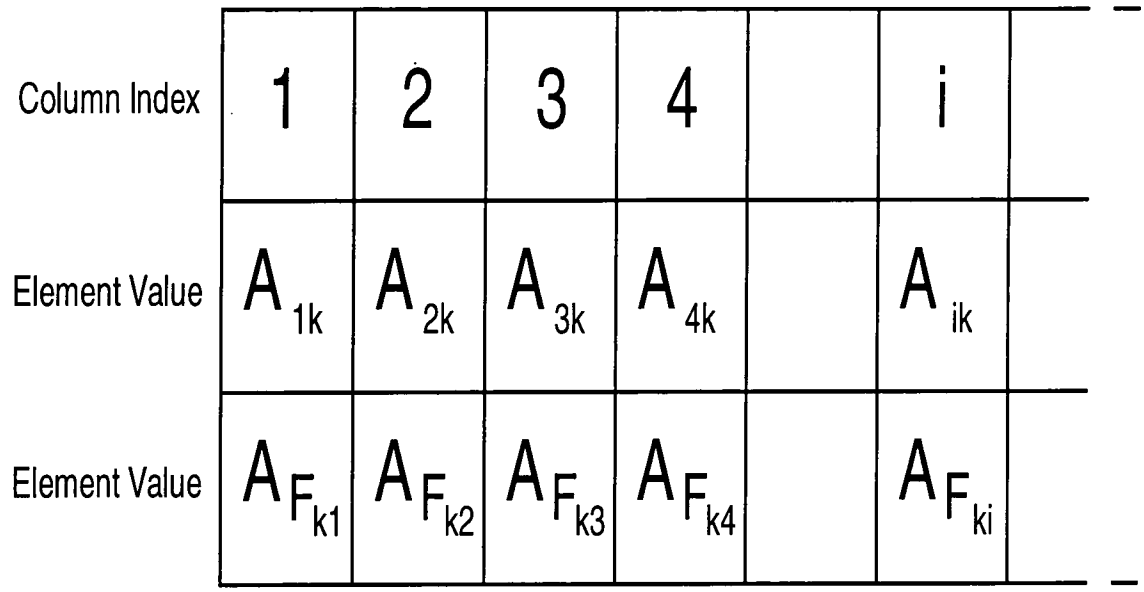


Figure 2.10: The information array used for updating

elimination. This is used to establish a mapping between physical system node numbers and the required new node numbers.

A selection sort algorithm, adapted from one presented by Sedgewick [58], can be used to examine the topology lists of the matrix and determine the one which has the lowest value of a given parameter as the next one to be eliminated. If more than one list has the lowest value then the next row to be eliminated is taken to be the first one occurring in the matrix. The selection sort routine is employed between simulated elimination steps to identify the next row/column for elimination. The factorisation routine forms the admittance matrix by reading in data from the disk file and renumbering each pair of node numbers before the data associated with them is inserted into the admittance matrix.

2.8.3 Coefficient Matrix Factorisation Using Bifactorisation

Having stored the coefficient matrix as a set of linked lists and determined the order of elimination of matrix rows, the factored matrix, $\mathbf{A}_F$, can be created as described previously. The formulae for obtaining the elements of the factored matrix are those presented in Section 2.4.4. Whilst determining the factors of the row being eliminated an array is created which holds holds the column index and the value of $\mathbf{A}_{F_{ki}}$ for each entry in the row linked list. This array provides for fast and efficient updating of the matrix elements as entries of the coefficient matrix are updated according to the entries of this array.

Suppose we are in the pth column of the array. $\mathbf{A}_{pp}$ is modified by adding to it $\mathbf{A}_{pk} * \mathbf{A}_{F_{kp}}$

yielding

$$\mathbf{A}_{pp}^{(k)} = \mathbf{A}_{pp}^{(k-1)} + \mathbf{A}_{pk}^{(k-1)} * \mathbf{A}_{F_{kp}}^{(k)}$$

Then the elements $p+1, \ldots n$ are used according to

$$\mathbf{A}_{pl}^{(k)} = \mathbf{A}_{pl}^{(k-1)} + \mathbf{A}_{pk}^{(k-1)} * \mathbf{A}_{F_{kl}}^{(k)}$$

where $l = p+1, \ldots n$. If at any step $\mathbf{A}_{pl}$ does not exist in the row linked list for row p it is assumed to be zero and

$$\mathbf{A}_{pl}^{(k)} = \mathbf{A}_{pk}^{(k-1)} * \mathbf{A}_{F_{kl}}^{(k)}$$

Once the factored matrix has been derived it is necessary to multiply its entries to achieve the right to left multiplication of equation (2.18) and hence the direct solution.

Multiplication of $\mathbf{x}$ by all n left hand factors is performed followed by multiplication of all the right hand factors. Throughout the multiplication only non zero elements are used and efficient use is made of the fact that multiplication by unity is equivalent to no multiplication at all. After multiplication by the last factor $\mathbf{x}$ contains the solution to the system of equations.

2.9 Summary

This chapter has introduced the subject of power system modeling. Basic models of power system components have been presented and the derivation of these models is given in detail in Appendix B. Particular attention has been given to the representation of the network and how this gives rise to the linear equations which have to be solved in many power system computations. Efficient sequential algorithms for obtaining a direct solution of the network equations have been presented and sparse matrix techniques were introduced as an efficient method for representing and processing the coefficient matrix within a computer program. The operation of these algorithms can be examined by resorting to the elimination tree, a simple structure which identifies the precedence relationships in the factorisation and substitution phases of the algorithms. Elimination trees are extremely important in analysing the behaviour of both parallel and sequential solutions and a simple method for deriving the elimination tree of any coefficient matrix has been introduced. The use of near-optimal ordering methods has been presented as a way of reducing the amount of computation needed to solve the equations and thus minimizing the solution time. In

addition practical considerations for the implementation of a solution program have been examined.

The next chapter extends the discussion to focus upon parallel algorithms for solving the network equations. Many of the direct parallel methods are based upon the direct sequential methods presented in this chapter. Iterative methods, which are of limited use for sequential solutions, also provide effective parallel solutions. The relative merits of both approaches are considered in respect to the solution of power system network equations.

Chapter 3

Parallel Methods of Solving the Network Equations

3.1 Introduction

The solution of large sparse sets of linear equations is a common computational problem encountered in many branches of science and engineering. Such systems of equations appear in fields as diverse as analysis/simulation of electrical networks, finite element methods, structural analysis and analysis/simulation of hydraulic networks. Sequential methods for solving these equations have been presented but it is also possible to solve them using parallel processing techniques. Tylavsky et al. [50] note that parallel dense matrix algorithms are not competitive with sequential sparse matrix algorithms, but it is possible to create parallel sparse matrix algorithms by exploiting independences in the equations. Sparse matrix computations contain more inherent parallelism than their dense matrix counterparts but due to the irregular pattern of sparsity in power system matrices it has been difficult to find efficient sparse parallel methods [50].

Two flavours of parallel sparse solution exist - direct and iterative. These two distinct methods have different computational characteristics and are best suited to different types of problem. Direct methods are more suitable for power system problems [50] and much research effort has been concentrated on the development of parallel LU solution methods. Outside the power system field much work has been done on parallel implementations of the Cholesky factorisation techniques.

This chapter examines some issues in the solution of sparse sets of linear equations

on parallel computer architectures. The chapter begins with a discussion of direct and iterative methods and assesses which is most suitable for power system problems. Issues faced in the design and use of direct methods are then considered before existing techniques are presented. Cholesky methods are examined to illustrate the principles of common approaches and a summary of the LU techniques used in power systems work is presented.

3.2 Iterative Methods for Solving Linear Equations

The fundamental difference between direct and iterative methods is the number of passes through the algorithm required to give the complete solution. Direct methods apply heuristic rules to manipulate the equations and achieve an exact solution with only one pass through the algorithm. As the name suggests, iterative methods require more than one pass through the algorithm and the solution algorithms come in two distinct flavours, *stationary* techniques and *gradient descent* techniques [62]. All iterative methods operate by choosing trial values for the unknown variables and using iterative correction to improve on previous values. The true solution will not be obtained in practice and the iterative method must be terminated once a suitably accurate solution has been achieved. Three common iterative techniques will now be examined.

3.2.1 The Jacobi Method

The Jacobi method was one of the first iterative techniques developed and it is a stationary iterative technique. As convergence is only guaranteed in the presence of a dominant leading diagonal the coefficient matrix is usually scaled to give unit coefficients on the leading diagonal. Hence the linear equations $\mathbf{Ax} = \mathbf{b}$ may be written as

$$\begin{bmatrix} 1 & a_{12} & a_{13} & a_{1n} \\ a_{21} & 1 & a_{23} & a_{2n} \\ a_{31} & a_{32} & 1 & a_{1n} \\ \\ a_{n1} & a_{n2} & a_{n3} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \\ b_n \end{bmatrix} \tag{3.1}$$

The coefficient matrix may be expressed in terms of an upper and lower triangular component, $\mathbf{U}$ and $\mathbf{L}$ respectively, as

$$\mathbf{A} = \mathbf{I} - \mathbf{L} - \mathbf{U} \tag{3.2}$$

where

$$\mathbf{L} = \begin{bmatrix} 0 & & & \\ -a_{21} & 0 & & \\ -a_{31} & -a_{32} & 0 & \\ & & & \\ -a_{n1} & -a_{n2} & -a_{n,n-1} & 0 \end{bmatrix}$$

and

$$\mathbf{U} = \begin{bmatrix} 0 & -a_{12} & -a_{13} & -a_{1n} \\ & 0 & -a_{23} & -a_{2n} \\ & & & \\ & & 0 & -a_{n-1,n} \\ & & & 0 \end{bmatrix}$$

To solve the equations using the Jacobi method a trial vector, $\mathbf{x}^{(0)}$ is selected. It is usual to set all elements of this initial vector to zero. A new trial vector is derived at each iteration by modifying the trial vector of the previous iteration. This process of iterative modification continues until the solution converges to within some desired tolerance. The benefit of the Jacobi method is that convergence is guaranteed for diagonally dominant matrices [63].

At the k^{th} iteration the elements of the new trial vector $\mathbf{x}^{(k+1)}$ are derived according to

$$x_i^{(k+1)} = b_i - a_{i1}x_1^{(k)} - \cdots - a_{i,i-1}x_{i-1}^{(k)} - a_{i,i+1}x_{i+1}^{(k)} - \cdots - a_{in}x_n^{(k)} \tag{3.3}$$

The complete iteration step may be expressed in matrix notation as

$$\mathbf{x}^{(k+1)} = \mathbf{b} + (\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)} = \mathbf{b} + (\mathbf{I} - \mathbf{A})\mathbf{x}^{(k)} \tag{3.4}$$

Defining the residual vector $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ allows (3.3) to be expressed in terms of this residual [63].

$$x_i^{(k+1)} = \frac{r_i^{(k)}}{A_{i,i}} + x_i^{(k)} \tag{3.5}$$

If the coefficient matrix has been scaled to give unit diagonals

$$x_i^{(k+1)} = r_i^{(k)} + x_i^{(k)} \tag{3.6}$$

The complete iteration step is expressed in matrix notation as

$$\mathbf{x}^{(k+1)} = \mathbf{r}^{(k)} + \mathbf{x}^{(k)} \tag{3.7}$$

Equation (3.6) provides the key to the efficient parallel implementation of the Jacobi method. If the vectors are assigned to processors such that $x_i^{(k)}$ and $r_i^{(k)}$ reside on the same processor then (3.6) may be computed without the need for interprocessor communication. If $x_{(k)}$ has n elements then it is possible to calculate all n elements of $x^{(k+1)}$ in parallel using n processors. Calculation of the residual vector $\mathbf{r}^{(k)}$ requires interprocessor communication as a result of the matrix-vector multiplication $\mathbf{A}\mathbf{x}^{(k)}$. The decision to terminate the iterations is based on the residual norm $|\mathbf{r}^{(k)}|$ and the termination condition is

$$\frac{|\mathbf{r}^{(k)}|}{|\mathbf{b}|} \leq tolerance \tag{3.8}$$

The tolerance value is normally set to less than 0.001 to ensure that the solution vector is correct to at least three significant figures.

3.2.2 The Gauss-Seidel Method

The Gauss-Seidel method is similar to the Jacobi method and is also a stationary iterative method. The equations are once again expressed as in (3.1) and (3.2). A trial vector is selected and the elements of this vector are iteratively modified until the solution converges to within an acceptable tolerance.

At the k^{th} iteration step the values of $x_1, x_2, \ldots, x_{i-1}$ of the vector $x^{(k+1)}$ will have been derived from the previous trial vector $x^{(k)}$ but the values of $x_i, x_{i+1}, \ldots, x_n$ remain to be determined. The i^{th} element of $\mathbf{x}$ is modified according to

$$x_i^{(k+1)} = b_i - a_{i1}x_1^{(k+1)} - \cdots - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - \cdots - a_{in}x_n^{(k)} \tag{3.9}$$

Equation (3.9) may be written in matrix notation as

$$(\mathbf{I} - \mathbf{L})\mathbf{x}^{(k+1)} = \mathbf{b} + \mathbf{U}\mathbf{x}^{(k)} \tag{3.10}$$

The successive correction procedure is iterated until the error in the solution falls below some specified tolerance limit. The decision to terminate this iterative process is based on

the calculation of a residual norm. Termination occurs when

$$\frac{|\mathbf{r}^{(k)}|}{|\mathbf{b}|} \leq tolerance \tag{3.11}$$

where $\mathbf{r}^{(k)}$ is the residual and is calculated as

$$\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} \tag{3.12}$$

Again the tolerance value is normally set to less than 0.001 to ensure that the solution vector is correct to at least three significant figures.

The Gauss-Seidel algorithm has often been used in sequential power system analysis programs. With suitable modifications it is also possible to implement a parallel version of the Gauss-Seidel method. Suppose that the coefficient matrix can be reorganized to give it a block structure. Each matrix block can then be assigned to an individual processor of a parallel machine. A central pool of values for $x_i^{(k)}$ is maintained. On each iteration each processor applies the Gauss-Seidel algorithm to the nodes within its own matrix block and uses the values available in the pool at the start of the iteration. Upon completion of an iteration the processors send the modified values of $x_i^{(k+1)}$ back to the pool. The algorithm is asynchronous and each processor may begin a new iteration once the previous iteration is complete and values have been sent to the pool. It is likely that the parallel Gauss-Seidel solution of a given system will require more iterations than a sequential solution of that system due to the asynchronous nature of the algorithm. Even if the same number of iterations occur the parallel algorithm will require more *total* processing time due to the contention between processors accessing the central pool of $x_i^{(k)}$ values.

3.2.3 The Conjugate Gradient Method

The Conjugate Gradient method is a specific example of a category of iterative solution techniques known as gradient descent methods. Gradient descent methods are based on the premise that solving a set of n simultaneous equations is equivalent to locating the minimum of an error function in n-dimensional space. At each iteration the set of trial values for the variables are used to create a new set of values which correspond to a lower value of the error function. The location of the global minimum of the error function in the n-dimensional space corresponds to the solution of the set of simultaneous equations.

If $\overline{\mathbf{x}}$ is the vector of trial values then a residual vector, $\mathbf{r}$, can be calculated as

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\overline{\mathbf{x}} \tag{3.13}$$

The error function, h, may be defined as

$$h = \mathbf{r}^T\mathbf{A}^{-1}\mathbf{r} \tag{3.14}$$

If the matrix is positive definite symmetric[1] then the error function will have a positive value for all vector $\overline{\mathbf{x}}$ except for the correct solution $\overline{\mathbf{x}} = \mathbf{x}$ where $\mathbf{r} = 0$ and $h = 0$. The vector $\mathbf{x}_k$ represents a point in the n-dimensional space and the equation

$$\overline{\mathbf{x}} = \mathbf{x}_k + \alpha\mathbf{d}_k \tag{3.15}$$

defines a line which passes through $\mathbf{x}_k$ with a direction determined by $\mathbf{d}_k$. α is a parameter which is directly proportional to the distance of $\overline{\mathbf{x}}$ from $\mathbf{x}_k$. Note that $\mathbf{x}_k$ is the value of $\mathbf{x}$ obtained at the k^{th} iteration. The error function h varies quadratically [65, 62] with α and has a local minimum at

$$\frac{\delta h}{\delta \alpha} = 2\mathbf{d}_k^T[\alpha\mathbf{A}\mathbf{d}_k - \mathbf{r}_k] = 0 \tag{3.16}$$

All the gradient descent methods use the location of this local minimum to derive the next value of the trial vector according to

$$\alpha_k = \frac{\mathbf{d}_k^T\mathbf{r}_k}{\mathbf{d}_k^T\mathbf{A}\mathbf{d}_k} \qquad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{d}_k \tag{3.17}$$

The only difference between the various gradient descent methods is the choice of the direction vectors $\mathbf{d}_k$. In the Conjugate Gradient method the direction vectors are chosen to be a set of vectors $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_k$ which represent the steepest descent of the points $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k$. Additionally the $\mathbf{p}_k$ vectors are chosen to be conjugate (*i.e.* orthogonal with respect to $\mathbf{A}$). The vectors thus satisfy the condition

$$\mathbf{p}_i^T\mathbf{A}\mathbf{p}_j = 0 \qquad i \neq j \tag{3.18}$$

[1] A symmetric matrix is positive definite if all of its eigenvalues are positive [64]. Alternatively, the $n \times n$ symmetric matrix $\mathbf{A}$ is positive definite iff $\mathbf{x}^t\mathbf{A}\mathbf{x} > 0$ for every n-dimensional column vector $\mathbf{x} \neq 0$. Power system admittance matrices do not obey these conditions and are not positive definite. In many conditions they are almost positive definite and these techniques have been used with limited success.

To solve a set of equations using the Conjugate Gradient method initial values ($k = 0$) must be specified for $\mathbf{p}$ according to

$$\mathbf{p} = \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0 \tag{3.19}$$

$\mathbf{x}_0$ may be initialized to zero. At the k^{th} iteration

$$\mathbf{u}_k = \mathbf{A}\mathbf{p}_k \tag{3.20}$$

$$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{u}_k} \tag{3.21}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{3.22}$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{u}_k \tag{3.23}$$

$$\beta_k = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k} \tag{3.24}$$

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k + \mathbf{p}_k \tag{3.25}$$

Theoretically the correct solution is obtained after n iterations but if the equations are ill-conditioned or the matrix is densely populated then it may take more than n iterations to reach convergence.

Equations (3.20) to (3.25) are calculated on each iteration and the potential parallelism in the method is visible in equation (3.22) to (3.23). Neither of these equations depends on values calculated in the other and they can be computed concurrently. If there are n simultaneous equations and n processors are available in the parallel machine then the minimum number of processing steps taken on each iteration is $2n + 3[\log_2(n)] + 10$. This is $3[\log_2(n)] + 10$ steps more than the $2n$ minimum steps required by the Gauss-Seidel method but the Conjugate Gradient method is likely to achieve convergence significantly quicker.

The Conjugate Gradient method is particularly well suited to solving large sparse systems of equations, such as those occurring in power systems analysis. Unfortunately power system admittance matrices are not positive definite and the Conjugate Gradient equations must be changed to a form suitable for symmetric indefinite systems. The error function changes to become

$$h = \mathbf{r}^T \mathbf{r} \tag{3.26}$$

and

$$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{A}\mathbf{r}_k}{\mathbf{u}_k^T \mathbf{u}_k} \tag{3.27}$$

$$\beta_k = \frac{\mathbf{r}_{k+1}^T \mathbf{A}\mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{A}\mathbf{r}_k} \tag{3.28}$$

$$\mathbf{p}_i^T \mathbf{A}^2 \mathbf{p}_j = 0 \qquad i \neq j \tag{3.29}$$

Decker et al. [28] proposed a parallel method for solving the power system network equations which combines both the LU decomposition and Conjugate Gradient methods. Given the set of network equations in Bordered Block Diagonal Form

$$\begin{bmatrix} \mathbf{I}_1 \\ \mathbf{I}_2 \\ \vdots \\ \mathbf{I}_p \\ \mathbf{I}_s \end{bmatrix} = \begin{bmatrix} \mathbf{Y}_1 & & & & \hat{\mathbf{Y}}_1 \\ & \mathbf{Y}_2 & & & \hat{\mathbf{Y}}_2 \\ & & \ddots & & \vdots \\ & & & \mathbf{Y}_p & \hat{\mathbf{Y}}_p \\ \hat{\mathbf{Y}}_1^t & \hat{\mathbf{Y}}_2^t & \cdots & \hat{\mathbf{Y}}_p^t & \mathbf{Y}_s \end{bmatrix} \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \vdots \\ \mathbf{V}_p \\ \mathbf{V}_s \end{bmatrix} \tag{3.30}$$

Decker notes that Block Gaussian elimination may be used to solve this set of equations in two stages

1. *Step 1 :* Solve

$$\hat{\mathbf{Y}}_s \mathbf{V}_s = \hat{\mathbf{I}}_s \tag{3.31}$$

where

$$\hat{\mathbf{Y}}_s = \sum_{i=1}^{p} \bar{\mathbf{Y}}_i^t \mathbf{Y}_i^{-1} \bar{\mathbf{Y}}_i$$

$$\hat{\mathbf{I}}_s = \mathbf{I}_s - \sum_{i=1}^{p} \bar{\mathbf{Y}}_i^t \mathbf{Y}_i^{-1} \mathbf{I}_i$$

2. *Step 2 :* For $i = 1, 2, \ldots, p$, solve

$$\mathbf{Y}_i \mathbf{V}_i = \mathbf{I}_i - \bar{\mathbf{Y}}_i \mathbf{V}_s \tag{3.32}$$

Step 2 is inherently parallel and if p processors are available then the solutions for all p subnetworks can be obtained simultaneously. Step 1 is inherently serial and although it can be solved by a parallel direct method Decker does not advocate this approach as

the parallel implementation of direct methods is not an easy task [28]

. Instead Decker proposes the use of the Conjugate Gradient method to solve the cutset equation (3.31). This requires the formation of $\hat{\mathbf{Y}}_s\mathbf{V}^0$, $\hat{\mathbf{I}}_s$,$\hat{\mathbf{Y}}_s d^k$ and the residual $\mathbf{r}^0$ at each iteration. If the subnetwork equations (3.32) are solved by direct LU factorisation techniques the factors of $\mathbf{Y}_{i;i=1,\ldots,p}$ can be used to efficiently calculate $\hat{\mathbf{Y}}_s\mathbf{V}^0$, $\hat{\mathbf{I}}_s$ and $\hat{\mathbf{Y}}_s d^k$ in parallel.

Decker [28] formulated this approach as part of a transient stability simulation implemented on an array of 8 Transputers connected in a hypercube configuration. The solution of the network equations was actually implemented using a combined LU factorisation and preconditioned conjugate gradient method. Preconditioning techniques modify the residual vector formed at each iteration to accelerate convergence. Decker notes that the use of this conjugate gradient based method produces substantial reductions in total computation time when compared to the best sequential method, although the sequential method he uses for comparison utilizes only LU factorisation. Speed-up figures are provided for the complete transient stability program and these show a speed-up of between 1.2 and 3.9 with 2 and 8 processors respectively. Unfortunately no results are provided for just the network solution phase of the simulation. The use of preconditioning reduces the number of iterations required by the conjugate gradient method but does not always produce a similar reduction in computation time dues to the extra overheads introduced by the preconditioning calculations. The decomposition of the network is significant as it affects the load balancing of the method. A poor partitioning produces an imbalanced load which adversely affects performance. The network decomposition also has an affect on the number of iterations required by the conjugate gradient method. A poor decomposition produces ill-conditioned equations which take longer to converge, although this can usually be corrected through the use of suitable preconditioning techniques.

3.3 Direct vs Iterative Methods

The difference between the direct and iterative approaches can be characterized in terms of the number of steps required to yield a solution [62]. Table 3.1 illustrates the computational requirements of the two approaches for two common algorithms operating on a set of equations with n variables.

Direct methods often require more operations in total to yield a solution than iterative techniques but there are two major advantages of the direct methods. Firstly the fac-

Method	Number of steps			
	Factorisation	Substitution	Total	Iterations
LU Decomposition	$3(n-1)$	$5n-4$	$8n-7$	1
Gauss-Seidel	-	-	$2nK+[\log_2(n+1)]-1$	K

Table 3.1: Operation counts for direct and iterative solution schemes

torisation and substitution operations of direct methods (*e.g.* LU decomposition) may be performed separately. This allows easy solution of systems with multiple right hand sides as the coefficient matrix only needs to be factorised once. After factorisation the substitution operation may be used any number of times to solve for different right hand sides. As the number of right hand side solutions becomes large, the number of steps taken to yield a solution tends to $5n-4$ and the direct methods become significantly more efficient than iterative methods. Iterative techniques do not provide separate factorisation and substitution operations and $2nK+[\log_2(n+1)]-1$ operations are required to yield a solution for each right hand side vector. In a power system simulation much of the time is taken up in solving the same system of equations with different right hand side vectors and this explains why direct methods find such widespread usage in the field of power systems analysis.

The second advantage of direct methods is that they do not suffer the convergence problems which are prevalent in iterative techniques. The value K in Table 3.1 is the number of iterations required for the iterative method to produce a solution which meets some predefined tolerance limits. If the tolerance limits are narrow or the equations are poorly conditioned K can become quite large, significantly increasing the time taken to reach a solution. For power system problems it has often been found that iterative methods do not converge quickly enough to a solution of sufficient accuracy. If iterative methods are employed either approximate solutions or long solution times must be accepted, neither of which is appropriate for accurate, real-time simulations. The problems with convergence have tended to preclude the use of iterative techniques for the solution of power system linear equations.

This thesis is concerned only with the direct method of solution and iterative methods will not be considered further. The development of efficient parallel formulations of direct methods for sparse linear systems is an area of research currently eliciting much interest. Substantial parallelism exists in direct methods but only limited success has been achieved in the development of parallel formulations. There are two reasons for this. Firstly the

amount of computation involved is small in relation to the size of the system to be solved. Even a small amount of interprocessor communication can significantly alter the balance between computation and communication and this results in poor efficiencies. The second reason for the inefficient parallel solutions developed to date is that most of them are based on good sequential algorithms. It is not always the case that parallelising the best sequential algorithm will give the best parallel algorithm [63, 66, 44]. The goals of a serial algorithm are often inappropriate in a parallel environment. Numerous parallel direct methods have been developed and they have all been based upon Gaussian elimination or Cholesky factorisation. This thesis examines ways to improve upon the existing methods to provide a faster and more efficient method of solution.

Heath et al. [67] and Kumar et al.[63] observe that a complete direct sequential solution consists of four phases

- Ordering - find a good ordering of the matrix such that minimal fill-in is introduced
- Symbolic Factorisation - Determine the structure of the factor matrices and set up data structures to hold them
- Numeric Factorisation - Compute the factor matrices of the coefficient matrix, A
- Triangular Solution - Using the factor matrices and the right hand side vector perform the forward/backward substitution operations to determine the values in the unknown vector $\mathbf{x}$

In most power system computations the ordering and symbolic factorisation operations are required infrequently and there is not the same need for high performance which arises from the frequently used operations. As the admittance matrix of the system seldom changes, the numerical factorisation operation is also infrequently used. In real-time simulations it may be necessary to refactorise the admittance matrix in response to a change in the network topology. Inefficient numerical factorisations have no place in real-time simulations as they may cause the simulation to leave real-time. Consequently there has been a significant amount of research into the development of efficient parallel numeric factorisations [35, 32, 68, 31, 50, 46, 29] although the performance of methods developed to date is not very impressive. Tylavsky et al. [50] observe that much of the algorithm development has been purely theoretical and little software has actually been developed for parallel machines. The software which has been produced [32, 40] shows that a full factorization can

be achieved with a speed-up of about 2, whilst a speed-up of about 10 may be achieved if factorization is halted before the densest part of the matrix is encountered. The most frequently used operation in power system computations, and particularly in simulations, is the triangular solution operation. If power system analysis and simulation algorithms are to be efficient then efficient parallel triangular solution methods are required. This has also been the focus of much research attention [31, 32, 26, 37, 36] but again the results are disappointing with speed-ups seldom exceeding 3 or 4 [69, 70, 57]. The reasons for the poor performance have been analysed by Bialek [49].

This thesis concentrates on the numeric factorisation and triangular solution operations. Symbolic factorisation is not considered as efficient sequential techniques have already been developed for this operation [67, 71, 72]. As Heath [67] points out, there is little that can be done to parallelize the symbolic factorisation operation and this operation is perhaps best performed sequentially on a single processor in a multiprocessor array. Ordering techniques are considered here but only in relation to improving the amount of inherent parallelism exploited in the numeric factorisation and triangular solution operations. No consideration is given to the development of parallel ordering algorithms as highly efficient sequential techniques already exist.

3.4 Parallel Algorithms for Direct Solution

3.4.1 Granularity of Solution

To solve any problem in parallel requires that problem to be divided into separate tasks which can be assigned to individual processors. The solution of power system network equations requires the set of equations to be decomposed into independent subsets. Unfortunately the algebraic equations are not easy to decompose as they are global and relate to all nodes in the network. Conventional diakoptics techniques make use of either node - tearing or branch cutting to split the network into subnetworks, thus allowing a partitioning of the equations. Direct methods can then be used to solve individual subnetworks concurrently with the solutions being combined to give an overall solution for the algebraic equations. The number and size of subnetworks required really depends on the capabilities of the target parallel machine. *Granularity* is a qualitative measure of the size of the parallel tasks. Three levels of granularity can be identified and Kumar [63] defines them as

- Fine-grain parallelism - Parallelism at the level of individual floating point operations

- Medium-grain parallelism - Parallelism in performing groups of floating point operations, for example on an entire matrix row or column
- Coarse-grain parallelism - Parallelism in operating on independent groups of matrix rows / columns

The exploitation of fine-grain parallelism is not suitable for message passing, distributed memory machines due to the excessively high amount of interprocessor communication required. Fine-grain parallel solutions are really only suited to massively parallel computing platforms containing thousands of relatively simple processing units or to shared memory architectures. Message passing architectures are better married with the exploitation of medium and coarse grain parallelism and research has been directed toward developing solutions at these levels. When designing a parallel algorithm it is not possible to ignore the architecture of the machine on which that algorithm is to be executed. As Lin and Van Ness [33] point out

> *... the architectural differences between the machines is a far more important factor (in speed-up) than the variations in the way that the algorithm is applied.*

Coarse grain parallelism arises as a direct result of the sparsity of the coefficient matrix. Due to the sparse nature of the matrix it is possible to identify independent groups of rows and columns which may be processed in parallel. No such groups exist in dense systems and coarse grain parallelism cannot be exploited. Dense parallel solutions are therefore restricted to the usage of medium grain parallelism.

For a coarse grain solution the groups of independent matrix rows/columns correspond to subnetworks within the network. Individual subnetworks are processed by separate processors with a central coordinating processor being used to combine individual subnetwork results to give the overall result. The use of the central coordinating processor introduces a large sequential stage in the parallel algorithm and is a bottleneck in the process, limiting the efficiency of the solution of the equations. Any method which increases the amount of parallelism extracted from these equations and reduces the size of the sequential step will significantly increase the speed of solution.

3.4.2 Task Mapping and Load Balancing

Two of the fundamental issues in the design of any parallel program are those of task mapping and load balancing. Chapters 4 and 6 consider these issues in more detail but a

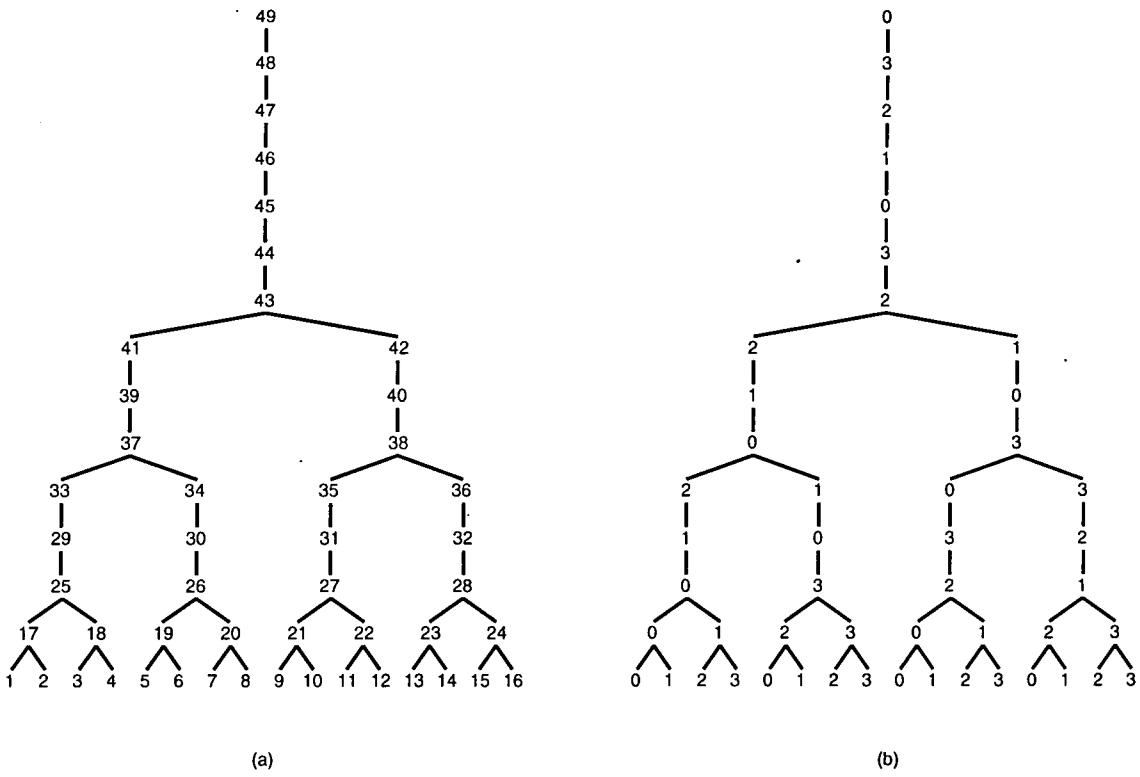


Figure 3.1: An elimination tree (a) and the wrap mapping strategy (b)

brief introduction is now presented.

Having partitioned the problem into a number of smaller subproblems, or tasks, it is necessary to assign processing resources to carry out these tasks. Tasks may be assigned to processors in any order but it is desirable to minimize the amount of communication between tasks on different processors. The task mapping operation must find the optimum placement of tasks which minimizes the delays associated with synchronisation and communication.

One of the most common task mappings used for Cholesky factorisations is known as the *wrap* mapping. Consider the elimination tree of Figure 3.1(a) which shows the precedence relationships between the tasks which make up the problem. Suppose that there are p processors available and there are i tasks in the problem. Generally $p < i$ and the wrap mapping assigns the i^{th} task to the processor $((i-1)\ mod\ p)$ [73]. The tasks in the tree are wrapped on to the available processors, as shown in Figure 3.1(b). (The numbers in the diagram refer to the processor to which each task is assigned). The wrap mapping assigns all potentially concurrent operations to different processors and distributes the communication load evenly across the processors. The volume of communication associated with this mapping scheme is high and Chapter 4 will introduce a more efficient strategy.

Load balancing is an issue of critical importance to the efficiency of any parallel algorithm. Maximum speed-up can only be attained if the computational workload is equally divided amongst all the processors and these processors are kept constantly busy [19]. Despite this many researchers seem to ignore the load balancing issue. In fact some go further and claim that load balancing is unimportant [44], a view which challenges the beliefs of the parallel computing community. Load balancing techniques may have one of two flavours - static or dynamic. Static approaches are the simplest and balance the load by a careful division of the problem into tasks and optimum assignment of tasks to processors based on some *a priori* knowledge. Dynamic strategies equalize the load whilst the program is running by moving tasks and data between the processors. Due to the relatively high communication requirements of dynamic strategies they can be inefficient when applied to distributed memory machines. The Transputer and its languages do not provide support for task migration [19] and other dynamic load balancing operations. This thesis considers only the simpler static load balancing strategies.

3.4.3 Ordering Strategies for Parallel Solutions

Sequential solutions employ ordering techniques to reduce the memory requirements and computational workload by minimizing the amount of fill-in resulting from eliminations. The same considerations apply to parallel solutions but there are additional reasons for reordering the coefficient matrix. Reordering the matrix alters the shape of the elimination tree associated with that system. In their natural order certain systems appear to have little exploitable parallelism. For example the elimination tree associated with a tridiagonal system is simply a linear chain of tasks. Applying an appropriate reordering can modify this system such that its elimination tree has a more traditional tree-like structure and this allows some concurrent execution of tasks to occur. One of the main goals of ordering for parallel solution is to rearrange the system to allow the exploitation of more of the potential parallelism in the problem.

Another frequently cited goal of ordering for parallel solution is to reduce the height of the elimination tree. Many authors [67, 69, 54, 74, 57] consider tree height to be critical in obtaining short execution times. Execution time is proportional to the length of the critical path through the elimination tree and critical path length is the same as tree height by definition. Applying orderings which minimize the height of the tree is an attempt to minimize the execution time of the parallel algorithm although Heath implies that there

is no proof that systems with shorter trees execute quicker than those with long trees. His view is that those authors who choose to use short trees base their choice on instinct rather than on theoretical proof. This author believes that in general shorter trees do give rise to shorter execution times and this view is supported by empirical observations made throughout the course of this research project.

3.5 Diakoptical Based Solution Methods

3.5.1 The Method of Diakoptics

The method of diakoptics may be used to divide the network equations into sets of equations corresponding to interconnected subnetworks. The techniques of diakoptical analysis [47, 75, 76, 77, 78] were developed in the 1950's by Gabriel Kron. The word diakoptic is derived from the Greek *kopto*, meaning to break or tear apart [47], and this neatly summarises the whole concept of diakoptical analysis. The essence of the technique is to solve a large system by tearing it into smaller subsystems which are then solved independently. The solution for the whole system can be found by combining and modifying the individual subsystem solutions. Developed as a method for solving large network problems, diakoptical analysis is an ideal approach for the treatment of electrical power systems. In fact the diakoptics method was devised by Kron as a solution to a particular power engineering problem [47]. For diakoptical solution the power network must be partitioned into subnetworks and the partitioning can be based on geographical or political considerations, on ownership of utilities comprising the network or on consideration of the complexity of computing a solution. Regardless of how the split is obtained, diakoptical analysis gives a mathematical structure which is particularly amenable to multiprocessing computer environments using either parallel or distributed computing techniques. Combining diakoptical methods with sparse matrix techniques and triangular decomposition methods provides a powerful tool for simple, rapid solution of large sparse network problems [3, 2, 78].

Two approaches exist for partitioning a given network into a set of subnetworks. The easiest method to visualise is the *branch cutting* method. This partitions the network by cutting some of the branches which connect the nodes in the network. The branches are chosen so that, when cut, they separate the nodes into independent regions, or subnetworks. The second partitioning method splits the network into subnetworks by tearing some of the network nodes apart. This is known as the *node tearing* method and the nodes are

chosen such that tearing them apart decomposes the network into distinct subnetworks. Appendix D provides a more detailed and mathematical treatment of the two methods.

Partitioning of the network using node tearing has an effect on the system's admittance matrix. The process of partitioning transforms the matrix into the Bordered Block Diagonal Form (BBDF) shown below.

$$\begin{bmatrix} Y_{11} & & & & Y_{1c} \\ & Y_{22} & & & Y_{2c} \\ & & \ddots & & \vdots \\ & & & Y_{kk} & Y_{kc} \\ Y_{c1} & Y_{c2} & \cdots & Y_{ck} & Y_{cc} \end{bmatrix} \cdot \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_k \\ V_c \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_k \\ I_c \end{bmatrix} \tag{3.33}$$

The name arises from the fact that the matrix has blocks of non-zero elements along the leading diagonal and along the lower and right edges. Elsewhere in the matrix the elements are all zero. If branch cutting is used the Bordered Block Diagonal Form is not produced directly as extra elements may exist in the matrix between the diagonal blocks and the borders. Appendix D shows that it is possible to restructure the problem such that the admittance matrix produced by branch cutting also exhibits BBDF.

The diakoptic method may be married with the triangular decomposition approach [78] to obtain a solution for network node voltages given the network branch currents. The BBDF admittance matrix $\mathbf{Y}$ is decomposed into a series of triangular factors and forward and backward substitution with the right hand side yields the solution for the unknown node voltage vector $\mathbf{V}$. It is observed that the BBDF is maintained in the factored admittance matrix.

The bordered block diagonal form (3.33) has important consequences for the parallel solution and it defines the algorithmic structure of conventional parallel solutions for (2.7). $Y_{11}, Y_{22}, \ldots, Y_{kk}$ are the admittance submatrices corresponding to the k independent subnetworks created by the removal of the tearing nodes or branches. Similarly V_k and I_k are the node voltages and branch currents for the kth independent subnetwork. As the subnetworks associated with $Y_{11}, \ldots, Y_{kk}$ are independent it is possible factorise the entries of Y_{ii}, Y_{ci} and Y_{ic} in parallel, where $i = 1, \ldots, k$. Y_{cc} must be factorised after this is completed as the presence of Y_{ci} and Y_{ic} results in the triangular decomposition updating the values of Y_{cc}. It is then possible to forward/backward substitute with I_c to yield values for the cutset voltages V_c. Once this has been done, forward/backward substitution can be

performed in parallel for each subnetwork to yield solutions for $V_1, V_2, \ldots, V_i$. Note that the distributive effect of Y_{cc} through forward and backward substitution prevents the commencement of parallel substitution for individual subnetworks until the cutset voltages have been determined.

3.6 The Multiple Factoring Method

The multiple factoring method [46, 29] is aimed at the substitution phase of the solution of linear equations and is based on a standard LU decomposition. An improvement in the substitution phase is achieved at the expense of factorisation, as extra work has to be introduced at the factorisation stage. The efficiency of the method lies in the fact that factorisation only needs to be performed infrequently. Multiple factoring is a prime example of an inefficient sequential algorithm being used as the basis of an efficient parallel algorithm.

Consider the equations $\mathbf{Ax} = \mathbf{b}$ factored into lower, $\mathbf{L}$, and upper, $\mathbf{U}$, triangular matrices such that $\mathbf{LUx} = \mathbf{b}$. The multiple factoring method requires these matrix factors to be factored further. The basic scheme, known as diagonal partitioning, is

$$\begin{bmatrix} \mathbf{I} & \\ \mathbf{K} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11} & \\ & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \\ & \mathbf{U}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{R} \\ & \mathbf{I} \end{bmatrix} x = \mathbf{b} \tag{3.34}$$

where $\mathbf{L}_{ii}$ and $\mathbf{U}_{jj}$ are lower and upper triangular matrices respectively, $\mathbf{I}$ is the identity matrix and $\mathbf{K}$ and $\mathbf{R}$ are rectangular matrices. (3.34) can be easily solved as

$$\begin{bmatrix} \mathbf{I} & \\ \mathbf{K} & \mathbf{I} \end{bmatrix} \mathbf{w}_1 = \mathbf{b} \tag{3.35}$$

$$\begin{bmatrix} \mathbf{L}_{11} & \\ & \mathbf{L}_{22} \end{bmatrix} \mathbf{w}_2 = \mathbf{w}_1 \tag{3.36}$$

$$\begin{bmatrix} \mathbf{U}_{11} & \\ & \mathbf{U}_{22} \end{bmatrix} \mathbf{w}_3 = \mathbf{w}_2 \tag{3.37}$$

$$\begin{bmatrix} \mathbf{I} & \mathbf{R} \\ & \mathbf{I} \end{bmatrix} \mathbf{x} = \mathbf{w}_3 \tag{3.38}$$

Given $\mathbf{b}$ the vectors $\mathbf{w}_1$, $\mathbf{w}_2$, $\mathbf{w}_3$ and $\mathbf{x}$ may be found successively. In examining the structure of (3.35) it is observed that the upper part of $\mathbf{w}_1$ will be identical to the corresponding part

of **b**. The lower part of $\mathbf{w}_1$ may be found by substituting the upper part into successive rows of the lower part. No precedence relationships exist in solving for the lower part of $\mathbf{w}_1$ so the rows may be processed in parallel and in any order. (3.36) is solved to yield values for $\mathbf{w}_2$ by forwarding substituting $\mathbf{L}_{11}$ and $\mathbf{L}_{22}$ with the values obtained for $\mathbf{w}_1$. Substitution with $\mathbf{L}_{11}$ can take place immediately as the upper part of $\mathbf{w}_1$ is known to be identical to the corresponding part of **b**. It is not necessary to wait until $\mathbf{w}_1$ has been completely obtained before processing begins with $\mathbf{L}_{11}$ and the generation of the upper part of $\mathbf{w}_2$ may be computed in parallel with the computation of the lower part of $\mathbf{w}_1$ if desired. The fact that no coupling exists between the upper and lower parts of this matrix also allows the upper and lower parts of $\mathbf{w}_2$ to be computed in parallel. (3.37) is similar to (3.36) but processing cannot commence until $\mathbf{w}_2$ is completely determined. Again no coupling exists between upper and lower parts of the matrix and upper and lower partitions of $\mathbf{w}_3$ can be computed in parallel. Solution of (3.38) is similar to the solution of (3.35) and once again the upper and lower parts of the vector **x** may be computed in parallel.

By multiplying together the first and last two matrices of (3.34)

$$\begin{bmatrix} \mathbf{L}_{11} & \\ \mathbf{K}\mathbf{L}_{11} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{11}\mathbf{R} \\ & \mathbf{U}_{22} \end{bmatrix} = \mathbf{A} \tag{3.39}$$

$\mathbf{L}_{11}$, $\mathbf{L}_{22}$, $\mathbf{U}_{11}$ and $\mathbf{U}_{22}$ can be seen to be simple submatrices of the lower and upper triangular factors of **A**. **K** and **R** may be found from the equalities

$$\mathbf{K}\mathbf{L}_{11} = \mathbf{L}_{21} \tag{3.40}$$

$$\mathbf{U}_{11}\mathbf{R} = \mathbf{U}_{12} \tag{3.41}$$

The generation of all the required factor information is a two step process. Firstly a standard LU decomposition must be used to determine the lower and upper triangular factors **L** and **U**. $\mathbf{L}_{11}$, $\mathbf{L}_{22}$, $\mathbf{U}_{11}$ and $\mathbf{U}_{22}$ may then be determined directly and **K** and **R** can be found from (3.40) and (3.41).

The power of the multiple factoring method becomes apparent when the **L** and **U** factor matrices are subdivided many times. Three possible schemes exist for subdividing the matrix factors of **A**.

Scheme 1 : Forward factoring

Scheme 1 subdivides the factor matrices according to

$$\begin{bmatrix} \mathbf{I} & & \\ \mathbf{K}_{21} & \mathbf{I} & \\ \mathbf{K}_{31} & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & & \\ & \mathbf{I} & \\ & \mathbf{K}_{32} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11} & & \\ & \mathbf{L}_{22} & \\ & & \mathbf{L}_{33} \end{bmatrix} \tag{3.42}$$

$$\begin{bmatrix} \mathbf{I} & & \mathbf{R}_{13} \\ & \mathbf{I} & \mathbf{R}_{23} \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{R}_{12} & \\ & \mathbf{I} & \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & & \\ & \mathbf{U}_{22} & \\ & & \mathbf{U}_{33} \end{bmatrix} \tag{3.43}$$

A consequence of this scheme is that extra fill-ins, over and above those resulting from LU decomposition, are introduced into the matrices by the subdivision process. With the factors divided in this fashion it can be seen that results from each step must be propagated to the next step of the solution process. For example, the computations involving $\mathbf{K}_{21}$ and $\mathbf{K}_{31}$ can commence immediately with these computations taking place concurrently. The computations involving $\mathbf{K}_{32}$ cannot proceed until the computations involving $\mathbf{K}_{31}$ are complete as substitution with $\mathbf{K}_{32}$ depends on values which are altered in substituting with $\mathbf{K}_{31}$.

Scheme 2: Backward factoring

The second strategy for division partitions the factor matrices according to

$$\begin{bmatrix} \mathbf{I} & & \\ & \mathbf{I} & \\ \mathbf{K}_{31} & \mathbf{K}_{32} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & & \\ \mathbf{K}_{21} & \mathbf{I} & \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11} & & \\ & \mathbf{L}_{22} & \\ & & \mathbf{L}_{33} \end{bmatrix} \tag{3.44}$$

$$\begin{bmatrix} \mathbf{I} & \mathbf{R}_{12} & \mathbf{R}_{13} \\ & \mathbf{I} & \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & & \\ & \mathbf{I} & \mathbf{R}_{23} \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & & \\ & \mathbf{U}_{22} & \\ & & \mathbf{U}_{33} \end{bmatrix} \tag{3.45}$$

Under this scheme there are no precedence relationships between the $\mathbf{K}$ ($\mathbf{R}$'s) blocks and this means that there is no propagation of results between the steps of the solution. Hence the potential for parallelism is great. However Van Ness [46] notes that this partitioning strategy introduces much more fill-in than the first strategy. As a result he suggests that the first strategy should be applied to the partitioning of the majority of the matrix, which

is sparse. The lower right hand corner of the matrix is usually fairly densely populated and here fill-in is of little consequence. Van Ness suggests that this region should be treated using the second partitioning scheme. The lowest submatrix in each of the **L** and **U** chains are both fully populated. These submatrices have to be processed consecutively (the sequential part of the method) and Van Ness advocates representing this fully populated section using its full matrix inverse.

Scheme 3 : Row oriented factoring

Both of Van Ness's partitioning schemes are based on the use of diagonal partitioning. Berry et al. [29] have proposed a third partitioning scheme which is row oriented. Under this scheme

$$\mathbf{U} = \begin{bmatrix} \mathbf{U}_{11} & \mathbf{R}_{12} & \mathbf{R}_{13} \\ & \mathbf{I} & \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & & \\ & \mathbf{U}_{22} & \mathbf{R}_{23} \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & & \\ & \mathbf{I} & \\ & & \mathbf{U}_{33} \end{bmatrix} \tag{3.46}$$

This is similar to the column oriented partitioning of **L** factors under Scheme 2. Solution of the equations requires scheme 2 partitioning of the **L** matrix with scheme 3 partitioning used for the **U** matrix. Berry et al. found that although this approach introduces a large amount of fill-in, the solution is achieved faster then when using either scheme 1 or scheme 2 factors alone.

Both Van Ness [46] and Berry [29] cite results in their papers but neither really sheds any light on the speed-up performance of the multiple factoring method. Berry gives absolute execution times for the solution of a 60 bus test system but gives no reference sequential execution time from which speed-up may be calculated. Van Ness provides the theoretical minimum computation time and and 'actual' computation time obtained from a simulation of the method. As with most parallel simulations no attempt is made to account for the effects of interprocessor communication or task switching overheads. The absence of a sequential reference time makes it impossible to calculate the speed-up. When a large number (50+) of processors are used Van Ness finds the 'actual' computation time to be similar to the theoretical minimum computation time, indicating that speed-up with a large number of processors is close to the theoretical predicted maximum speed-up, whatever that may be. When a more modest number of processors (≈ 20) is used the 'actual' computation time is three times greater than the theoretical minimum time, indicating that speed-up

can be no more than $\frac{1}{3}$ of the theoretical maximum. Evidently a large number of processors are needed to give maximum speed-up, making the solution both expensive and inefficient. Reducing the number of processors used to a more economic level has the effect of drastically reducing the performance.

3.7 Parallel LU Decomposition Techniques

A survey of the available literature reveals a number of publications on the parallel solution of power system equations by LU based methods. Some of these publications detail interesting and innovative work but most are nothing more than subtle modifications to existing methods. This section presents a survey of some of the parallel LU decomposition methods currently available. All parallel techniques developed to date have a structure similar to that of sequential LU decomposition algorithms (*i.e.* factorisation of the admittance matrix followed by forward and backward substitution). The factorisation phase sees each subnetwork being factorised in parallel on an individual processor. Once all subnetworks have been factorised any information relating to the cutset block must be passed to the processor responsible for that block. The cutset block is then factorised and forward and backward substitution of this block with the current vector is performed. This generates information relating to all the other subnetworks and hence the result of this forward/backward substitution must be sent to all other processors before substitution can be completed. The parallel algorithm has the structure shown in Figure 3.2. The cutset block must be processed by a central coordinating processor which receives information from and passes information back to all the other processors, controlling their operation in a Master - Slave fashion.

A simple example of a 5×5 matrix with Bordered Block Diagonal Form is shown in Figure 3.3. The matrix consists of two 2×2 blocks are arranged on the diagonal and there is a single element cutset block. Figure 3.3 also shows all the operations that must be performed to factorise this matrix. The independence of operations on each block is clearly illustrated in this example. Factorising rows 1 and 2 causes the cutset element in row 5 to be modified but no modifications to rows 3 or 4 take place. Similarly, factorising rows 3 and 4 results in the the cutset element of row 5 being modified but no modifications to rows 1 or 2 take place. The operations on blocks 1 & 2 and 3 & 4 can be performed concurrently with the cutset block being processed once the processing of the other two blocks is complete

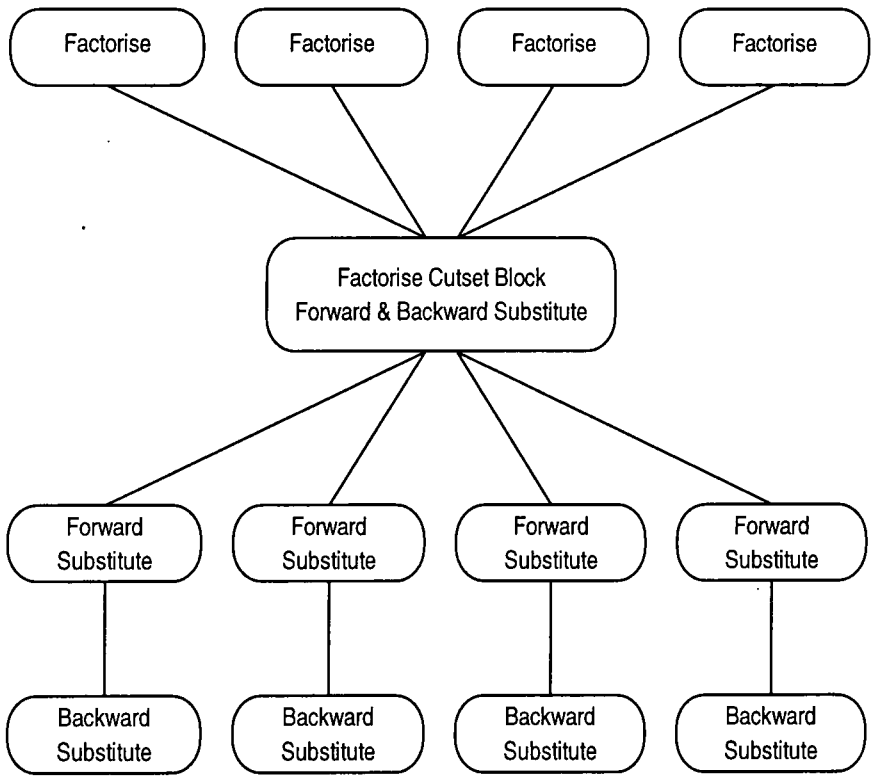


Figure 3.2: Structure of conventional parallel solution algorithm

and all modifications have been made to the cutset block.

Several researchers [79, 57, 80] have noted that when more than about five processors are used the benefits of partitioning the network and solving it in parallel tend to saturate - *i.e.* introducing more processors does not decrease the overall solution time. This saturation is attributed to the significant size of the cutset block, which must be processed sequentially, and the amount of interprocessor communication resulting from the Master - Slave approach of the conventional parallel solution. Despite the significant research effort devoted to this problem speed-ups for both the factorisation and substitution steps of the LU decomposition remain disappointing. The results of Lau's approach [32] to parallelizing the factorisation part of the method reveal a maximum speed-up of two even when as many as 32 processors are used. Techniques for parallelizing the substitution phase have fared little better. Abur's technique [26] requires 57 processors to achieve a speed-up of 3.8 for the substitution stage of the solution of the IEEE 118 node test system and as many as 281 processors are required to achieve a speed-up of 5 from a system containing only 590 nodes. Whilst this method gives a better speed-up than many previous methods it is extremely inefficient due to the large number of processors required.

	1	2	3	4	5
Calculate	$L_{11} = \frac{1}{A_{11}}$	$R_{12} = -\frac{A_{12}}{A_{11}}$ $L_{21} = -\frac{A_{21}}{A_{11}}$			$R_{15} = -\frac{A_{15}}{A_{11}}$ $L_{51} = -\frac{A_{51}}{A_{11}}$
Update		$A_{25} = A_{25} - \frac{A_{21}A_{15}}{A_{11}}$ $A_{22} = A_{22} - \frac{A_{21}A_{12}}{A_{11}}$			$A_{52} = A_{52} - \frac{A_{51}A_{12}}{A_{11}}$ $A_{55} = A_{55} - \frac{A_{51}A_{15}}{A_{11}}$
Calculate		$L_{22} = \frac{1}{A_{22}}$			$R_{25} = -\frac{A_{25}}{A_{22}}$ $L_{52} = -\frac{A_{52}}{A_{22}}$
Update					$A_{55} = A_{55} - \frac{A_{52}A_{25}}{A_{22}}$
Calculate			$L_{33} = \frac{1}{A_{33}}$	$R_{34} = -\frac{A_{34}}{A_{33}}$ $L_{43} = -\frac{A_{43}}{A_{33}}$	$R_{35} = -\frac{A_{35}}{A_{33}}$ $L_{53} = -\frac{A_{53}}{A_{33}}$
Update				$A_{45} = A_{45} - \frac{A_{43}A_{35}}{A_{33}}$ $A_{44} = A_{44} - \frac{A_{43}A_{35}}{A_{33}}$	$A_{54} = A_{54} - \frac{A_{53}A_{34}}{A_{33}}$ $A_{55} = A_{55} - \frac{A_{53}A_{35}}{A_{33}}$
Calculate				$L_{44} = \frac{1}{A_{44}}$	$R_{45} = -\frac{A_{45}}{A_{44}}$ $L_{54} = -\frac{A_{54}}{A_{44}}$
Update					$A_{55} = A_{55} - \frac{A_{54}A_{45}}{A_{44}}$
Calculate					$L_{55} = \frac{1}{A_{55}}$

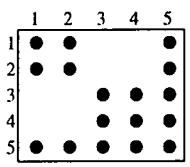


Figure 3.3: Simple BBDF factorisation example, showing independence between operations

The fact that most of the existing LU factorisation techniques have very similar algorithms is not surprising. All of the techniques rely on the use of BBDF matrix structure for exploiting parallelism. The structure of the algorithm is a function of the BBDF structure *not* a function of the particular LU-based method chosen. The bottleneck introduced by the cutset solution is also a function of the BBDF matrix topology. This implies that any improvement over existing parallel methods must begin by improving the structure of the matrix to allow greater exploitation of parallelism. The actual LU or LDU-based factorisation chosen will have only a minor effect on performance. The main differences between existing techniques are the way in which they store data and the architecture of the target parallel machine.

3.7.1 Chan's Method

One recently developed technique is that of Berry, Chan and Dunn [36]. Their technique is designed for MIMD machines but they do not specify in their paper whether shared or distributed memory machines are the target architecture. A personal communication with the authors [81] has subsequently revealed that the target architecture was a distributed shared memory machine. Communication of all data was performed via the global shared memory and the message passing facilities of the architecture were only used for synchronizing the operation of the program tasks. The need for a custom parallel machine renders this approach unsuitable for use with off-the-shelf distributed memory machines but it is still a useful and interesting method. BBDF is the prominent feature of the approach but the authors successfully divorced the BBDF structure from the conventional algorithm structure. Algorithms which use BBDF normally assign the cutset block to a central coordinating processor. All other processors in the machine must communicate their updates to the central processor which then becomes the bottleneck in the solution process. Chan dispenses with the need for the central coordinating processor and the Master-Slave architecture by assigning a copy of the cutset block to each processor in the network. Instead of communicating updates to a central processor each processor broadcasts its updates to every other processor in the network. All the other processors then update their local copy of the cutset matrix data. When all the processors have performed all the updates supplied by all the processors they each solve their own local version of the cutset block in parallel before solving their respective subnetworks. There are two interesting points to this approach

1. The elimination of the need for a central coordinating processor means that this method uses one processor less than a more traditional BBDF approach and should be more efficient

2. The assignment of a copy of the cutset block to each processor removes a communication step from the algorithm. It is no longer necessary to broadcast the results of the cutset solution throughout the network as each processor holds its own local solution. Provided that each processor maintains a coherent copy of the cutset matrix data prior to its solution, all the processors should obtain the same answer from the solution of the cutset block. The extra communication step is eliminated at the expense of implementing a method for ensuring coherency between local copies of data structures held by all processors.

That there are differences between this method and the traditional methods is quite obvious from its algorithm structure, shown in Figure 3.4. The bottleneck of communications to the central processor is no longer present but has been replaced by a much larger collection of intertask communications. The explicit communication following cutset solution has been eliminated. Chan actually found that the duplicated computation of the cutset by all processors is the dominant feature in the total amount of computation. As the inverse of the cutset block is not significantly denser than the cutset block itself, Chan found it more efficient to solve the cutset block by calculating its full inverse using a single processor. This is carried out after the parallel factorization of subnetwork blocks is complete and an extra communication step is required to pass the result of inverting the cutset to the subnetwork processors. The algorithm structure once again becomes the same as that of the conventional structure (Figure 3.2).

Whilst the method is interesting its speed-up performance is little better than that of other approaches. Speed-ups are only marginally improved, if at all, but the benefit of the approach is its increase in efficiency. Similar speed-ups to other methods can be achieved using fewer processors.

3.7.2 The W-matrix Method

The W-matrix method was first proposed by Alvarado [37] and adapted by Padhila and Morelato [44] in 1992. The approach is based on a sequential solution technique known as matrix inverse factors, or the W-matrix method. The authors note that the W-matrix

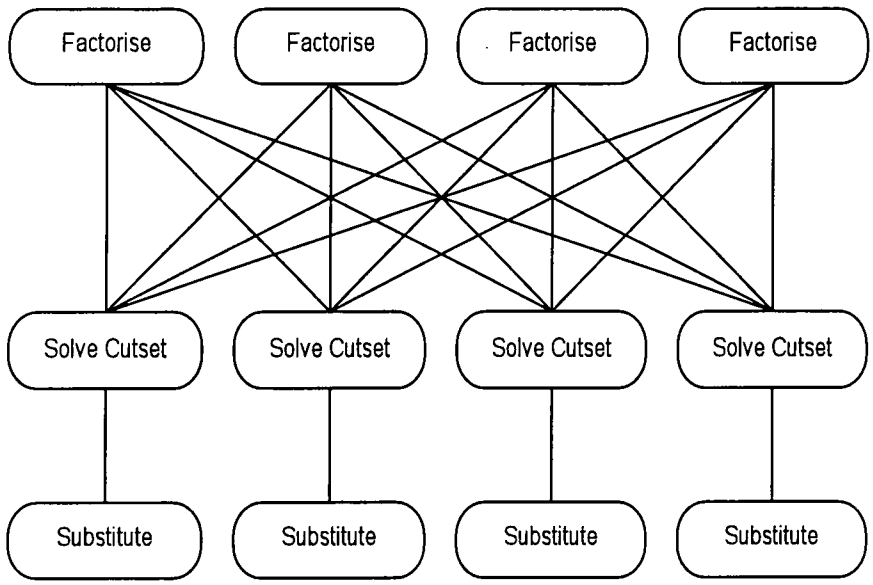


Figure 3.4: Algorithm structure of Chan's method, using duplicate cutset computation

method is not the most efficient solution algorithm for sequential machines but it holds promise for parallel solutions as the techniques gives a degree of independence between elementary operations.

The W-matrix method is based on LDU factorisation and it decomposes the coefficient matrix, **A**, into three matrix factors such that

$$\mathbf{A} = \mathbf{LDU} \tag{3.47}$$

The solution to the set of equations is given by

$$\mathbf{x} = \mathbf{U}^{-1}\mathbf{D}^{-1}\mathbf{L}^{-1}b \tag{3.48}$$

A matrix, **W**, is defined as

$$\mathbf{W} = \mathbf{L}^{-1} \tag{3.49}$$

and this can be expanded into n separate factors, $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_n$, where $W_i = L_i^{-1}$ and L_i^{-1} is a matrix which differs from the unit matrix in the i^{th} column, which is the same as the i^{th} column of **L**. Hence

$$\mathbf{x} = \mathbf{W}^T\mathbf{D}^{-1}\mathbf{W}\mathbf{b} = \mathbf{W}_1^T\mathbf{W}_2^T \ldots \mathbf{W}_n T\mathbf{D}^{-1}\mathbf{W}_1\mathbf{W}_2 \ldots \mathbf{W}_n\mathbf{b} \tag{3.50}$$

A parallel formulation of the method is achieved by partitioning the W-matrix such that

multiplications between columns of **W** and elements of the righthand side vector become independent [44]. The partition is accomplished by horizontal division of the elimination tree associated with the system and if p partitions are produced then these must be processed consecutively. The solution of (3.50) can be seen as an ordered sequence of updating tasks operating on the components of the right hand side vector, **b**. Each update task is formed from multiplication and addition operations and is of the form

$$\mathbf{b}_j = \mathbf{b}_j + \mathbf{w}_{k,j}\mathbf{b}_k \tag{3.51}$$

The W-matrix method allows all multiplications between **W** matrix elements and elements of **b** to be performed concurrently within each partition. As each multiplication requires read-only access to one row of **W** ($\mathbf{W}_k$) and read-only access to the vector $\mathbf{b}_k$, each row-by-vector multiplication is independent and can be performed concurrently without introducing any problems of data coherency. Unfortunately the additions cannot all be performed concurrently as each addition requires the latest value of $\mathbf{b}_j$ and this may be under processing when it is required for addition. Hence some of the additions (not previously known [44]) must be performed after the parallel multiplications are complete.

The approach described by Padhila and Morelato is aimed at improving the parallel substitution operations required to solve for the unknown vector once the matrix has been factorised. The disadvantages of the method are

- An extra stage is required in the solution process to calculate the $\mathbf{W}_i$ matrices. This would introduce a large overhead into a one-off solution but in applications such as power system simulation where repeated solution with multiple right hand sides is required the extra overhead introduced in not all that large.

- Although the method can be used on a message passing machine the best performance is obtained when the right hand side vector is stored in common shared memory. The method is therefore best suited to distributed shared memory or shared memory machines.

The results quoted for the substitution phase speed-up are better than those for many other methods and Padhila's results are better than those obtained by Chan. For similar size systems Chan achieves a speed-up of 3.36 using 8 processors whereas Padhila observes a speed-up of 5.14 with the same number of processors. In a recent paper [33] Lin and

Van Ness show that the W-matrix method is mathematically equivalent to the multiple factoring method and give performance results for a shared memory implementation of the multiple factoring method. These results are very similar to Padhila's results for the W-matrix method, highlighting the equivalence between the two approaches. Unfortunately the results for a distributed memory implementation of the multiple factoring method are not nearly as good. The speed-ups produced are less than half of those produced by the shared memory implementation.

3.8 Cholesky Factorisation Techniques

Cholesky factorisation is a commonly used method for solving sparse symmetric positive definite linear equations. Power system linear equations are not positive definite and Cholesky methods may not be used to solve them but it is worth considering these methods for the sake of completeness. Certain forms of Cholesky factorisation suggest ways in which parallel LU decomposition based solutions may be improved.

Cholesky factorisation was originally developed as a sequential method of solving equations. With the advent of high performance computers much research effort has been applied to the development of parallel Cholesky techniques, particularly within the applied mathematics community. A number of issues have been confronted in the development of these parallel techniques which are also encountered with other parallel factorisation methods.

Given a system of linear equations

$$\mathbf{Ax} = \mathbf{b} \tag{3.52}$$

where $\mathbf{A}$ is the coefficient matrix, $\mathbf{b}$ is the known vector and $\mathbf{x}$ is the unknown vector, a solution for $\mathbf{x}$ may be obtained by computing the Cholesky factorisation

$$\mathbf{A} = \mathbf{LL}^T \tag{3.53}$$

$\mathbf{L}$ is the Cholesky factor and it is a lower triangular matrix which has positive entries on its leading diagonal. The unknown vector $\mathbf{x}$ can be be easily computed as

$$\mathbf{Ly} = \mathbf{b} \qquad \mathbf{L}^T\mathbf{x} = \mathbf{y} \tag{3.54}$$

Two substitutions are required to calculate first **y** and then **x**. As in LU decomposition no pivoting is required to maintain numerical stability and ordering techniques may be applied to enhance the preservation of sparsity.

Cholesky factorisation is a variant of Gaussian elimination, as is LU decomposition, and is therefore based around the equation

$$a_{ij} = a_{ij} - \frac{(a_{ik}a_{kj})}{a_{kk}} = a_{ij} - \frac{a_{ik}}{\sqrt{a_{kk}}}.\frac{a_{kj}}{\sqrt{a_{kk}}} \tag{3.55}$$

The difference between LU decomposition and Cholesky factorisation is that element values in Cholesky factorisation are normalized by the square root of the pivot. The matrix **L** must be real but if **A** is not symmetric positive definite **L** will not be real. Hence the coefficient matrix for Cholesky factorisation must be symmetric positive definite. Jennings [62] notes that it is possible to apply Cholesky factorisation to a matrix which is symmetric but *not* positive definite but the procedure is made significantly more complicated by the introduction of imaginary elements in **L**.

Cholesky factorisation is similar to both LU and LDU factorisation. However Cholesky factorisation has the disadvantage of requiring the calculation of n square roots. As square root calculations are usually the slowest arithmetic operations on a computer Cholesky factorisation may be slower than LU factorisation.

Note that three subscript indices i, j, k exist in equation (3.55) and in a computer implementation three nested loops are used to increment these indices to scan through the coefficient matrix. Three different Cholesky factorisation algorithms will result depending upon which of these indices is controlled by the outer loop.

1. Row Cholesky algorithm - i is controlled by the outer loop and successive rows of the matrix **L** are computed on each iteration. The inner loops solve a triangular system in terms of previously computed rows.

2. Column Cholesky algorithm - j is controlled by the outer loop and successive columns of **L** are computed. The inner loops perform matrix-vector multiplication of previously computed columns on the current column.

3. Submatrix Cholesky algorithm - k is controlled by the outer loop and successive rows of the factor matrix are computed on each iteration. The inner loops update the submatrix to the right of the current column.

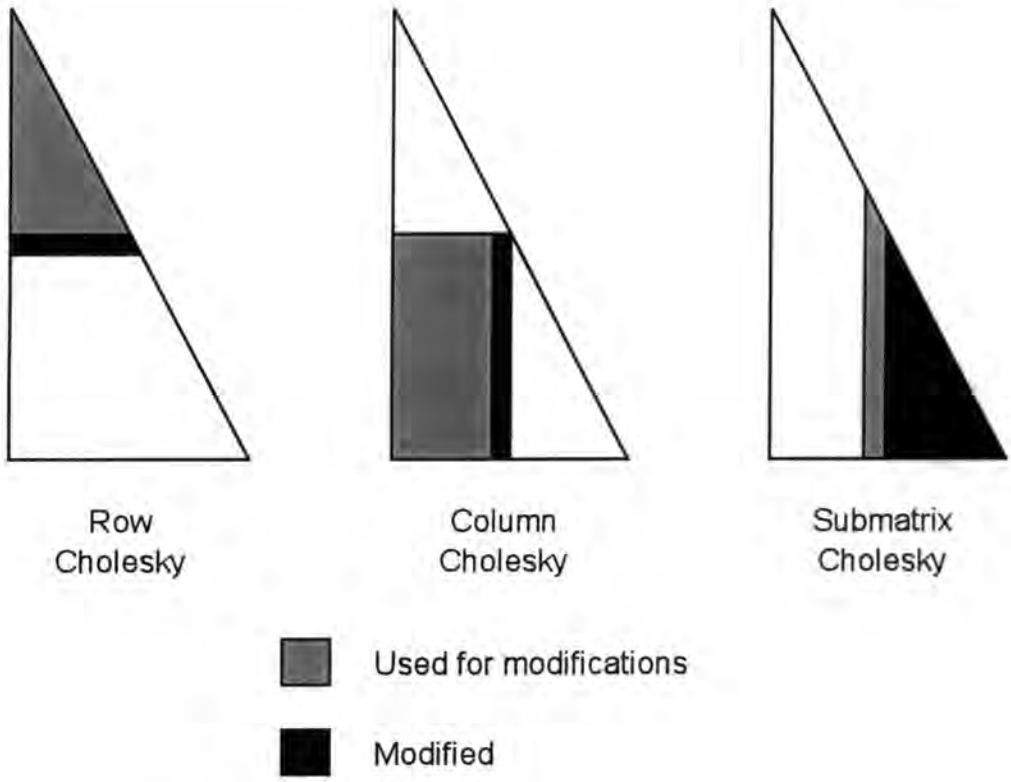


Figure 3.5: The three flavours of Cholesky factorisation

The row and column Cholesky algorithms are leftward looking algorithms as all the information required to modify the current row/column can be found to its left. Submatrix Cholesky factorisation is a rightward looking algorithm as all the information in the current column is used to modify the submatrix to the right of it. LU decomposition is also a rightward looking method and similar in its operation to submatrix Cholesky. The three flavours of sequential Cholesky algorithm, illustrated in Figure 3.5, develop directly into two parallel Cholesky algorithms. Row/column Cholesky factorisation gives rise to the parallel method known as the 'fan-in' algorithm whilst submatrix Cholesky gives rise to the 'fan-out' algorithm.

3.8.1 The Parallel Fan-In Algorithm

The parallel fan-in algorithm, originally proposed by Ashcraft, Eisenstat and Liu [82], is based on the sequential column Cholesky method. Columns of the coefficient matrix are assigned to processors and each processor performs a local processing and data reduction phase before participating in a global data reduction phase. The algorithm is demand driven and it requires aggregated update columns to be passed between processors. If the results

of processing on a given processor relate to columns on other processors then that processor sends the aggregated update columns to the appropriate recipient, which then incorporates them into its calculations. The fan-in algorithm has a very regular communication structure and a low volume of communication traffic, making it more efficient than the fan-out Cholesky algorithm described in the following section.

3.8.2 The Parallel Fan-Out Algorithm

The fan-out algorithm, which is based on the sequential submatrix Cholesky algorithm, assigns columns to processors. Unlike the fan-in algorithm, the fan-out algorithm is data driven and the data which pass between processors are matrix columns. Each processor continually checks for incoming columns, which may be used to update the local matrix data, and when the processor has completed the processing of its own columns it sends the results to all other processors which may require them.

The fan-out algorithm is not very efficient and this is a function of its poor communication performance. The algorithm gives rise to a large number of long messages and although aggregation techniques have been employed to improve the performance it is still less efficient than the fan-in algorithm or multifrontal methods. Complicated message structures are required and these must be assembled before transmission and unpacked upon receipt. This, and the searching that is required to allow updating of the matrix, results in low serial efficiency, E_s, where

$$E_s = \frac{t_s}{t_{up}} \tag{3.56}$$

where t_s is the execution time of the best sequential solution and t_{up} is the execution time of the parallel algorithm executed on a single processor.

3.8.3 Frontal Methods

Frontal methods provide another approach to the parallel solution of linear equations. These method are sophisticated variations of submatrix Cholesky techniques and are significantly more complicated than any of the basic Cholesky algorithms. They were originally developed [67] to make more efficient use of auxiliary storage (*i.e.* disk storage) in the days when memory was expensive and limited in size. The essence of the method is that a small 'window' or front of active computation moves through the coefficient matrix and a full matrix representation is used to store the frontal submatrix in memory. The use of full

matrix representation makes solution more efficient on scalar machines and allows vector architectures to be used to speed up the computations on the frontal submatrix. Parallel multifrontal methods have been developed but they appear to be no more efficient than the fan-in algorithm. The success of frontal methods depends on keeping the active window as small as possible so as to keep to a minimum the amount of dense storage required. The main disadvantage of these methods is the complexity of the algorithms. Continual conversion between sparse and full matrix representations is required and sophisticated data management methods are needed. However Chapter 6 will show that the adoption of full matrix representation for parts of an LU solution can lead to an increase in the efficiency of parallel LU factorisation.

3.9 Summary

This chapter has examined existing parallel methods for solving linear equations. Two classes of solution have been identified. Jacobi iteration, Gauss-Seidel iteration and the Conjugate Gradient method have been introduced as examples of iterative solutions. Methods in this class of solution make an initial guess at a solution and refine it through a procedure of iterative correction until an acceptably accurate result is achieved. Iterative techniques suffer from two main drawbacks. Firstly it is not possible to achieve multiple solutions of the same system of equations with different right hand side vectors quite as readily as it is using direct methods. Secondly, iterative techniques may require many iterations to reach convergence if the equations are ill-conditioned or if a high precision answer is required. Direct methods, of which LU and Cholesky factorisation are examples, provide an exact solution for the equations but may require more total computation than iterative methods. Direct methods do not suffer from convergence problems and have the advantage that separate factorisation and substitution operations are available. Factorisation of the matrix only needs to be performed once and then the substitution operation may be used to solve for multiple right hand sides. In applications such as power system simulation, where most of the processing involves solving the same set of equations with different right hand sides, direct solutions are significantly more efficient than iterative solutions. Iterative solutions are used in power systems computations but direct methods hold the greatest promise for high performance solutions.

Existing direct solution methods have been examined to determine the relative advan-

tages and disadvantages of the various approaches. Cholesky techniques are not suitable for power system applications as they can only be applied to symmetric positive definite systems and power system equations do not fall into this category. This leaves parallel LU and LDU-based methods as the only really suitable solution methods for power system applications. Numerous different techniques have been attempted but the speed-up resulting from them is unimpressive and quickly saturates as the number of processors increases. The inefficiencies are attributed to a sequential bottleneck present in all these methods. The algorithms for each of the methods are similar and arise as a direct result of the use of the BBDF structure as a device for exploiting parallelism. It is the BBDF structure and not the particular flavour of LU-based solution which gives rise to the sequential bottleneck and performance limitations.

Improving upon existing LU methods requires the BBDF structure to be improved. The sequential bottleneck must be eliminated and more parallelism exploited. The review of the various existing methods presented in this chapter suggests ways in which improvements can be made to this structure. Chapter 5 takes these suggestions and uses them to develop an efficient new matrix structure suitable for parallel LU-based solutions. Before examining this structure it is necessary to consider the elimination tree again and what implications it has for network partitioning and balancing the computational load.

Chapter 4

Elimination Trees, Network Partitioning and Load Balancing

4.1 Introduction

In a sequential computer the processor executes a program by operating on the required data using the sequence of operations prescribed by the program until all operations are complete. Throughout the lifetime of the program the processor is always busy and spends most of its time performing useful[1] work. In a parallel computer each task executes on its own in isolation from the other tasks. When cooperation with other tasks is required some time may be spent waiting for the other tasks to reach their synchronisation points and valuable processing time is wasted in these idle wait states. In order to achieve maximum efficiency it is desirable to have all the processors performing useful work all of the time. Maximum speed-up can only be achieved if all of the processors are constantly kept busy. Ensuring that processors are always busy requires the partitioning of the total workload into equally sized tasks which are then assigned to each available processor. The process of spreading the workload over the processors is known as *load balancing* and a *balanced* load occurs when equal portions of the workload are assigned to each processor.

Consider the computer solution of the linear equations associated with a large network problem. The workload in this case is the solution of the linear equations and the division of the workload requires the solution of the equations to be divided equally across the

[1] 'Useful' in this context means work that is of value to the user and is generally taken to mean the execution of the user's program. Non-useful work would include such things a servicing operating system interrupts, hardware interrupts *etc.*.

processors. Splitting the equations up is equivalent to tearing the network into subnetworks using Kron's method of diakoptics, thereby reducing the load balancing problem to one of network partitioning. A balanced loading can be seen as the partitioning of a network into a number of subnetworks which require equal amounts of processing. This partitioning is seldom easy and ideal load balancing is rarely achieved in all but the most trivial of cases.

Within the general literature of parallel computing much emphasis is placed on the importance of achieving an ideal balanced load. Strategies are presented for achieving this balance but most authors [19, 25] observe that if the workload is known *a priori* then near-optimal load balance can be achieved through a static distribution of workload at compile time. A knowledge of the true workload requires a detailed knowledge of the program, the operating system and the hardware on which it is executed. Many programmers resort to using approximate load balancing techniques. One technique often used when solving network equations is to partition the network into subnetworks such that each subnetwork contains an equal number of nodes. This method does not guarantee that each subnetwork requires an equal amount of processing and this chapter describes the development of a better technique which is based upon an analysis of the computational complexity of the solution and the use of the elimination tree.

4.2 Balancing the Computational Load

In executing any program on a multiprocessor computer the aim is to achieve maximum speed-up using the available processors. It was stated previously that this is only possible if all of the available processors are constantly busy. This is achieved through dividing the processing load into equal portions which are assigned to individual processors (balanced loading). Careful consideration must be given to the decomposition of the algorithm into logical tasks and the partitioning of the data into distinct subsets. These two issues are closely interrelated. Of particular importance to the efficiency of a parallel program is task synchronisation and the use of interprocessor communication. A well partitioned problem will require only a minimal amount of interprocessor communication and good speed-ups can be expected. A poorly partitioned problem is characterized by large amounts of interprocessor communication and many process synchronisation requests. Interprocessor communication incurs overheads and if the amount of interprocessor communication is large then the penalty incurred can seriously limit the speed-up obtained. Poorly partitioned

TASK A	TASK B
begin	begin
calculate $F(D_a)$	calculate $F(D_b)$
send results to task B	get results from task A
get results from task B	send results to task A
end	end

Figure 4.1: Simple load balancing example

programs can be less efficient than sequential programs designed to solve the same problem simply because of the penalty incurred by interprocessor communication. Few problems are easy to partition well and many problems fall into the 'hard to partition' category. Badly partitioned programs are more often a function of the problem and its inherent lack of parallelism than of the designer.

As an example of the need to obtain a balanced load consider the simple program whose algorithm is given in Figure 4.1. This program consists of two tasks with similar structures which are executed in parallel on independent processors. Each tasks applies the function F() to its dataset before passing the results to the other task. Function F() has the property that its execution time is proportional to the size of the dataset on which it operates. Suppose also that the dataset for the problem, D, is partitioned into two subsets ,D_a and D_b, such that

$$D = D_a \cup D_b \tag{4.1}$$

Data subset D_a is used by task A whilst D_b is used by task B.

Let us assume that D is not partitioned equally and that D_a holds one tenth of the information from D with the remaining nine tenths being held in D_b. As the execution time of $F()$ is proportional to the size of the data it operates on, $F(D_b)$ will take nine times longer to execute than $F(D_a)$. Task A will complete the application of $F()$ to D_a and try to send the results to task B long before B has finished the operation $F(D_b)$. Task A will then be forced to wait until task B finishes $F(D_b)$ before it can send its information. If we assume that the time taken to perform $F(D_a)$ to be the basic unit of time, then task A will waste 8 units of time waiting for task B to finish the operation $F(D_b)$. This is shown graphically in Figure 4.2(a). Suppose now we alter the partitioning of D such that D_A and D_B both contain half of the information from D. $F(D_A)$ and $F(D_B)$ will now execute in identical times of 5 time steps. Task A no longer has to wait before sending its information and the processor spends all of its time performing useful computations. The

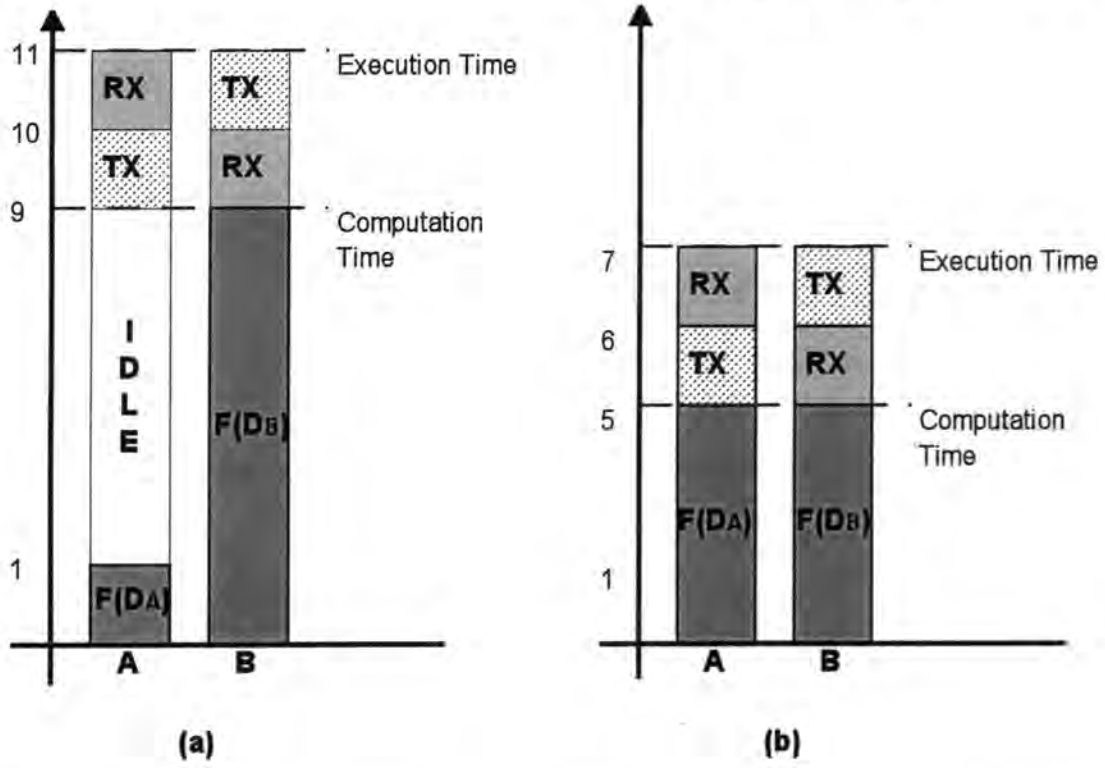


Figure 4.2: Graphical depiction of the execution of the example program with a) imbalanced load b) balanced load

elimination of the idle time reduces the overall execution time for the program to 7 time steps, as opposed to 11 time steps for the previous case. This is the optimum execution time and it arises from the equalization of the computational load between the processors. Any imbalance in the division of the computational load between the processors results in one of the processors taking longer to execute. The other process is forced into an idle wait state and this increases the total execution time of the program. The situation for the balanced load is shown graphically in Figure 4.2(b).

The effect of imbalanced loading manifests itself in the execution time of a program, and consequently in the speed-up. The speed-up, $S(n)$, is the ratio of the sequential execution time to parallel execution time on n processors. For the example above consider only the computations, that is the operation $F()$, and ignore the interprocessor communication. Amdahl's Law allows us to predict the maximum speed-up we can hope to obtain provided we know the sizes of W_p, the amount of work which can be performed in parallel, and W_s, the amount of work which cannot be parallelised and has to be processed sequentially. The simple program of Figure 4.1 has all the work performed in parallel. Hence

$$W_p = 1 \qquad W_s = 0$$

Load Condition	Computation Time		Speed-up
	Parallel	Sequential	
Balanced	5	10	$S = \frac{10}{5} = 2$
Imbalanced	9	10	$S = \frac{10}{9} = 1.11$

Table 4.1: Speed-ups for the load balancing example

Amdahl's Law predicts that, for $W_s = 0$, $S(n) = n$ indicating that linear speed-up is achievable. As the example program uses two processors the maximum speed-up that can be achieved is two. Assuming that the function $F()$ obeys the principle of linear superposition then the time taken to execute $F(D)$ (*i.e.* the sequential solution) will simply be the sum of the times taken to execute $F(D_A)$ and $F(D_B)$, which is 10 time steps. This allows the calculation of the speed-ups for the two loading examples, which are shown in Table 4.1. As Lewis predicts [19], maximum speed-up is achieved under conditions of balanced loading whilst the imbalanced load case performs little better than the sequential program. The importance of this result is that in trying to extract the maximum speed-up from any program it is essential to achieve a load balance which is as near as possible to the ideal balanced load. If the load balance is ignored then the resulting speed-ups will not be optimal. Table 4.2 shows similar speed-up figures obtained from an LU-based solution program implemented as three tasks running on three processors. The data for this test, the CEGB 734 node system, is similarly divided into three subnetworks. The program has the structure shown in Figure 4.3. Tasks A and B operate in parallel and pass their results to task C which then operates sequentially. One subnetwork is processed by each task and the workload for each task is changed by altering the partitioning of the network into subnetworks. The amount of the problem which has to be processed sequentially, W_s, is the workload of task C. Table 4.2 shows the effect on speed-up of changing the workloads assigned to tasks A and B - the more unequal the load the further the resulting speed-up is from the ideal speed-up predicted by Amdahl's Law. Note that it is usually not possible to achieve an exact load balance and the first row of Table 4.2 gives figures for the most equal loading possible with this data.

4.2.1 The Two Approaches to Load Balancing

The previous section has shown the need for, and the benefits of, computational load balancing but has not discussed the method of achieving a balanced load. Section 4.1 has

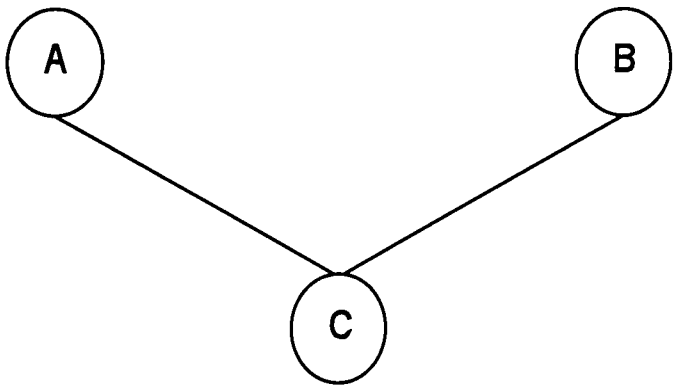


Figure 4.3: Three task implementation of the LU-based solution

Percentage of work performed by				Amdahl's Law
Task A	Task B	Task C	Speed-up	Predicted Speed-up
48.5	50.9	0.4	2.23	2.98
42	56	2.7	1.98	2.85
25	70	4	1.55	2.77
22	75	3	1.48	2.83

Table 4.2: Effect of load balancing on speed-up for an LU-based solution

indicated that if the structure of the algorithm and the size and nature of the workload are known *a priori* then it is possible to achieve *static* load balancing by assigning parts of the problem to tasks at compile time. Static load balancing techniques can be categorized into four distinct approaches with graph theoretical approaches being the most common. The technique relies on the use of two graphs - a graph representing the target machine (*i.e.* the physical processor interconnections) and a task graph which shows the relationship between tasks and the communication requirements of each task. The problem of load balancing reduces to one of mapping the task graph to the hardware graph in a manner which minimizes interprocessor communication and execution time. This is a graph theory problem and much work has been undertaken in this area [83, 19, 84]. *Dynamic* load balancing, where the distribution of the computational workload changes during the course of the program's execution, is also possible but harder to achieve. The major difference between the two approaches is that static load balancing can be performed by the programmer but dynamic load balancing is controlled by the operating system or applications software and is beyond the control of the programmer/user.

Distributing the workload throughout the processors in the system can be performed either in a *domain decomposition* or *control decomposition* based manner [19]. Control de-

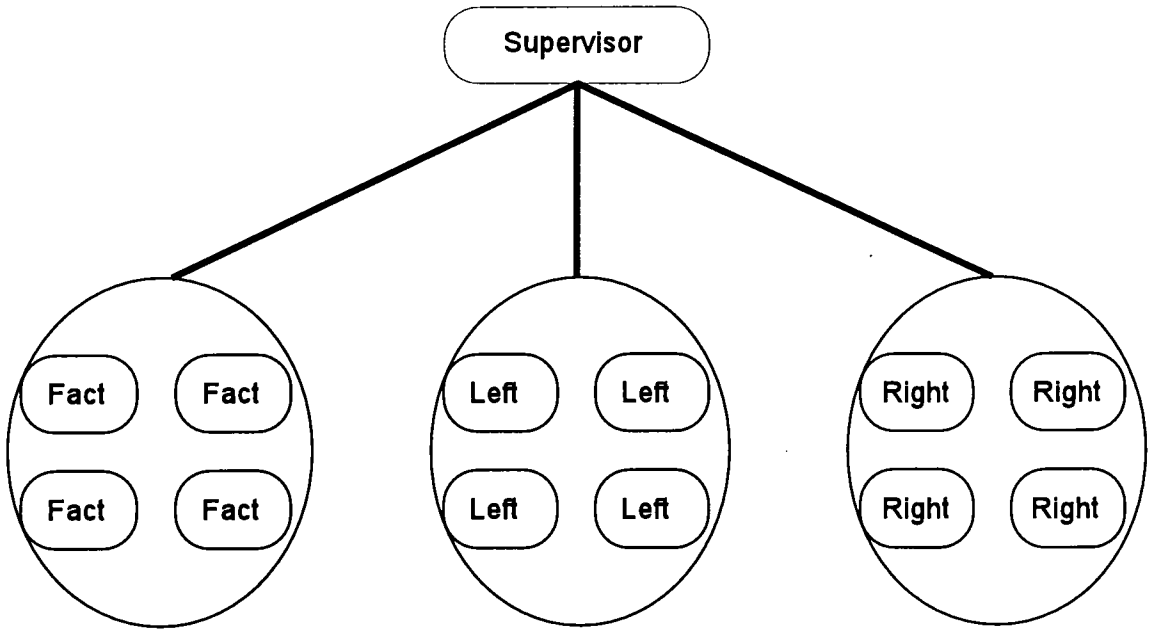


Figure 4.4: A supervisor/worker approach to parallel bifactorisation

composition is concerned with the structure of the algorithm and the way it is split into logical tasks which can then be assigned to individual processors. One typical approach to control decomposition is the supervisor/worker approach which is based on the client/server and processor pool models of distributed computing [85]. There are a number of worker tasks which all perform the same function. The supervisor is responsible for allocating work to each worker task. When a worker finishes its work it sends the results back to the supervisor and if there is still work to be performed the supervisor allocates another job to the worker. In this manner all the workers are kept busy performing the work scheduled by the supervisor until all the work has been completed. This approach often has groups of tasks which perform certain functions, each group consisting of multiple instances of the required task. This is shown in Figure 4.4 which depicts a possible supervisor/worker approach to a parallel implementation of bifactorisation. Three groups of tasks exist; one group consists of tasks to perform factorisation, the second has tasks to perform left multiplication and the third consists of tasks which perform right multiplication. Note that the supervisor/worker approach is a dynamic load balancing approach implemented in the applications software. Domain decomposition on the other hand is concerned with the data to be operated upon by the program. The domain of the input data is partitioned and the different subsets are assigned to the available processors.

Both control and domain decomposition are used in implementing an LU-based solution.

There is domain decomposition in that the input data (*i.e.* the network) is partitioned into subnetworks which are then assigned to individual processors. The program consists of multiple instances of the same task, one for each subnetwork. This single task performs all the necessary processing for a subnetwork and each task is assigned to an individual processor, thus embodying the principle of control decomposition. The implementation is also loosely based on the supervisor/worker model in that there needs to be a supervisor task which partitions the data and forwards it to the subnetwork tasks before solution commences. Each task is assigned only one unit of work throughout the program's lifetime. Once the data has been distributed by the supervisor it becomes redundant and the worker tasks operate autonomously.

The supervisor/worker model is often used as a dynamic load balancing strategy to allow a supervisor, usually the operating system, to distribute work across the system in a dynamic fashion according to the resources (*i.e.* worker tasks) available at the time. The parallel LU-solution is only loosely based on the supervisor/worker model in that the partitioning of data is statically determined *a priori* by the domain decomposition. A dynamic supervisor/worker scenario has no prior knowledge about the workload and must adapt the distribution of work as the program executes. A different approach to dynamic load balancing is based on the use of task migration. Tasks are initially assigned to certain processors but as the program executes the operating systems can reassign tasks and their data to different processors in order to equalize the computational loading. Suppose two tasks are assigned to each processor in the array and that on a certain processor, A, these tasks complete before the corresponding tasks complete on processor B. A is now idle whilst B is busy. A better computational loading is achieved if one of B's tasks is moved to processor A and allowed to complete its computation there. Whilst the concept of task migration is well understood for both parallel and distributed systems, it remains a largely theoretical concept as few practical implementations exist in commercially available operating systems [85].

Most off-the-shelf Transputer systems are not supplied with an operating system which has global control of the Transputer array. Each processor has built in hardware which is responsible for providing local process scheduling and facilities for interprocess communication. This basic microcoded kernel is sufficient to allow most applications to be implemented with careful programming. It is possible to buy operating systems which give global control of the network but these are expensive and for the amount of usage it would receive, it was

decided that the expense was not warranted for this research project. Consequently the programs developed during the project had to rely on facilities provided by the Transputer hardware. As the Transputer provides no support for dynamic load balancing a static load balancing approach with domain decomposition was used throughout. It is the author's view that the use of dynamic load balancing techniques for this problem would merely provide a fine tuning mechanism for wringing the last little bit of speed-up out of the system but the benefits of implementing a dynamic load balancing mechanism are probably outweighed by the complexity of its implementation.

4.2.2 Load Balancing Methodologies Adopted by Other Parallel Solutions

Chapter 3 surveyed some of the existing parallel algorithms for solving sparse sets of linear equations. The same load balancing problem has been encountered in the development of these methods and various techniques for achieving approximately balanced loads have been developed. Many of the researchers avoid the subject of load balancing in their publications and it is not clear whether they have addressed the issue or simply ignored it. Padhila and Morelato [44] go one step further and state that

> 'load imbalancing is not an issue ... The solution with the best load balance is not necessarily the fastest.'

This is certainly a different view from that held by most of the parallel computing community which believes load balancing to be an issue of critical importance, as the previous sections have shown. Where load balancing has been considered it is usually performed in a simple and inefficient way. Many authors [86, 27] simply assign each column of the matrix to an individual processor but this is expensive in terms of the computing hardware required and it ignores the fact that columns with different numbers of off-diagonal elements require different amounts of computation. Columns with many off-diagonals give rise to a large amount of computation whereas columns with few off-diagonals require relatively little computation and their processors spend most of their time idling. Whilst this scheme divides the load across the processors it takes no account of the characteristics of the load and results in an ill balanced computational loading. Variations on this scheme exist [86] which assign multiple columns to processors but these also ignore the characteristics of the load and fare little better in achieving a balanced load. This is the load balancing strategy adopted by Padhila and Morelato [44] in their parallel solution, despite their claim that

load balancing is unimportant.

Other researchers [57, 32] assign multiple columns to individual processors in a manner that tries to equalize the number of non-zero elements processed by each processor. It is not clear whether it is the number of non-zeroes before or after fill-in that is equalized. As this scheduling is based on the use of the elimination tree, and the coefficient matrix must be at least symbolically factorised before the tree can be derived, it is reasonable to assume that it is the number of non-zeroes after fill-in which is equalized. If the number of non-zeroes prior to elimination was equalized then the schedule would fail to take account of fill-ins that occur as a result of elimination. As factorisation proceeded the load across the network would become significantly imbalanced. This would be especially true for rows at the lower right of the matrix which become much more densely populated than rows at the top left of the matrix.

Liu [52] proposes a different approach based on the use of elimination trees to determine the scheduling strategy which gives the best load balance. Geist and Ng [73] also use this approach to balance the load in the parallel Cholesky factorisation method. This method gives a reasonably even load balancing and has been adapted for use with the LU-based solution.

4.3 The Elimination Tree and Parallel Processing

The factorisation path for each node in the elimination tree determines the precedence relationship in the factorisation and substitution phases of the triangular solution of the set of equations represented by matrix **A**. Nodes which do not belong to the same factorisation path are independent and may be processed simultaneously by assigning them to different processors in a multiprocessor array. This is true for both the factorisation and substitution operations of the solution. For example, nodes 5 and 6 in the 10 node example of Figure 2.6 belong to different factorisation paths and may be processed in parallel.

The power of the elimination tree lies in the fact that it is a useful tool for visualising the factorisation/substitution processes and the parallelism inherent in them. Recall that factorising a column of the coefficient matrix is equivalent to eliminating a node from the graph $G(A)$ and hence from the tree $T[A]$. It is easy to see that removing all the leaf nodes from the tree exposes a new layer of leaf nodes. Repeating the process again exposes another layer of leaf nodes, and so on. As each leaf node belongs to a different factorisation path

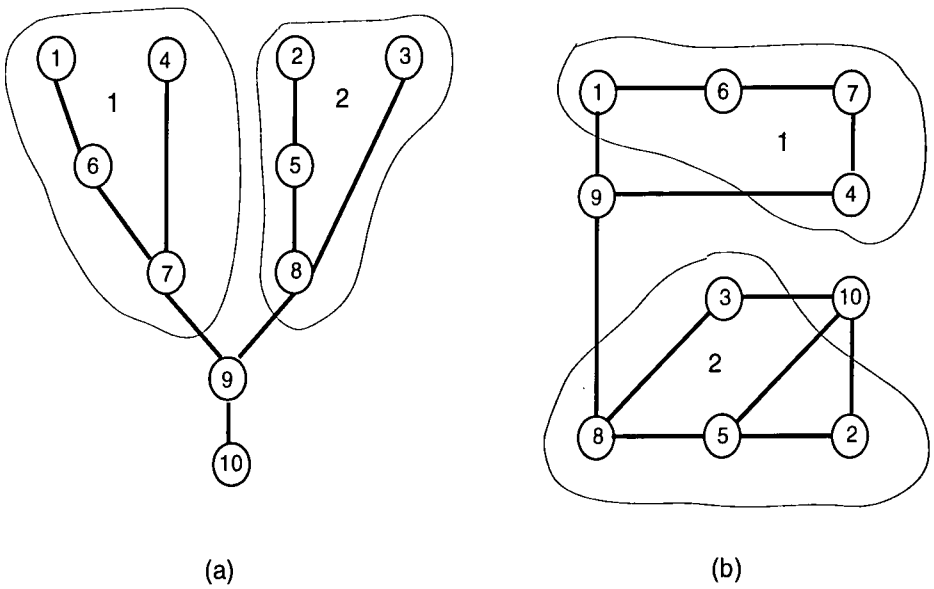


Figure 4.5: Partitioning of the elimination tree (a) and corresponding network partitions (b)

they can be simultaneously eliminated. This is true for each exposed layer of leaf nodes and exemplifies one typical approach to parallelising the triangular solution [73].

A different approach to the exploitation of parallelism is based on an analysis of the elimination tree. The essence of the method is to group nodes within the tree into subtrees and this is equivalent to grouping the nodes of the network into subnetworks. Many researchers who have used the elimination tree in their approach to parallel triangular decomposition have not explicitly recognized this fact [87]. As the subtrees contain independent factorisation paths they may be processed in parallel by independent processors. This method gives a coarse grain parallel solution whereas the wrap mapping method described above produces a fine grain solution. Figure 4.5(a) shows how the 10 node example tree of Figure 2.6 could be partitioned into subtrees. Subtrees 1 and 2 are independent and may be processed in parallel. The remainder of the tree corresponds to the cutset in the partitioned network for this system and it must be processed after the subtrees have been processed. Figure 4.5(b) shows how the elimination tree partitioning corresponds to the partitioning of the network into subnetworks

Within any elimination tree it is always possible to find a longest path through the tree. This path is referred to as the *critical path* of the tree and the length of this path has important consequences for parallel solutions. The longer the critical path the slower the elimination process will be. As a consequence, systems which have very short critical

paths can be processed much faster than systems with long critical paths. Consider the fine grain approach which exposes successive layers of leaf nodes. If the critical path has p nodes on it then p parallel steps will be required to complete the elimination. This can be verified by assuming that the processing of each node on the critical path is performed in unit time. Hence a system with p nodes on the critical path will execute in p units of time and a system with 10 nodes on the critical path will take twice as long to execute as a system with 5 nodes on the critical path. A similar argument can be made for the coarse grain approach and the result is the same in both cases.

Unfortunately this assumption is not valid for real triangular solution programs. The processing of each node is achieved in a time which is proportional to the number of nonzero elements in the row of the matrix corresponding to that node. Consequently a short path consisting of nodes with a large computational overhead may take longer to execute than a long path consisting of nodes with a small computational overhead. The critical path is not necessarily the longest path, but the path with the largest aggregate computational overhead.

4.3.1 The Elimination Tree and Network Partitioning

Section 2.7 states that it is possible to swap rows and columns of a matrix before performing Gaussian elimination. This is equivalent to reordering the nodes in the network, and hence in the graph $G(\mathbf{A})$. In his paper on the role of elimination trees Liu [52] introduces the concepts of topological and equivalent reorderings. A *topological* reordering of the elimination tree reorders the nodes such that all child nodes are numbered before their parents. The consequence of such an ordering is that the last row/column of the matrix always corresponds to the root of the elimination tree. An *equivalent* reordering is defined as follows; if there is a symmetric matrix $\mathbf{A}$ and two orderings $\mathbf{P}$ and $\mathbf{Q}$, then $\mathbf{P}$ and $\mathbf{Q}$ are said to be equivalent if the filled graphs of $\mathbf{PAP}^T$ and $\mathbf{QAQ}^T$ are the same (*i.e.* isomorphic). Furthermore $\mathbf{P}$ (or $\mathbf{Q}$) is said to be an equivalent reordering if the filled graph of $\mathbf{PAP}^T$ has the same structure as the filled graph of $\mathbf{A}$. The benefit of equivalent reorderings is that the reordered matrix incurs the same computational and storage costs as the original matrix and in terms of performance, the equivalent reordering is every bit as good as the original ordering. This implies that if the matrix $\mathbf{A}$ has been ordered with some form of near optimal ordering algorithm applying an equivalent reordering will not destroy the optimal nature of the solution.

Liu proves a number of theorems which are needed to explain how elimination tree based partitioning works. The proofs of these theorems are given in Appendix D.

Theorem 1 *For each node x_j in $G(\mathbf{A})$, the subgraph of $G(\mathbf{A})$ (or $G(\mathbf{F})$) which consists of nodes in the tree $T[x_j]$ is connected, where $T[x_j]$ is the subtree rooted at node x_j.*

This theorem implies that partitioning the tree into disjoint subtrees is the same as partitioning the network into subnetworks. Considering Figure 4.5(a), partitioning the tree into disjoint subtrees $T[7]$ and $T[8]$, rooted at nodes 7 and 8, is the same as clustering the corresponding nodes in the network into two subnetworks, as shown in Figure 4.5(b).

Theorem 2 *Given the matrix, $\mathbf{A}$, and an equivalent reordering, $\mathbf{P}$, the filled graphs of $G(\mathbf{A})$ and $G(\mathbf{PAP}^T)$ are isomorphic if they are treated as unlabeled structures.*

Theorem 2 implies that every topological reordering of $\mathbf{A}$ is an equivalent reordering of the matrix $\mathbf{A}$. The corollary to this theorem is that the tree $T[\mathbf{PAP}^T]$ is isomorphic to $T[\mathbf{A}]$ if they are treated as unlabeled structures.

Combining the results of these two theorems allows us to perform a partitioning of the network into subnetworks based upon an inspection of the elimination tree. If the matrix $\mathbf{A}$ has been ordered using a near optimal ordering strategy, subtrees can be identified within $T[\mathbf{A}]$ to partition the network into the required number of subnetworks, as in Figure 4.5. At this stage the choice of subtrees is arbitrary but Section 4.3.2 introduces criteria for selecting the roots of the subtrees so that the resulting network partition gives an approximately balanced load across the processors. The results of theorem 2 allow the optimally ordered coefficient matrix to be reordered so as to give the coefficient matrix a particular desired structure. As this reordering is an equivalent and topological reordering no extra fill-ins will be introduced when processing the matrix and the same number of arithmetic operations are required to process both this and the original matrix. The importance of this result cannot be overstated as it is the foundation of the proposed parallel method. Given a system of equations a near optimal ordering may be applied to minimize the fill-in resulting from the factorisation of that system. Theorem 2 allows this minimum fill-in reordering to be ordered again using an equivalent reordering to give some desired matrix structure *without* introducing any extra computations or information. Examining the elimination tree of this system and using theorem 1 allows independent subtrees to be identified within the elimination tree. These subtrees may be processed in parallel by assigning them to

different processors in a multiprocessor array. A parallel solution has been created by simply rearranging the system of equations.

Theorem 1 allows the parallelism inherent in the system to be exploited and theorem 2 ensures that both sequential and parallel solutions will require the same total amount of computation. This contrasts with a number of existing parallel solutions [44, 29, 35] which rely on the introduction of extra information or computations to allow the exploitation of parallelism. The new approach will be more efficient than existing methods, assuming that all methods exploit the parallelism in a given problem to the same extent and in the same way. Note that no reference has been made to the particular triangular decomposition method adopted. This is because the parallelisation based on theorems 1 and 2 depends *only* on the structure of the coefficient matrix and is *independent* of the decomposition technique. Hence a parallel solution can be devised which utilizes *any* of the available triangular decomposition methods.

It has been stated that partitioning the elimination tree into subtrees is the same as partitioning the network into subnetworks. Partitioning the elimination tree has the effect of giving the coefficient matrix a block based structure, in the same way that partitioning the network does. Each block in the matrix corresponds to one or more subtrees in the elimination tree. Allocating the main row/column blocks to individual processors, as described in Section 3.4.1, is the final stage of the partitioning process.

4.3.2 Using the Elimination Tree to Achieve Load Balancing

Section 4.3 describes how the computational load may be spread across an array of processors by assigning subtrees of the elimination tree to separate processors. This approach is based on a method due to George et. al [88] known as the *subtree-to-subcube* mapping. George et. al pioneered the method to assign subtrees of the elimination tree onto disjoint subcubes of a hypercube multiprocessor. Section 4.3 made no attempt to discuss how suitable subtrees are identified so that there are always exactly the same number of subtrees as processors, or how to assign subtrees to processors if there are more subtrees than processors. The problem of dealing with the portion of the tree below the chosen subtrees (*e.g.* nodes 9 and 10 in Figure 4.5) has also not been considered.

Geist & Ng [73] observe that the subtree-to-subcube mapping is only efficient when applied to balanced trees with a regular structure, such as those resulting from grid-based problems (*e.g.* finite element problems). Applying the method to imbalanced trees results

in a large increase in the amount of interprocessor communication which degrades the performance of the solution algorithm. Furthermore the method assumes there to be exactly the same number of subtrees as there are processors. Geist & Ng discuss ways of treating the lower portion of the matrix when using the subtree-to-subcube mapping and they put forward two approaches. The first uses a fine grain mapping to assign alternate nodes from the lower portion of the tree to the different processors whilst the second approach considers the entire lower portion of the tree to be a subtree, rooted at the root of the full tree. The latter technique was adopted here as it is more in keeping with the concept of minimally interconnected subnetworks. Consider treating nodes 9 and 10 of Figure 4.5(a) as a third subtree. This causes nodes 9 and 10 in the associated graph (Figure 4.5(b)) to be encapsulated into a third subnetwork lying between the two existing subnetworks (Figure 4.6). This new subnetwork has the property that, if it and the branches connected to it are removed, it separates the remaining network into disjoint subnetworks. Recalling the discussion of the diakoptic method from Section 3.5.1 highlights the fact that this new subnetwork has all the properties of the cutset block of the diakoptic method, and it is in fact the cutset of the system. This can easily be verified by considering the elimination tree and the role played by the third subtree (Figure 4.6(a)). Removing this subtree separates the tree into two disjoint subtrees. This cutset can be assigned its own processor but it cannot be processed in parallel with the other subtrees. As the structure of the elimination tree shows this cutset subtree must be processed *after* the other subtrees have been processed in parallel. This gives rise to the Master-Slave structure of Figure 3.2.

Geist & Ng refine the subtree-to-subcube mapping to create a method which is suitable for use with unbalanced trees. Given an arbitrary tree and a set of n processors the technique finds the smallest set of branches in the tree which can be partitioned into exactly n subsets whose solution requires approximately the same amount of work. The key to method is the use of a weighted elimination tree. Each node i in the tree is assigned a weight equal to the number of operations required to eliminate node i plus the sum of the weights of its child nodes. The weight of node i corresponds to the number of operations required to eliminate the subtree rooted at node i and the weight of the root node corresponds to the total number of operations required to factorise the matrix. An heuristic bin packing technique [73, 89, 90] is used to partition the tree into n subsets of approximately equal weighting. The initial step is to select the first n branches and place their weights in the n bins. A breadth first search then scans the tree selecting nodes and adding their weights to

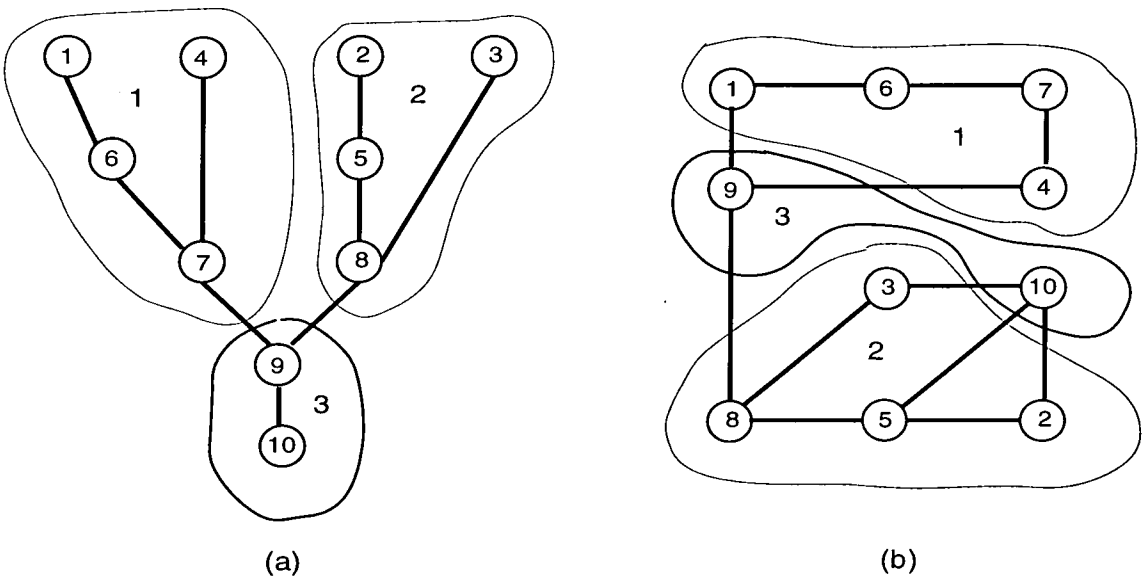


Figure 4.6: Treating the lower portion of the tree as a separate subnetwork a) partitioned elimination tree b) partitioned network

each of the bins in turn. The algorithm continues until the difference between the weights in the bins falls below some user defined tolerance. The contents of each bin (*i.e.* the nodes associated with it) are then assigned to individual processors. The remaining portion of the tree, referred to as the separator set, can then be treated using a wrapping assignment or a separate subtree. Figure 4.7 shows the results of this mapping for a simple tree, to be processed using 4 processors. A wrapping technique is used to assign the members of the separator set to different processors. Members of the separator set are denoted by double circles.

As Figure 4.7 shows, a significant portion of the tree is actually contained within the separator set. Treating the whole separator set (cutset) as a subtree is inefficient as this has to be processed sequentially after the other subtrees have been eliminated. The more nodes there are in the separator set, the more processing has to be performed sequentially, reducing the amount of potential parallelism in the problem. Geist's technique becomes inefficient when the cutset is treated as a separate subtree.

4.3.3 Advantages of the Tree-based Approach

The use of weighted elimination trees has a number of advantages besides those already mentioned. It has been suggested that the execution time of the parallel algorithm is proportional to the length of the critical path. More precisely, the parallel execution time is

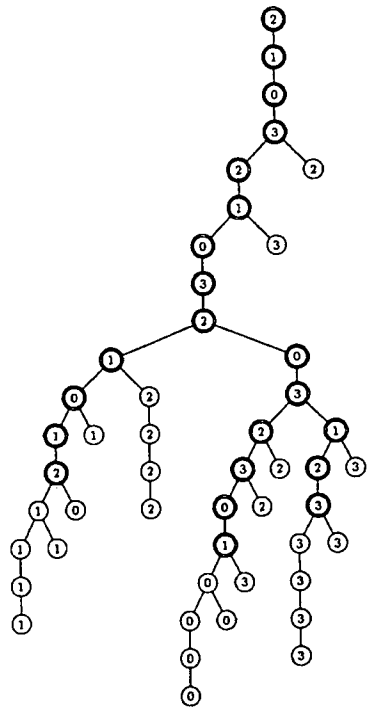


Figure 4.7: Geist & Ng's partitioning method applied to a simple tree (after Geist & Ng)

directly proportional to the number of operations that have to be performed to eliminate the nodes on the critical path during the factorisation stage (*i.e.*the weight of the critical path W_{cp}). The total number of operations required to factorise the whole matrix is equal to the weight of the root node, W_{root}. Hence the time taken to factorise the matrix sequentially is proportional to W_{root}. In solving any problem on a parallel computer we are interested in the amount of speed-up which the parallel solution gives us. The speed-up, S, is the ratio of the execution time of the best sequential program, t_{seq}, to the execution time of the parallel program, t_{par}. Combining this with the two observations made above allows the derivation of a simple rule of thumb for estimating the speed-up that can be obtained for the parallel solution of a set of equations. As

$$t_{seq} = \alpha_1 W_{root} \tag{4.2}$$

and

$$t_{par} = \alpha_2 W_{cp} \tag{4.3}$$

where α_1, α_2 are constants of proportionality, speed-up can be expressed as

$$S = \frac{t_{seq}}{t_{par}} = \frac{\alpha_1 W_{root}}{\alpha_2 W_{cp}} \tag{4.4}$$

If the sequential and parallel solutions are executed on the same type of processor running at the same clock speed

$$\alpha_1 \approx \alpha_2 \tag{4.5}$$

and hence

$$S \approx \frac{W_{root}}{W_{cp}} \tag{4.6}$$

Overheads and interprocessor communication have been ignored and (4.6) is a simple rule-of-thumb for estimating the speed-up that can be expected for a given system.

When considering networks with many thousands of nodes the elimination tree can become very large and it is difficult to plot this on a reasonably sized sheet of paper. Associating weights with each node in the tree allows the tree to be pruned to a more manageable size by replacing entire subtrees with a single node of the same weight. If a toleranced threshold weight is set it is possible to replace all subtrees with a weight that falls inside the tolerance band by single nodes of equivalent weight. During the course of this research project a software package was developed to derive and plot the elimination tree for any desired system. The toleranced threshold technique was used to automatically reduce the elimination tree diagram to an acceptable and easily managed size. The tolerance band in the program was set to $\pm 10\%$. The threshold value is calculated for each system according to the number of subnetworks required, n, and the total weight of the elimination tree. Hence

$$W_{thresh} = \frac{w_{root}}{n} \pm 10\% \tag{4.7}$$

As the partitioning of the elimination tree into subtrees is currently performed by inspection it is advantageous to have trees of a manageable size. The reduced trees can be used to determine the network partitioning but nodes replacing the pruned subtrees must be clearly marked so that when nodes are assigned to processors the pruned subtrees can be expanded back into their original form.

System Size	Subnetworks	Speed-up S_A	Speed-up S_B	Predicted speed-up
734	3	2.29	2.27	1.94
734	7	5.68	4.00	3.46
734	15	5.52	5.33	5.40

Table 4.3: The effect of load balancing on speed-up

4.3.4 Performance of the Tree-based Load Balancing

Table 4.3 shows speed-up results for a number of test systems partitioned into different numbers of subnetworks. For each case the speed-up resulting from two different load balancing techniques is shown. Scheme A equalizes the number of nodes in each subnetwork whilst scheme B equalizes the computational complexity of each subnetwork based on the weighted elimination tree analysis. S_A is the speed-up resulting from the former scheme whilst S_B is the speed-up resulting from the latter. The speed-up predicted by the rule-of-thumb of (4.6) is also shown for the sake of comparison.

Table 4.3 clearly shows the efficacy of the tree-based load balancing strategy, which consistently results in higher speed-ups. The speed-ups produced in practice are similar to those predicted using (4.6). At present, partitioning of the elimination tree is performed by visual inspection. However the process is an heuristic one and it should be possible to develop a set of heuristic rules which will form the basis of an automatic partitioning algorithm. This idea is developed further in Chapter 7.

4.4 Summary

This chapter has considered the role played by the elimination tree in the creation of parallel solution methods. It has been demonstrated that the elimination tree is an indispensable aid to be used in meeting the combined goals of network partitioning and load balancing. The need for a balanced computational load has been stressed in this and other chapters and a method has been presented which allows the network to be partitioned to give the most balanced load. The method is based on the use of a weighted elimination tree and it partitions the network in a way which tries to equalise the computational requirements of each subnetwork. Results from a number of test systems have shown that this method produces good parallel performance. A simple technique for estimating the maximum speed-

up that can be expected from the parallel solution of a set of equations has also been introduced.

The most significant point emerging from this chapter is that it is possible to improve the performance of a parallel solution of a given system simply by rearranging the elimination tree associated with that system. The concept of an equivalent reordering has been introduced and a theorem has been presented which states that applying an equivalent reordering changes the matrix structure and elimination tree of that system but not the amount of computation required to yield a solution. This allows a parallel solution to be formulated which requires the same total amount of computation as the best sequential solution and this is an improvement over existing parallel formulations, many of which introduce extra computation steps to allow the exploitation of parallelism. The new method is independent of the solution algorithm chosen as it depends only on the structure of the elimination tree. Hence it is possible to formulate a parallel solution using any desired triangular solution method. The next chapter discusses the design of a Transputer implementation of this new parallel formulation, based around the bifactorisation algorithm.

Chapter 5

An Improved Parallel Factorisation

5.1 Introduction

This chapter considers in detail how the insight provided by the elimination tree can be used to create an improved parallel triangular factorisation and solution. For power system simulations it is most important to improve the performance of the forward/backward substitution operations of the algorithm as it is these operations which are continuously repeated in a dynamic simulation. Factorisation only needs to be performed once before repeated solution can occur and poorer performance can be tolerated. Any event which causes a change in the topology of the system network requires a refactorisation of the matrix before further solutions can be obtained. When the topology does change it is vital that the refactorisation be performed as quickly as possible so that real-time solutions remain in soft real-time[1]. Therefore it is also desirable to improve the performance of the factorisation part of a triangular method. 'Performance' in this context equates to the speed-up obtained from the method under consideration.

The previous chapter has discussed existing parallel Cholesky and LU-based methods for the solution of linear equations. In the development of these methods the authors have adopted various algorithmic approaches. The method described in this chapter draws upon these different approaches and uses a combination of the best techniques to give an

[1]Soft real-time systems are those in which response time is important but the system still functions correctly if some deadlines are missed. More specifically, soft real-time systems are real-time systems which are tolerant of the occasional missed deadline or deadlines which are not missed by much [91].

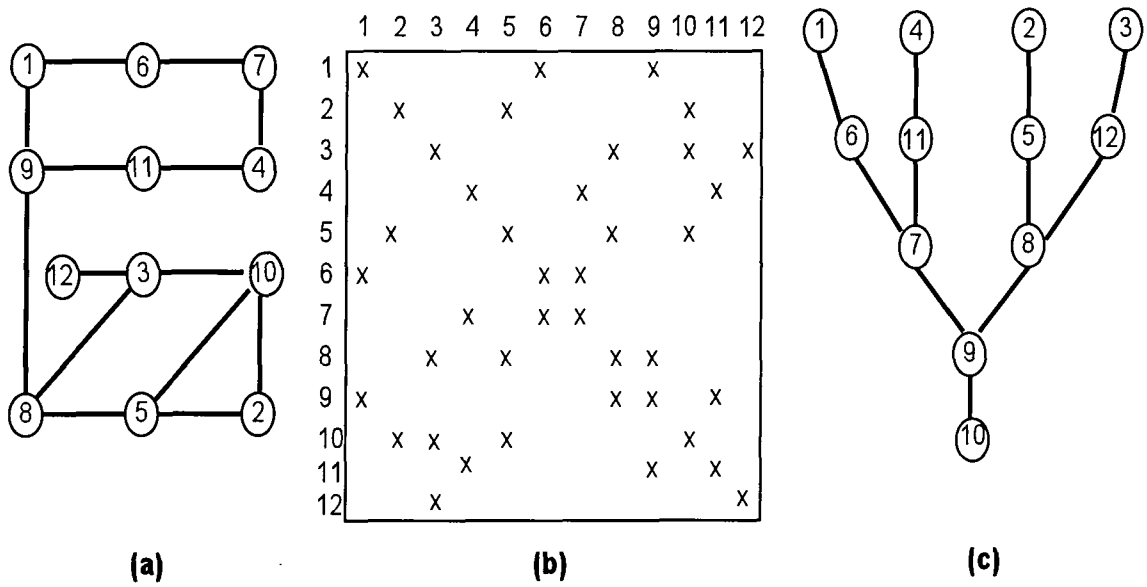


Figure 5.1: The simple 12 node example system a) network graph b) coefficient matrix structure c) elimination tree

improvement in both the factorisation and substitution steps of a parallel LU-based solution algorithm.

5.2 Development of the Recursively Parallel Method

5.2.1 Identifying the Potential Parallelism

The discussion on the identification of potential parallelism is best treated with regard to a specific example. The example used here is a simple 12 node system and the network, its associated matrix and elimination tree are shown in Figure 5.1. The elimination tree of the system, which has already been ordered using an optimal ordering algorithm, shows the parallelism existing in the problem.

To implement a parallel solution the workload must be divided up and assigned to the individual processors. Consider the factorisation phase of the solution algorithm (a similar argument applies to the substitution phase). The workload in this case is the set of nodes which must be eliminated from the network (*i.e.* the nodes in the elimination tree). Dividing the workload is equivalent to assigning nodes to processors for processing, or in terms of the elimination tree, dividing up the tree and assigning the partitions to the available processors. Two approaches exist for partitioning the tree - the first considers each node in the tree individually whilst the second groups nodes into collections of nodes

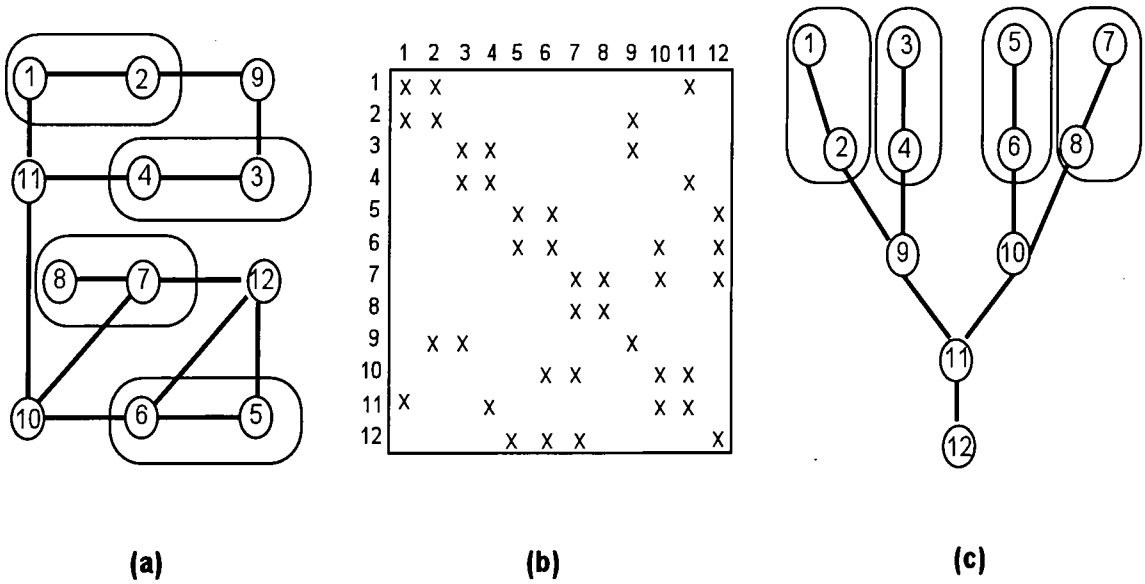


Figure 5.2: a) Partitioned topologically reordered network, b) admittance matrix structure and c) effect on elimination tree

which are then assigned to individual processors. The first method also clusters tree nodes into groups but each group contains nodes which are scattered throughout the tree. This approach is equivalent to a wrap mapping and results in a large volume of interprocessor communication. As a result this form of partitioning is really only useful for algorithms implemented on parallel machines with shared memory. The latter method is more suitable for distributed memory machines as it groups together nodes in the same region of the tree and results in a lower volume of interprocessor communication.

Consider the example network - it is possible to identify four subtrees which may be processed concurrently. To partition the network into four independent subnetworks a topological reordering must be applied. It is assumed that the network of Figure 5.1 has already been optimally ordered. The four independent subtrees are {1,6},{4,11},{2,5},{3,12} and the remainder of the network ({7,8,9,10}) are the cutset. The topological reordering will reorder the network such that nodes in the independent subtrees are numbered first and cutset nodes are numbered last. The nodes in the first subtree {1,6} will be renumbered as {1,2} whilst the nodes in the last subtree will be renumbered as {7,8}. The cutset nodes will be renumbered as {9,10,11,12}. Figure 5.2(a) shows the renumbered network after topological reordering and Figure 5.2(c) shows how the elimination tree is divided into the four independent subtrees which correspond to the four subnetworks. Parallel factorisation can be accomplished by assigning the four subnetworks to four different processors. The

problem then arises of how to deal with the cutset nodes. These nodes may be assigned to the four processors along with the subnetworks using a wrap mapping [73] but this introduces a large volume of communication during the cutset solution. A better approach [73] is to treat the cutset as a separate subtree and assign it to its own (fifth) processor. Figure 5.2(c) implies that the cutset must be processed after the processing of the other four subnetworks is complete. Information from all the other processors must be passed to the processor which hosts the cutset so the cutset must be processed by a central coordinating processor. The algorithm structure is that of Figure 3.2.

The problem in treating the cutset as a separate subtree is that it ignores potential parallelism existing within the cutset. In the cutset of Figure 5.2 nodes 9 and 10 may be processed simultaneously, as indicated by Figure 5.2(c). Three further subtrees may now be created {9}, {10} and {11,12}. These subtrees are derived by partitioning the cutset and are referred to as *minor* subtrees, or *minor* subnetworks. Subtrees {1,2},{3,4},{5,6},{7,8} are created by the existence of the cutset and are referred to as *major* subtrees, or *major* subnetworks. Subtrees {9} and {10} can be assigned to separate processors and factorised in parallel whilst subtree {11,12} is factorised after these two have completed. Extra parallelism has been exploited by partitioning of the cutset thus reducing the amount of sequential computation. The elimination tree now consists of three distinct levels. Each level contains a number of subtrees (subnetworks) and all the subtrees in a level are independent and may be processed in parallel. For the example of Figure 5.2 the levels are

Level 1	{1,2},{3,4},{5,6},{7,8}
Level 2	{9},{10}
Level 3	{11,12}

Although the processing within levels can be performed concurrently, the levels themselves must be processed in sequence (*i.e.* Processing of level 2 cannot begin until the processing of level 1 is complete).

The scheduling strategy can be used to make this approach more efficient as it is observed that subtrees {1,2},{3,4},{5,6} and {7,8} must be factorised before {9} and {10} can commence factorisation. The processors which dealt with {1,2},{3,4},{5,6},{7,8} are lying idle and may be used to factorise {9} and {10}. The same argument can be applied to subtree {11,12} and only four processors are required to factorise the seven subtrees. Each subtree constitutes a separate computational task and more than one task is assigned

to each processor. For example, one processor will host the tasks (subtrees) {1,2}, {9}, {11,12} whilst another might host {5,6} and {10}. This is a departure from existing approaches which divide the problem into computational tasks and assign each task to its own processor. The essence of the method is to exploit the parallelism which exists within the cutset by making use of idling processors and the method has been termed the Recursively Parallel (RP) method.

5.2.2 The Recursive Bordered Block Diagonal Form

If the Recursively Parallel solution is to be efficient it is necessary to ensure that exploiting parallelism within the cutset actually reduces the volume of communications to the last task. This has been achieved by constraining the RP method to make use of a particular coefficient matrix structure known as the Recursive Bordered Block Diagonal Form (RBBDF).

The RBBDF matrix can be derived by considering a graph of the network described by the linear equations. Normally the network is comprised of a number of subnetworks connected in some arbitrary fashion. Suppose there exists a system with fifteen subnetworks, eight major and seven minor, and that the interconnections are constrained such that the subnetworks are arranged in binary tree structure. Figure 5.3(a) shows the tree structure and Figure 5.3(b) shows the matrix associated with the network. This matrix structure is similar to the BBDF structure in that there are blocks along the leading diagonal and a border region along the bottom and right of the matrix. Most of this border region is empty and it contains only two lines of blocks which run diagonally across to the lower right corner of the matrix. There is a significant amount of parallelism available for exploitation in this structure as the constrained interconnection creates very little dependence between the matrix blocks. There are no dependencies between the first eight blocks and these may be processed in parallel. Significantly these first eight blocks are all the subnetworks in the leaf level (level 0) of the tree-based network (*i.e.* the major subnetworks). Dependencies do exist between the major and the minor subnetworks themselves. During factorisation of blocks 1 to 8 updates will be made to blocks 9 to 12 and hence 9 to 12 cannot be processed until all processing on blocks 1 to 8 is complete. There are no dependencies between blocks 9 to 12 and these may also be processed in parallel. These blocks together constitute level 1 of the tree-based network graph. Processing of blocks 9 to 12 updates blocks 13 and 14 and the processing of these two blocks must commence after the processing of blocks 9 to 12 has completed. The lack of dependencies between 13 and 14 allows these two blocks

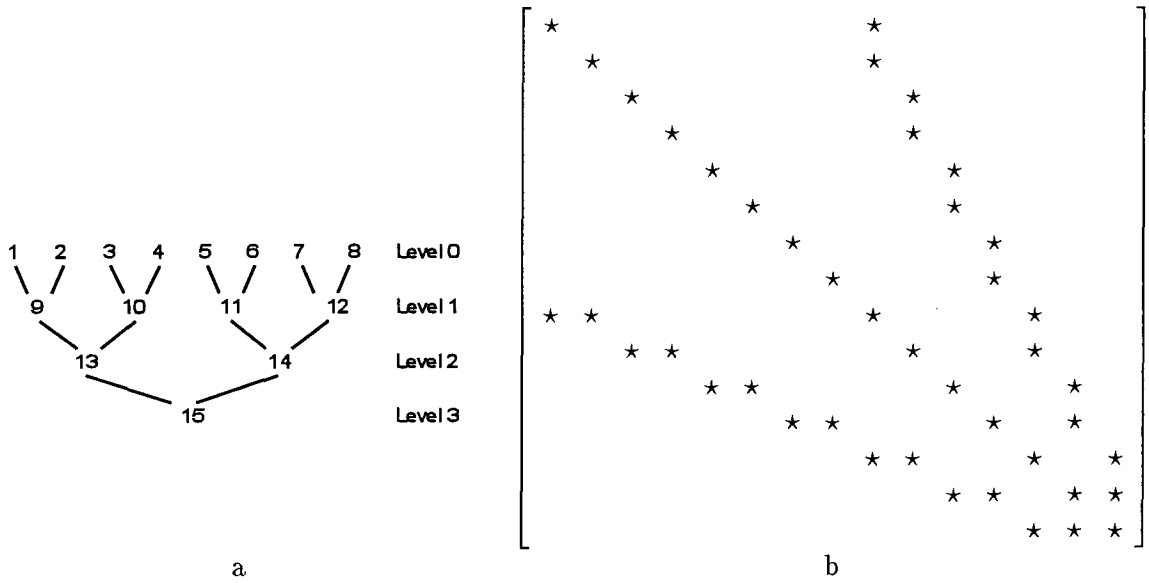


Figure 5.3: Subnetworks constrained to a binary tree connection structure a) elimination tree b)coefficient matrix structure

to be processed in parallel and they constitute level 2 of the network graph. Finally block 15 may be processed after 13 and 14 have completed and this block corresponds to the root node of the tree structured network graph. The four levels of the tree give rise to four regions of independence within the matrix. All the blocks within each of these regions may be processed in parallel. However the regions must be dealt with one after another in a sequential manner. Constraining the network graph to have a binary tree structure immediately introduces four phases of parallelism into the factorisation and substitution operations on the coefficient matrix. Unfortunately the network is somewhat artificial and it would be very difficult to partition a real network, particularly one as complicated as a power system, such that its constituent subnetworks were connected together to form a binary tree.

Suppose that the strict binary tree connection constraint is relaxed so that an additional connection is allowed between every subnetwork and the last subnetwork (*i.e.* the root) of the tree. Adding in the extra connections creates a border in the last row and column of the matrix but this does not affect the exploitation of parallelism described in the preceding paragraph. Four phases of parallelism still exist corresponding to the four levels in the tree and all subnetworks which lie in the same level of the tree may be processed in parallel. The only consequence of allowing the extra connections in the network graph is that extra updates have to be performed between the parallel phases. For example the processing of

level 0 (*i.e.* blocks 1 to 8 in parallel) requires updates to block 9 to 12 but also to block 15. Relaxing the connection constraints still further allows any subnetwork to be connected to any other subnetwork below it in the tree. Connections across the tree are not allowed. Implementing this strategy would allow subnetwork 1 to be connected to subnetworks 9, 13 and 15. This is depicted in Figure 5.4(a) and the associated matrix is given in Figure 5.4(b). This matrix exhibits Recursive Bordered Block Diagonal Form. If the diagonal blocks corresponding to the subnetworks in level 0 of the tree are considered it can be seen that the remaining blocks form a border block below and to the right of them. The diagonal blocks corresponding to the first two levels (1 to 12) are also bordered below and to the right by the remaining network blocks. Similarly the diagonal blocks for the first three levels are also bordered below and to the right by the remaining network blocks. The bordered block diagonal form is recurrent in the matrix giving rise to the name of Recursive Bordered Block Diagonal Form.

Interconnecting the subnetworks such that the associated matrix has RBBDF imposes constraints on the network partitioning but these are reasonably loose constraints. Many connections are allowable in the network and it is relatively easy to partition any real network such that the coefficient matrix is in RBBDF. The structure is even more flexible in that some of the connections may be missing. For example if the connection between 1 and 15 is missing it does not significantly alter the matrix structure and the same four phases of parallelism can be exploited. In fact any of the connections may be missing and more than one may be missing simultaneously. The only requirement is that there must be at least one connection to each subnetwork which is present. An even greater degree of flexibility is offered when it is recognized that not all of the subnetworks have to be present either. If insufficient network partitions can be found to give the required number of subnetworks without violating the connection constraints then some of the subnetworks can be missed out. For example if only 14 subnetworks can be found, subnetwork 1 in Figure 5.4(a) can be missed out [2] so that only seven subnetworks exist in the first level. Multiple subnetworks can be missing simultaneously, the only condition being that the root of the tree must always exist.

The constraints which must be imposed on the network partitioning in order to produce

[2]The numerical labeling of the subnetworks must be contiguous. In this example, level 0 would consist of nodes 1 to 7, level 1 consists of nodes 8 to 11, blocks 12 and 13 lie in level 2 and the root of the tree is block 14.

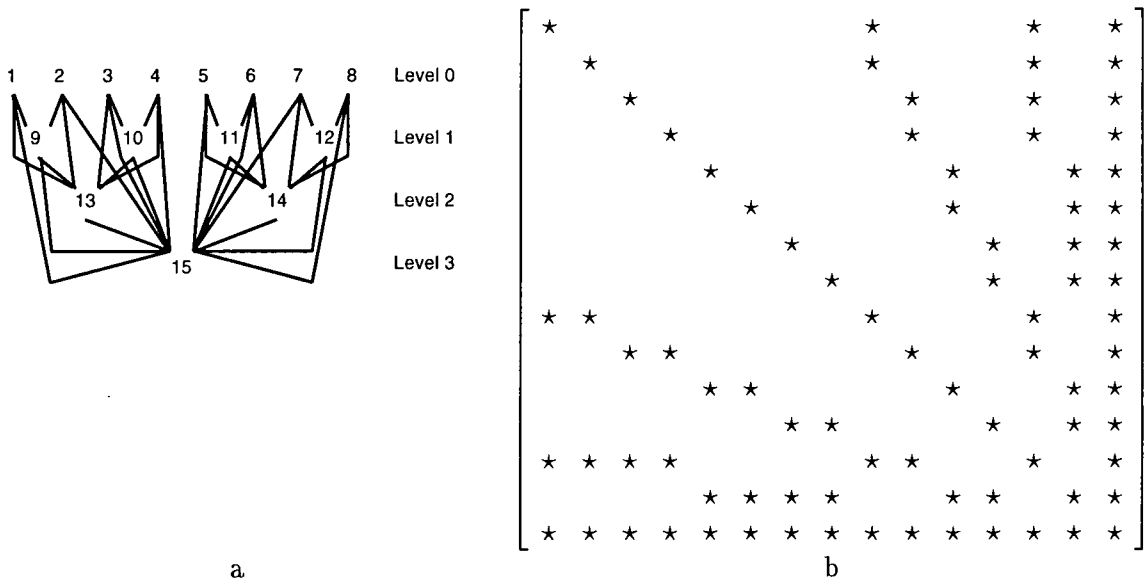


Figure 5.4: The interconnections giving rise to RBBDF a) elimination tree b) coefficient matrix structure

an RBBDF coefficient matrix may be summarized as

- The interconnection of subnetworks is based on a binary tree with additional connections
- There may be *up to* $2^m - 1$ subnetworks connected in a tree-like fashion, where m is the number of levels in the tree and indicates the number of parallel phases that can be exploited in either factorisation of substitution
- The subnetwork which forms the root of the tree *must* always be present - any other subnetworks may be missing
- The subnetworks which are present must be labeled contiguously
- Connections may be missing from the tree structure but there must be *at least* one connection to every subnetwork which is present. In other words, subnetworks which are present must be connected to the tree and cannot be isolated.
- Any subnetwork is allowed to connect to any other subnetwork *below* it in the tree. Connections across the tree are not allowed

Figure 5.5 shows the possible interconnections for networks with 3, 7, 15 and 31 subnetworks.

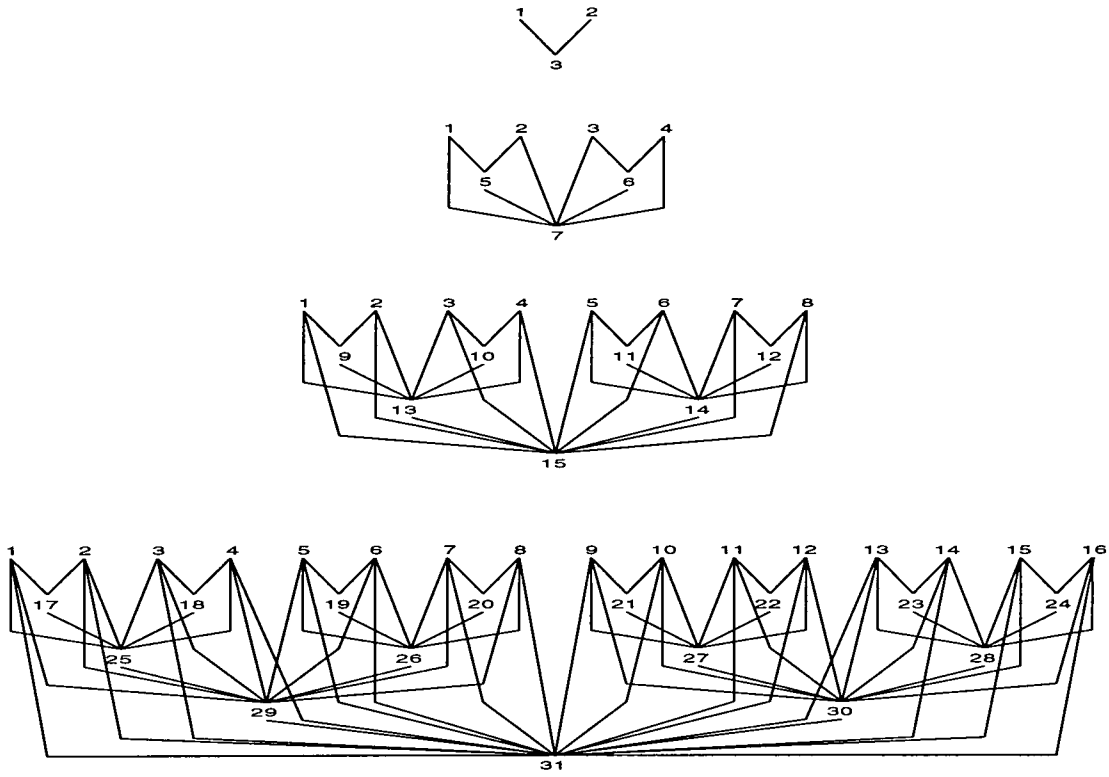


Figure 5.5: The constrained subnetwork interconnections

Given the coefficient matrix of a system it may be placed into RBBDF in three simple steps

1. Apply an optimal ordering to this system to ensure minimum fill-in and short path lengths

2. Determine the elimination tree of the system

3. Using the elimination tree, apply the equivalent reordering which converts the matrix into RBBDF.

The equivalent reordering is determined by visual inspection of the elimination tree. The tree is divided into the required number of subtrees and these subtrees must be arranged within the tree such that their interconnections do not breach the connectivity constraints necessary for RBBDF. The individual nodes of the tree must then be renumbered. Suppose that k subtrees have been identified and that subtree 1 encloses n_1 nodes, subtree 2 encloses n_2 *etc.* The individual tree nodes are renumbered a subtree at a time in ascending subtree order. The n_1 nodes in subtree 1 are renumbered as 1 to n_1. The nodes in subtree 2 are then renumbered as $n_1 + 1$ to $n_1 + n_2$ *etc.* Once all the nodes in the tree have been renumbered

the matrix associated with the tree will have been restructured to give RBBDF.

5.2.3 Balancing the Load

The subtree-to-subcube mapping (Section 4.3.2) is not adequate for partitioning trees for solution with the Recursively Parallel method. A variation of this technique has been developed to partition the tree so as to exploit the potential parallelism existing in the cutset itself. As with subtree-to-subcube mapping, the approach is based on the use of a weighted elimination tree. For each node i in the tree the number of multiplication-additions required to eliminate that node from the network is calculated. This is the *computational complexity*[3] $C(i)$ of eliminating node i and if row i of the matrix contains k non zero elements

$$C(i) = 1 + k + \frac{k(k+1)}{2} \tag{5.1}$$

The nodes in the tree are assigned a weight, $W(i)$, defined to be the computational complexity of node i plus the weight of the descendant nodes. The weight of the root node is the computational complexity of factorising the whole matrix. The method then assigns nodes to processors by selecting the correct number of subtrees from the elimination tree. For example, to partition the network into eight main subnetworks up to fifteen subtrees have to be identified - the eight main subtrees and seven minor ones. These subtrees must be connected in the manner shown in Figure 5.5.

When partitioning an elimination tree it is necessary to pick subtrees such that those subtrees which lie in the same level of the appropriate tree in Figure 5.5 have roughly equal weights. For example subtrees 1 to 8 lie in the same level and they should have approximately equal weights. Consider the tree of the reduced CEGB 734 node system shown in Figure 5.6 and how it is partitioned into subtrees for a solution with eight main subnetworks on eight processors. The subtree rooted at * has a weight of 1228 whilst the subtree rooted at ** has a weight of 1246. In real systems it is not usually possible to identify subtrees with exactly the same weight and some imbalance in the computational loading has to be accepted. As following chapters will show, this imbalance can be used to improve speed-up.

[3]Several other formulae for determining computational complexity may also be used *e.g.* total number of nonzeroes in the corresponding matrix row, total number of machine level arithmetic operations for factorisation, total number of machine level arithmetic operations for substitution

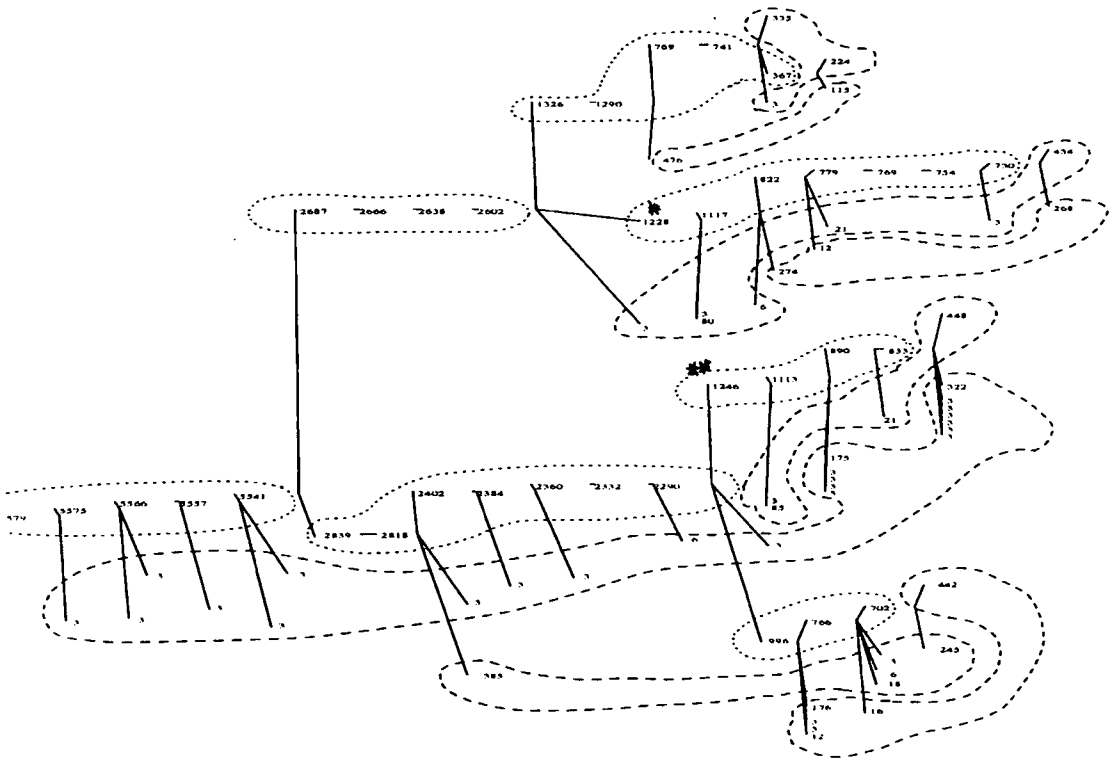


Figure 5.6: Partitioning of the reduced CEGB 734 node system for solution on 8 processors

5.2.4 Reducing the Sequential Part of the Method

The RBBDF matrix structure may be used to reduce the size of the sequential part of a conventional parallel LU solution and can increase performance. Section 5.2.1 showed that potential parallelism exists in the processing of the cutset and the use of RBBDF allows this parallelism to be exploited.

Consider the network represented in Figure 5.4(a). The sequential part of the algorithm is reduced by partitioning the cutset blocks into minor subnetworks found in level 1 and below. In Figure 5.4(a) minor subnetworks 9 to 15 together constitute the cutset block and they would be solved sequentially as a single block in a conventional triangular solution. Blocks 1 to 8 are the subnetworks in the traditional approach and parallelism is only exploited in the processing of these subnetworks. Giving RBBDF structure to the coefficient matrix and partitioning the cutset into minor subnetworks allows parallelism to be exploited in the solution of both main subnetworks and the cutset. Parallelising cutset processing reduces the size of the sequential part of the method and will result in improved speed-ups. For example, consider the system of Figure 5.4(a) and assume that the processing of each subtree requires one unit of computation. In a conventional BBDF-based parallel solution

subtrees 1-8 would be processed in parallel whilst subtrees 9-15 would be aggregated into a single subtree and processed sequentially. Processing of 9-15 would take seven units of computation whilst parallel processing of 1-8 would take one unit of computation. The BBDF-based solution requires eight units of computation time to yield a solution. Now consider the RBBDF-based triangular solution. Subtrees 1-8 are again processed in parallel requiring one unit of computation. Now 9-12 are also processed in parallel, requiring one unit of computation. Processing 13 and 14 concurrently requires one unit of computation and processing of 15 requires one further unit of computation. The RBBDF-based solution requires only 4 units of computation time to yield a solution, which is a significant improvement over the BBDF-based solution. Note that both solutions require the same amount of *total* (sequential) computation but the RBBDF solution allows idle processors to be used to process the cutset in parallel. The exploitation of parallelism within the cutset is made possible by the use of the RBBDF coefficient matrix.

5.3 A Simulation of the Recursively Parallel Method

In order to verify the effectiveness of the Recursively Parallel method a simulation of the approach was implemented on an IBM PC-AT clone. A suite of four test systems was used in the simulation and these were partitioned into various numbers of subnetworks for solution. There were three aims to the simulation

1. To verify that the RP method worked correctly, particularly in the presence of missing connections and subnetworks.

2. To verify that the RP method, when executed on a parallel machine, would give a faster solution than the best sequential method. Also to verify that the total computation time for the RP method was the same as that of the best sequential method.

3. To verify that the RP method, executed on a parallel machine, would give a faster solution than existing parallel LU based solution techniques.

This section discusses the implementation of the simulation and the results obtained from it.

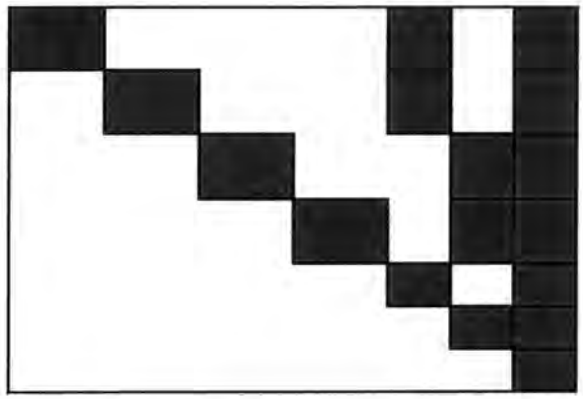


Figure 5.7: The block oriented data structure

5.3.1 Implementation

The simulation of the RP method is based around Zollenkopf's bifactorisation method. Bifactorisation is used to decompose the RBBDF coefficient matrix into the relevant factor matrices and these are multiplied together to yield the solution for the unknown vector. The use of the RBBDF coefficient matrix allows the factorisation and substitution operations to be performed in a number of parallel phases, as described in the previous section. Timing mechanisms have been built into the simulation to allow the computations to be accurately timed. This enables an estimate to be made for the computation time of a multiprocessor implementation of the method. The overall execution time may be monitored and it is also possible to time the processing of individual regions of the coefficient matrix. To maintain maximum accuracy timing is performed using an external digital counter/timer connected to the PC's parallel port. This timer has a resolution of $\frac{1}{100}$th of a millisecond and timing is triggered by start and stop signals sent from the PC. The times recorded by the timer are manually entered into the simulation by the user.

The heart of the simulation, and of the multiprocessor implementation, is the data structure used to store the coefficient matrix. Two data structures were used in the simulation, each giving rise to a slightly different algorithm. The first structure, depicted in Figure 5.7, treats the coefficient matrix in a blockwise manner. Each populated region in the matrix, shown shaded in Figure 5.7, is treated as a separate block and bifactorisation is accomplished by performing elementary operations on the individual matrix blocks. Computation time is monitored by timing the elementary operations on individual blocks. The method is described further in Chapter 8 where it is shown that this block oriented treatment of the matrix may form the basis of a different type of parallel solution. For

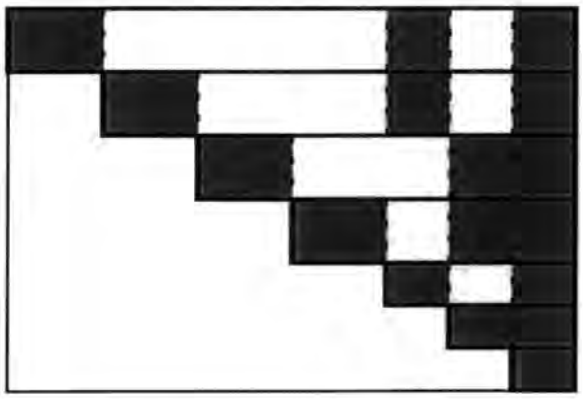


Figure 5.8: The row oriented data structure

the method described in this chapter the block-oriented approach does not give the most efficient solution. The second data structure treats the matrix in a row-oriented fashion. Again the matrix is stored in sections, as in Figure 5.8, and if there are n subnetworks then there will be n sections in the matrix. Each of the sections extends from the diagonal to the righthand edge of the matrix and covers both populated (shaded) and unpopulated (unshaded) regions of the matrix. It is observed that the lower right corner of the matrix is more densely populated and sections in this region may employ full array storage for matrix elements whilst those in the rest of the matrix make use of sparse linked list storage. The location of the changeover between sparse and full storage is specified by the user. For many of the simulation runs only the last section made use of full array storage. Bifactorisation is accomplished using the bifactorisation rules of equations (2.28) to (2.30). Timing of the computation is restricted to monitoring the processing time for each section but this is sufficient to allow a prediction of parallel computation time.

During a simulation run a time is returned for the factorisation, left multiplication and right multiplication operations on each block. If the system is partitioned into n blocks there will be n factorisation times $t_f[1] \ldots t_f[n]$, n left multiplication times $t_l[1] \ldots t_l[n]$ and $(n-1)$ right multiplication times $t_r[1] \ldots t_r[n-1]$. These $(3n-1)$ timing statistics may be manipulated to produce a value for the total computation time of the RP method and an estimate of the execution time of the RP method on a parallel computer. Total computation time is easily found by summing all $(3n-1)$ times and this should be similar to the total computation time for the best sequential method. If this does not hold then the RP method will be inherently less efficient than the sequential method. Total computation time may be subdivided into total times for each of the three solution operations (*i.e.* factorisation,

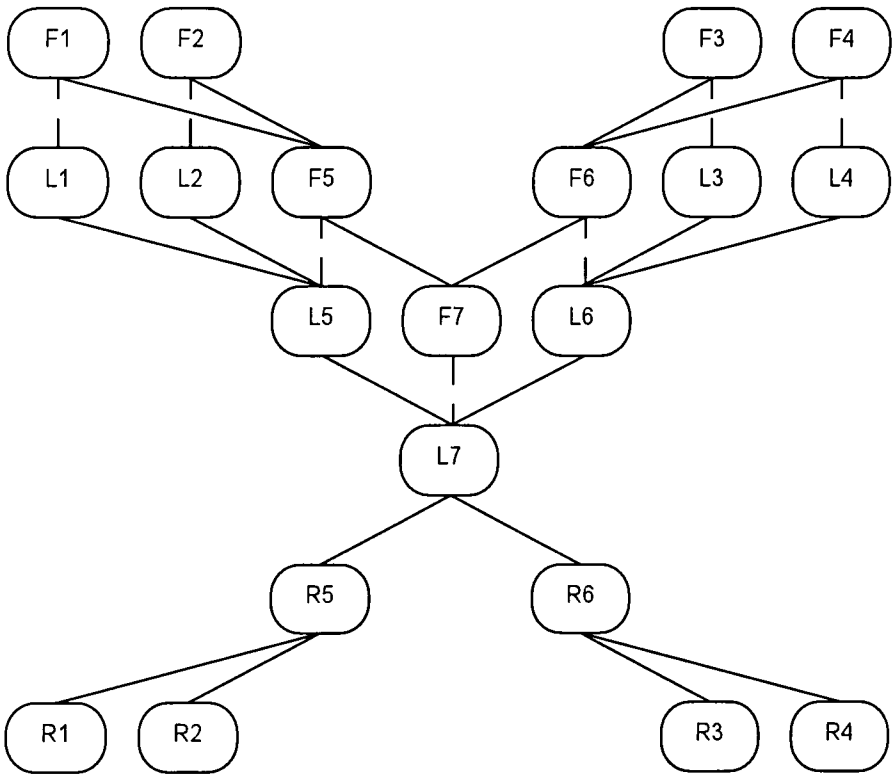


Figure 5.9: Algorithm structure for the Recursively Parallel method

left multiplication and right multiplication). The following relationships will be true if the RP method is as efficient as the best sequential method

$$t_{SEQ_f} \approx \sum_{i=1}^{n} t_f[i] \tag{5.2}$$

$$t_{SEQ_l} \approx \sum_{i=1}^{n} t_l[i] \tag{5.3}$$

$$t_{SEQ_r} \approx \sum_{i=1}^{n} t_r[i] \tag{5.4}$$

The estimated execution time for the parallel RP solution can be determined by considering the tree-based interconnection of subnetworks. This gives rise directly to the algorithm structure of the RP method which is of the form shown in Figure 5.9. Consider the factorisation operations. F_1, F_2, F_3, F_4 all commence at the same time. F_5 cannot commence until both F_1 and F_2 complete whilst F_6 cannot commence until F_3 and F_4 are complete. F_7 commences when both F_5 and F_6 are complete. Suppose that F_1 takes much longer to perform than F_2, F_3, F_4. F_3 and F_4 will complete and F_6 will be able to commence its operation. F_2 will also have completed but F_5 is held up awaiting the completion of F_1. When

F_1 finally finishes F_5 will commence but this will occur some time after the commencement of F_6. If F_5 and F_6 take equal times to process then F_6 will complete before F_5 and F_7 will be held up awaiting the completion of F_5. The long processing of F_1 can be seen to ripple down through the tree causing other operations to wait and F_1 has a direct effect on the total time for parallel factorisation. In fact the longest operation in the first level of the tree defines a *critical path* from this operation to the root of the tree. Summing the times of factorisation operations on this critical path gives the total time for parallel factorisation, t_{PAR_f}. Summing the left and right multiplication times along the same path defines the total left and right multiplication times, t_{PAR_l} and t_{PAR_r} respectively.

A cursory examination of the algorithm structure shows that the total parallel execution time is *not* the sum of the total parallel times for each of the three operations. Many of the left multiplication operations occur in parallel with the factorisation operations and are effectively hidden behind the factorisations. Only one left multiplication operation is not performed in parallel with the factorisations and all of the right multiplications have to be performed after the left multiplications are complete. The total parallel execution time for the RP method is therefore given by

$$t_{PAR_{total}} = t_{PAR_f} + t_{PAR_r} + t_l[n] \tag{5.5}$$

The speed-ups can be calculated as

$$S_f = \frac{t_{SEQ_f}}{t_{PAR_f}} \tag{5.6}$$

$$S = \frac{t_{SEQ_{total}}}{t_{PAR_{total}}} \tag{5.7}$$

where S_f is the factorisation speed-up and S is the overall speed-up. Substitution speed-up must be calculated in a different manner. In a power system simulation, where the network equations are solved for many different right hand sides, factorisation is performed only once but substitution is performed repeatedly. Under these conditions the structure of the algorithm changes to that shown in Figure 5.10. None of the left multiplications are hidden behind other operations and the time for parallel substitution, $t_{PAR_{subst}}$, becomes

$$t_{PAR_{subst}} = t_{PAR_l} + t_{PAR_r} \tag{5.8}$$

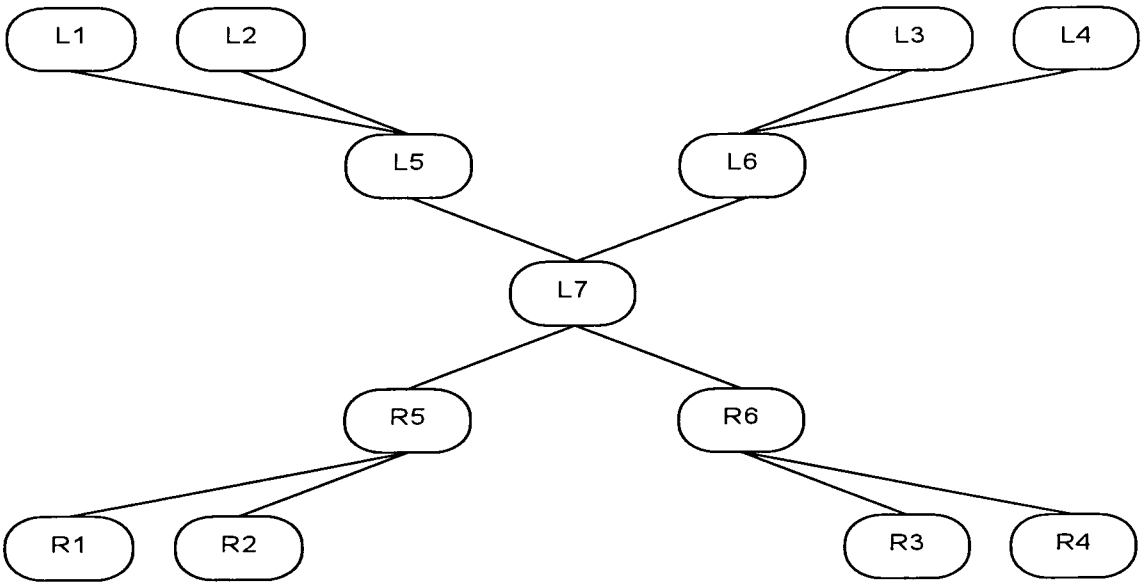


Figure 5.10: Algorithm structure for repeated substitution with multiple right hand sides

Hence the substitution speed-up, S_s, is given by

$$S_s = \frac{t_{PAR_l} + t_{PAR_r}}{t_{SEQ_l} + t_{SEQ_r}} \tag{5.9}$$

5.3.2 Results of the Simulation

The results of the simulation are encouraging and bear out all the theoretical predictions presented earlier in this chapter concerning the performance of the RP method. Table 5.1 lists the speed-up results for both the RP method and a standard parallel LU method for the four test systems. Figure 5.11 to Figure 5.13 show these results graphically.

It is easy to see from the graphs that the RP method performs better than the existing approaches to parallel solution. Higher absolute speed-ups are obtained and as the number of processors increases the speed-ups carry on increasing and do not appear to suffer from the saturation which affects the standard solution. This does not mean that the RP solution never saturates, simply that saturation is not observed within the region of interest. Given that the RP method returns higher speed-ups than the standard method from the same number of processors it is obviously more efficient.

The results obtained from the simulation are somewhat optimistic as there are certain characteristics of a parallel program that are not included in the simulation. The main omission is that of interprocessor communication and the overheads it introduces. A communication between two processors takes a finite amount of time and this depends not only

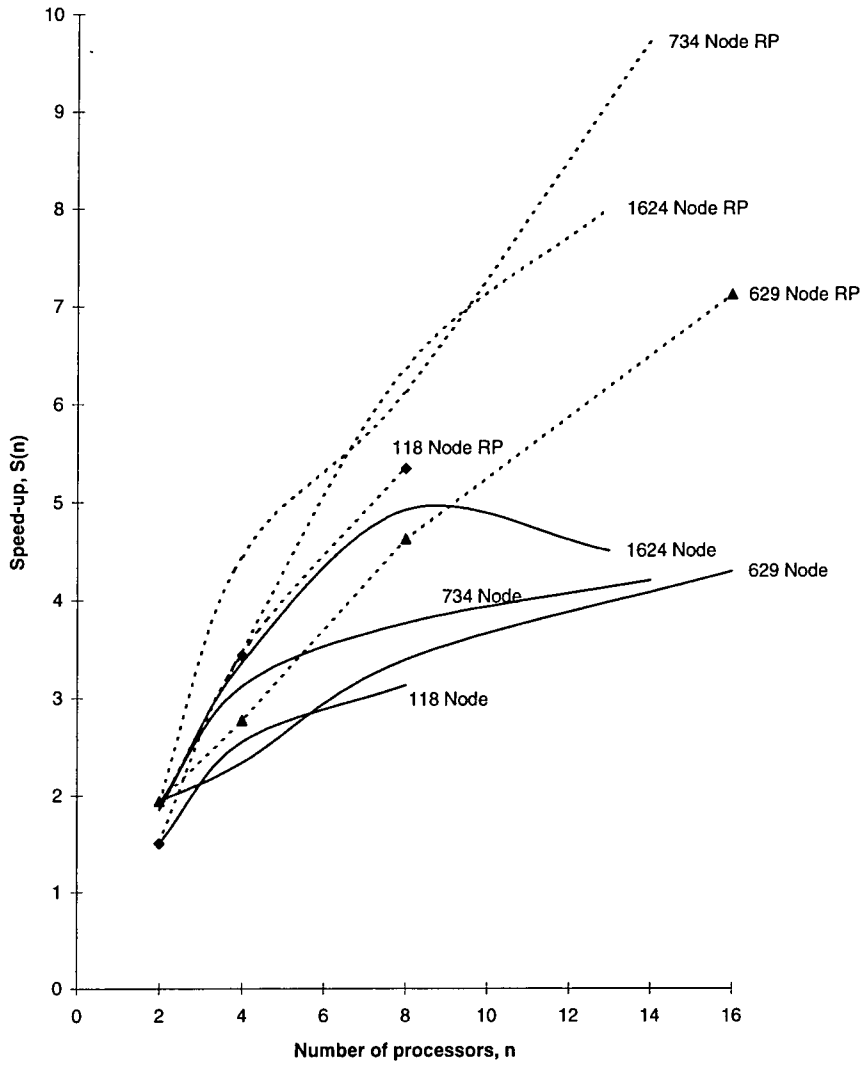


Figure 5.11: Overall speed-up results of simulated solution of the four test systems - solid lines correspond to the standard parallel method whilst dashed lines correspond to the RP method

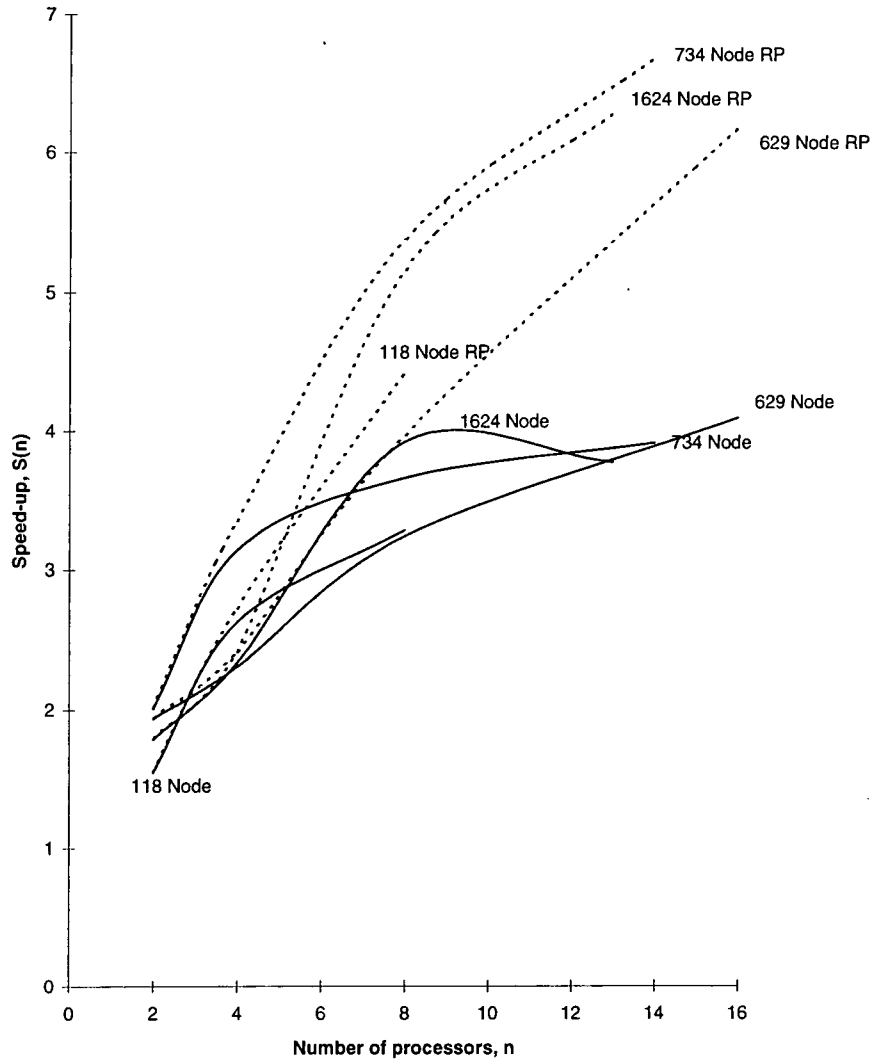


Figure 5.12: Factorisation speed-up results of simulated solution of the four test systems - solid lines correspond to the standard parallel method whilst dashed lines correspond to the RP method

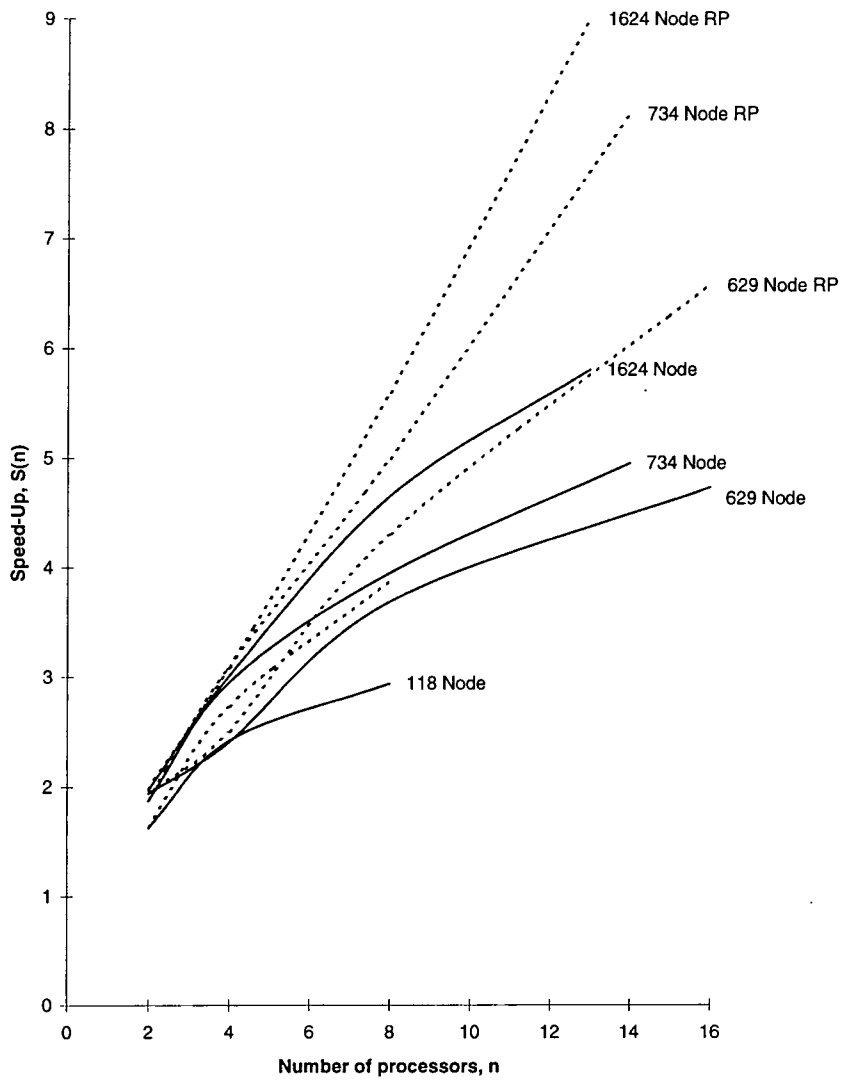


Figure 5.13: Substitution speed-up results of simulated solution of the four test systems - solid lines correspond to the standard parallel method whilst dashed lines correspond to the RP method

System	CPU's	Recursively Parallel Speed-up			Standard Speed-up		
		Overall	Factorise	Substitute	Overall	Factorise	Substitute
118	2	1.50	1.55	1.62	1.50	1.55	1.62
118	4	3.43	2.72	2.72	2.55	2.63	2.42
118	8	5.34	4.41	3.86	3.13	3.28	2.94
629	2	1.94	1.94	1.94	1.94	1.94	1.94
629	4	2.77	2.41	2.49	2.34	2.31	2.41
629	8	4.62	3.96	4.29	3.39	3.24	3.69
629	16	7.12	6.17	6.57	4.29	4.09	4.73
734	2	1.92	2.01	1.87	1.92	2.01	1.87
734	4	4.43	3.34	3.08	3.12	3.14	2.95
734	8	6.12	5.37	4.98	3.76	3.66	3.95
734	14	9.71	6.67	8.14	4.2	3.91	4.95
1624	2	1.85	1.79	1.97	1.85	1.79	1.97
1624	4	3.46	2.41	3.07	3.36	2.34	3.01
1624	8	6.36	5.15	5.58	4.92	3.92	4.65
1624	13	8.00	6.27	8.99	4.5	3.77	5.80

Table 5.1: Results of the simulated solution of the four test systems

on the length of the message but also on the state of the receiving task. If the intended recipient is not ready to receive information the communication will be blocked until the receiving task reaches its synchronisation point. Whilst the transfer time of the message is proportional to the length of the message the delays due to blocking are non-deterministic and difficult to simulate. Accounting for the communication delays would make the simulation significantly more complicated. The results for the standard solution are also obtained by simulation and neither set of results includes the effects of communications. Given that the volume of communications involved in each method is roughly similar this allows a relative comparison of the performance of the two methods. The absolute performance of either method will be worse than that given in Table 5.1 due to the delays resulting from communication and scheduling. In a multiprocessor environment the communication delays will be affected by the target processor topology and communication routing protocols used. Omitting communication from the simulation means that it cannot be used to assess the suitability of different target architectures for the RP method.

It was stated earlier that if the RP method of solution is efficient the sum of its computation times should be approximately equal to the computation times of the best sequential method. Table 5.2 compares the total computation time of the RP solution with that of the best sequential solution for each of the test systems. In most cases the results are

similar, indicating the efficiency of the RP method. However there are a few differences which should be noted. In certain cases (*e.g.* the 118 node system using 8 processors) the total computation time of the Recursively Parallel method is greater than that of the best sequential method. Where there are differences they are not that large, ranging from 15% for the 118 node, 8 processor case down to 6% for the 734 node, 16 processor case. For the 1624 node systems it is noted that the total computation time of the RP method is always less than the total computation time of the best sequential method, by up to 3%. All these discrepancies are due to the fact that the data structures used in the two solution methods are different. The best sequential method uses a linked list data structure which optimises storage and processing for a sequential solution. The RP method uses data structures that mimic the data structures which would be used in a parallel implementation. This introduces extra overhead into the solution and its effect is more noticeable in the solution of smaller systems. The RP method uses a hybrid storage scheme so that some of the elements in the dense lower right corner of the matrix are stored in arrays. When the lower right corner becomes densely populated, array storage is more efficient and leads to faster processing. The effect is more noticeable for larger systems and this explains why the total times for the 1624 node RP solution are less than the best sequential times. The overhead introduced into the simulated RP solution by the more complex data structures still exists but its effect is counteracted by a decrease in execution time due to array storage. This effect probably occurs in some of the smaller systems but here it is likely that the reduction in execution time produced by hybrid storage in not sufficient to counteract the adverse effects of the overhead introduced by the more complex data structures. Hybrid storage is not implemented in the best sequential solution but the fact that it reduces the execution time of the more complex RP method to below that of the sequential method provides further evidence to support the claim that hybrid storage may play a significant part in improving existing sequential methods.

One of the main causes of inefficiency in a parallel program is the overhead introduced by interprocessor communication. If the communication overheads introduced into the parallel implementation can be kept to a minimum the performance achieved will be similar to that predicted by Table 5.1. Low communication overheads will lead to near-unity serial efficiency. As Tylavsky et al. [50] observe, the serial efficiency of a parallel algorithm relates to its scalability. An algorithm which has a unity serial efficiency will display increasing speed-up as the number of processors used to execute the algorithm increases. An efficiency

System	CPU's	Recursively Parallel Total Computation Time	Best Sequential Computation Time
118	2	13.63	13.24
118	4	14.47	13.24
118	8	15.32	13.24
629	2	80.07	76.49
629	4	82.43	76.49
629	8	81.91	76.49
629	16	82.00	76.49
734	2	95.87	94.89
734	4	99.55	94.89
734	8	99.82	94.89
734	14	100.45	94.89
1624	2	273.04	275.40
1624	4	274.32	275.40
1624	8	268.08	275.40
1624	13	269.19	275.40

Table 5.2: Simulated RP solution vs best sequential solution

greater than unity indicates that the speed-up will saturate as the number of processors increases and may eventually fall off. The greater the serial efficiency, the quicker the speed-up will saturate. Tylavsky et al. note that an efficiency of less than about 4 or 5 is needed in order to achieve a speed-up greater than unity.

5.4 Summary

The previous chapter discussed existing parallel methods for solving systems of linear equations. This chapter has considered how those methods can be modified to create a parallel solution which has a better performance. The increase in performance is achieved through the use of a particular coefficient matrix structure, the Recursive Bordered Block Diagonal Form. This results from constraining the subnetwork interconnections so that the network graph has a tree-like structure. Within the RBBDF matrix there are regions of independence and all subnetworks contained in these regions may be processed in parallel. Several independent regions exist in the matrix and solution involves sequential execution of several parallel phases.

A simulation of the method has been implemented and the results for the solution of four test systems have been presented. The simulation results support all the claims made for

the RP method and show that it offers better speed-up performance than standard methods in all phases of the solution. As the total amount of computation involved is the same as in the best sequential method it is possible to create a highly efficient parallel implementation if the communication overheads can be kept to a minimum.

The Recursively Parallel method is not a revolutionary new algorithm for the solution of linear equations. It is simply a restructuring of the problem to allow existing methods to exploit more of the potential parallelism in the problem. This restructuring is achieved by constraining the topology of the network interconnections to a particular form. Despite the constraints, this form of interconnection is very flexible and should allow the solution of any real system.

Chapter 6

Issues of Parallel Implementation

6.1 Introduction

In the previous chapter the general algorithm of the Recursively Parallel solution method was discussed without reference to how it might be implemented on a MIMD computer. Chapter 4 considered the partitioning of the problem and argued the case for obtaining an equal division of work through a careful analysis of the system to be solved. This chapter considers some aspects of implementing the Recursively Parallel solution on a MIMD machine, in particular on an array of INMOS Transputers. These issues relate to the algorithm (*e.g.* number of tasks, data structures, methods of communication *etc.*) and to the architecture of the solution, both the physical architecture of the target machine and the software architecture of the interconnected tasks and their placement on the available processors.

The INMOS Transputer was designed to provide an implementation of Hoare's Communicating Sequential Processes (CSP) paradigm of parallel computation [92, 93]. CSP considers a parallel program to be made of a collection of independently executing tasks, where each task is an autonomous unit of sequential code which executes in the same manner as a normal sequential program. Tasks synchronise their actions and share data through explicit interprocess communications along virtual, unidirectional communication channels. Many parallel languages (*e.g.* all the INMOS parallel languages, Ada *etc.*) are based on the CSP paradigm and the comments made in this chapter concerning an INMOS 'C' implementation apply equally well to implementations in other CSP-based languages. Certain specific issues of a Transputer implementation are considered but it should be noted that

the Transputer has only been used as a cheap testbed on which the methodologies may be proven. Having verified the methods on a Transputer system it is easy to generalize them for any MIMD/CSP implementation and also implementations on distributed computer networks (*e.g.* workstation clusters [85]).

6.2 Algorithmic Issues

6.2.1 Program Structure and Task Design

The design of a parallel program requires careful consideration of the algorithm it embodies. The algorithm must be decomposed into individual tasks in a manner which maximizes the speed-ups that can be obtained. Particularly important is matching the number of tasks to the available processing hardware. If there are more tasks than there are processors the performance can be degraded due to overheads introduced by multitasking. Hence it is important to choose the right grain of parallelism for the implementation.

The Recursively Parallel program implemented in this research project is based upon Zollenkopf's Bifactorisation method of triangular decomposition. Within this method three distinct operations can be identified - factorisation of the coefficient matrix, multiplication of left hand factors and multiplication of right hand factors. For the RP solution of a given system, p distinct subnetworks will be created within the network. The three basic operations must be applied to each subnetwork and it is perhaps easiest to consider the parallel program to be made of $3p$ tasks; p factorisation tasks, p left factor multiplication tasks and p right factor multiplication tasks. One each of the factorisation, left and right multiplication tasks would be associated with each subnetwork. This approach can be seen to be inefficient when the bifactorisation process is considered as a whole. First the factorisation operation is performed followed by left factor multiplication and then right factor multiplication. Left factor multiplication cannot commence until factorisation completes and right factor multiplication cannot commence until left factor multiplication completes. Applying this to an individual subnetwork we have the same sequence of operations. As neither of the multiplications can occur at the same time as factorisation both multiplication tasks would be idle whilst the subnetwork is being factorised. A similar argument shows that at any point in time only one of the three tasks associated with a given subnetwork is active, the other two being idle (Figure 6.1(a)). A single task can be created which performs the three operations one after another on a given subnetwork. Now only p tasks are required to

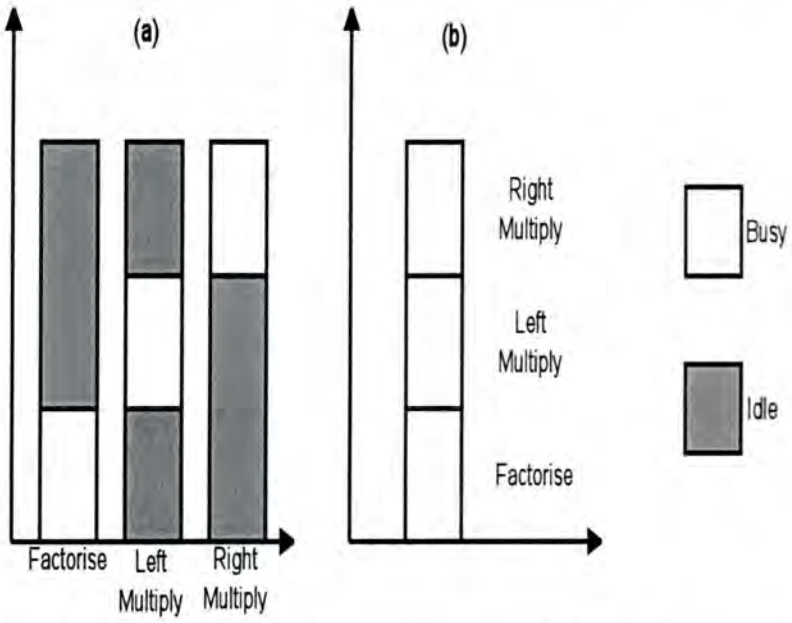


Figure 6.1: Task structures with different granularity a) fine grain b) coarse grain

process the p subnetworks and each of these tasks is busy at all points in time. Referring to the inherent sequentiality of the bifactorisation process reveals that there is little point using $3p$ tasks when p tasks will suffice. From the programmer's point of view it is much simpler to manage only p tasks. Figure 6.1(b) shows the structure of a task in the p task solution.

Having determined the number and structure of the tasks it is necessary to connect up the tasks so that they can cooperate and share data. For the remainder of this discussion it is assumed that task T_i, where $i = 1 \ldots p$, is responsible for processing the ith subnetwork. Consider Figure 6.2 which shows the structure of the coefficient matrix for a system with 4 main subnetworks and 3 minor ones. The analogy of blocks in the matrix diagram being elements in a 7×7 matrix with the same structure is used. In factorising the first row of the matrix updates will be made to elements (5,5), (5,7) and (7,7) as defined by the equations of Section 2.4.4. In terms of the subnetworks in the RP solution, processing of the first subnetwork leads to values in the fifth and seventh subnetworks being modified. As the three subnetworks involved are processed by separate tasks these tasks must cooperate and share the information between themselves. In other words T_1 must send information on modifications to T_5 and T_7. An analysis of the communications required for the other subnetworks in all three stages of the Recursively Parallel solution yields the dataflow diagram of Figure 6.2. Circles in the diagram represent tasks and the arcs linking them show the communications between the various tasks. The flow of data is shown by the

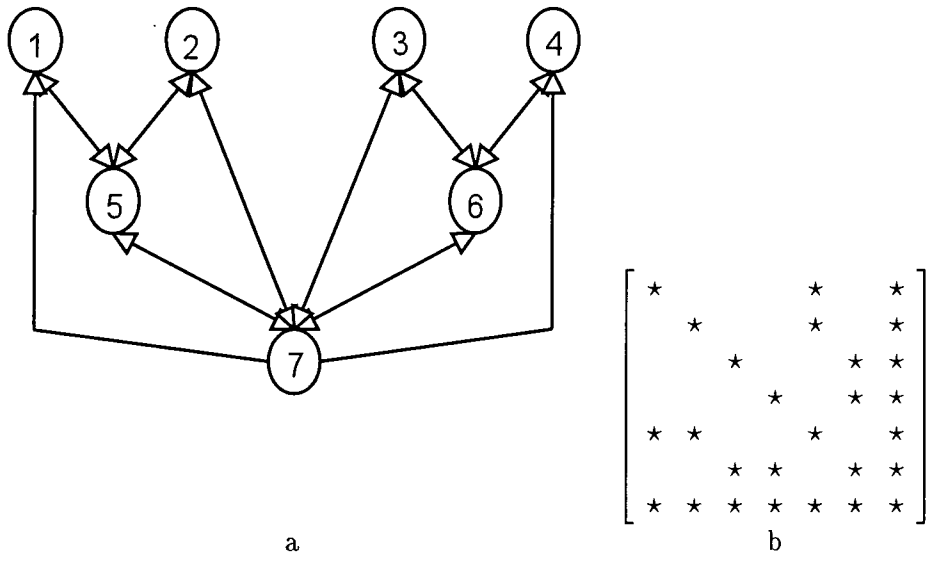


Figure 6.2: Intertask communications in a 7 subnetwork solution a) and corresponding coefficient matrix structure b)

arrowheads.

If one task is used for each subnetwork it is obvious that the tasks are connected in a tree form. Figure 6.3 shows the task intercommunications that are required for solutions with 3 and 15 subnetworks. Observe that no connection is ever required between tasks in the same level of the tree. These observations will be used in Section 6.2.3 to realize a reduction in the number of intertask communications.

The basic task structure of Figure 6.1(b) can be refined to take account of the communications that occur between tasks in the program. Figure 6.2 shows how data moves in two directions through the tree. The factorisation and left factor multiplication operations both produce data that flows down the tree from the leaves towards the root. The right factor multiplication operation generates data which flows in the opposite direction back towards the leaves. After factorisation a task T_i must send any data pertaining to the modification of other subnetworks to the tasks responsible for processing those subnetworks. These tasks all lie below (*i.e.* toward the root) task T_i in the tree. Modifications to other subnetworks resulting from left multiplication with subnetwork i again requires task T_i to send data to tasks lower down the tree. Modifications resulting from right multiplication require task T_i to send data to tasks higher up the tree (*i.e.* toward the leaves). When a task is the recipient of any of these communications it must add the modifications to its data before it performs the relevant operations on its subnetwork. The refined task structure is shown in

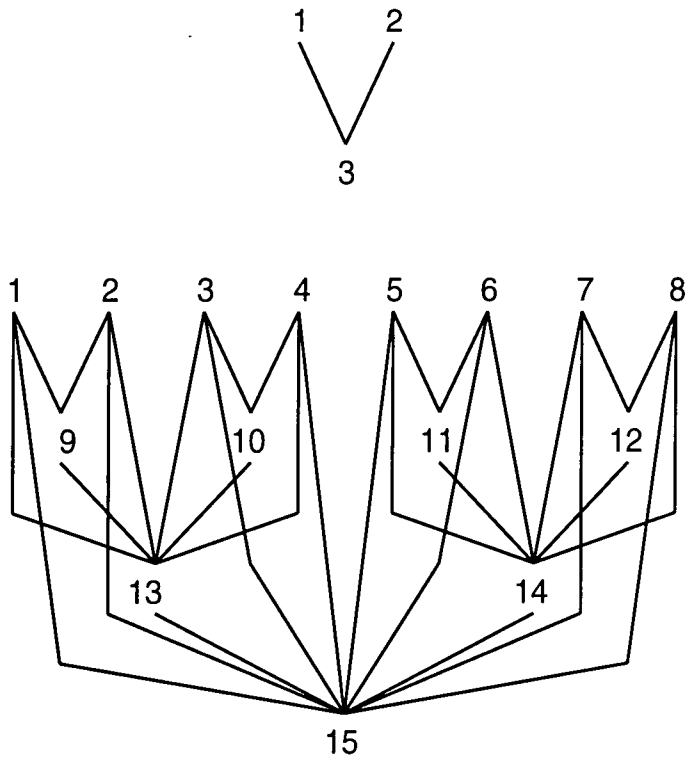


Figure 6.3: Intertask communications in 3 and 15 subnetwork solutions

Figure 6.4. Note that stages 2 and 12 do not occur for tasks at the leaves of the tree and that stages 5, 9 and 10 do not occur for the task at the root of the tree.

One important issue not yet addressed is where do the tasks originally obtain their data from ? The collection of generic tasks which perform the computation of the solution can be viewed as a set of worker tasks in a supervisor / worker scenario (Figure 4.4). A supervisor task is needed to take control over the generic workers. This supervisor is responsible for passing the appropriate subnetwork data to each worker task. Stage 1 of the generic task is responsible for accepting the initial subnetwork data from the supervisor task. The supervisor is also responsible for gathering the results from each worker once computation is complete (stage 13) and assembling them into a single resultant vector. For the purposes of monitoring the performance of the Recursively Parallel method, the supervisor also synchronizes the start of computation within the set of workers. This is necessary to allow an accurate time to measured for the computation of the solution. Appendix G gives further consideration to the role of the supervisor in monitoring the performance of the computations.

```
1:   Receive initial subnetwork data from supervisor task
2:   Receive modification data from all connected tasks higher up the tree
3:   Modify subnetwork data
4:   Factorise the subnetwork
5:   Send modifications to all connected tasks lower down the tree
6:   Receive modifications from all connected tasks higher up the tree
7:   Modify right hand side vector data
8:   Left multiply with the subnetwork
9:   Send modifications to all connected tasks lower down the tree
10: Receive modifications from all connected tasks lower down the tree
11: Right multiply with the subnetwork
12: Send modifications to all connected tasks higher up the tree
13: Send results to supervisor task
```

Figure 6.4: The generic task of the Recursively Parallel solution

6.2.2 Data Storage and Data Structures

The worker task must store, modify and process the data associated with the subnetwork for which it is responsible and the choice of the correct data structures is crucial to the performance of the parallel program. For the solution of linear equations the data is the coefficient matrix and the known right hand side vector of the set of equations. Matrices are usually stored by computer as a two dimensional array of an appropriate type but sparse matrix techniques are a more efficient method of storing the large coefficient matrices associated with network problems. These techniques store only the non-zero matrix elements and these are held in linear linked lists. The known right hand side vector is stored as a one dimensional array as it is usually densely populated, particularly as computation draws to a close. In a parallel solution there is no longer a single coefficient matrix structure corresponding to a single network. Instead there are a number of subnetworks, which correspond to a number of submatrices of the coefficient matrix. Consideration has to be given as to what are the best methods of storing these submatrices and the portion of the right hand side vector associated with each subnetwork.

If a system is divided into m subnetworks to be solved using the Recursively Parallel approach then the coefficient matrix of the system will have Recursive Bordered Block Diagonal Form with $p = 2m-1$ 'r' shaped segments stacked along the diagonal, as in Figure 6.5. Each task requires the data held in a single 'r' shaped segment and is responsible for storing this segment. For example, the data associated with subnetwork 1 is contained in the shaded 'r' shape segment of Figure 6.5 and task T_1 must store this data in an appropriate

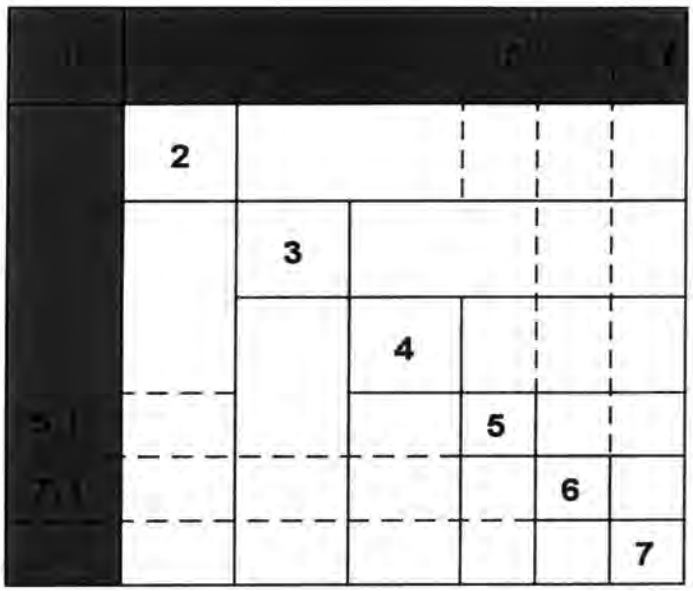


Figure 6.5: RBBDF matrix structure showing 'r' segments

data structure. Suppose that the ith 'r' segment begins at row a, extends to row b and that there are n rows in the matrix. If the traditional two dimensional array is used to store the 'r' segment for this task then a square array of the same height and width as the 'r' segment is required and this is extremely wasteful of memory, especially when the subnetworks of the first level of the tree are considered. Within the 'r' segment of these subnetworks there are only four possible regions where non-zero elements can be located (*e.g.* blocks 1,5 1,7 5,1 7,1). A more efficient storage scheme stores only these non-zeros and this is implemented through the use of a sparse matrix linked list representation. Under this scheme $(n-a)$ lists will be required for the given subnetwork. Most of these lists will contain few, if any, entries and substantial savings on memory can be realized. Further savings result from examining the nature of the coefficient matrices for power system problems. As Chapter 2 has shown, the coefficient matrix is often symmetric and if this is the case it is not necessary to replicate data by storing both arms of the 'r' segment. A single arm contains all the information held within the 'r' segment and only $(b-a)$ linked lists are now required. Figure 6.6 shows the portion of the coefficient matrix stored in the linked list data structures by task T_1. This storage scheme proves to be the most efficient for storing the major subnetwork. Inspecting the off-diagonal blocks corresponding to the minor subnetworks reveals a different picture. These blocks contain many more non-zeros than zeros and in moving from the top left corner of the matrix to the bottom right corner the density of the off-diagonal blocks increases. Most of the off-diagonal blocks corresponding to the minor (cutset) subnetworks are in fact

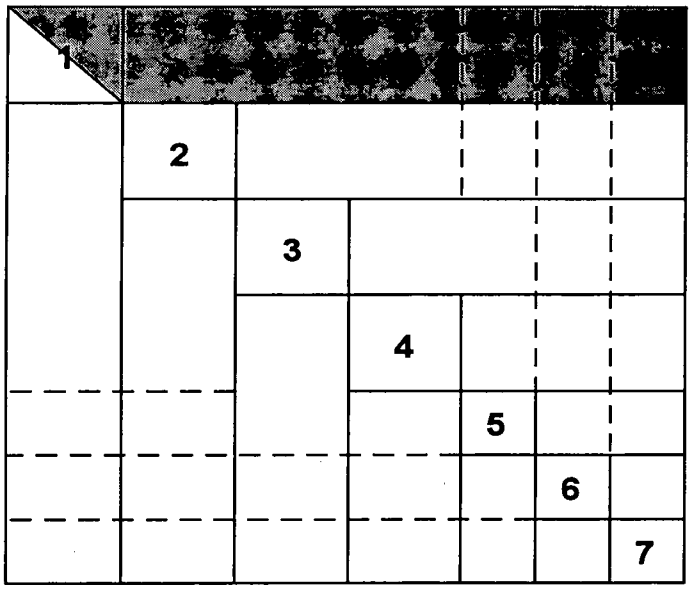


Figure 6.6: Portion of the coefficient matrix stored by a single task

full of non-zero entries. Brameller et. al [2] observe that there is a threshold beyond which it is inefficient to use sparse matrix techniques and in the minor subnetwork off-diagonal blocks this threshold is easily exceeded. The empty spaces between the blocks in these regions of the matrix are small, accounting for a only a small percentage of the total length of the row to the right of the diagonal. It is more efficient to process the minor subnetworks as two dimensional arrays rather than as sparse matrix linked lists. Unfortunately the point at which the density exceeds the threshold value does not usually coincide with the boundary between major and minor subnetworks. The changeover point differs from system to system and it often found that it is more efficient to treat the first few minor subnetworks using sparse matrix techniques, reserving the use of array storage for the remaining minor subnetworks. Figure 6.7 shows the parts of the coefficient matrix stored in the Recursively Parallel solution and the type of storage scheme used by each task/subnetwork.

When the optimal ordering routine is applied prior to solution, a simulation of the factorisation process is performed to determine the effect of fill-ins. This simulation can be adapted so that it assesses the density of the coefficient matrix after the simulated factorisation is complete. Starting with the element at the bottom right corner of the matrix, the algorithm works back up the diagonal and as it goes it examines the square region below and to the right of the diagonal (Figure 6.8). The number of non-zeros in this region is counted and the total number of elements enclosed by the region is calculated. If the density of the region exceeds the sparsity threshold the search continues moving back

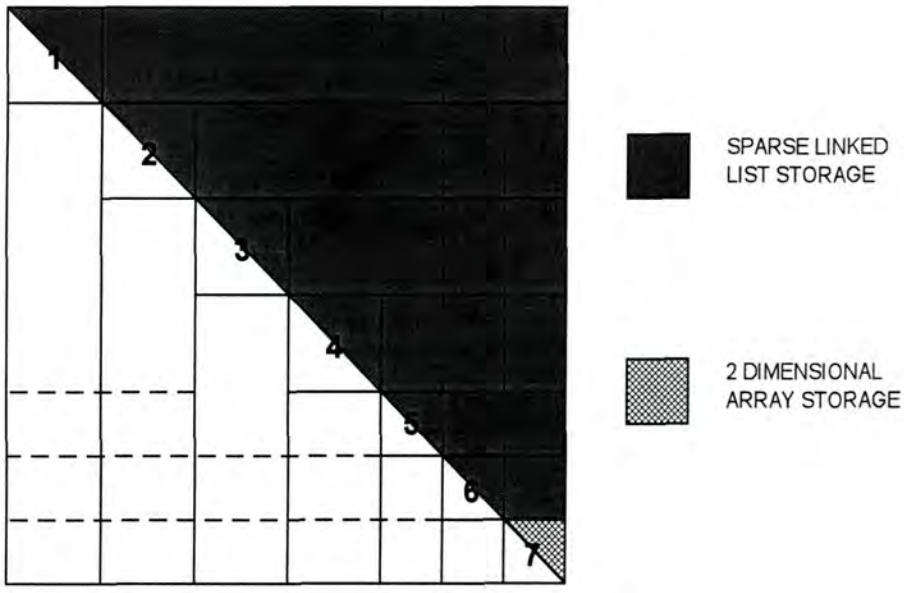


Figure 6.7: Storage techniques used by the Recursively Parallel method

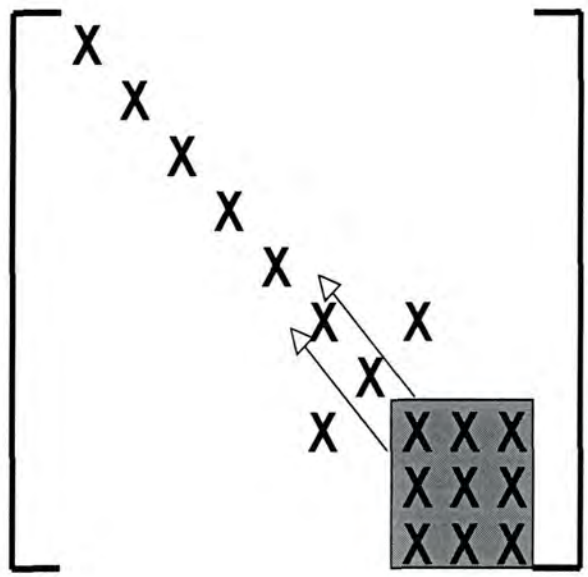


Figure 6.8: Assessing the density of the coefficient matrix

System	Subnetworks	Speed-up with original storage	Speed-up with hybrid storage
118	3	1.42	1.42
118	7	1.93	2.08
118	15	2.12	2.47
629	3	2.18	2.23
629	7	2.52	2.70
629	15	3.71	4.04
734	3	2.46	2.46
734	7	3.59	4.10
734	15	4.56	5.83
734	31	4.71	5.98
1624	3	2.64	2.64
1624	7	4.00	4.15
1624	15	6.54	7.55
1624	31	7.26	9.38

Table 6.1: The effect of storage scheme on speed-up

up the diagonal until the density of the region falls just below the threshold. The location of this point allows the data structures of the Recursively Parallel program to be tailored to give the most efficient storage and processing of each individual system. Everything above and to the left of the changeover point is most efficiently processed using sparse matrix techniques whilst all rows below and to the right are most efficiently dealt with when stored in two dimensional arrays. Performing this sort of density test on a system prior to processing determines which subnetworks should store their submatrix in linked lists and which should store it in arrays. Future work may allow the changeover point to occur *within* a subnetwork but for the present the changeover point is constrained to lie on a subnetwork boundary. The program implemented for this project allows any of the minor subnetworks, except the last, to use either sparse matrix or array storage as the need requires. The last subnetwork is always processed using arrays and the major subnetworks employ sparse matrix storage techniques.

The benefits of this hybrid storage scheme are evident in the speed-up results. Table 6.1 shows the factorisation speed-ups for a number of systems processed both with and without hybrid storage. Figure 6.9 shows the variation in speed-up for the 1624 node US power system as the changeover between sparse and array storage is moved between the minor subnetworks. Changing the number of subnetworks which are stored using array storage has a dramatic effect on the speed-up. Notice that it is possible to vary the factorisation speed-

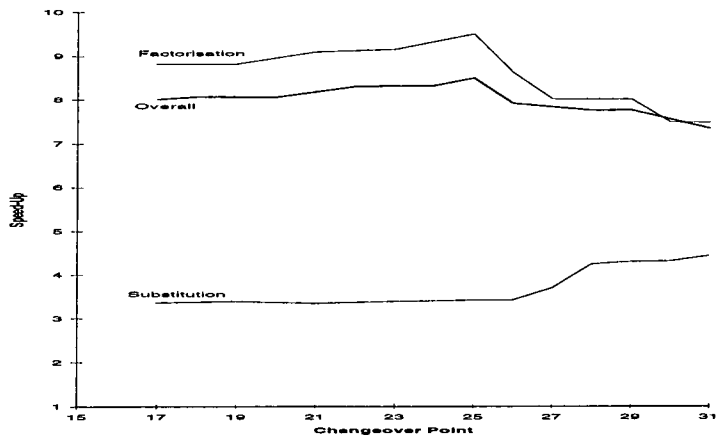


Figure 6.9: Variation of speed-up with location of changeover point in hybrid storage, for US 1624 node system

up between 7.3 and 9.5 simply by altering the point at which dense storage is introduced. If little use is made of array storage (changeover point to the right of the graph) then the factorisation speed-up is poor. Increasing the amount of array storage (changeover point moving towards the left of the graph) has the effect of increasing the speed-up, although there is a certain point beyond which it is not efficient to use array storage as the factorisation speed-up begins to decrease again. Examining a number of graphs like this has revealed that there is often a clearly defined optimum location for the changeover point which results in the maximum factorisation speed-up. For the system shown in Figure 6.9 the maximum factorisation speed-up is obtained when subnetworks 25 to 31 are stored using array storage. The effect of hybrid storage on the substitution speed-up is somewhat different. It can be seen from the graph that the substitution speed-up can be varied between 3.3 and 4.3 simply by altering the point at which array storage is first introduced. The graph shows that increasing the amount of array storage actually reduces the substitution speed-up. Again, an examination of a number of graphs like this has revealed that maximum substitution speed-up is obtained when only the last subnetwork is stored using array methods. If any other subnetworks are stored using arrays the speed-up rapidly drops off. The use of hybrid storage has different effects on the performance of factorisation and substitution. Using a significant amount of array storage improves the performance of factorisation as the factorisation algorithm performs many update operations to the lower right of the matrix. If this

is region is stored using linked lists then each update requires a search through the linked list for the relevant row to find the desired element for updating. If this region is stored using arrays then it is possible to jump directly to the desired element, thus removing the need for time-consuming searches. The extra overhead of examining zero elements during the factorisation of this region is countered by the significant reduction in processing time produced by the elimination of a large number of linked list searches. The situation is somewhat different with substitution where increasing the use of array storage has a detrimental effect on performance. There are two effects which contribute to this phenomenon. Firstly, substitution simply involves the multiplication of matrix rows or columns by a vector. The algorithm for performing this operation simply scans through the stored data and performs an element by element arithmetic operations. No lengthy searches are involved when linked lists are used so the use of array storage would not eliminate any list searches. However, when array storage is used the multiplication algorithm must perform element by element arithmetic on all elements of the array, even if they are zero. It is possible to reduce the amount of work by first examining the current element to check its value. Only if the element is non-zero is any arithmetic operation involving that element performed: However the examination of each element to determine its value still adds a computational overhead. Linked list storage is more efficient as no zero elements are stored and no examination of element values needs to be performed. All elements in the list must automatically take part in multiplication and the algorithm simply performs element by element arithmetic using all the values in the list. The computational overhead associated with the multiplication using linked lists is less than that using arrays, making it more efficient for substitution to store as much of the coefficient as possible using linked lists. The last subnetwork in the matrix is always stored using arrays as this subnetwork is always fully populated. It makes little difference to the amount of computation involved whether linked list or array storage is used.

The effect illustrated by Figure 6.9 has profound implications for the parallel solution. It is possible to maximize speed-up by tailoring the storage scheme to the characteristics of either the factorisation or substitution phase, but not both. If the storage strategy is chosen to maximize substitution speed-up then factorisation speed-up is still reasonable. However if the situation is reversed and the storage strategy is chosen to maximize factorisation speed-up then the substitution speed-up is poor. The ideal requirement is for a storage scheme which maximizes both speed-ups, but as Figure 6.9 shows, this is clearly

not possible. In power system simulation the factorisation operation is only required when the topology of the network changes. This occurs infrequently and it is perhaps possible to use the optimum storage for factorisation when factorising and then transform the data storage so that data is then stored in the optimum manner for substitution. This would maximize the performance of both the factorisation and substitution operations but would introduce a large overhead when the network topology changes. An alternative is to use two data storage mechanisms. The solution implemented on the Transputer saves on memory requirements by overwriting the original coefficient matrix with the factored matrix as factorisation progresses. If memory size is not a limitation then it would be possible to store the coefficient matrix in the optimum form for factorisation and to store the factored matrix separately in the optimum form for substitution. No transformation overheads are introduced when the topology changes and both factorisation and substitution would be able to achieve maximum performance. Some small overhead would be introduced into the factorisation computations.

It is interesting to note that the hybrid storage scheme can also be used to improve the performance of sequential solution techniques. With a sequential solution the coefficient matrix does not need to be arranged into BBDF or RBBDF. Optimal ordering is applied to minimize the amount of fill-ins that occur. If the filled matrix is examined the increase in density in moving from the top left to the bottom right of the matrix is again observed. Usually all the matrix rows are stored in sparse linked lists but in the lower right corner of the matrix many of the rows may contain only non-zero elements. Once again the threshold of efficiency for sparse matrix storage is exceeded and it would be more efficient to store these rows in conventional arrays. Search operations would be performed faster with these rows stored as arrays and this would decrease the execution time of the sequential solution. This theory has not been tested in practice and the implementation of hybrid storage in a sequential solution has been left as an item of further work. However the argument presented for the sequential solution is identical to that for the parallel case. Having observed the beneficial effect of hybrid storage on the performance of the parallel solution it is reasonable to assume that a similar effect would be observed if hybrid storage were to be implemented for sequential solutions.

Given that it is not necessary for each task to store the entire coefficient matrix, perhaps similar savings can be achieved in storing the right hand side vector by only storing the relevant portion of the vector. Unfortunately this does not turn out to be the case, as can

be seen by considering the multiplication stages of the bifactorisation algorithm. Taking left multiplication as the example, consider the multiplication of the right hand side vector by the first left factor, $L^{(1)}$. Assume $L^{(1)}$ and the right hand side vector have the form shown below.

$$\mathbf{L^{(1)}b} = \begin{bmatrix} \times & & & & & & \\ & \times & & & & & \\ & & \times & & & & \\ & & & \times & & & \\ \times & & & & \times & & \\ & & & & & \times & \\ \times & & & & & & \times \end{bmatrix} \begin{bmatrix} o \\ \times \\ \times \\ \times \\ o \\ \times \\ o \end{bmatrix} \tag{6.1}$$

Performing this multiplication results in a new vector ,b', which differs from b in elements 1, 5 and 7. Assuming that the resultant vector is generated by successively overwriting the original right hand side vector, b, left multiplication by the i^{th} subnetwork could possibly alter elements a to n of the right hand side vector where a is the first node in the i^{th} subnetwork. Thus it is necessary to for the i^{th} task to store elements a to n of the right hand side vector otherwise the result of left factor multiplication within that task will be incorrect. A similar analysis of right factor multiplication shows that the same range of RHS vector elements must be stored to obtain correct results from right multiplication. A simple examination for the RBBDF matrix structure shows that for a given subnetwork only certain elements of the RHS vector are modified and others are never referenced. For example, multiplication by subnetwork 1 only ever references elements in blocks 1, 5 and 7 of the RHS vector. An efficient scheme for storing only the RHS vector blocks required by each task could be devised but the saving on memory does not really warrant the extra overhead and complexity involved in managing such a structure. The largest test system used in this research project required less than 12 kB of memory to store the entire right hand side vector. Given the amount of memory available in modern computing systems it is simplest for each task to store all the nodes a through to n of the RHS vector even though some of them are never referenced.

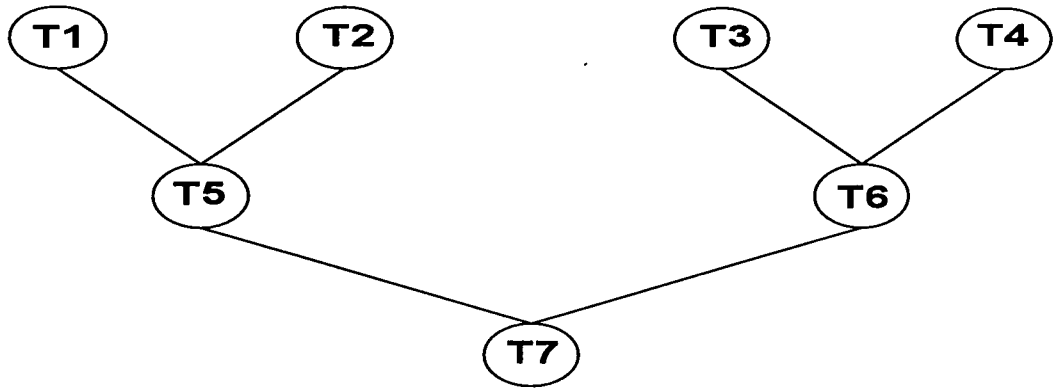


Figure 6.10: Modified intertask communications for a 7 subnetwork system

6.2.3 Reducing the Communication Overhead

Although the basic program structure and intertask communications have been defined there are several refinements that can be made which significantly improve the performance of the parallel program. These refinements are based on aggregating the intertask communications.

The first improvement aggregates communication messages so as to reduce the total number of messages passed between the tasks in the system. Consider the dataflow diagram of Figure 6.2. Suppose that task T_1 has performed its factorisation operation. This results in data which has to be passed to T_5 and T_7. The transmission of data to T_5 is critical to the next stage of processing and this communication must be performed first. When this finishes data can be sent explicitly to task T_7 as this communication is not critical to the next stage of processing. In fact T_7 does not require the data from T_1 until T_5 has completed its factorisation operation. The use of a suitable store-and-forward aggregation scheme eliminates the need for the explicit communication between T_1 and T_7. If T_1 aggregates the data it sends to T_5 and T_7 into a single message sent to T_5, T_5 can use the data pertaining to itself and simply ignore the data relating to T_7. When T_5 has factorised its submatrix it adds the data it created for T_7 to the stored data generated by T_1 and sends the single message to T_7. The explicit communication between T_1 and T_7 is thus avoided. Applying this technique across the whole task graph reduces the intertask communications shown in Figure 6.2 to those of Figure 6.10. The worker tasks which make up the parallel program are connected in a simple binary tree structure. The use of this aggregation technique eliminates eight communications in a seven subnetwork solution and forty communications in a fifteen subnetwork solution resulting in a significant improvement in efficiency.

The second improvement is at a much lower level and reduces communication times by

decreasing the length of the intertask messages using a data grouping technique. This technique is described fully in Appendix F but it basically operates by minimizing the amount of addressing information needed to uniquely identify an element within the coefficient matrix by making use of implicit information about the matrix structure.

6.3 Architectural Issues

In examining the architectural issues concerning parallel program design there are two aspects which must be considered. The software architecture describes the tasks which make up the parallel program and how they are interconnected to give the best performance. The hardware architecture is concerned with the physical processors of the target machine. The way these processors are connected and the way in which data is routed between them can have an effect on the performance of the parallel program.

6.3.1 The Software Architecture

A parallel program is made up of a set of autonomous tasks which cooperate through synchronizing their actions and sharing data using explicit intertask communications [92, 94]. These tasks must be assigned to the available processors. Given a parallel program consisting of n tasks, these must be placed on the p processors available and two scenarios can be envisaged

- $n > p$ More than one task must be assigned to each processor and support for a multitasking environment is required to execute the tasks simultaneously
- $n \leq p$ Each task in the program can be assigned to its own processor

Whilst the latter scenario provides the easiest implementation it may not be the best solution. Assigning more than one task to each processor is often most economic as it requires fewer processors. The efficiency of a parallel program, E, is given by

$$E = \frac{S(n)}{n} \tag{6.2}$$

and (6.2) clearly shows that program efficiency is a direct trade-off between the number of processors used and the speed-up obtained. The more processors that are thrown at the problem to achieve a desired speed-up the less efficient that solution becomes. Programs

which assign a number of tasks to each processor generally achieve a higher efficiency at the expense of lower speed-up.

A simple examination of the structure of the Recursively Parallel program quickly identifies the optimum number of processors required. Taking a seven subnetwork (four major and 3 minor subnetworks) system as an example, the program structure is as shown in Figure 5.9. Operations with the same subscript are all parts of the same task (*e.g.* task T_1 comprises the operations F_1, L_1 and R_1). As the factorisation operation is the most computationally intensive part of the solution only the factorisation operations of Figure 5.9 will be considered in the following argument. At level 1 there are four concurrently executing factorisation operations and the fastest computation occurs when each operation is assigned to its own processor. Hence level 1 requires four processors. Level 2 has only two concurrently executing operations requiring only two processors whilst the final operation in level 3 requires only a single processor. By virtue of the way in which the program operates, all the first level operations (*i.e.* F_1, F_2, F_3, F_4) must have completed before the second level operations (*i.e.* $F_5, F6$) begin. The four processors which executed $F_1 \ldots F_4$ are thus idle at the end of level 1 and two of them can be used to perform the operations of level 2. Similarly the two processors responsible for executing level 2 operations are idle at the end of level 2 and one of them can process level 3. Consequently the maximum number of processors required is the same as the number of concurrent operations in the first level and a possible task to processor allocation is shown in Figure 6.11(a). This allocation strategy results from the fact that operations $F_1 \ldots F_7$ are constituent parts of the tasks $T_1 \ldots T_7$. Consider the allocation strategy of Figure 6.11(a) from the point of view of the factorisation operations and recall the previous discussion of load balancing and recall that maximum speed-up only occurs when all of the processors are busy all the time. This strategy results in processors P_3 and P_4 being unused for more than half of the total computation time and this is wasteful of processing resources. The efficiency of the task allocation can be improved by moving task T_7 from processor P_1 onto one of the 'spare' processors P_3 or P_4. The modified strategy is shown in Figure 6.11(b).

The allocation of tasks to processors is often considered using two acyclic directed graphs [19, 83]. One graph is used to represent the processors of the target machine and their interconnection. The second graph is the task graph and this describes the interconnection between the tasks which make up the program. The task graph nodes are often annotated with computation and communication requirements. Figure 6.12 gives an example of these

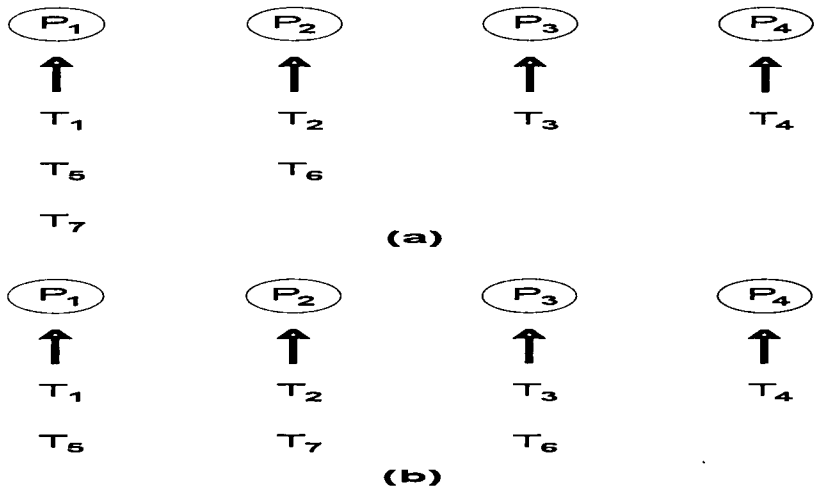


Figure 6.11: A many to one task allocation strategy a) initial task allocation b) improved task allocation

two graphs The task allocation process then becomes one of mapping the task graph to the target machine graph in the way that minimizes computation and communication requirements. Berman [83] considers the task allocation process as a 'contraction' of the task graph to fit the target machine graph. To describe the task allocation strategy for a general Recursively Parallel solution it is useful to think of the contraction of the factorisation operations of Figure 5.9. The first step of the process is to squash levels 2 and below into a single level by superimposing the levels onto one another such that they become interleaved. Vertically adjacent tasks of level 1 and the new level 2 are then assigned to the same processor. The contraction for a 15 task system is shown in Figure 6.13. Note that there are never more than two tasks assigned to each processor but there is always one processor which hosts a single task. As there is one task for each subnetwork in the system it is easy to verify that

$$n_p = \frac{n+1}{2} \tag{6.3}$$

are required for the RP solution of a system with n subnetworks.

This argument has so far ignored the effects of left and right factor multiplication on the task allocation strategy. Returning to the allocation strategy of Figure 6.11(b) we see tasks T_1 and T_5 sharing the same processor P_1. This allocation was derived on the assumption that F_1 must complete before F_5 starts and by making these two tasks share the same processor each will effectively have the whole processor at its disposal. When F_5 is being executed F_1 must have completed and F_5 will be the only operation available for processing on processor P_1. Recalling that the task T_i consists of the operations $\{F_i, L_i, R_i\}$ executed

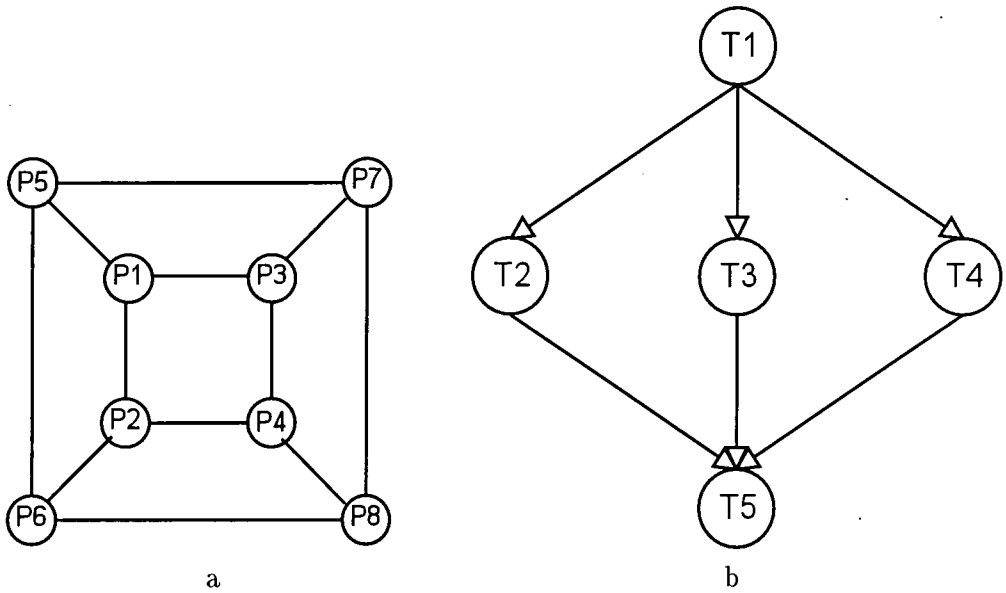


Figure 6.12: The acyclic graphs used in determining task allocations a) the hardware graph showing interconnection of processors b) the task graph showing dataflow between tasks

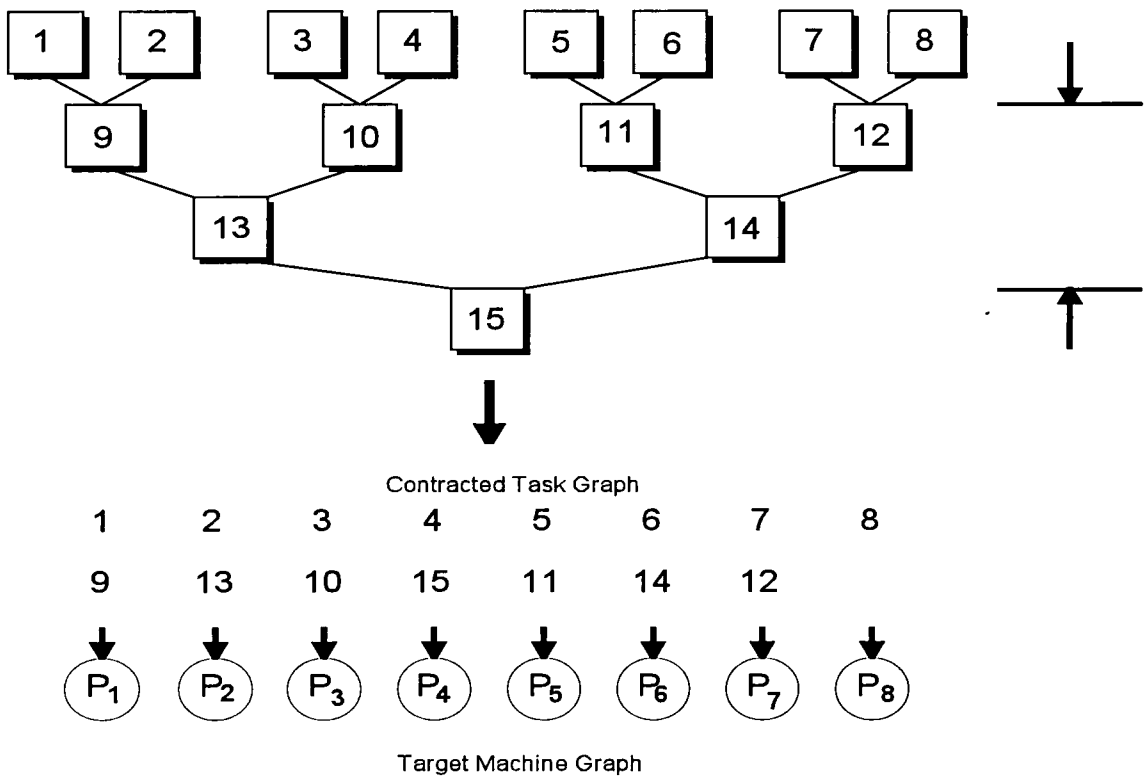


Figure 6.13: Contraction mapping for Recursively Parallel task allocation

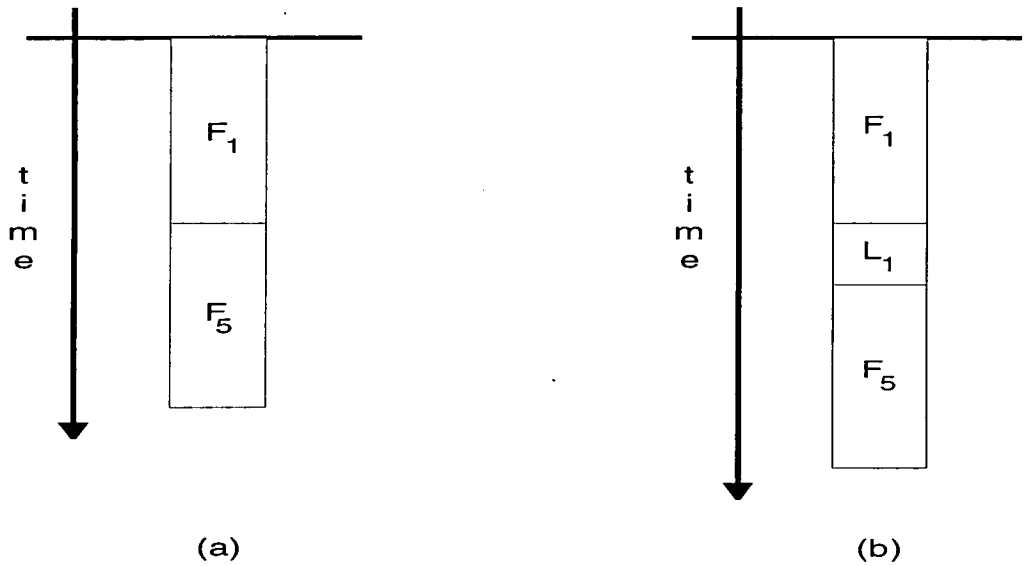


Figure 6.14: Task execution, showing the effect of multitasking (a) Naive view of operation execution (b) true execution of individual operations

in sequence and referring to level 2 in Figure 5.9 reveals that this assumption is invalid. When task T_1 completes the operation F_1 it immediately commences the operation L_1, and this requires processing at the same time as F_5. As both T_1 and T_5 reside on the same processor P_1 resorts to multitasking to timeslice execution between $T_1(L_1)$ and $T_5(F_5)$. Now neither task has the whole processor at its disposal and the efficiency of the allocation strategy is reduced. However the left multiplication operation L_i is not as computationally intensive as the factorisation F_k and as a result it only executes for a short time compared to F_k. Once L_i is completed task T_i suspends itself waiting for data to be passed to it following the operation R_i. Now F_k has the whole processor at its disposal and it executes F_k to completion without interruption, as shown in Figure 6.14(b). The effect of the left multiplication operations is to increase the overall execution time of the parallel program and reduce its efficiency. For the seven subnetwork system of Figure 5.9 execution time is increased by

$$MAX|L_1 + L_5|, |L_2 + L_5|, |L_3 + L_6|, |L_4 + L_6| \tag{6.4}$$

A method exists for improving the efficiency of the allocation strategy and this relies upon eliminating the contention between L_i and F_k. The contention arises because these two operations wish to execute concurrently. If it were possible to prevent F_k from executing until L_i completes then the contention could be avoided. Figure 5.9 shows the prerequisite of F_5's operation is that both F_1 and F_2 have completed. If F_2 takes much longer to

System Size	Subnetworks	Balanced Load Execution Time (ms)	S(n)	E(n)	Imbalanced Load Execution Time (ms)	S(n)	E(n)
118	3	27.24	1.28	0.64	27.24	1.28	0.64
118	7	18.88	1.85	0.46	20.22	1.72	0.43
118	15	14.53	2.40	0.30	14.53	2.40	0.30
629	3	112.81	2.00	1.00	110.06	2.05	1.03
629	7	88.38	2.55	0.64	83.33	2.71	0.68
629	15	60.22	3.75	0.47	55.1	4.09	0.51
734	3	129.97	2.16	1.08	127.60	2.20	1.10
734	7	77.06	3.64	0.91	70.27	3.99	1.00
734	15	54.46	5.15	0.64	50.37	5.57	0.70

Table 6.2: Effect of load imbalance on performance

execute than F_1, task T_1 can perform operation L_1 whilst $T_5(F_5)$ is suspended awaiting the completion of F_2. When F_2 completes T_5 commences its factorisation operation F_5 and has the whole processor at its disposal. For this method to work

$$t(F_2) > t(F_1) + t(L_1) \tag{6.5}$$

where $t(x)$ is the time taken to perform operation x. Obviously if $t(F_2)$ is much greater than $t(F_1) + t(L_1)$ processor P_1 will waste time in idle wait states, thus reducing the efficiency. Optimum performance occurs when

$$t(F_2) = t(F_1) + t(L_1) \tag{6.6}$$

To make this method work it is necessary to be able to adjust the execution time of each of the tasks and their constituent operations. The amount of time taken to complete one of the three fundamental operations within a task is directly proportional to the amount of computational work to be performed. Modifying the execution time of the operations can be achieved by altering the load balancing strategy and assigning a greater proportion of the workload to T_2 than to T_1, thus satisfying the inequality of (6.5). For the test systems used here it has been found that significant performance benefits result from increasing the workload of some tasks by up to 25% and similarly decreasing the loading of other tasks. Table 6.2 shows the benefits of this approach over the equally loaded case. Whilst it may appear that there is an imbalanced load it must be remembered that it is only the workload of each *task* that is imbalanced. The loading on each processor will in fact be balanced.

This technique of imbalancing the load is essentially a *latency hiding* method. Latency hiding [95] is a way of using the available multitasking facilities to keep processors busy during communication events. A program task will block and wait on the arrival of a message it requires if the sender is not ready to send, thus placing the processor into an idle state. If other program tasks are available for execution the processor may continue computing whilst the first task is awaiting the arrival of its message. In this manner the latency introduced into the first task by the necessity of waiting for a communication is hidden by the processing of other tasks. In the technique described above, increasing the execution time of F_2 allows task T_1 to complete F_1 and L_1. Once L_1 is complete the task blocks and waits until it can send a message to a subsequent task. This now allows task T_5, which resides on the same processor, to be executed whilst L_1 is waiting to send it's message. This ensures that processor P_1 is kept continuously busy. The hiding of some of the latency in the program results in a decrease in execution time and an increase in speed-up.

6.3.2 The Hardware Architecture

Given a set of processing elements there are many different ways in which those processors may be connected. Section 1.4.4 and Figure 1.6 describe some of the standard processor interconnection topologies and it was observed that different topologies are suited to certain types of problem. For general parallel problems the hypercube topology often gives the best performance. Chapter 1 observed that the choice of the interconnection network can make or break the performance of the parallel machine and the parallel algorithm executed on it. The general literature of the field of parallel computing observes that parallel computers are often used to solve a single parallel problem and in this case the hardware architecture becomes an intrinsic part of the problem and its solution. It is essential to use the interconnection network that gives the optimum performance and this section considers which of the many interconnection topologies is most suitable for a dedicated Recursively Parallel solution.

Architecture and Performance

The hardware architecture of a parallel computer affects performance through its implications for intertask communication. When the parallel tasks reside on different physical processors intertask communication then becomes interprocessor communication. Messages traveling between the tasks flow physically along the wires interconnecting the processors.

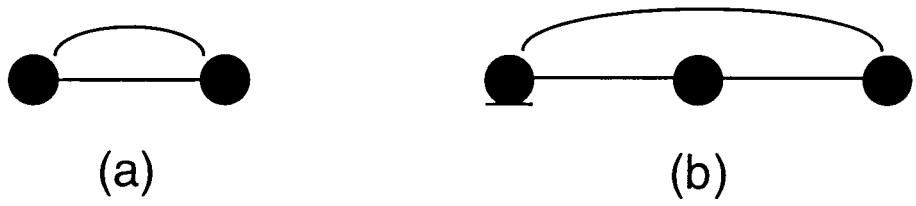


Figure 6.15: Direct 1 hop communication (a) and indirect 2 hops communication (b)

There is a finite delay introduced into the execution of the program which is due to the finite amount of time taken for the message to travel between the two communicating processors. When tasks on adjacent processors communicate this delay is negligible. If the tasks reside on non-adjacent processors and communicate indirectly via an intermediate processor then these delays can become significant. The intermediate processor uses a store-and-forward protocol to route messages to other processors. It receives incoming messages and stores them in a buffer. When this processor makes time for communication, and when the receiving processor is ready, the intermediate processor forwards the message from the buffer to the receiver. If each message is acknowledged, as in Transputer systems, the original sending processor cannot continue executing the task which requested the communication until the acknowledgment from the receiver travels back through the intermediate processors. Routing through any number of intermediate processors is possible but the more intermediaries there are, the greater the delays involved in communication.

Two adjacent processors in a network are said to be separated by a distance of one hop (Figure 6.15(a)) whilst two processors communicating via a single intermediary are separated by a distance of two hops (Figure 6.15(b)). The concept of distance is important in understanding the effect of different interconnection topologies on performance. The delay in communications between two processors is proportional to the distance between them and the various interconnection networks of Figure 1.6 can be characterized in terms of their maximum, minimum and mean communication distances. For example, the star network has a minimum distance of 1, a maximum distance of 2 and a mean distance of slightly less than 2. These distances are constant irrespective of the number of processors used. The ring network on the other hand has distances which vary with the number of processors - the minimum is constant at 1 but the maximum is $\frac{n}{2}$. These characteristics of the different topologies make them suitable for certain applications but not for others. A program based on the supervisor/worker approach would achieve good performance on the star network with the supervisor located at the centre and the nodes at the radii. Communication always

takes place between adjacent processors as the only communications are those between the supervisor and the workers. For other programs communication may have to be routed through the centre making the central processor a bottleneck in the system which limits its performance. In general the average communication delay for a given topology will be proportional to the average communication distance. A topology with a low average distance will often be the best choice. This is one of the reasons for the hypercube's popularity. Of all the topologies in Figure 1.6 the hypercube has the lowest average communication distance. Another explanation of its popularity is that a number of other topologies (*e.g.* pipeline, ring, tree) can be embedded into a hypercube. By purchasing a machine configured as a hypercube users may also make use of any of the interconnection topologies which can be embedded into a hypercube. The hypercube arrangement is one of considerable flexibility.

The benefits of an efficient communication topology can be undermined if the task to processor allocation is not undertaken carefully. Assigning two communicating tasks to opposite ends of a pipeline would be extremely inefficient. The ideal is to place communicating tasks on adjacent processors, thus minimizing the communication delays. This is not always possible and a compromise often has to be reached. The best system design is a close marriage between efficient hardware connection and sensible task allocation.

A Suitable Architecture for the Recursively Parallel Solution

Processor interconnection networks are often represented as undirected graphs in which nodes represent processors and edges represent physical bidirectional communication links between them. If this convention is adopted, Figure 6.2 immediately suggests a possible interconnection for the processors in a system dedicated to the Recursively Parallel solution, assuming each task is assigned to an individual processor. As the Transputer has four bidirectional communication links per processor this topology is only realisable for systems of seven subnetworks or less. For system with more than seven subnetworks indirect routing of communications would be required. Assigning two tasks to each processor improves the situation somewhat and it becomes possible to realize systems with fifteen subnetworks. Systems with more than fifteen processors again require indirect communications. Early in the project a novel architecture, the Mutated Tree architecture [96], was proposed based on an analysis of Figure 6.2. Alterations made to the method since the design of this architecture have rendered it non-optimal as some of the assumptions on which it was based are no longer valid. It is worth considering the Mutated Tree architecture for the

sake of completeness.

The Mutated Tree architecture was conceived prior to the introduction of the hierarchical message aggregation scheme (Section 6.2.3) and its design is based on the assumption that explicit communications are allowed to occur between the last task and every other task in Figure 6.2. If the adjacency constraints for communicating tasks are to be satisfied it is clear that every task must be directly connected to the last one in a radial fashion. This suggests the use of a star network but such a topology cannot be easily realized with the Transputer due to the large number of connections to the central processor. Considering how the algorithm works reveals that the communications $P_1 - P_7$, $P_2 - P_7$, $P_3 - P_7$, $P_4 - P_7$ are not critical to the next stage of processing whereas $P_1 - P_5$, $P_2 - P_5$, $P_3 - P_6$, $P_4 - P_6$, $P_5 - P_7$, $P_6 - P_7$ are. To minimize execution time the critical communications must be performed without delay whilst the non-critical communications can withstand some degree of delay. This observation allows a better interconnection of the processors to be achieved by allowing non-critical communications to be routed via intermediate processors whilst ensuring that critical communications occur only between adjacent processors. The resulting interconnection is the Mutated Tree topology and Figure 6.16(a) shows the 4, 8 and 16 processor Mutated Tree interconnections. Redrawing Figure 6.16(a) gives the diagrams in Figure 6.16(b) and these clearly show a modified form of binary tree, hence the name Mutated Tree. If the assumption concerning explicit last task communications is valid then the Mutated Tree architecture has better communication characteristics than a hypercube with the same number of processors. In each case the Mutated Tree has a total communication distance as good as, if not better than, its hypercube counterpart. Unlike the hypercube the Mutated Tree never requires critical communications to be routed via intermediate processors. Figure 6.16 reveals another useful property of the Mutated Tree, that of scalability. The architecture scales easily to 2^p processors, where p is a positive integer, and never requires more than four interconnections per processor. Large Mutated Tree structures are therefore realizable using Transputers. The limitation of four links per processor means that it is not easy to build Transputer-based hypercubes with more than sixteen processors.

The use of an hierarchical message aggregation protocol invalidates the assumption concerning explicit last task communications and the dataflow diagram of Figure 6.2 is reduced to a simple binary tree. There is no longer any need for the extra connections provided by the Mutated Tree structure and a binary tree interconnection will suffice. The

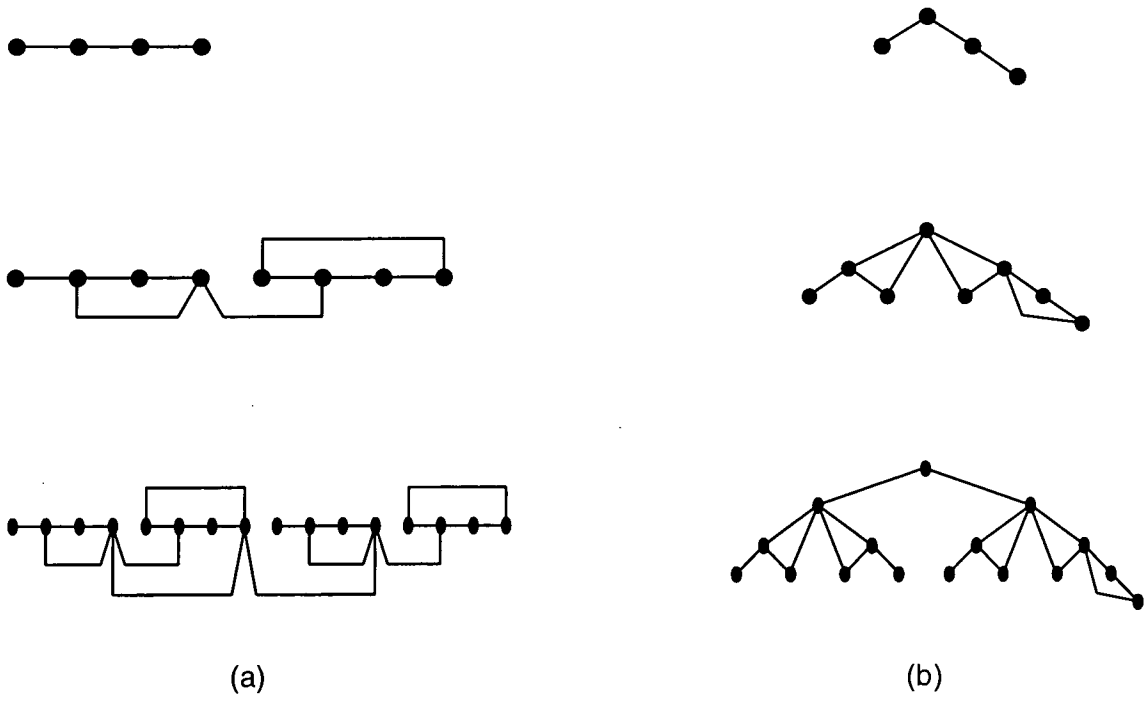


Figure 6.16: The Mutated Tree interconnection network a) shown as modified pipeline b) shown as modified binary tree

simplest topology which satisfies the adjacency constraints is a straightforward pipeline of processors, as shown in Figure 1.7. This is ideal for Transputer implementation as any length of pipeline can be constructed. One interesting feature of the pipeline is that it can be embedded into many of the other structures in Figure 1.6 (*e.g.*ring, mesh, hypercube, chordal ring). Most of the results shown in this thesis were taken with the Transputer network configured as a pipeline, although hypercubes and rings were occasionally used. The results obtained are independent of topology *iff* the topology supports the embedding of a pipeline. This argument is supported by the results of Table 6.3 which shows the same systems solved by the RP method using pipeline and hypercube processor interconnection topologies. The results are identical for both topologies. Results for sixteen processor hypercubes have not been obtained as it was not possible to configure the experimental hardware as a four dimensional (sixteen processor) hypercube. If the RP method were to be used to successively solve different systems of equations then the hypercube topology would probably give the best results. The embedded pipeline within the hypercube satisfies the requirements for the computation phase whilst the low communication delays of the hypercube are beneficial in minimizing the time taken to transfer data from the supervisor task to the worker tasks.

System	Subnetworks	Pipeline			Hypercube		
		Overall	Factorise	Substitute	Overall	Factorise	Substitute
118	3	1.42	1.42	1.39	1.42	1.42	1.39
118	7	2.04	2.08	1.89	2.08	2.08	1.84
118	15	2.50	2.47	1.89	2.50	2.49	1.90
629	3	2.13	2.23	1.84	2.13	2.23	1.84
629	7	2.73	2.70	2.17	2.65	2.67	2.19
629	15	3.97	4.04	3.06	3.94	4.05	3.08
734	3	2.20	2.46	1.84	2.29	2.46	1.84
734	7	3.96	4.10	3.00	3.89	4.02	3.01
734	15	5.68	5.83	4.14	5.56	5.78	4.18
734	31	5.90	5.98	3.98	-	-	-
1624	3	2.51	2.64	1.82	2.51	2.64	1.82
1624	7	4.02	4.15	2.88	4.02	4.15	2.88
1624	15	7.31	7.55	4.52	7.31	7.56	4.55
1624	31	8.84	9.38	5.46	-	-	-

Table 6.3: Comparison of performance using pipeline and hypercube architectures

Consider again the task graph of Figure 6.13 and the intertask communications that are required. At some point in the solution tasks 13 and 14 must communicate with task 15. If the task allocation of Figure 6.13 is adopted and the processors are configured in a pipeline then tasks 13 and 14 both reside on processors which lie a distance of two hop from the processor hosting task 15. Communications with task 15 must therefore take place via an intermediate processor and this will increase the communication time. The communication time would be reduced if a processor topology was used which directly linked the processors hosting tasks 13, 14 15 and this can be achieved by adding two extra connections to the basic pipeline. However when this topology is implemented it is observed that it has very little effect on the speed-up obtained (a difference in speed-up of less than 0.08 for the CEGB 734 node system). There are two reasons for the apparent ineffectiveness of this modification. Firstly, the messages that are passed in the indirect communications are short and the transfer time for these communications is negligible. The extra connections do reduce the transfer time but as these times are small anyway the effect is not noticeable. Secondly it is observed that when a simple pipeline is used the latency of the indirect communications can be hidden by the introduction of an imbalance in the computational loading of each processor (Section 6.3.1). The pipeline topology is therefore the simplest topology for this application.

In relation to the Recursively Parallel solution there seems to be little point in adding any extra connections to the basic pipeline as there are only a few indirect communications which

System	Nodes	Non-zeros	Sparsity(%)
IEEE Test System	118	476	3.4
Reduced CEGB System	629	2301	0.58
CEGB System	734	2696	0.50
Reduced US Eastern Seaboard	1624	6050	0.23

Table 6.4: Characteristics of the test systems

would benefit marginally from their inclusion. A benefit is perceived when the supervisor task is considered. As this task has to communicate with all the processors the additional connections can help in reducing the time taken to transfer data before and after solution. If the Recursively Parallel method were to be applied in a situation where different sets of equations needed to be solved in succession then the provision of extra connections to reduce indirect communications would be of paramount importance in reducing the start up overhead of each solution. Extra connections would also be useful for a full dynamic simulation in which the RP method is used to solve linear equations. Although the extra connections have little benefit for the RP solution, they may be necessary for the other simulation algorithms to achieve maximum speed-up.

6.4 Performance of the Recursively Parallel Solution

Having discussed the development of the Recursively Parallel solution this section presents the results of applying the method to various test systems. Observing the performance of the program requires the accurate recording of its execution time and this is not straightforward. Appendix G gives some consideration to the problems of performance monitoring and visualisation.

The performance of the Recursively Parallel solution has been evaluated using a number of test systems drawn from power system applications. One of the systems is a standard test system whilst the others are derived from real power system networks. Table 6.4 characterizes the members of the test suite. Using the two-to-one task allocation allowed each of the test systems, except the IEEE 118 node system, to be partitioned and solved as 2, 4, 8 or 16 main subnetworks. The IEEE 118 node system is too small to be effectively partitioned into 16 main subnetworks but all the other partitions can be achieved.

In examining the performance of the RP solution the most important result is the speed-up. Speed-up, $S(n)$, is defined as the ratio of uniprocessor execution time to multiprocessor

System	Solution Time (ms)	Factorisation Time (ms)	Substitution Time (ms)
118	34.895	25.23	9.67
629	225.619	175.81	49.81
734	280.725	221.91	58.81
1624	969.49	825.56	143.69

Table 6.5: Performance of the best sequential algorithm

execution time for the same algorithm. To adhere to this strict definition requires the calculation of the ratio of the execution times of the RP solution executed on one processor to the execution time of the RP solution executed on n processors. This figure will be referred to as *speed-up over uniprocessor*. The end user is not interested in this largely theoretical figure as it does not reflect the realistic benefits of using a parallel solution. The true benefits of parallel solution are characterized by the *speed-up over best sequential solution*. This quantity is defined as the ratio of the execution time of the best sequential solution to the multiprocessor execution time. Only if the multiprocessor and best sequential algorithms are identical do the two speed-ups have the same value. In most cases there are necessary differences in the two algorithms and the two speed-ups are different for the same system. The speed-up over best sequential method is the important figure as it reflects the performance benefit of implementing a parallel solution.

The performance of the best sequential method must be considered before the performance of the multiprocessor method can be examined. Chapter 2 described the sequential solution and this is implemented in INMOS C on a single Transputer. The partitioning of the parallel solution is performed by applying the near optimal MDMLLRU ordering and then applying a topological reordering to the system to achieve RBBDF structure. The data supplied to the sequential solution also undergoes the MDMLLRU ordering and topological reordering so that both sequential and parallel programs solve identical systems. Table 6.5 details the performance of the best sequential solution of the test systems. It is found that the topological reordering of the network makes no difference to the sequential solution times. This is to be expected as a topological reordering is defined as one which introduces no extra operations or fill-ins. Factorisation and substitution times are independently identified for the purposes of later analysis.

The performance of the multiprocessor solution has been evaluated for each of the test systems using 2, 4, 8 and 16 processors. Table 6.6 gives the results for all the test cases.

System	CPU's	Complete Solution			Factorisation			Substitution		
		t_e	$S(n)$	$E(n)$	t_e	$S(n)$	$E(n)$	t_e	$S(n)$	$E(n)$
118	2	24.574	1.42	0.71	17.768	1.42	0.71	6.957	1.39	0.70
118	4	17.105	2.04	0.51	12.130	2.08	0.52	5.116	1.89	0.47
118	8	13.598	2.50	0.31	10.215	2.47	0.31	5.116	1.89	0.24
629	2	105.924	2.13	1.07	78.839	2.23	1.12	26.354	1.84	0.92
629	4	82.644	2.73	0.60	65.115	2.70	0.68	22.954	2.17	0.54
629	8	56.831	3.97	0.53	43.517	4.04	0.51	16.278	3.06	0.38
734	2	122.587	2.29	1.15	90.207	2.46	1.23	31.962	1.84	0.92
734	4	70.890	3.96	0.99	54.124	4.10	1.03	19.603	3.00	0.75
734	8	49.423	5.68	0.68	38.063	5.83	0.73	14.205	4.14	0.52
734	14	47.581	5.90	0.42	37.109	5.98	0.43	14.776	3.98	0.28
1624	2	386.257	2.51	1.26	312.720	2.64	1.32	78.951	1.82	0.91
1624	4	241.167	4.02	1.01	198.930	4.15	1.04	49.892	2.88	0.72
1624	8	132.625	7.31	0.91	109.346	7.55	0.94	31.790	4.52	0.57
1624	13	109.671	8.84	0.68	88.013	9.38	0.72	26.317	5.46	0.42

Table 6.6: Performance of the Recursively Parallel solution of the test systems

Figure 6.17 offer a graphical interpretation of these results and shows the scalability of the RP method. As the problem gets larger (*i.e.*more nodes in the network) the greater the speed-up that can be obtained from the parallel solution. This is because it becomes easier to partition the system into a suitable number of equally sized subnetworks as the system gets larger. Obviously there is a limit to the speed-up that can be obtained and this upper limit is the linear speed-up, n, produced by n processors In practice the linear speed-up is unlikely to be achieved and the upper bound on speed-up will be given by Amdahl's Law for the system under consideration.

The factorisation time and substitution time are identified independently in Table 6.6. This is made possible by the detailed timing results returned from the performance monitoring schemes described in Appendix G. Simple inspection of the timing results or the Gannt chart allows the critical path through the subnetworks to be identified and the factorisation and substitution times are obtained by summing the relevant values along the critical path. Note that the execution times for factorisation and substitution in Table 6.6 do not add together to give the overall execution time. This is because the substitution operation is timed as a stand-alone operation (Figure 5.10) and the latency of some of the computations is no longer hidden by factorisation computations (Figure 5.9). It should also be noted that the overall and factorisation execution times were obtained with the storage scheme optimized for factorisation. The substitution time was obtained with the storage scheme optimized for substitution. Table 6.6 also lists efficiency figures for each phase of

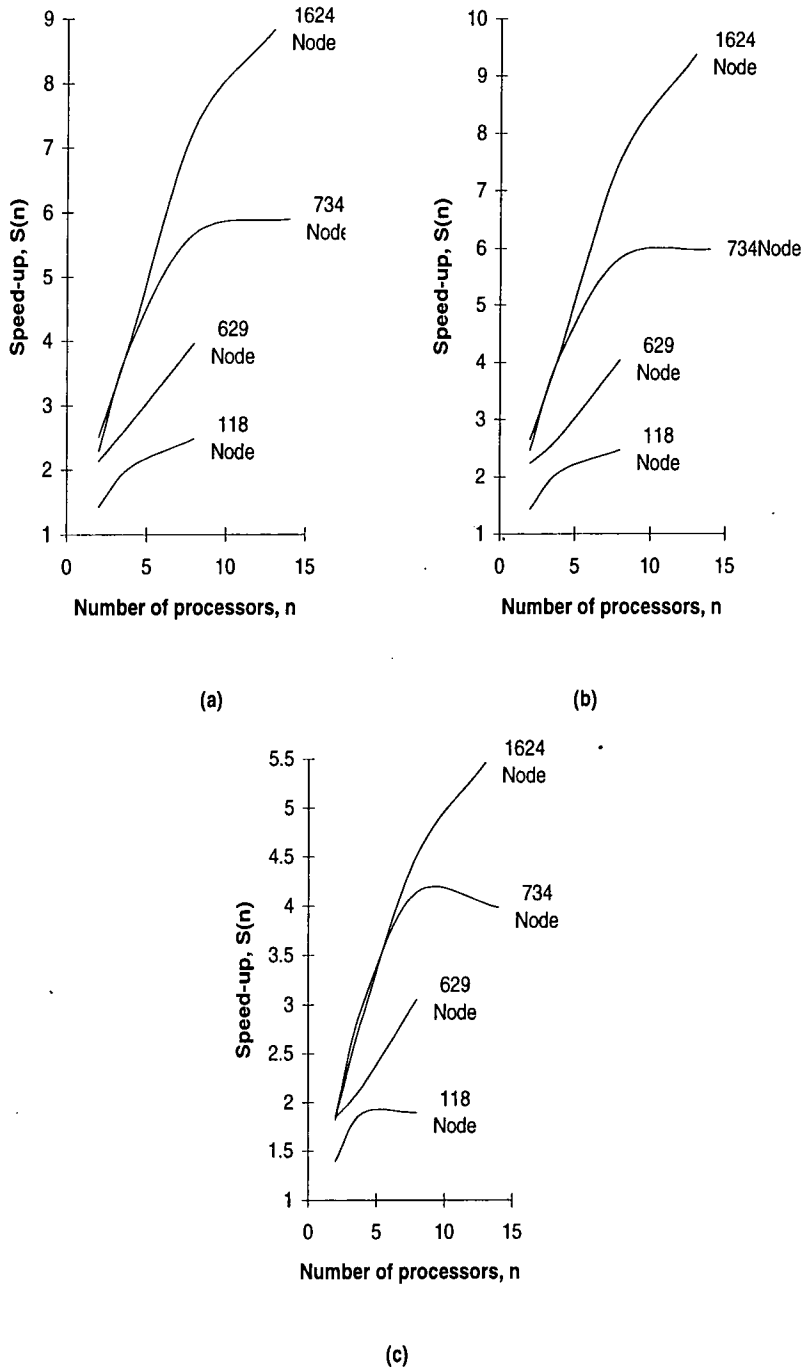


Figure 6.17: Performance curves for the Recursively Parallel method, with 2-1 task allocation a) overall b) factorisation c) substitution

System	CPU's	Complete Solution			Factorisation			Substitution		
		t_e	$S(n)$	$E(n)$	t_e	$S(n)$	$E(n)$	t_e	$S(n)$	$E(n)$
118	2	24.77	1.41	0.71	17.73	1.42	0.71	7.2	1.35	0.68
118	4	16.96	2.06	0.52	12.22	2.06	0.52	5.63	1.72	0.43
118	8	13.95	2.50	0.31	10.30	2.45	0.31	5.57	1.74	0.22
629	2	106.30	2.12	1.06	80.51	2.18	1.09	27.33	1.82	0.91
629	4	85.57	2.64	0.66	66.11	2.66	0.67	23.32	2.14	0.54
629	8	57.47	3.93	0.49	43.20	4.07	0.51	17.54	2.84	0.36
734	2	122.69	2.29	1.15	90.11	2.46	1.23	32.13	1.83	0.92
734	4	72.96	3.85	0.96	55.23	4.02	1.01	20.10	2.93	0.73
734	8	50.30	5.85	0.73	35.53	5.76	0.72	17.98	3.27	0.41
1624	2	386.43	2.51	1.26	313.10	2.64	1.32	78.98	1.82	0.91
1624	4	241.28	4.02	1.01	198.98	4.15	1.04	50.43	2.85	0.71
1624	8	132.93	7.29	0.91	109.31	7.55	0.94	36.42	3.95	0.49

Table 6.7: Performance of the Recursively Parallel solution of the test systems, using 1-1 task allocation

the solution of the test systems.

For the purposes of comparison Table 6.7 details the performance of the multiprocessor solution when a one-to-one task allocation is used. The figures show that the one-to-one allocation gives greater speed-ups for systems with only a small number of subnetworks. As the number of subnetworks increases the speed-up rapidly saturates. The two-to-one task allocation gives lower speed-ups for small systems but the speed-up saturates more slowly as the number of subnetworks increases.

Note that in Table 6.7, superlinear speed-up is recorded for some of the systems. Superlinear speed-up is the name given to speed-ups greater than n produced using n processors. Superlinear speed-up is a rare event and usually only occurs under certain well-defined algorithmic conditions [97, 98, 99] such as combinatorial implosiveness [100]. In this case such conditions do not arise and the superlinear speed-up is a function of the differences between the best sequential and parallel algorithms. The major algorithmic difference relates to the data structures and storage schemes employed by the algorithms. The parallel algorithm uses the highly efficient hybrid storage scheme whilst the sequential algorithm uses less efficient sparse linked list storage. For a true comparison the speed-up figures should be calculated using the execution times from a sequential method which also makes use of hybrid storage. As Section 6.2.2 points out, this storage scheme has not yet been implemented in a sequential solution so the comparison cannot be made here. It is still reasonable to compare multiprocessor performance to that of the best sequential method as this sequential method is the current 'industry accepted' standard and it is interesting

to quantify the improvement provided by the multiprocessor solution over the standard method. Perhaps a fairer analysis of performance can be made by resorting to the speed-up over uniprocessor figures (Table 6.8 and Figure 6.18). In calculating the speed-up over uniprocessor the same program is used for both uniprocessor and multiprocessor solutions so there can be no differences in the data structures and algorithms used. Therefore the speed-up over uniprocessor reflects more accurately the benefit that can be gained from increasing the number of processors used to obtain a solution. In examining the results (Table 6.8) it is immediately apparent that superlinear speed-up no longer occurs. Hence it may be concluded that the superlinear speed-up recorded in Table 6.6 is due entirely to the different data structures used by the parallel and best sequential algorithms. The speed-up curves (Figure 6.18) are slightly different to those for the speed-up over the best sequential solution. The absolute speed-ups are similar for the factorisation, substitution and overall solutions. This is different than for the speed-up over the best sequential solution where it was observed that factorisation and overall speed-ups were similar but substitution speed-up was smaller. The curves of Figure 6.18 confirm the expectation that factorisation and substitution speed-up should be similar. Another striking feature of these curves is that saturation of the speed-up is much less noticeable than in Figure 6.17 particularly for the 734 node system. This indicates that the speed-up saturation demonstrated in Figure 6.17 is a function of the differences in data structures between the best sequential and parallel algorithms. The lower rate of saturation in the speed-up over uniprocessor results is manifested in Figure 6.18 as 'straighter' curves which are much closer to an ideal straight line than those of Figure 6.17. The lower rate of saturation implies that the RP algorithm is easily scaleable and that few overheads are introduced as the number of processors is increased.

In evaluating the performance of the RP solution it must be compared with the speed-ups obtained in simulation and the speed-up predicted by an analysis of the elimination tree. To establish the validity of the RP solution it is also necessary to compare the performance of the RP solution with the performance of other multiprocessor solutions.

Table 6.9 and Figures 6.19 and 6.20 show the theoretically predicted speed-up and the simulated speed-ups for each of the test systems. It can be seen that the performance of the Transputer implementation matches, and sometimes exceeds, the predicted performance of the method. The factorisation speed-up is encouraging as it is often greater than that predicted or obtained in simulation. The main reason for this is that the simulation uses the

System	CPU's	—Complete Solution— $S(n)$	—Factorisation— $S(n)$	—Substitution— $S(n)$
118	2	1.51	1.49	1.56
118	4	2.47	2.58	2.40
118	8	3.98	3.98	3.35
629	2	1.87	1.90	1.91
629	4	2.62	2.54	2.39
629	8	4.05	3.96	3.70
734	2	1.90	1.92	1.88
734	4	3.52	3.44	3.21
734	8	5.51	5.36	5.05
734	14	7.08	6.84	6.05
1624	2	1.76	1.69	1.91
1624	4	2.93	2.78	3.23
1624	8	5.48	4.98	5.24
1624	13	7.43	7.04	6.43

Table 6.8: Performance of the Recursively Parallel solution of the test systems - speed-up over uniprocessor

System	Processors	Subnetworks	Predicted Speed-Up	Simulation Speed-Up	Observed Speed-Up vs Sequential	Observed Speed-Up vs Uniprocessor
118	2	3	1.55	1.55	1.42	1.49
118	4	7	2.93	2.72	2.08	2.58
118	8	15	4.17	4.41	2.47	3.98
629	2	3	1.92	1.94	2.23	1.90
629	4	7	2.38	2.41	2.70	2.54
629	8	15	3.91	3.96	4.04	3.96
734	2	3	1.94	2.01	2.46	1.92
734	4	7	3.46	3.34	4.10	3.44
734	8	15	5.40	5.37	5.83	5.36
734	14	31	7.67	6.67	5.98	6.84
1624	2	3	1.80	1.79	2.64	1.69
1624	4	7	3.07	2.41	4.15	2.78
1624	8	15	5.21	5.15	7.55	4.98
1624	13	31	8.78	6.27	9.38	7.04

Table 6.9: Simulated, predicted and observed factorisation speed-ups for the RP solution of the test systems

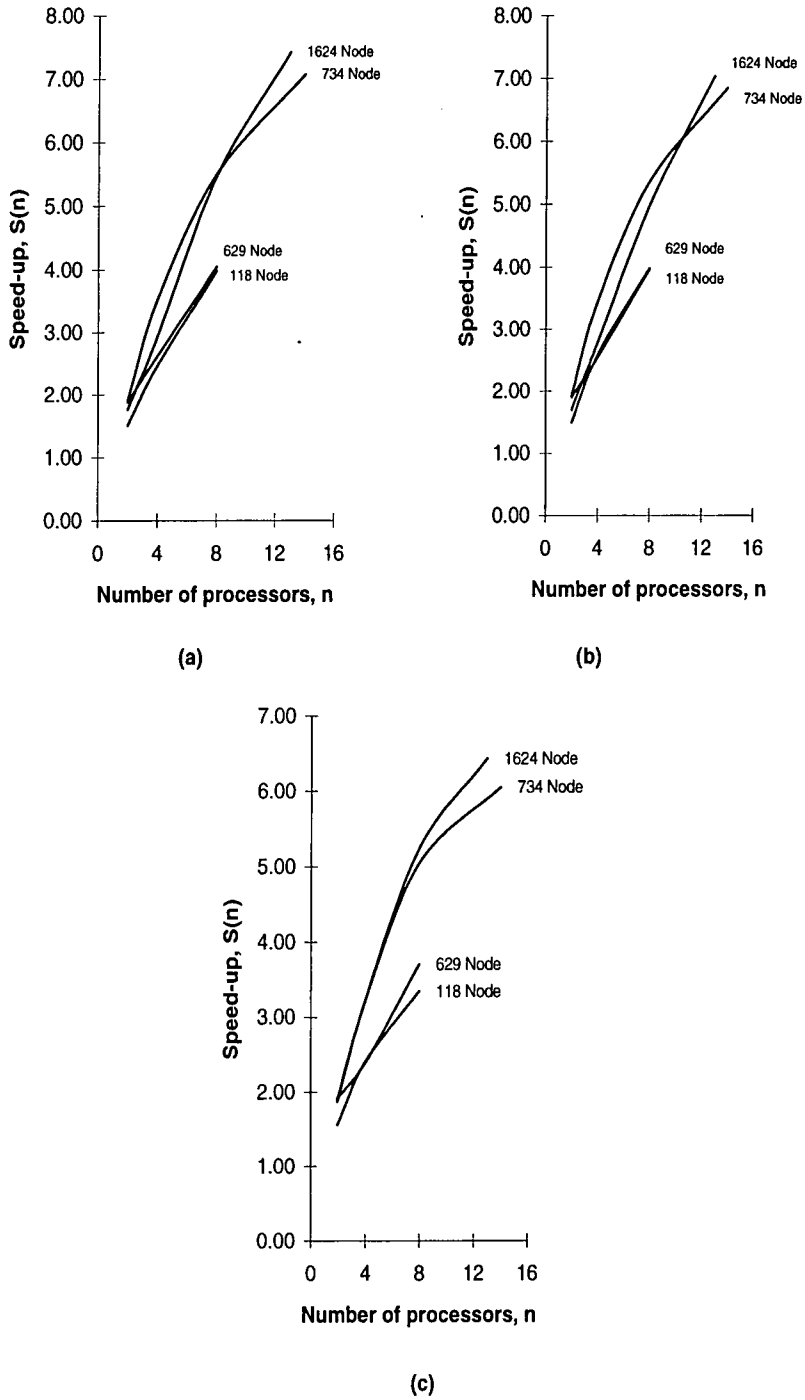


Figure 6.18: Speed-up against uniprocessor for the Recursively Parallel method, with 2-1 task allocation a) overall b) factorisation c) substitution

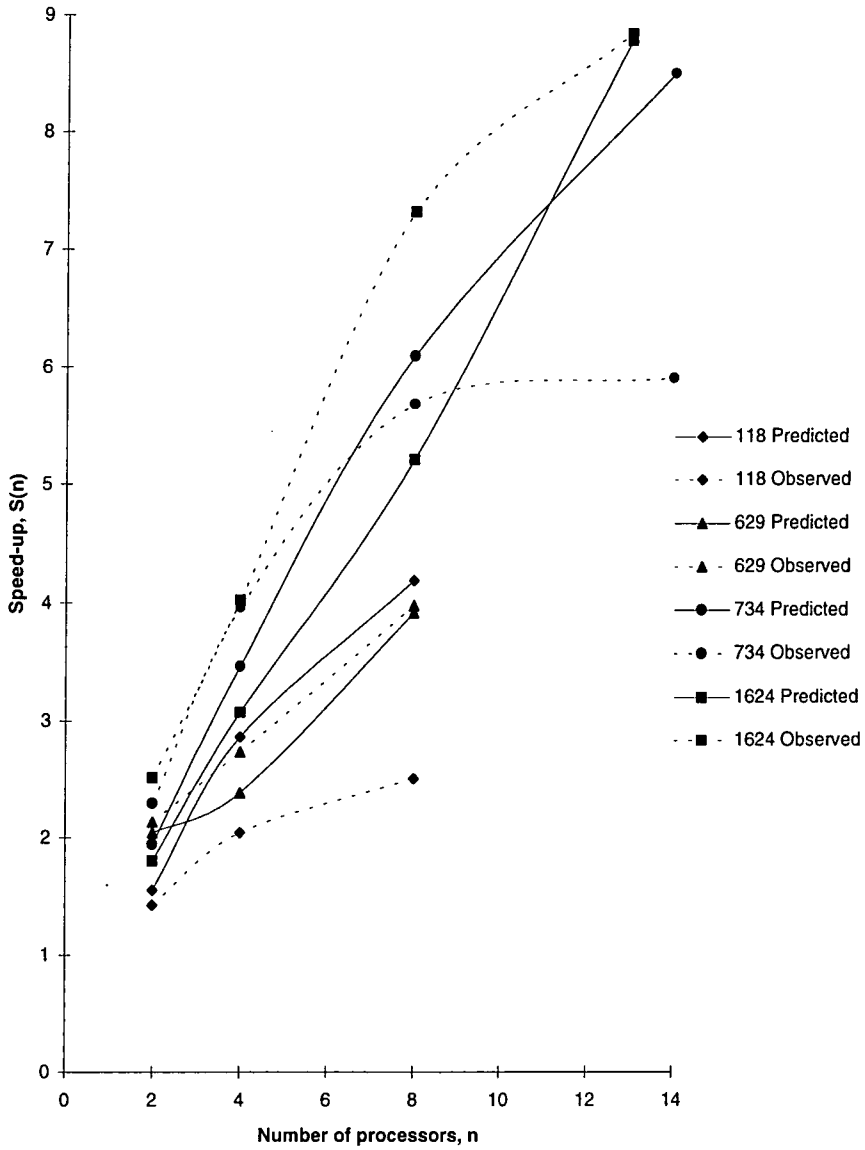


Figure 6.19: RP method compared to predicted performance (factorisation only - speed-up over best sequential)

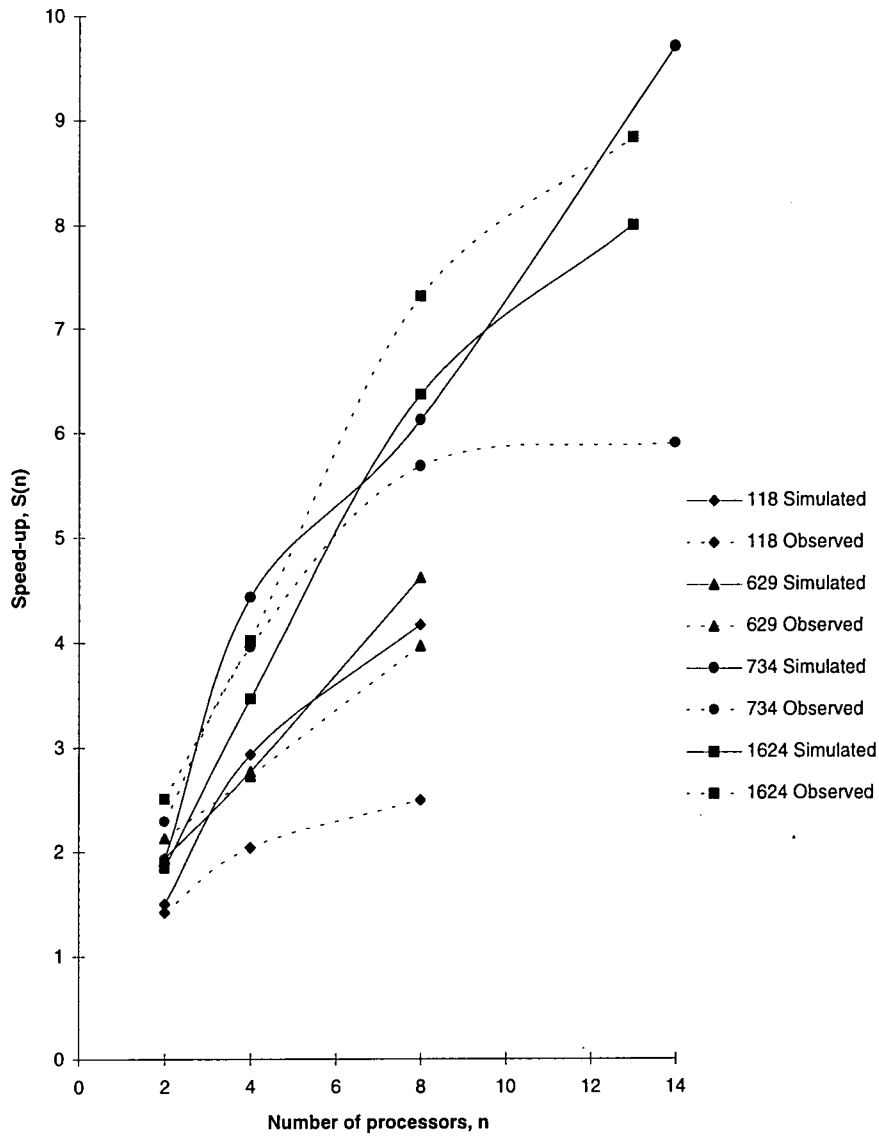


Figure 6.20: RP method compared to simulated performance (factorisation only - speed-up over best sequential)

hybrid storage mechanism to a lesser extent and the parallel implementation is therefore inherently more efficient. As the factorisation requires many search operations to find the relevant matrix elements hybrid storage improves the performance by keeping the number of less efficient linked list searches to a minimum. The effect is more noticeable on larger systems thus explaining why the performance of the 734 and 1624 node systems are better than predicted. A second effect is also at play in the parallel factorisation which is not accounted for in the simulation and that is the effect of interprocessor communication. This has the effect of reducing the speed-up and is particularly noticeable in smaller systems. It is partly this effect that is responsible for keeping the speed-up of the 118 node system below that expected of it. In larger systems this effect tends to reduce the benefits produced by the hybrid storage scheme. The influence of communication becomes greater as the number of processors is increased and it is the interprocessor communication which is responsible for the tailing off of the speed-up as more processors are introduced [25]. When the speed-up over uniprocessor results are compared with the predicted speed-ups of Table 6.9 it is found that observed speed-ups match very closely to the theoretical predictions. When eight (or fewer) processors are used the observed speed-ups are similar to the predicted speed-ups and for the 734 node system the two sets of figures are almost identical. As the number of processors increases beyond eight the observed speed-up starts to fall away from the theoretical predictions and this provides evidence to support the claim that increased communication overheads are responsible for the tailing off of speed-ups as the number of processors increase. It also confirms that the improved performance offered by the RP method is due to the improved partitioning of the problem.

The substitution speed-up is much less encouraging as it seldom comes close to the predicted or simulated performance. This is due to the inherent difficulties in parallelising the substitution operation. Highly efficient sequential substitution algorithms exist which execute extremely rapidly. Because the sequential substitution algorithms are so fast it is difficult to create parallel versions which do not introduce extra overheads into the computation and detract from the performance. The interprocessor communication which results from parallelising the substitution operation introduces the greatest overheads. Although these communications are responsible for only a very small amount of time in factorisation they become much more significant in substitution due to its short execution time. This explains the almost universal difficulty in creating a parallel substitution algorithm which has a speed-up greater than about 3. Although the actual performance (over the best se-

quential method) falls short of the theoretical and simulated performance it is still good when compared to the performance of other multiprocessor substitution algorithms and some relatively high speed-ups have been achieved using only a small number of processors. When substitution performance over uniprocessor is compared to predicted performance the outcome is much more encouraging. The speed-ups are much closer to those predicted and the absolute speed-ups are greater than the speed-ups over best sequential. This indicates that the poor speed-ups over best sequential are once again a function of the differences in data structures between the best sequential and parallel algorithms. As with the speed-up over best sequential, substitution speed-up over uniprocessor falls off as the number of processors increases and it falls off at a faster rate than the factorisation or overall speed-up. The implication of this is that interprocessor communication and other scaling overheads are primarily responsible for saturation of the substitution speed-up.

Unfortunately it is not very easy to directly compare the performance of the RP method against the performance of other multiprocessor solutions. Few authors cite actual speed-up figures in their publications, although there is an argument that it is more valid to present actual execution times [101]. Unfortunately it is not possible to compare the same method executed on different architectures using only absolute execution times as the execution time depends on characteristics of the machine, such as the instruction set and the clock speed. Comparisons are made possible if authors publish absolute execution times for their parallel algorithms along with execution times for the best sequential algorithms. Some authors do give absolute execution times for their methods but fail to give the execution time for the best sequential solution on the same processor. This rules out the possibility of calculating speed-ups from the quoted timings. Where results are available they seldom refer to the same systems as those used here and some interpolation of results is necessary to allow comparison. A few authors have published performance figures which do allow some comparison to be attempted. In the following pages the performance of other parallel methods are compared with the speed-ups (over best sequential) produced by the RP method.

Padhila & Morelato [44] cite results for three systems similar to those used here. Padhila's results,shown below, were obtained using both 8 and 16 processors.

System	Speed-Up Over Sequential		Efficiency	
	8 Processors	16 Processors	8 Processors	16 Processors
118	4.76	≈ 8	0.595	0.5
725	5.14	≈ 9	0.643	0.563
1729	5.95	≈ 11	0.744	0.688

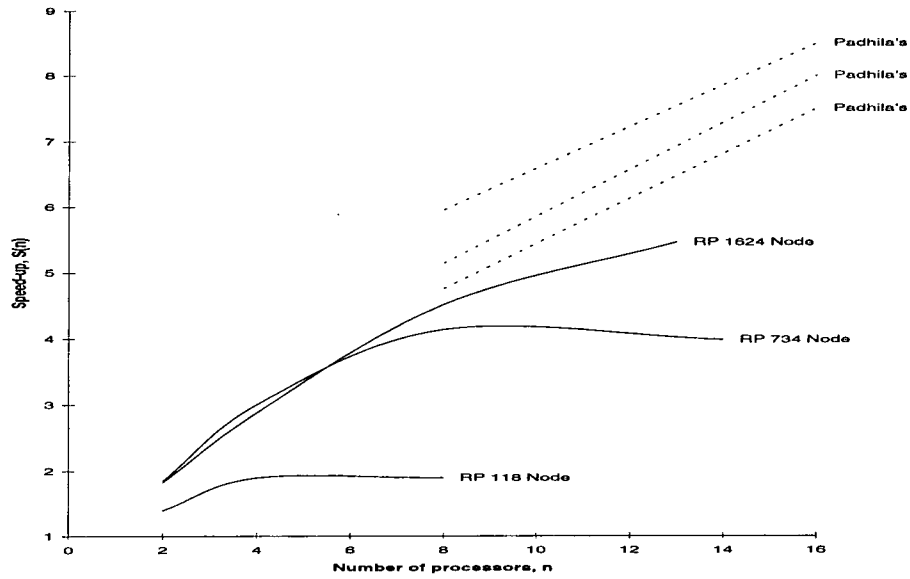


Figure 6.21: RP method compared to Padhila's W-matrix method

These results, which are for substitution only, appear impressive at first sight. The 8 processor results for the larger systems are not significantly different from those produced by the RP method. The 16 processor results seem to significantly better than those obtained from the RP solution. However the 16 processor results are only simulated results and Padhila points out that the simulation neglects communication and also does not account for the processing of diagonal elements. It is not possible to compare these figures to the RP method as actual results are not available but is suspected that the effect of diagonal processing and communication will be to reduce the 16 processor performance to something more akin to that of the RP method. Another factor which is significant in making Padhila's results more impressive is the architecture on which the parallel program was implemented. The machine was a distributed shared memory machine in which each processing node

consisted of an Intel 80286 processor coupled with an Intel 80287 floating point maths coprocessor. The presence of the coprocessor introduces parallel processing at each node as the coprocessor can perform mathematical operations in parallel with the main processor. The fact that a large shared memory was present makes one suspect that communication between processors was achieved via the shared memory and few, if any, interprocessor communication delays were introduced.

Lau [32] presents performance figures for systems similar in size to the smaller of the two test systems used here. The performance quoted by Lau is for the factorisation operation and no results are presented for substitution. Table 6.6 clearly shows the performance of the RP method to be far superior to that of Lau's method. The performance of Lau's method is surprisingly poor as the method itself is similar to the RP method. Lau bases network partitioning on an analysis of the elimination tree and tries to achieve a balanced computational loading. The major difference is that Lau's method uses a standard BBDF coefficient matrix which requires the cutset to be processed sequentially. Even when hybrid storage is not used the RP factorisation speed-ups are significantly better than Lau's. This would imply that the improved performance of the RP solution is due mostly due to the exploitation of the extra parallelism introduced by the RBBDF coefficient matrix structure.

Processors	118 Node System		662 Node System	
	$S(n)$	$E(n)$	$S(n)$	$E(n)$
1	1	1	1	1
2	1.6	0.8	1.4	0.7
4	1.9	0.95	1.7	0.85
8	1.8	0.9	1.65	0.83
16	1.4	0.7	1.5	0.75

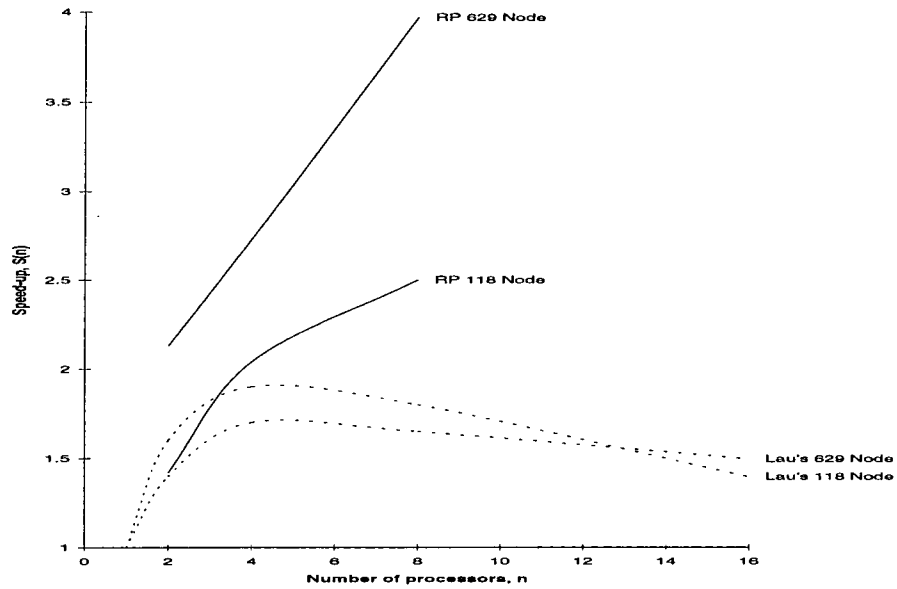


Figure 6.22: RP method compared to Lau's method

Chan [36] provides result for the CEGB 811 node network. These results, which are for substitution, are useful as the CEGB 734 node network used here is a reduced version of the 811 node network used by Chan. The results for the RP solution of the CEGB 734 node network are superior to those obtained by Chan and the results for the US 1624 node network, which is more than twice the size of Chan's system, are also superior. Chan's results are disappointing when it is considered that they were obtained using a distributed shared memory architecture [81]. As all interprocessor communication is performed via the shared memory Chan's method should not suffer from the communication delays inherent in message passing distributed memory machines. One would therefore hope that Chan's

method would be more efficient than a distributed memory solution. The fact that the results from RP solution were collected using a distributed memory multiprocessor serves to highlight the improvements in performance offered by the RP method.

Processors	Speed-Up $S(n)$	Efficiency $E(n)$
2	1.77	0.94
4	2.55	0.78
8	3.36	0.58
16	3.65	0.33

RP method compared to Chan's method

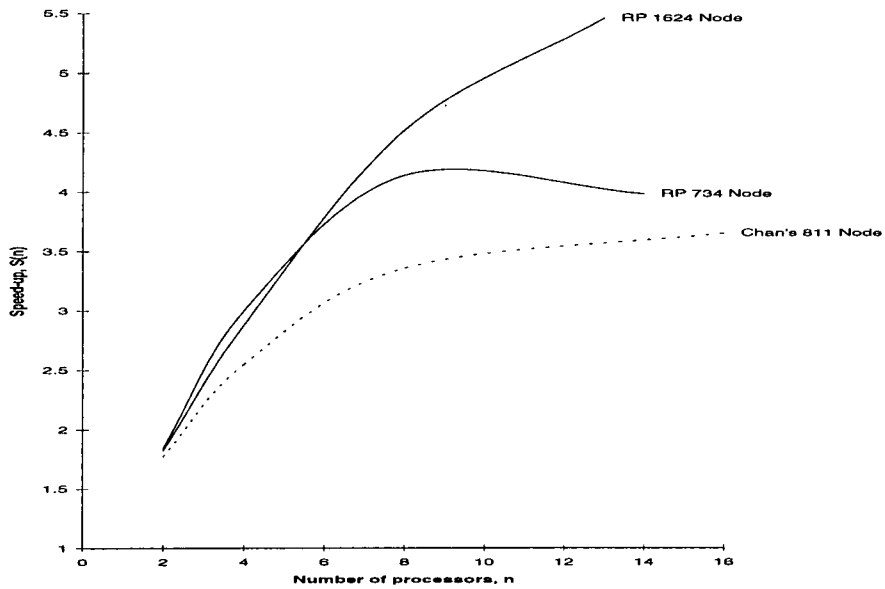


Figure 6.23: RP method compared to Chan's method

Van Ness [46] provides results for the substitution of a 1723 node system using the multiple factoring method. As with many other authors, Van Ness provides only the results of a simulation and does not provide real results from a practical implementation of the method. Based on the figures given it would appear that the multiple factoring method produces a speed-up of 15 using 20 processors and a speed-up of 9.7 using 50 processors. The introduction of parallel overheads and interprocessor communication would reduce the speed-ups somewhat but they do appear to be greater than those produced by the RP method. In a recently published paper [33] Van Ness does provide results from implemen-

tations of the multiple factoring method on both shared and distributed memory machines. Again these results are for a 1723 node system and the results for the distributed memory implementation are disappointing, with the highest speed-up being 2.47 using 8 processors. If the number of processors is increased further the speed-up drops off and a speed-up of 2.02 is achieved using 12 processors. The Transputer-based RP solution, which is also a distributed memory implementation, fares much better. The 1624 node system is solved with a speed-up of 4.52 using 8 processors and a speed-up of 5.46 using 13 processors. Not only does the RP method give greater speed-ups but it also has quicker factorisation then the multiple factoring method, which requires an extra factorisation stage to partition the factored coefficient matrix. The multiple factoring method performs much better when implemented on a shared memory machine. For the same 1723 node system the speed-ups are now 5.55 with 8 processors, 5.96 with 12 processors and 7.48 with 16 processors. Although this performance is almost three times better than that of the distributed memory implementation it is still comparable with that of the distributed memory RP implementation. The main drawback of the multiple factoring method is that extra information has to be introduced into the problem to enable a parallel solution to be implemented.

Several other authors also provide performance statistics in their publications but these all relate to the IEEE 118 node system which is perhaps too small to provide a valid test of a parallel solution. Abur [26] quotes a speed-up of 3.8 for the IEEE 118 system which appears encouraging until one realizes that between 48 and 57 processors are required to achieve this performance. Not only is Abur's method extremely inefficient ($0.079 \leq E(n) \leq 0.067$) it is also extremely expensive when one considers the amount of processing power needed. El-Kieb et al. [30] provide results for the performance of their parallel solution but these results are for system that have no more than 30 nodes and again these systems are too small to provide a valid test of performance. However El-Kieb does claim a fourfold speed-up for the solution of the IEEE 118 node system using 10 processors, although he gives no specific results to support this claim.

The comparisons drawn have shown the RP method to give a better performance than a number of existing LU-based multiprocessor solutions. The RP method produces higher speed-ups at greater efficiencies over a range of test systems. An extensive comparison of the methods is not possible as insufficient data is available for the methods developed by other authors. Similarly, the four systems used in assessing the RP method do not make for a comprehensive test. Obtaining the data for real systems is not easy as the required data

is often commercially sensitive. Using the limited data available one must conclude that the RP method offers performance advantages over existing methods. As the performance compares closely with theoretical predictions one must assume that the RP method is an efficient and effective method for solving linear network equations in parallel.

6.5 Summary

This chapter has considered many of the implementation issues encountered in developing an RP solution program for a distributed memory MIMD environment. Following from the initial specifications and basic algorithms of Chapter 5, refinements have been introduced which significantly improve the performance of such an implementation. Results of the Transputer-based implementation have been presented and the performance of the method has been compared with that of other methods. It has been demonstrated that the RP method offers comparable performance to other methods using fewer processors, making the RP method more efficient and economically viable.

Chapter 7

Further Work

Whilst the Recursively Parallel method performs almost as well as the theory predicts there is always room for improvement. Throughout this discourse various suggestions have been made regarding further work which may be undertaken to improve the method and make it more amenable to incorporation into power system analysis applications. These suggestions are concerned with the partitioning of the network, the selection of an optimal ordering and the use of vector processing techniques. Outlines of the problems requiring further investigation are given along with suggestions as to how these problems may be solved.

7.1 Automatic Network Partitioning

Chapter 4 showed how the elimination tree may be used as a tool to assist in the partitioning of the network. At present the partitioning is achieved by visual inspection of the elimination tree. This is far from ideal and a significant amount of preparation is required prior to the solution of a system. A method which automatically partitions a system is more appropriate and will allow for faster processing of systems. Fortunately the techniques used in dividing the tree are heuristic and a set of rules can be derived which form the basis of an automatic partitioning algorithm. The method will be based upon a bin packing approach [89, 90] and the heuristic rules are used to select which nodes or subtrees to place in each bin. Whilst many of the rules are already known it is unclear how certain parts of the method will be implemented and it is these topics which require further investigation.

The bin packing approach works by designating a number of 'bins' into which parts of the problem are placed. For network partitioning the bins correspond to subnetworks and

subtrees of the elimination tree are assigned into each subnetwork. Hence each subnetwork is a collection of subtrees of the elimination tree. It is important to note that the subnetworks may consist of disjoint subtrees. A subnetwork can therefore be constructed from a number of small subtrees which are not directly interconnected.

Consider the method for partitioning a system for solution by a typical Master/Slave type parallel solution algorithm [87]. A number of subnetworks are to be produced along with a single cutset block. The first step in the method is to chose a threshold weight which is used in analyzing the weighted elimination tree (Section 4.3.2). If the entire tree has a weight of W and m subnetworks are to be created then the ideal threshold weight is $\frac{W}{m}$. Taking into account the presence of the cutset block, the threshold weight is given by

$$W_{th} = \frac{W}{(m+1)} \tag{7.1}$$

An upper and lower bound on subnetwork(bin) weight is also required and the idea is to assign subtrees to a subnetwork until the weight of the subnetwork lies in between the upper and lower bound on subnetwork weight. If has been found empirically that setting the upper and lower bounds (W_u and W_l) to $W_{th} \pm 5\%$ gives good results. Hence

$$W_u = 1.05W_{th} \qquad\qquad W_l = 0.95W_{th} \tag{7.2}$$

The partitioning algorithm has six steps. In the following outline algorithm W_i is the weight of the i^{th} subnetwork and $i = 1 \ldots m$.

1. Scan the elimination tree, starting from the root, and pick the first nodes encountered on each branch which have a weight lower than or equal to W_{th}. Order these nodes, which are the root nodes of subtrees, in descending order of weight.

2. Choose the subtree corresponding to the node of the largest weight and assign it to subnetwork i

3. If $W_i < W_l$ try adding other subtrees (in descending weight order) until the weight of subnetwork i obeys $W_l < W_i < W_u$.

   If $W_i > W_u$ remove subtrees from the subnetwork until $W_l < W_i < W_u$ and add the removed subtrees into other subnetworks so that their weights fall in the desired range.

4. Repeat steps two to four until all the subtrees have been assigned to the subnetworks and each of the m subnetworks consists of at least one subtree.

5. It may be necessary to fine tune the partitioning if there are some subnetworks for which $W_i < W_l$. Fine tuning may also be required if some subnetworks have $W_i \approx W_u$ whilst others have $W_i \approx W_l$. Fine tuning is accomplished by removing constituent subtrees from subnetworks with large weights and assigning them to subnetworks with smaller weights in an attempt to balance out the weight of the subnetworks. This stage may be repeated until the best balance is found.

6. All the nodes which lie between the root of the elimination tree and the identified subtree roots constitute the tearing nodes and these are grouped together to form the last subnetwork (*i.e* the cutset).

This algorithm works well for partitioning a network for solution by a conventional parallel approach where m subnetworks and a single cutset are required. The problem becomes more complicated when an automatic method of partitioning for RP solution is required. With the RP method there are several distinct levels of subnetworks within the task graph. All the subnetworks in each level should have roughly the same weight and these weights should decrease in moving through the levels from the leaves to the root of the task graph tree. If there are k levels in the tree and WL_j is the desired weight of all nodes in level j then

$$WL_j > WL_{j+1} \qquad j = 1 \ldots k-1 \tag{7.3}$$

Instead of a single upper and lower bound on subnetwork weight an upper and lower bound is now required on WL_j, where $j = 1 \ldots k$. Again the $\pm 5\%$ tolerance can be applied to yield

$$WL_{u_j} = 1.05 WL_j \qquad WL_{l_j} = 0.95 WL_j \tag{7.4}$$

but some heuristic must be found to determine the relationships between the various values of WL_{l_j},WL_{u_j},WL_j and W_{th}. At present no heuristic has been derived and further work is required. Assuming that the values of WL_j can be found the algorithm for partitioning for RP solution is

1. Scan the elimination tree, starting from the root, and pick the first nodes encountered

on each branch which have a weight lower than or equal to W_{th}. Order these nodes, which are the root nodes of subtrees, in descending order of weight.

2. Use steps two to four of the previous algorithm to allocate subtrees to the m subnetworks in level j, where $j = 1 \ldots k$.

3. Repeat step two until subtrees have been allocated to all the subnetworks in the k levels of the tree.

4. Fine tune the partitioning to achieve the best inter-level balancing.

There are variations on this algorithm but these will not be considered further as the aim is only to highlight the issue as a topic for further investigation rather than to solve the problem. Another complication for RP partitioning also requiring further study is implicit in steps three and four of the first algorithm. In partitioning for a standard parallel solution the choice of which subtree to add to which subnetwork is not constrained in any way. In partitioning for RP solution subtrees must be selected so that they do not violate RBBDF structure constraints as well as satisfying the weight constraints. An additional heuristic must be derived which checks to see whether the inclusion of a given subtree violates the RBBDF constraints.

7.2 The Search for an Optimal Ordering

Section 2.7.6 compares the performance of a number of near-optimal ordering strategies and describes a simple computer program which was written to assess the performance of each of these methods. The performance of a given ordering strategy is observed by iteratively applying that ordering to a given system. Before each iteration the system is randomly reordered and performance is quantified in terms of the fill-in introduced and the length of the critical path. Whilst most of the path lengths and fill-ins are close to the mean values, certain random reorderings give significantly shorter paths and fewer fills. Other random orderings give rise to much higher fill and longer paths. The ordering which gives the shortest path and minimum fill-in is of interest as this is the optimum ordering of the system.

Chapter 2 observed that a graph with n nodes may be reordered in $n!$ different ways, some of which are, in some sense, 'more optimal' than others. When n is of the order of 1000 the problem is NP-complete and it is not realistic to examine all the possible orderings

to find the optimum. However the use of the optimum ordering can have a profound effect on the performance of the solution algorithm and if it is possible to find this ordering without incurring large computational overheads then it is surely worth doing. Existing near-optimal strategies go some way to optimizing the elimination but they are sensitive to the initial network ordering and the orderings they produce can only be considered as locally optimal.

As the problem is NP complete it would be inefficient to perform an exhaustive search to locate the optimum ordering. Genetic algorithm techniques [102] suggest ways in which the space of all possible reorderings may be searched to find the optimum solution. Genetic algorithms are based upon the biological processes of evolution and natural selection, often referred to as survival of the fittest. Each organism in nature carries a blueprint of itself in its genes and each gene consists of smaller segments called chromosomes which define certain aspects of the organism and its behaviour. Evolution occurs when the organism reproduces and is accomplished through the action of mutation and crossover. In reproduction each of the parent organisms passes some of its genetic structure to the child. Crossover is the operation which splices together the two parent genes to produce the child's gene which is slightly different from that of either parent and this makes each child a unique individual. Mutation is the process which may randomly alter some of the child's chromosomes. The alteration to genetic structure provided by crossover and mutation gives the child slightly different structure and behaviour to its parents and this enables organisms to adapt to their environment over a period of several generations. Those organisms which are best adapted to their environment reproduce vigorously and produce many offspring with similar genetic make up. Those organisms least suited to the environment produce few offspring. Over a number of generations the strong genes survive and are spread throughout the population whilst the weak genes die out.

Genetic algorithms (GA's) are a type of search algorithm which 'evolve' toward the most optimum solution of a given problem. They are especially useful for problems in which the solution space is very large or for NP complete problems. In order to use a GA it is necessary to parameterize the characteristics of a solution to the problem. These parameters are the 'chromosomes' of the solution and are concatenated to form a gene string. An initial population of gene strings is required and it is usual to use a population with tens of members. The members of the population initially have their strings initialized with random values. Some form of fitness function is required to assess how good each

gene string is as a solution to the problem. Those strings which are found unfit are killed off and reproduction of the remaining strings is used to generate an equivalent number of new strings. Crossover is used in reproduction and a small (less than 1%) percentage of the strings in the population are mutated. The cycle of fitness testing and reproduction continues until the majority of the population converges to an acceptable solution.

The most difficult part of any genetic algorithm is knowing when to terminate the search. The operation of a GA can be seen as a sort of parallel search of the problem's solution space. Somewhere within that space lies the optimum solution. Starting from random locations within that space the selection, mutation and crossover operations cause the search points to jump through the space towards the region where the optimum solution lies. Most of the population will eventually hold the gene string representing the optimum solution but not all strings will converge to this solution due to the action of the mutation operation, which is needed to prevent the search getting trapped by local optima. The search can be terminated when a defined number of the population have converged to the same solution.

The search for the optimum matrix reordering can be coded as a GA by making each gene a string which defines how the naturally ordered network is to be reordered. Each gene in the population is initially given a random reordering. The selection, mutation and crossover operations are applied to produce new populations. Mutation simply shuffles some of the elements in a given gene and effectively produces a new random reordering. Crossover splices together two genes to give a child gene which encodes a different reordering. Care must be taken to ensure that the ordering specified by the child does not contain any repeated entries. The selection operation is the fitness function and this must assess which reorderings are good and discard those that are not good. This is the most difficult part of the proposed method and it is this aspect of the problem which requires further work.

The fitness of a given ordering can be quantified in terms of the fills and critical path length, as in the program described in Section 2.7.6. The determination of path length and fill-in can only be obtained by simulating the elimination of nodes from that system and although this is quicker than actual elimination it is still a slow process. The simulated elimination must be performed for each gene in the population. Considering that hundreds of generations may be required to achieve convergence it is easy to see that this approach will lead to long run times and the benefits of using the optimal ordering may be outweighed by the computational overhead required to identify that ordering. What is needed is a different method of assessing fitness that does not rely on simulated elimination. Any such method

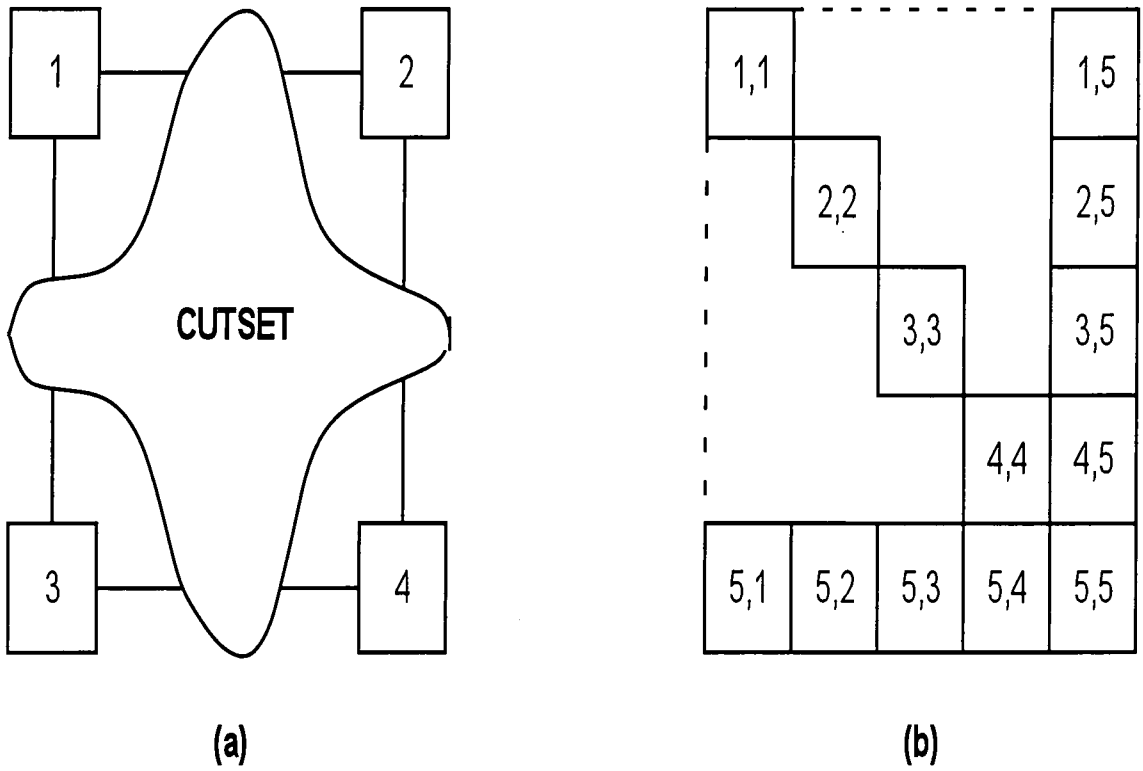


Figure 7.1: Conceptual view of a simple four subnetwork system (a) and its BBDF matrix (b)

must be fast so as to minimize the time to convergence. If convergence is still too slow the search could be accelerated by using a number of copies of the GA executing on different processors in a parallel machine. Any gene which fails the fitness test represents a location in the search space at which the optimum solution does not exist. If this information is shared between the parallel copies of the algorithm then the search space of each copy can be reduced and the optimum solution may be found more quickly. The increase in performance is produced by the combinatorial implosiveness [100] of the parallel GA method.

7.3 Block-oriented Solution and Vector Processing

Chapter 4 stated that it is possible to achieve a parallel solution of the linear equations in a block oriented manner rather than in the rowwise manner which has been adopted throughout this thesis. Consider the network of Figure 7.1(a) which consists of four subnetworks separated by a cutset. Figure 7.1(b) shows the BBDF coefficient matrix associated with this network.

The rowwise method of processing factorises the coefficient matrix into left and right

hand factor matrices by applying the bifactorisation rules of Section 2.4.4 to individual matrix elements. For the matrix in Figure 7.1(b) the bifactorisation rules may be modified such that they act on entire matrix blocks. The solution for the unknown vector, x, is still given by

$$x = R^1.R^2 \cdots R^{n-1}.R^n.L^n.L^{n-1} \cdots L^2.L^1.b \tag{7.5}$$

but the rules for creating $L^k and R^k$ are

$$L^k_{ik} = -A^{(k-1)}_{ik}(A^{(k-1)}_{kk})^{-1} \qquad i = k+1 \ldots n \tag{7.6}$$

$$R^k_{kj} = -(A^{(k-1)}_{kk})^{-1}A^{(k-1)}_{kj} \qquad j = k+1 \ldots n \tag{7.7}$$

$$A^k_{ij} = A^{(k-1)}_{ij} + L^k_{ik}A^{(k-1)}_{kj} \qquad i,j = k+1 \ldots n \tag{7.8}$$

For a symmetrical matrix

$$A^{(k-1)}_{ik} = (A^{(k-1)}_{ki})^T \qquad R^k_{ik} = (L^k_{ki})^T \tag{7.9}$$

The left and right hand factor matrices can be obtained entirely by operating on the matrix at block level. Block manipulations simply require multiplication and additions of matrix blocks, which can be treated as matrices in their own right. Only one step is more complicated and that is the step which requires a multiplication of the form

$$C = A^{-1}_{kk}B \tag{7.10}$$

where A_{kk}, B and C are matrix blocks. It is not necessary to obtain the full inverse of A_{kk} although it may be more efficient to do so if A_{kk} consists mostly of non-zeros. The effect of A^{-1}_{kk} can be obtained by factorising the matrix block A_{kk} to yield the factored form $\tilde{A_{kk}}$ and the multiplication can be performed using the algorithm

```
loop i from 1 to n
        C_i = Ã_kk.B_i
end i loop
```

where C_i and B_i are the i^{th} columns of C and B respectively. The multiplication by A^{-1}_{kk} can thus be reduced to a matrix by matrix multiplication if the full inverse is used, or a series of n matrix by vector multiplications if the factored form is used. Again the basic

operation is that of block multiplications.

The benefit of this method is its suitability for use with vector processors. A vector processor achieves high performance through the use of a number of pipelined arithmetic units. If there are n arithmetic units then an increase in performance of at almost n times can be obtained. For example if two vectors are to be multiplied then the pipeline of arithmetic units performs the operations on individual vector elements and the multiplication is performed almost n times faster than on a non-pipelined (scalar) processor. The advantage of vector processors is that they have built in support for vector based operations. A vector by vector multiplication can be performed using a single machine instruction and it will be performed n times faster than on a scalar processor. If the matrix blocks in Figure 7.1(b) are assumed to be densely populated and stored as arrays then the block bifactorisation can be performed using only a small number of of machine instructions. An increase in performance over a scalar processor will be observed due simply to the pipelined vector processor. An even bigger increase in performance may be achieved by using either vector processors as the processing elements in a parallel machine. As the four subnetworks in Figure 7.1(b) are independent it is possible to process them in parallel and a parallel vector processor with 5 CPU's could be used to give a 3 or 4 fold increase over the performance of the single vector processor solution. The number of pipelined arithmetic units, n, depends the processor used but it is typically less than 12 for a simple vector microprocessor. Hence a single vector microprocessor could theoretically give a 12 fold increase in performance over a scalar processor. Using a parallel vector processor could give as much as a 48 fold increase in performance over the best sequential method executed on a single processor scalar machine.

The disadvantage of vector processing is that it is only efficient if operations are performed on dense vectors. When sparse matrices and vectors are used scalar processors are often found to be more efficient. To enable good speed-ups to be obtained from a parallel vector processor it is necessary to ensure that all matrix blocks are sufficiently dense for the vector processor to operate efficiently. This is not a problem with the cutset as it generally tends to be densely populated. Unfortunately the matrix blocks associated with the subnetworks are more sparsely populated and it may be inefficient to operate on these blocks using vector methods. A hybrid parallel machine consisting of both scalar and vector processors can be envisaged. The scalar processors may be used to process the subnetwork blocks whilst a vector processor may be used to process the cutset blocks, giving a highly

efficient solution. This approach has benefits for both standard parallel algorithms and for the Recursively Parallel algorithm. Under the RP scheme the main subnetworks would be processed using sparse matrix techniques on scalar processors whilst the vector processors could be used to process the minor subnetworks which constitute the cutset block. Little work has been done on using parallel vector machines and hybrid scalar/vector parallel machines for solving the linear equations associated with network problems. It would be interesting to investigate the performance of both standard and RP solution techniques on these architectures as great performance benefits may lie in store.

7.4 Summary

Although it has been demonstrated that the Recursively Parallel method exhibits good performance there is always room for improvement in any method. This chapter has considered some suggestions for improving on the methods described in this thesis to make the Recursively Parallel method more appropriate for use in power systems analysis applications.

The method presented for partitioning the network prior to RP solution works well but relies on visual inspection by the user. This is time consuming and it would be advantageous to have an automatic method for partitioning the system. Based on the existing method which utilizes a weighted elimination tree, an heuristic bin packing method has been proposed to partition systems for solution by a conventional parallel method. Unfortunately the situation is more complicated for the RP method as the heuristic partitioning must cope with multiple levels of parallelism and ensure that the chosen decomposition does not violate RBBDF constraints. Further work is required to develop the rules which will allow the automatic heuristic partitioning approach to cope with the constraints imposed by the RP solution.

Near-optimal ordering strategies are often used to optimize the elimination process during triangular solution but these strategies are sensitive to the initial network ordering and the orderings that they produced can only be considered locally optimum. Throughout the space of all $n!$ possible reorderings, some orderings may be found which are 'more optimal' than those produced by the near-optimal strategies and these can have a profound effect on improving the performance of the multiprocessor solution. It would be advantageous to find the (globally) optimum ordering if this does not require the expenditure of too much computational effort. An approach has been proposed which uses a genetic algorithm to

rapidly search the space of all possible reorderings. The problem with the method is that a fitness function is required which can be used to assess the optimality of each potential solution produced by the genetic algorithm. Optimality is characterized by the path length and fill-in produced by the ordering and this can only be determined by performing a simulated elimination using that ordering. This is computationally intensive and it would require more computational effort to find the optimum ordering than would be saved by using that ordering during elimination. Further work needs to be undertaken to find a less intensive method of assessing the fitness of potential solutions. If such a method can be found, the genetic algorithm-based optimal ordering may be useful in improving the performance of all parallel methods.

The RP method solves the network equations in a row-oriented manner but it is also possible to solve them in a block-oriented manner. Block-oriented solution is particularly suited to the use of vector processors which use pipelines of multiple arithmetic units to provide machine-level instructions for performing vector arithmetic operations. A vector microprocessor with a pipeline of n ALU's can, theoretically, show an n-fold increase in performance over an ordinary scalar microprocessor. A multiprocessor computer made from vector microprocessors could combine the performance benefits of vector computers with those of parallel computers to give speed-up which are not possible from either vector or parallel processing alone. Such a computer could be used to step over the current performance limitations and to significantly reduce the time taken to solve large sets of linear equations. Unfortunately vector processing requires vectors and matrices to be stored as arrays and the use of array storage may detract from the possible performance benefits. A hybrid parallel scalar-vector computer can be envisaged in which parallel vector processors are used to solve the dense cutset block whilst parallel scalar processors are used to solve the remainder of the matrix. This type of architecture certainly deserves further investigation.

Chapter 8

Conclusions

8.1 Conclusions

This thesis has examined how the linear equations associated with network problems may be solved using parallel computing techniques. In particular, the solution of the linear equations arising from the study of large electrical power systems has been considered. For most real-time power system simulations it is necessary to solve these equations as fast as possible and the work described here has focussed on the development of a method which offers faster and more efficient solutions than existing parallel techniques.

Parallel methods have been developed for the solution of general sets of linear equations and many more methods have been developed to solve the equations which arise from specific problems. The field of power system analysis is no exception to this and numerous researchers have attempted different methods of solving the network equations. Most of the methods use some form of Gaussian elimination and diakoptical techniques are used to partition the problem into independent parts. The fine details of the Gauss-based algorithm and the target parallel architecture are the main differences between the methods. Different Gauss methods allow different levels of parallelism to be exploited and the parallel architecture can have an effect on the efficiency of the method. Shared memory machines can give better performance due to the lower interprocessor communication overheads but cheap, off-the-shelf distributed memory machines are widely available. From the user's point of view massively parallel computers are perhaps not appropriate for a power system simulation due to the large capital outlay required in the purchase of such a machine. The aim of the work described here has been to develop a parallel solution method suitable for use with

distributed memory machines. An array of Transputers has been used as the development platform but the solution has not been developed specifically for Transputers. This platform has been used to verify the effectiveness of the method but any method which works on the Transputer array should also work on other platforms including distributed memory multiprocessors, distributed systems of workstations and multitasking sequential machines. Existing solutions are inefficient in that speed-up for either factorisation or substitution seldom exceeds a value of three or four and a large number of processors are required to achieve this speed-up. The aim was to develop a solution technique that would give greater speed-ups than existing methods with only a small number of processors. It was hoped that this could be achieved through the elimination of the speed-up saturation observed in existing solutions due to the existence of a large sequential section in the solution algorithm.

The development of the Recursively Parallel method has been described and, like existing methods, it is based on diakoptics and Gaussian elimination. The method has been devised by considering the structure of the coefficient matrix and the precedence relationships which arise from it. Using the insight provided by the elimination tree a novel coefficient matrix structure has been created by constraining the interconnection of subnetworks. This structure introduces greater independence into the treatment of cutset elements and allows the cutset to be processed in parallel as a number of subnetworks. The size of the sequential part of the method has been significantly reduced and more parallelism has been exploited.

Partitioning the system for parallel solution is simplified by resorting to the use of the elimination tree and analyzing the complexity of processing nodes. It has been demonstrated that subtrees are equivalent to subnetworks and selecting subtrees is equivalent to partitioning the network into subnetworks. Load balancing has been introduced as an important issue in parallel computing and the need for achieving a balanced load has been identified. Using the weighted elimination tree reduces the problem of load balancing to one of selecting subtrees with appropriate weights. Using an aggregate weighting technique it is possible to find subtrees of the elimination tree with nearly equal weights and an ideal balanced load is achieved when the subtrees all have equal weights. Examining the partitioned elimination tree allows a prediction to be made for the speed-up that can be obtained by solving the system in parallel. Using this technique it has been possible to partition all the test systems described in this thesis. These test systems are based on real networks and the method is powerful enough to allow partitioning of any power system network.

The performance of the RP method depends heavily on the efficiency of the data struc-

tures and algorithms used in its implementation. Suitable data structures and outline algorithms of efficient parallel tasks have been presented throughout the course of the discussion. A hybrid data storage scheme has been proposed and this allows maximum performance to be obtained by tailoring the data storage and processing to the characteristics of the individual system. This hybrid parallel data structure has suggested a way of improving the performance of existing sequential solution methods. In addition it is also possible to optimize the network partitions and data structures for either the factorisation or substitution phases of the solution thereby improving the performance of these operations. An optimal assignment of tasks to processors has been derived to minimize interprocessor communications and number of processors required whilst still allowing for easy scalability. It has also been demonstrated that the performance of the method is independent of the target architecture if a pipeline can be embedded in that architecture. This is true for most topologies apart from trees. A method of visualising the operation of the parallel program has also been described along with a strategy for accurately timing the execution of the solution and events within the solution.

The performance of the new method is very encouraging. Simulations of the method have shown its effectiveness, producing speed-ups which are better than those of many other LU-based solution methods when using the same number of processors. The method is effective even with a large number of processors and scales easily from a small number of processors to a larger numbers of processors. An actual parallel implementation on an array of Transputers concurs with the results of the simulations. The resulting speed-ups are similar to those predicted by both the theory and the simulation. Whilst the overall speed-ups and factorisation speed-ups are significantly better than those obtained from existing methods the substitution speed-ups are somewhat disappointing but still show an improvement over other methods. This highlights the difficulties of speeding-up the substitution phase. Highly efficient sequential substitution algorithms already exist and due to the short execution times of these algorithms it is difficult to exploit parallelism without introducing overheads which detract from performance. When the differences in the data structures used by the parallel and sequential solutions are ignored the performance of the substitution phase becomes similar to that of the overall and factorisation performance. Although saturation is still observed it is less significant than in other methods and is primarily due to the overheads associated with interprocessor communication.

It has been demonstrated that the performance of both the simulated and actual parallel

implementations is dependent on the load balance. Static load balancing is employed by the RP method and the load balance is adjusted by altering the partitioning of the network. The ability to predict speed-up based upon an analysis of the partitioned elimination tree allows the load balance to be assessed prior to solution. The adjustments to the load balancing strategy presented in Chapter 6 allows the optimum network partitioning, and hence load balance, to be easily determined.

The major benefit of the RP method is that it is more efficient than existing solution methods as fewer processors are required to yield comparable speed-ups. This property of the method offers important economic advantages for the user. The strategy used for assigning tasks to processors offers the advantage of easy scalability. The architectural independence of the RP method allows it to be implemented on the simplest of architectures, or on more complicated ones. The fact that the method works as well on the simple pipeline as it does on any other architecture is advantageous as the pipeline topology is cheap and easy to implement. Although the RP method works well on a distributed memory machine it should be simple to adapt the RP solution to work with shared memory machines. The method is also suitable for implementation on a distributed network of workstations. This is important when it is considered that the RP method is designed to be the engine for solving the linear equations in a power system simulation. Whilst the performance of the method on networked workstations will be poorer than on a dedicated multiprocessor, it may still be acceptable for simulation applications. If this is the case then the end user of the simulation would not necessarily have to invest in a dedicated multiprocessor but could make use of existing networked computing facilities.

Although the RP method performs as expected there is always room for improvement and suggestions have been made for further work on the method. If the RP method is to be used as a solution engine in power system analysis software then it will be necessary to implement an automatic method of partitioning the system. It has been shown that the elimination ordering has an effect on the performance of the solution and it would be advantageous to find a fast method for locating the most optimum ordering for a given system. Parallel processing techniques have been successfully applied to speed-up the solution of the network equations but vector processors also show promise for accelerating solutions. Significant increases in the speed of solution could be achieved by combining the techniques of parallel processing and vector processing through the use of a parallel vector processor architecture and this approach to solution deserves further research.

In conclusion, the RP method proposed here has proved to be an effective method for solving linear network equations in parallel. It is more efficient and produces better speed-ups than many other parallel solutions. For the method to be of any use in a real-time power system simulation it must solve the network equations as fast as possible on each iteration, and each iteration must completed in a time shorter than the integration step of the simulation. Given that the integration step is of the order of 1 second and that the RP method solves a 1624 node network in 36 milliseconds, the RP method is certainly suitable for use as part of a real-time power system simulator.

Bibliography

[1] C.A. Gross, *Power Systems Analysis.* Wiley, second ed., 1986.

[2] A. Brameller, R.N. Allan, and Y.M. Hamam, *Sparsity - Its practical application to systems analysis.* Pitman, 1976.

[3] W.F. Tinney and J.W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorisation," *Proceedings of the IEEE*, November 1967.

[4] B. Stott, "Power system dynamic response calculations," *Proceedings of the IEEE*, pp. 219–241, February 1979.

[5] Balu et. al, "Online power system security analysis," *Proceedings of IEEE*, pp. 260–280, February 1992.

[6] I. Susmago et al., "Development of a large scale dispatcher training simulator and training results," *IEEE Transactions on Power Systems*, pp. 67–75, May 1986.

[7] H. Biglari et al, "A dispatcher training simulator design with multi-purpose interfaces," *IEEE Transactions on Power Apparatus and Systems*, pp. 1276–1280, June 1985.

[8] R. Podmore et al, "An advanced dispatcher training simulator," *IEEE Transactions on Power Apparatus and Systems*, pp. 17–25, January 1982.

[9] J. Arillaga and C.P. Arnold, *Computer Analysis Of Power Systems.* Wiley, 1990.

[10] C. Lazou, *Supercomputers And Their Use.* Oxford Science Publications, 1988.

[11] A. Trew and G. Wilson, eds., *Past, Present and Parallel: A survey of available parallel computing systems.* Springer-Verlag, 1991.

BIBLIOGRAPHY

[12] G. Sabot, *The Paralation Model - Architecture independent parallel programming.* MIT Press, 1988.

[13] G. S. Almasi and A.J. Gottlieb, eds., *Highly Parallel Computing.* Benjamin/Cummings, second ed., 1994.

[14] J.P. Hayes, *Computer Architecture And Organisation.* McGraw-Hill, 1988.

[15] K.H. Hwang and F.A. Briggs, *Computer Architecture And Parallel Processing.* McGraw-Hill, 1984.

[16] M.J. Flynn, "Very high speed computing systems," *Proceedings of IEEE*, pp. 1901–1909, 1966.

[17] K.H. Hwang, *Advanced Computer Architecture - Parallelism, Scalability, Programmability.* McGraw Hill Series In Computer Science, McGraw Hill, 1993.

[18] A.L. DeCegama, *Parallel Processing Architectures And VLSI Hardware*, vol. 1. Prentice Hall, 1989.

[19] T.G. Lewis and H. El-Rewini, *Introduction To Parallel Computing.* Prentice Hall, 1992.

[20] J. Hinton and A. Pinder, *Transputer Hardware And System Design.* Prentice Hall International, 1993.

[21] I. Graham and T. King, *The Transputer Handbook.* Prentice Hall, second ed., 1990.

[22] R. Taylor, *Selected Notes On Transputers.* University of York, 1991.

[23] M. Minsky and S. Papert, "On some associative parallel and analog computations," in *Associative Information Techniques* (E.J. Jacks, ed.), Elsevier, 1971.

[24] G.A. Amdahl, "Limits of expectation," *International Journal of Supercomputing*, pp. 88–94, Spring 1988.

[25] E. Gelenbe, *Multiprocessor Performance.* Wiley Series In Parallel Computing, Wiley, 1989.

[26] A. Abur, "A parallel scheme for the forward/backward substitutions in solving sparse linear equations," *IEEE Transactions on Power Systems*, pp. 1471–1478, November 1988.

[27] G. Cafaro, P. Pugliese, and F. Vacca, "Parallel solution of torn network equations," *Electrical Power and Energy Systems*, pp. 131–138, July 1984.

[28] I.C. Decker, D.M. Falc ao, and E. Kaszkurewicz, "Parallel implementation of a power system dynamic simulation methodology using the conjugate gradient method," *IEEE Transactions on Power Systems*, pp. 458–465, February 1992.

[29] T. Berry, A.R. Daniels, and R.W. Dunn, "Parallel processing of sparse power system equations," *IEE Proceedings C*, pp. 68–74, January 1994.

[30] A.A. El-Kieb, H. Ding, and D. Maratukulam, "A parallel load flow algorithm," *Electric Power Systems Research*, pp. 203–208, 1994.

[31] J. Fong and C. Pottle, "Parallel processing of power system analysis problems via simple parallel microcomputer structures," *IEEE Transactions on Power Apparatus and Systems*, pp. 1834–1841, September/October 1978.

[32] K. Lau, D.J. Tylavsky, and A. Bose, "Coarse grain scheduling in parallel triangular factorisation and solution of power system matrices," *IEEE Transactions on Power Systems*, pp. 708–714, March 1991.

[33] S. Lin and J.E. Van Ness, "Parallel solution of sparse algebraic equations," *IEEE Transactions on Power Systems*, vol. 9, pp. 1117–1125, May 1994.

[34] M. Rafian, M.J.H. Sterling, and M.R. Irving, "Parallel processor algorithm for power system simulation," *IEE Proceedings Part C*, pp. 285–290, July 1988.

[35] D. Yu and H. Wang, "A new parallel LU decomposition method," *IEEE Transactions on Power Systems*, pp. 303–310, February 1990.

[36] T. Berry, K.W. Chan, and R.W. Dunn, "A parallel computer algorithm for real-time electro-mechanical transient power system simulation," in *Proceedings of 27 Universities Power Engineering Conference*, vol. 1, pp. 32–12, University of Bath, 1992. University of Bath, ENGLAND, 23-25 September 1992.

[37] F.L. Alvarado, D.C. Yu, and R. Betancourt, "Partitioned sparse A^{-1} methods," *IEEE Transactions on Power Systems*, pp. 452–459, May 1990.

BIBLIOGRAPHY

[38] M.La Scala, G. Sblendorio, and R. Sbrizzai, "Parallel in-time implementations of transient stability simulations on a Transputer network," *IEEE Transactions on Power Systems*, pp. 1117–1125, May 1994.

[39] G.P. Granelli, M. Montagna, M. La Scala, and F. Torelli, "Relaxation-Newton methods for transient stability analysis on a vector/parallel computer," *IEEE Transactions on Power Systems*, pp. 637–643, May 1994.

[40] S.Y. Lee, H.D. Chiang, K.G. Lee, and B.Y. Ku, "Parallel power system transient stability analysis on hypercube multiprocessors," in *Proceedings of the IEEE Power Industry Computer Applications Conference*, May 1989.

[41] F. Sato, A.V. Garcia, and A. Monitcelli, "Parallel implementation of probabilistic short-circuit analysis by the Monte-Carlo approach," *IEEE Transactions on Power Systems*, no. 2, pp. 826–832, 1994.

[42] A.A. El-Kieb, J. Nieplocha H. Singh D.J. Maratukulam M.K. Celik, and A. Abur, "A decomposed state estimation technique suitable for parallel processor implementation," *IEEE Transactions on Power Systems*, no. 3, pp. 1088–1097, 1992.

[43] D.M. Falc ao, E. Kaszkurewicz, and H.L.S Almeida, "Application of parallel processing techniques to the simulation of power-system electromagnetic transients," *IEEE Transactions on Power Systems*, no. 1, pp. 90–96, 1993.

[44] A. Padhila and A. Morelato, "A W-matrix methodology for solving sparse network equations on multiprocessor computers," *IEEE Transactions on Power Systems*, pp. 1023–1036, August 1992.

[45] D.J. Tylavsky, S. Nagaraj, and P.E. Crouch, "Parallel-vector processing synergy and frequency domain transient stability simulations," *Electric Power Systems*, pp. 89–97, 1993.

[46] J.E. VanNess and G. Molina, "The use of multiple factoring in the parallel solution of algebraic equations," *IEEE Transactions on Power Apparatus and Systems*, pp. 3433–3438, October 1983.

[47] H.H. Happ, *Piecewise Methods And Applications To Power Systems.* Wiley, 1980.

[48] G. Kron, "A set of principles to interconnect the solutions of a physical system," *Journal of Applied Physics*, pp. 965–980, 1953.

[49] J. Bialek, "Parallel solution of torn networks for power system simulation," in *Proceedings of 6th International Conference on Present Day Problems In Power Systems*, vol. 2, pp. 75–82, 1993. Gliwice, Poland.

[50] IEEE Committee Report by a Task Force of the Computer and Analytical Methods Subcommittee of the Power Systems Engineering Committee, "Parallel Processing in Power Systems Computation," *IEEE Transactions on Power Systems*, pp. 629–638, May 1992.

[51] B. Carré, *Graphs and networks.* Oxford applied mathematics and computing science series, Oxford University Press, 1979.

[52] J.W.H. Liu, "The role of elimination trees in sparse factorisation," *SIAM Journal of Matrix Analysis Applications*, pp. 134–172, January 1990.

[53] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Transactions on Mathematical Software*, pp. 256–276, 1982.

[54] J.A. Jess and G.H. Kees, "A data structure for parallel LU decomposition," *IEEE Transactions on Computing*, pp. 231–239, March 1982.

[55] W.F. Tinney, V. Brandwajn, and S.M. Chan, "Sparse vector methods," *IEEE Transactions on Power Apparatus and Systems*, pp. 295–301, February 1985.

[56] R. Betancourt, "Efficient parallel processing algorithm for inverting matrices with random sparsity," *IEE Proceedings Part E*, pp. 235–240, July 1986.

[57] A.J.E. Taylor, *Techniques For Power System Simulation Using Multiple Processors.* PhD thesis, University of Durham, UK, 1990.

[58] R. Sedgewick, *Algorithms, 2nd edition.* Addison-Wesley, 1988.

[59] I. Peterson, *The Mathematical Tourist - Snapshots of modern mathematics.* W. H. Freeman & Co., 1988.

[60] I.S. Duff, A.M. Erisman, and J.K. Reid, *Direct Methods For Sparse Matrices.* Oxford Press, 1986.

[61] A. Kelley and I. Pohl, *A Book on C.* Benjamin/Cummings, second ed., 1984.

[62] A. Jennings and J.J. McKeown, *Matrix Computations.* Wiley, second ed., 1992.

[63] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction To Parallel Computing - Design and analysis of algorithms.* Benjamin/Cummings, 1994.

[64] R.L. Burden and J.D. Faires, *Numerical Analysis.* PWS-Kent, fourth ed., 1988.

[65] G.H. Golub and C.F. Van Loan, *Matrix Computations.* North Oxford Academic, first ed., 1983.

[66] M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill Series in Supercomputing and Artificial Intelligence, McGraw-Hill, 1987.

[67] M.T. Heath, E. Ng, and B.W. Peyton, "Parallel algorithms for sparse linear systems," *SIAM Review*, pp. 420–460, September 1991.

[68] D.J. Tylavsky, "Quadrant interlocking factorization: a form of block LU factorization," *Proceedings of the IEEE*, 1986.

[69] J. Bialek and D.J. Grey, "The application of clustering and factorisation tree techniques for the parallel solution of sparse network equations," *IEE Proceedings Part C*, pp. 609–616, November 1994.

[70] J.S. Chai and A. Bose, "Bottlenecks in parallel algorithms for power system stability analysis," *IEEE Transactions on Power Systems*, pp. 9–15, February 1993.

[71] A. George and W.H. Liu, "An optimal algorithm for symbolic factorization of symmetric matrices," *SIAM Journal of Computing*, pp. 583–593, 1980.

[72] A. George and W.H. Liu, *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[73] G.A. Geist and E. Ng, "Task scheduling for parallel sparse Cholesky factorisation," *International Journal of Parallel Processing*, no. 4, pp. 291–313, 1989.

[74] J.W.H. Liu, "Reordering sparse matrices for parallel elimination," *Parallel Computing*, pp. 73–91, 1989.

[75] F.F. Wu, "Solution of large scale networks by tearing," *IEEE Transactions On Circuits and Systems*, pp. 706–713, December 1976.

[76] L.O. Chua and L.K. Chen, "Diakoptic and generalized hybrid analysis," *IEEE Transactions on Circuits and Systems*, pp. 706–713, December 1976.

[77] Z. Boming, X. Niande, and S. Wang, "Unified piecewise solution of power system networks combining both branch cutting and node tearing," *Electrical Power and Energy Systems*, pp. 238–288, October 1989.

[78] I.S. Duff, "A survey of sparse matrix research," *Proceedings of the IEEE*, pp. 500–535, April 1977.

[79] J. Bialek, D.J. Grey, and J.R. Bumby, "Parallel decomposed network solution method for power system simulation," in *Proceedings of 27th Universities Power Engineering Conference*, vol. 1, pp. 193–196, University of Bath, 1992. University of Bath, ENGLAND, 23-25 September 1992.

[80] M.R. Irving and M.J.H. Sterling, "Optimal network tearing using simulated annealing," *IEE Proceedings Part C*, pp. 69–72, January 1990.

[81] R.W. Dunn. Personal Communication, 1993. Stafford.

[82] C. Ashcraft, S.C. Eisenstat, and J.W.H. Liu, "A fan-in algorithm for distributed sparse numerical factorisation," *SIAM Journal on Scientific and Statistical Computing*, pp. 593–599, 1990.

[83] F. Berman and L. Snyder, "On mapping parallel algorithms into parallel architectures," *Journal of Parallel and Distributed Computing*, pp. 439–458, 1987.

[84] R.L. Graham, "Bounds on multiprocessor timing anomalies," *SIAM Journal of Applied Mathematics*, pp. 416–429, March 1966.

[85] G.F. Coulouris and J. Dollimore, *Distributed Systems.* Addison Wesley International Computer Science Series, Addison Wesley, 1988.

[86] T. Oyama, T. Kitshara, and Y. Serizana, "Parallel processing for power system analysis using band matrix," *IEEE Transactions on Power Systems*, pp. 1010–1016, August 1990.

BIBLIOGRAPHY

[87] J. Bialek and D.J. Grey, "An automatic clustering algorithm using factorisation tree for parallel power system simulation," in *Proceedings of MELECON 1994*, 1994.

[88] J.A. George, J.W.H. Liu, and E.G.Y. Ng, "Communication results for parallel sparse Cholesky factorisation on a hypercube," *Parallel Computing*, pp. 287–298, 1989.

[89] M.A. Weiss, *Data Structures And Algorithms.* Benjamin/Cummings, 1992.

[90] H.S. Wilf, *Algorithms And Complexity.* Prentice-Hall, 1986.

[91] A. Burns and A. Wellings, *Real-time systems and their programming languages.* Addison-Wesley International Computer Science Series, Addison-Wesley, 1989.

[92] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, pp. 666–777, 1978.

[93] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice-Hall International ?, 1980.

[94] P. Brinch Hansen, "Distributed Processes: A concurrent programming concept," *ACM Transactions on Programming Languages*, no. 4, pp. 405–430, 1978.

[95] K.M. Chandy and S. Taylor, *An introduction to parallel programming.* Jones and Bartlett Publishers, 1992.

[96] J. Bialek and D.J. Grey, "A mutated tree architecture for real-time parallel power system simulation," in *Proceedings of 28th Universities Power Engineering Conference*, vol. 2, pp. 458–461, University of Staffordshire, 1993. University of Staffordshire, ENGLAND, 21-23 September 1993.

[97] D.P. Helmbold and C.E. McDowell, "Modeling Speedup (n) greater than n," *IEEE Transactions on Parallel and Distributed Systems*, pp. 250–256, April 1990.

[98] D. Parkinson, "Parallel efficiency can be greater than unity," *Parallel Computing*, pp. 261–262, 1986.

[99] B.W. Weide, "Modeling unusual behaviour of parallel algorithms," *IEEE Transactions on Computers*, pp. 1126–1130, November 1982.

[100] W.A. Kornfeld, "Combinatorially implosive algorithms," *Communications of the ACM*, pp. 734–738, October 1982.

[101] L.A. Crowl, "How to measure, present and compare parallel performance," *IEEE Parallel and Distributed Technology*, pp. 9–25, Spring 1994.

[102] D.E. Goldberg, *Genetic Algorithms.* Addison-Wesley, 1989.

[103] Inmos, "IMS D7314A ANSI C Compiler Language Reference Manual," 1992.

[104] W.D. Stephenson, *Elements Of Power System Analysis.* McGraw Hill, third ed., 1975.

[105] B.M. Weedy, *Electric Power Systems.* Wiley, third ed., 1987.

[106] H. El-Rewini and T.G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel And Distributed Computing*, pp. 138–153, 1990.

[107] N.T. Karonis, "Timing parallel programs that use message passing," *Journal of Parallel And Distributed Computing*, pp. 29–36, 1992.

Appendix A

The INMOS Transputer

The INMOS Transputer first entered the microprocessor marketplace in 1982 with the release of the T212 Transputer. It was designed to be a general purpose microprocessor which could be used as a building block for creating high performance parallel processing systems. Nowadays the transputer is a cheap component that can be connected into large arrays to form a high performance machine. Taylor [22] notes that the following three main features can be identified in the design of the transputer;

- Support for multitasking so that many logically concurrent tasks may reside on the same processor.
- Support for synchronous communications so that tasks on the same or different processors may communicate efficiently with one another.
- Modular design and support for the construction of systems of processors.

An additional feature not explicitly noted by Taylor is that of creating a hardware platform for an implementation of Hoare's [92, 93] Communicating Sequential Processes paradigm of parallel computing.

The architecture of the transputer is such that it allows the size of a system to be easily scaled. The multi-tasking scheduler built into the chip allows multiple tasks to be executed on a single transputer. Large parallel programs may be executed on one or many transputers.

The communication facilities that are provided allow synchronous communication between tasks on the same or different processors. The synchronous nature of the communication primitives requires messages to be acknowledged and this gives a reliable protocol

for data transfer. Complete multiprocessor systems are built simply by interconnecting the communications links of the constituent transputers. Communications are point-to-point and each serial link consists of two unidirectional communication channels, one in either direction. The links operate concurrently with the processor in order to maximize performance and maximum data transfer rate is 20 Mbit/sec.

To address the modular design issue the transputer incorporates many of the supporting systems required by other microprocessors onto the same piece of silicon as the processor itself [20]. Each transputer consists of a microprocessor, serial communication links, fast cache memory, external memory interfacing, floating point coprocessor, real-time clocks and a hardware implemented multi-tasking scheduler. As so much is implemented on the transputer itself, systems can be built with only a few external components. Printed circuit boards can be small and power consumption is lower than in many comparable designs which use other processors. All the resources required by a transputer were designed to be local and not globally shared, eliminating the need for complicated bus interfaces. In fact the only external signals a transputer needs to enable it to boot and run a program from an attached ROM are the power, ground and clock signals. This makes the building of large systems straightforward.

A.1 The Architecture of the Transputer

The first generation of transputers, known as the Txxx family of processors, are essentially similar in their design. Although three distinct generations exist with slightly different characteristics, all generations share the same basic architecture and communication interfaces and have similar instruction sets. Figure A.1 illustrates the basic architecture of the Txxx series. In recent months the first of the second generation transputers, the T9000, has been released to market. This has a different architecture to the Txxx family and is not directly compatible with the earlier processors. It claims to give an order of magnitude greater performance over the Txxx family but problems are being experienced with this processor and it does not yet perform as well as expected. As it has only just been released it was not available for use in this research project and will not be considered further.

The Transputer processors are based around the RISC philosophy. Each instruction is 8 bits long, consisting of a 4 bit instruction code and a 4 bit operand. This allows 78% of all instructions to be coded in a single byte [22] and each memory access fetches 3 or 4

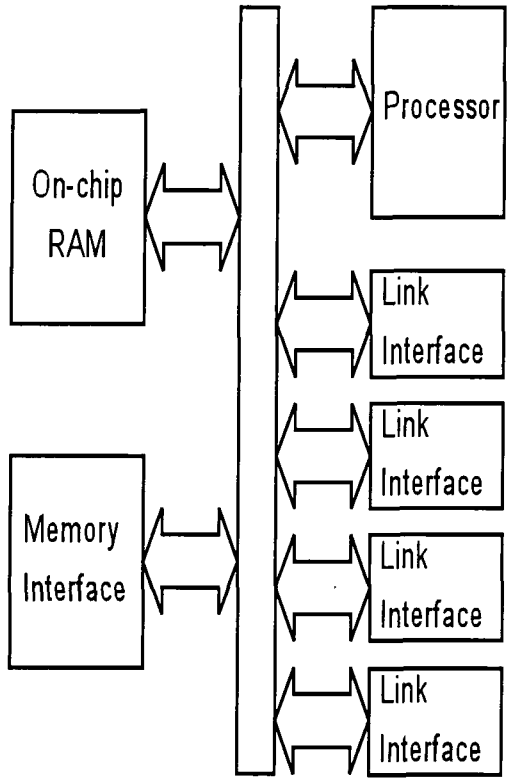


Figure A.1: Basic internal architecture of a Transputer

instructions. This makes the transputer's instruction set extremely efficient. The transputer also has a minimal register set which speeds up the context switching that occurs during multitasking.

Depending upon the processor, 2, 4 or 8 KBytes of memory are included onboard the transputer. This memory is single cycle static memory and access to it is extremely fast. External memory addresses are contiguous with internal addresses and to improve performance the external memory interface is only utilized when an invalid internal memory address is generated. The external memory interface is designed such that external memory systems may be connected with the minimum of extra components. Given the compact instruction set and fast internal memory it is possible to use transputers without external memory, although this severely limits program size and is insufficient for many applications.

Each transputer has four serial communications links and each link may be connected to any link of any other processor. Communication channels (Section A.2.1) between processors are implemented across these links. The hardware links themselves are implemented using direct memory access to allow high speed data transfer concurrently with the operation of the processor. Communication between tasks on the same processor simply requires

the copying of information held in the memory. This is implemented using an efficient block move instruction in the instruction set.

Most computer systems provide multitasking through the use of a software scheduling kernel. The transputer implements this multitasking scheduler in hardware for maximum efficiency. Executable tasks are arranged into two process queues; one for low priority processes and one for high priority processes. Low priority tasks are given a 1024 microsecond timeslice whilst high priority processes are allowed to execute until they become blocked. Tasks may block if they are waiting for a communication, waiting on a timer or waiting on an interrupt. When a task blocks, or its timeslice expires, it is placed at the bottom of its process queue and the task at the head of the process queue then executes. If the high priority process queue becomes non-empty at any time then the low priority task currently executing is pre-empted and the high priority task is executed until it blocks. If any further high priority tasks are available they will run until the high priority queue is empty. At this point the processor returns to handling the low priority tasks.

The transputer has two hardware timers onboard which can be accessed by the programmer. These timers are autonomous and run concurrently with the processor. The high priority timer ticks every 64 microseconds whilst the low priority timer ticks every 1024 microseconds. These timer speeds are independent of the transputer's clock speed. If a high priority task makes a call to the timer then it accesses the high priority timer whilst low priority tasks access the low priority timer. The timers are used by the scheduler to control timeslicing and may be used by the programmer to time events or implement delays.

A.1.1 The T2 Family

The T2 family of Transputers consists of the T212, T222 and T225 processors. The basic architecture is that of Figure A.1 and all members of this family are 16 bit processors. The T212 was the first Transputer released and has 2KB onboard RAM, 10 MBit/sec link speed and 20 MHz clock speed. The T222 has 4KB onboard RAM, 20 MHz clock speed and 20 MBit/sec link speed. The T225 processor was the last in this family and is identical to the T222 but with a clock speed of 30 MHz. This gave the T225 a bidirectional data transfer rate across its links of 2.4 MByte/sec and the processor can achieve 0.06 MFLOPS.

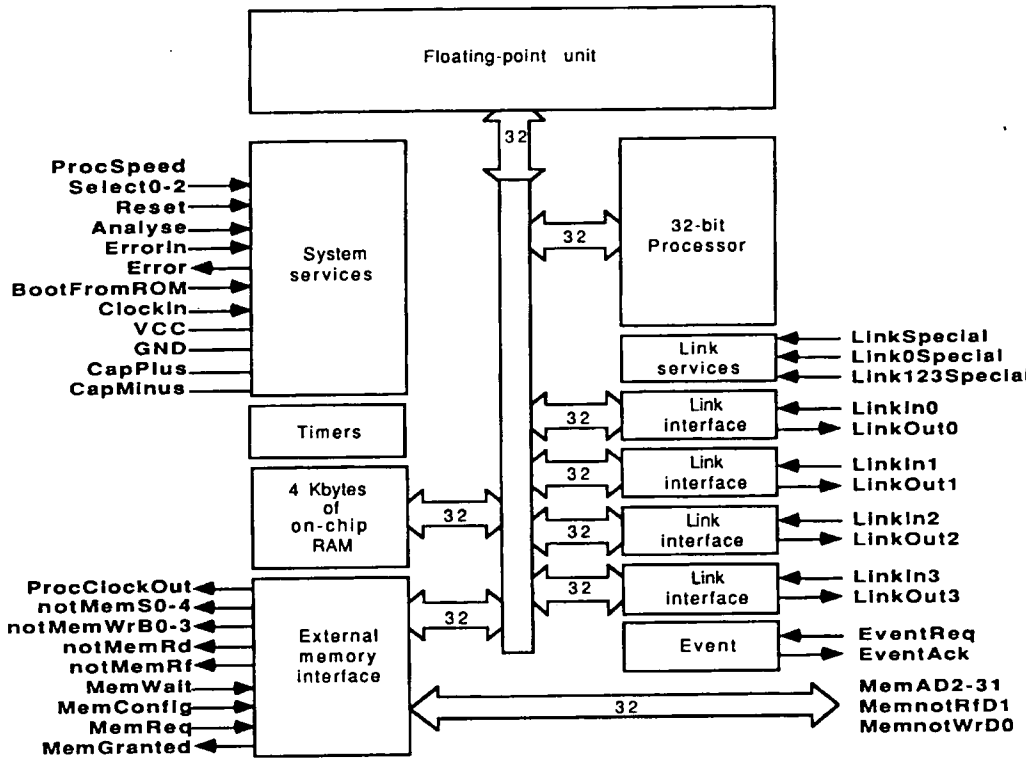


Figure A.2: Basic internal architecture of a the T8 Transputer family

A.1.2 The T4 Family

The T4 family consists of the T414 and T425 processors. The basic architecture is again similar to that of Figure A.1 but the T4 family are 32 bit processors. The instruction set was modified to improve floating point performance. The T414 runs at a clock speed of 20 MHz and incorporates 2KB of onboard RAM. The communication links operate at 20 MBit/sec. The T425 is capable of running at 30 MHz and has 4KB of onboard RAM. Enhanced links are incorporated which are capable of a bidirectional transfer rate of 2.4 MByte/sec. An enhanced instruction set improves error checking. The T425 is capable of 30 MIPS and 0.13 MFLOP.

A.1.3 The T8 Family

The 32-bit T8 family is the last of the first generation of Transputers and includes the T800, T801 and T805 processors. The architecture of this family includes a 64 bit floating point coprocessor unit and the basic architecture is shown in Figure A.2. The floating point unit (FPU) gives single and double precision floating point operations to the ANSI-IEEE 734 standard. The addition of the FPU makes this family of processors well suited to compu-

tationally intensive scientific computing applications.

The T800 was the first processor of the T8 family and is basically a T425 with the addition of the FPU and an expanded instruction set. Clock speed is 30 MHz and 2.4 MByte/sec link transfer rate is achievable.

The T805 is an improved T800 offering an enhanced memory interface and improved interrupt handling. The instruction set is also enhanced to facilitate debugging. Running at 30 MHz, the T805 is capable of 30 MIPS and 4.3 MFLOPS peak. The sustained rate for floating point operations is 3.3 MFLOPS, making the T805 a powerful processor in its own right.

Both the T800 and T805 have an external memory access rate limited to 40 MByte/sec due to their multiplexed data and address busses. The T801 is a repackaged T805 which has the address bus and data bus separated allowing an external memory access rate of 60 MByte/sec.

A.2 Programming the Transputer

A.2.1 Tasks and Channels

According to the CSP paradigm [92, 93], parallel programs are made up of sequential modules called *tasks*. Each task is an autonomously executing unit but two tasks may synchronise their operation or share data through communication. An explicit message passed between the two tasks is used to transfer data and the communication automatically forces them to synchronise their operations as neither task can continue until the communication is complete. Tasks can run concurrently on separate processors or may reside on the same processor and execute through timeshared multitasking. A typical transputer program consists of both concurrent and timeshared tasks. A simple configuration file is used to describe how tasks are placed on the available processors.

Communications between tasks are performed using *channels*. A channel is a logical, unidirectional communication link which exists between two tasks. In an intertask communication the sending task inserts data into the channel at one end and the receiving task removes the data at the other end. Channels can exist between tasks on two different processors or between tasks on the same processor. A channel between tasks on the same processor is known as a soft channel and is implemented using memory copy instructions. If the tasks reside on different processors then the channel is assigned to the physical con-

nection between the processors and is known as a hard channel. The Transputer's links can accommodate two channels, one in each direction. As there are only four links per processor it would appear that only four pairs of hard channels may be assigned by each processor. It is possible to increase this number through the use of channel multiplexing and virtual channels. As far as the programmer is concerned virtual channels are identical to hard channels in their use and operation. Many virtual channels may be assigned by each processor and more than two virtual channels may be assigned to each Transputer link. This is achieved by multiplexing the data from the virtual channel communications and passing it through the two hard channels which are assigned to the link. Demultiplexing at the receiving end splits the data up again and maintains the appearance of virtual channels passing across the link. When two processors which are not directly connected need to communicate a virtual channel between the two processors may be used. The virtual channel is implemented by passing messages from the sender to the receiver via intermediate processors. The latest versions of the INMOS Toolsets [103] incorporate virtual channels and provide routing software which automatically handles communications involving processors which are not directly linked.

A.2.2 Programming Languages

Occam was developed by INMOS as the language for programming the Transputer. It provides a strict implementation of Hoare's Communicating Sequential Processes (CSP) paradigm [92, 93] and allows all aspects of parallelism to be easily expressed. Occam is a small and somewhat terse language that allows the parallelism in a problem to be expressed in terms of jobs which must be performed in parallel and jobs which must be performed sequentially. The concept of a task still exists in Occam but fine grain parallelism is also possible as the language allows parallel execution of individual statements. In fact Occam views a program as an hierarchical arrangement of tasks and communication between tasks is synchronous via channels. Guarded commands are the other main feature of the language and these are implemented through the ALT construct. A number of C and Fortran compilers are available and compilers for other languages such as Ada, Modula-2, Lisp and Parlog are available commercially, although these are not very widely used. All the C and Fortran compilers provide extensions to standard versions of these languages in terms of functions for exploiting parallelism and performing communications. Parallel implementations of C are based on extensions to standard ANSI C whilst parallel versions

of Fortran are usually based on the Fortran 77 standard.

The INMOS C Toolset was used throughout this project and all the code written for the transputer system was written in INMOS' implementation of parallel C. This is an extended ANSI C which incorporates channel communication functions, task creation and management functions, functions to control the scheduling of tasks and functions to access the other Transputer features such as event handling and the hardware timers. Each parallel task can be written as a separate C *main()* routine and compiled individually into binary units. A configuration language is also provided to create the configuration files which describe how the tasks in the system are connected to form the complete program. Once the program has been configured the individual task binaries are linked to produce a single binary program file which may be loaded on to the transputer array. The benefit of using the C language is that existing sequential code written in C can easily be ported to the parallel environment, providing a much quicker route to software implementation than through the use of Occam. The latest version of the C Toolset [103] was used throughout this project and this provided support for virtual channels and channel routing.

A.3 Building Parallel Systems with the Transputer

A.3.1 The TRAM Standard

To enable the easy building of scaleable parallel systems INMOS have created a modular system for building Transputer based machines. This standard is based around the use of Transputer Applications Modules (TRAM's) and TRAM motherboards. The TRAM is a small circuit board measuring 3.6 inches by 1.1 inches. Each TRAM hosts a single Transputer, RAM and interfacing logic and is a complete computer in its own right. All TRAM's have a simple 16 pin interface which allows them to be connected to a motherboard and the power and control signals from the motherboard are passed through this 16 pin interface. Processors on different TRAM's are connected through their links via the motherboard. Many motherboards configure two links of each processor into a pipeline and take the remaining links to an external patch board. Different interconnections are built using jumper leads plugged into the patch board. Some motherboards provide an additional C004 reconfigurable electronic crosspoint switch to which the spare links are connected. The settings of this switch can be controlled using the vendor-supplied software and this allows different interconnection networks to be established without the need to physically rewire

the machine. The C004 switch is a static switching device and interconnection topologies cannot be changed whilst a program is running.

A TRAM motherboard must be interfaced to a host machine. Motherboard cards are usually designed so that they may reside within the host computer enclosure. PC's and Sun workstations are the usual hosts although other hosts may be used. Connecting the TRAM's to the host via the motherboard-host interface allows the transputers to utilise the facilities offered by the host in terms of disk storage, screen output and keyboard input. All I/O communications between the transputers and the host must be performed via the first processor in the network as this is the *only* processor which connects directly to the host. Any other processor that wishes to communicate with the host must use the first processor as an intermediary.

A.3.2 The Experimental Setup

The parallel computing system used throughout the duration of this research project consisted of 16 INMOS T805 30MHz Transputers and one INMOS T805 20 MHz Transputer. The 20 MHz processor was supplied with 16 MB of fast RAM and was used as the root processor in the Transputer network. Fifteen of the 30 MHz processors were supplied with 1 MB of fast RAM whilst the other 30 MHz processor was equipped with 4 MB of RAM. All of the Transputers were mounted on two INMOS B008 compatible motherboards and hosted by an IBM PC AT clone. Each motherboard could accommodate up to 10 Transputers and was equipped with an electronic crosspoint switch which allowed the Transputer interconnection network to be reconfigured from software. An overview of the machine used for the experiments described in this thesis is shown in Figure A.3.

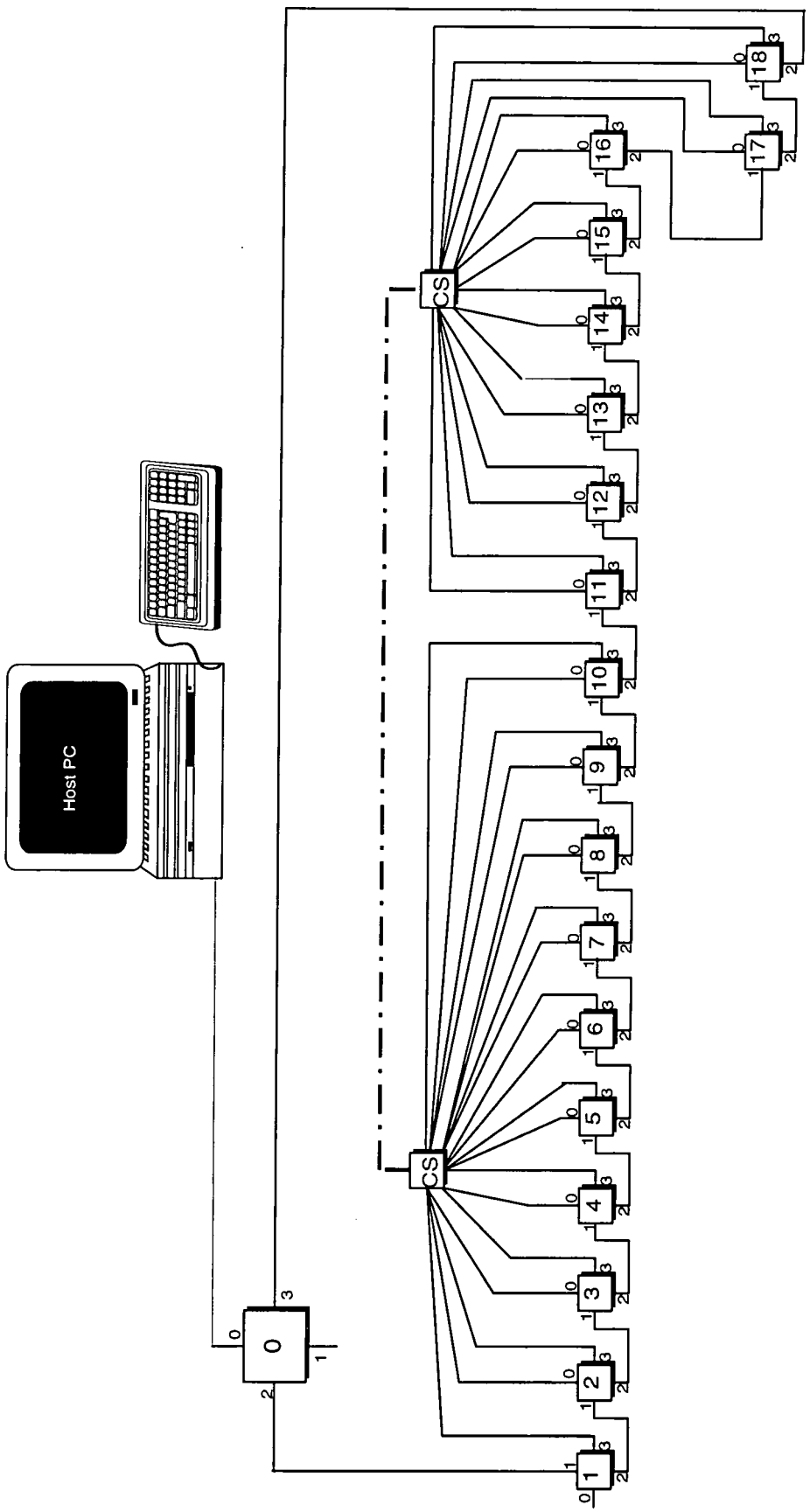


Figure A.3: Overview of the experimental Transputer-based parallel machine

Appendix B

Derivation Of The Models of Power System Elements

B.1 The Generator Model

Consider a non-salient two pole machine which has the rotor field winding supplied by a constant current source. If magnetic saturation is ignored and the rotor spins at a constant velocity balanced three phase sinusoidal voltages will be induced in the stator windings. These voltages are independent of the stator currents as the rotor field current is constant - hence they can be modelled as ideal voltage sources. The stator windings, separated spatially by 120 °, are inductively coupled with equal mutual impedances, Z_m. The self-impedances, Z_s, of the stator windings are also equal. Under these conditions the generator can be modelled as an equivalent circuit consisting of an ideal voltage source driving an impedance, one for each phase. Figure B.1 shows the equivalent circuit model of the generator, consisting of the three ideal voltage sources connected through the self impedances, Z_s and coupled by the mutual impedances, Z_m.

Assuming balanced three phase steady state conditions, only positive sequence networks [104] are involved. This allows circuit resistance and other machine losses to be ignored for the sake of clarity and the system can now be modelled by the equivalent circuit of Figure B.2.

Resorting to Kirchoff's Laws

$$\bar{E} = jX_d\bar{I} + \bar{V} \tag{B.1}$$

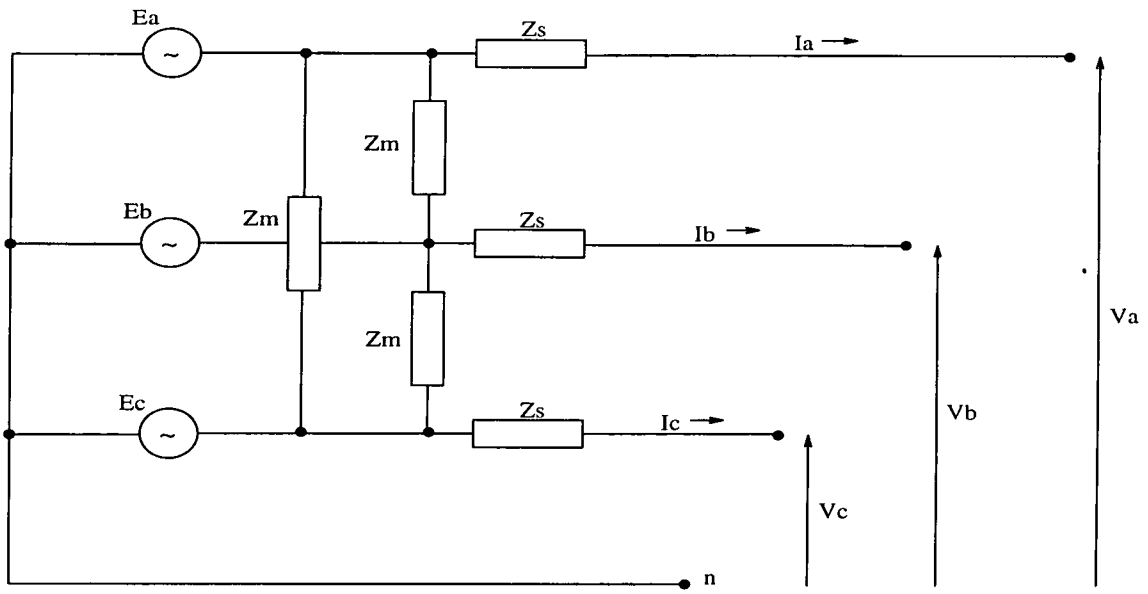


Figure B.1: Equivalent circuit model of the synchronous generator

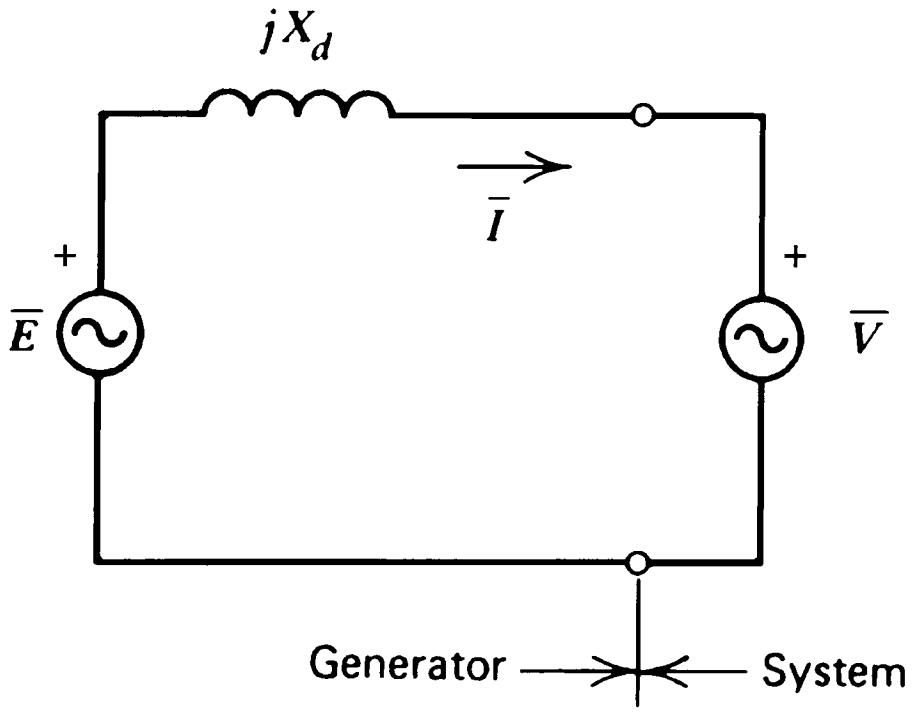


Figure B.2: Positive sequence equivalent circuit model of the synchronous generator

where

$\bar{V} = V\angle 0° =$ - generator terminal voltage

$\bar{E} = E\angle\delta =$ - stator voltage

$\delta =$ power angle

$X_d =$ direct axis synchronous reactance - this is reactive component of stator self-impedance [1]

The complex power for the system is given by

$$S = VI^* \tag{B.2}$$

Substituting from (B.1)

$$S = \bar{V}\left[\frac{\bar{E} - \bar{V}}{jX_d}\right]^* \tag{B.3}$$

Simplifying yields

$$\bar{S} = \frac{VE}{jX_d}\angle\underline{90 - \delta} - j\frac{V^2}{X_d} = \frac{VE}{X_d}\sin\delta + j\left[\frac{VE}{X_d}\cos\delta - \frac{V^2}{X_d}\right] \tag{B.4}$$

The real power, P, is the real part of (B.4)

$$P = \Re[S] = \frac{VE}{X_d}\sin\delta \tag{B.5}$$

and reactive power, Q, is the imaginary part of (B.4)

$$Q = \Im[S] = \frac{VE}{X_d}\cos\delta - \frac{V^2}{X_d} \tag{B.6}$$

Taken together (B.1) to (B.6) provide a mathematical description of the synchronous generator which allows the determination of real and reactive power delivered and the voltages and currents which can be measured both within the machine and at its terminals.

B.2 The Transmission Line Model

Three different equivalent circuit models of a transmission line may be derived depending upon the length of the line. This section discusses these models in detail. In discussing transmission lines it should be noted that parameters such as resistance and impedance are

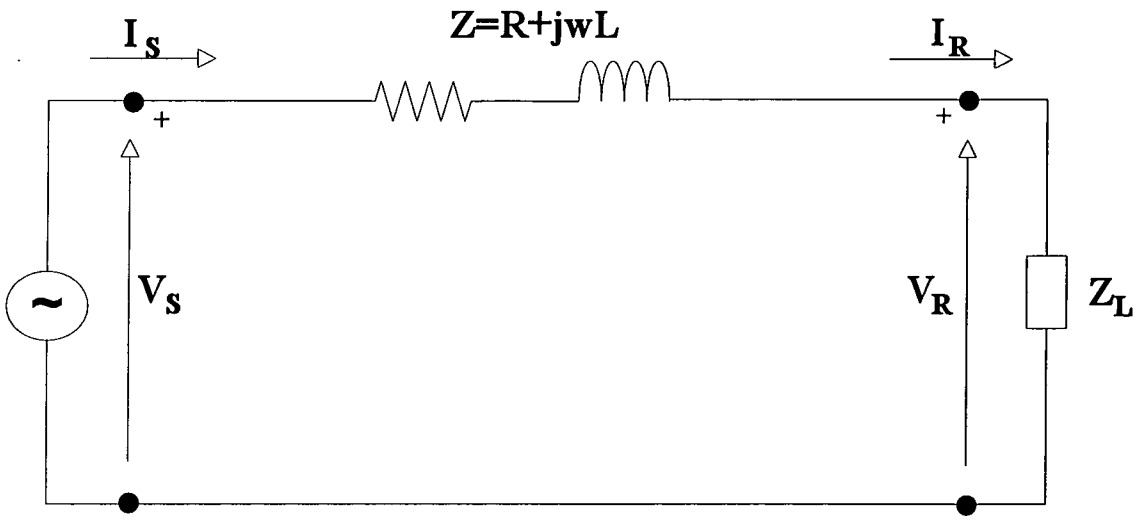


Figure B.3: Single phase equivalent circuit model of a short transmission line

distributed throughout the length of the line. The models can be derived by considering these parameters to be uniformly distributed along the line although it is more usual to make use of lumped parameter models [104]. These models concentrate the resistance and inductance of the line into single parameters in the equivalent circuit model.

B.2.1 Short Lines

Lines which are less than 80 km in length are considered to be short lines. For a line of this length the shunt capacitance between the line and the neutral return is negligible. Only the series resistance, R, and the series inductance, L, of the line need to be considered and a lumped parameter model can be devised. The single phase equivalent circuit is shown in Figure B.3. The current is the same at the sending and receiving ends of the line and thus

$$I_S = I_R \qquad V_S = \frac{V_R}{Z} \tag{B.7}$$

B.2.2 Medium Length Lines

Lines which are over 80 km in length and less than 240 km are defined as medium length lines. It is possible to represent a medium length line with a lumped parameter model using line series resistance and line series inductance. However the shunt admittance can now longer be neglected and it is included as a lumped parameter Y. The shunt admittance

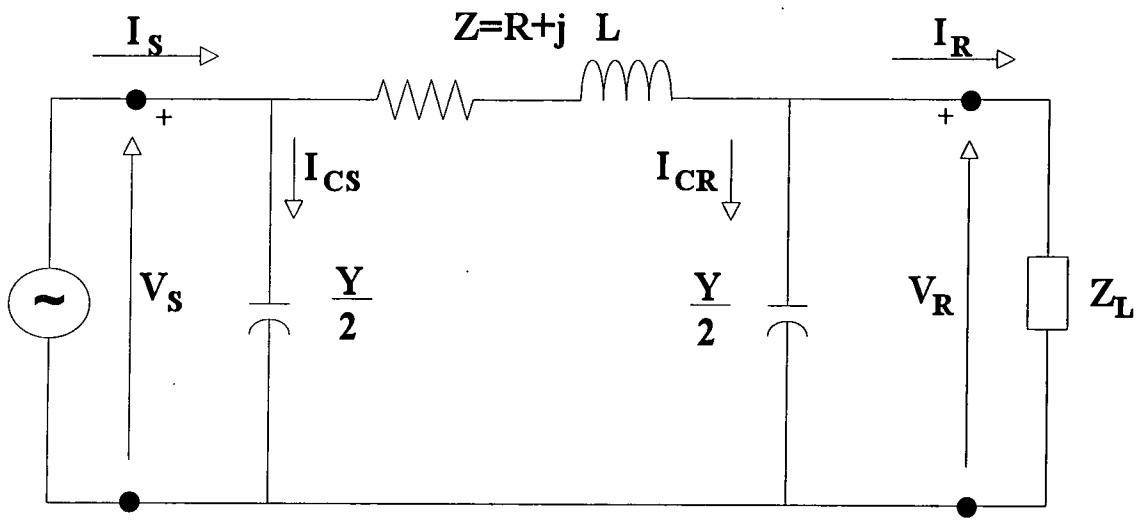


Figure B.4: Single phase equivalent circuit model of a medium length transmission line

is usually purely capacitive and it is customary to place half of the admittance at each end of the line. The equivalent circuit, shown in Figure B.4, is referred to as the nominal-π equivalent circuit model.

From simple circuit theory

$$I_{cr} = V_R \frac{Y}{2} \tag{B.8}$$

The current flowing in the series arm of the circuit is $I_R + V_R \frac{Y}{2}$ and thus

$$V_S = \left(V_R \frac{Y}{2} + I_R\right) Z + V_R = \left(\frac{ZY}{2} + 1\right) V_R + Z I_R \tag{B.9}$$

Looking at the sending end gives

$$I_{cs} = V_S \frac{Y}{2} \tag{B.10}$$

and the current flowing in the series arm is

$$I_S = V_S \frac{Y}{2} + V_R \frac{Y}{2} + I_R \tag{B.11}$$

Substituting (B.9) into (B.11) yields

$$I_S = \left[\left(\frac{ZY}{2} + 1\right) + Z I_R\right] \frac{Y}{2} + V_R \frac{Y}{2} + I_R = V_R \left(1 + \frac{ZY}{4}\right) + \left(\frac{ZY}{2} + 1\right) I_R \tag{B.12}$$

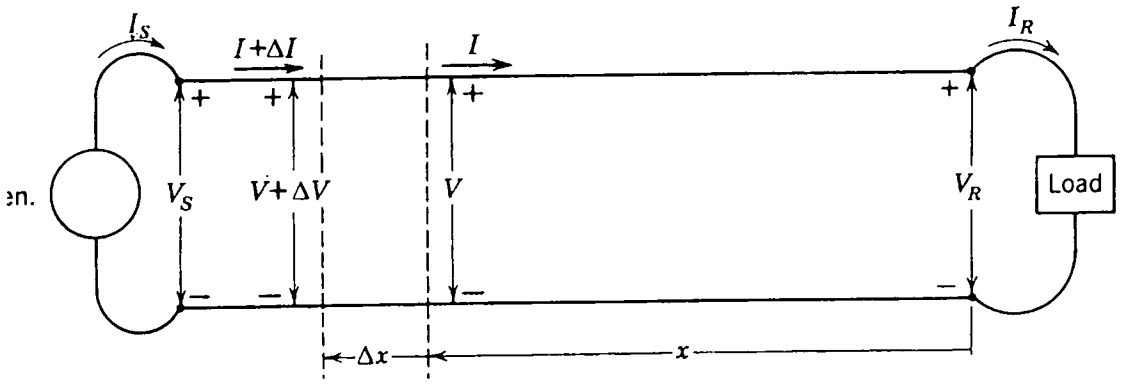


Figure B.5: Single phase representation of a long transmission line

B.2.3 Long Lines

Long lines are defined as those lines whose length exceeds 240 km. The mathematical description of a long line must account for the fact that the series resistance, series inductance and shunt admittance are distributed throughout the line rather than being lumped together. A long line may be represented by a single phase circuit of the form of Figure B.5. Series resistance, series inductance and shunt admittance are assumed to be uniformly distributed along the length of the line. Consider the voltage and current differences between the ends of the line element of length Δx, where x is the distance of the element from the receiving end. The end of the element nearest the receiving end of the line is referred to as the receiving end of the element whilst the end closest to the sending end of the line is the sending end of the element. The series impedance and shunt admittance of the element are $z\Delta x$ and $y\Delta x$ respectively, where

z = series impedance of per unit length

y = shunt admittance to neutral per unit length

If V is the voltage of the element at the receiving end then the voltage at the sending end is $V + \Delta V$ as the voltage increases by ΔV over the length of the element. If I is the current flowing at the receiving end of the element then

$$\frac{\Delta V}{\Delta x} = Iz \tag{B.13}$$

As $\Delta x \rightarrow 0$

$$\frac{dV}{dx} = Iz \tag{B.14}$$

The current entering the sending end of the element is $I + \Delta I$, where $\Delta I = Vy\Delta x$. Hence

$$\frac{dI}{dx} = Vy \tag{B.15}$$

as $x \rightarrow 0$. Resorting to calculus, it is possible to prove [104] that

$$V = \frac{V_R + I_R Z_c}{2} e^{\gamma x} + \frac{V_R - I_R Z_c}{2} e^{-\gamma x} \tag{B.16}$$

$$I = \frac{\frac{V_R}{Z_c} + I_R}{2} e^{\gamma x} - \frac{\frac{V_R}{Z_c} - I_R}{2} e^{-\gamma x} \tag{B.17}$$

where

$Z_c = \sqrt{\frac{z}{y}}$=characteristic impedance of the line

$\gamma = \sqrt{yz}$=propagation constant of the line

Recalling that

$$\sinh\theta = \frac{e^{\theta} - e^{-\theta}}{2} \tag{B.18}$$

$$\cosh\theta = \frac{e^{\theta} + e^{-\theta}}{2} \tag{B.19}$$

it is possible to rewrite (B.16) and (B.17) in hyperbolic form as

$$V = V_R \cosh\gamma x + I_R Z_c \sinh\gamma x \tag{B.20}$$

$$I = I_r \cosh\gamma x + \frac{V_R}{Z_c} \sinh\gamma x \tag{B.21}$$

Setting $x = l$, where l is the length of the line gives

$$V_S = V_R \cosh\gamma l + I_R Z_c \sinh\gamma l \tag{B.22}$$

$$I_S = I_R \cosh\gamma l + \frac{V_R}{Z_c} \sinh\gamma l \tag{B.23}$$

A lumped parameter equivalent circuit can be obtained for the long line. Consider the π equivalent circuit of Figure B.6. Substituting the lumped parameters into (B.9) yields

$$V_S = \left(\frac{Z'Y'}{2} + 1\right) V_R + Z' I_R \tag{B.24}$$

To make the equivalent circuit model the transmission line accurately the coefficient of V_R and I_R in (B.22) must equal the coefficients of V_R and I_R in (B.24). Equating the

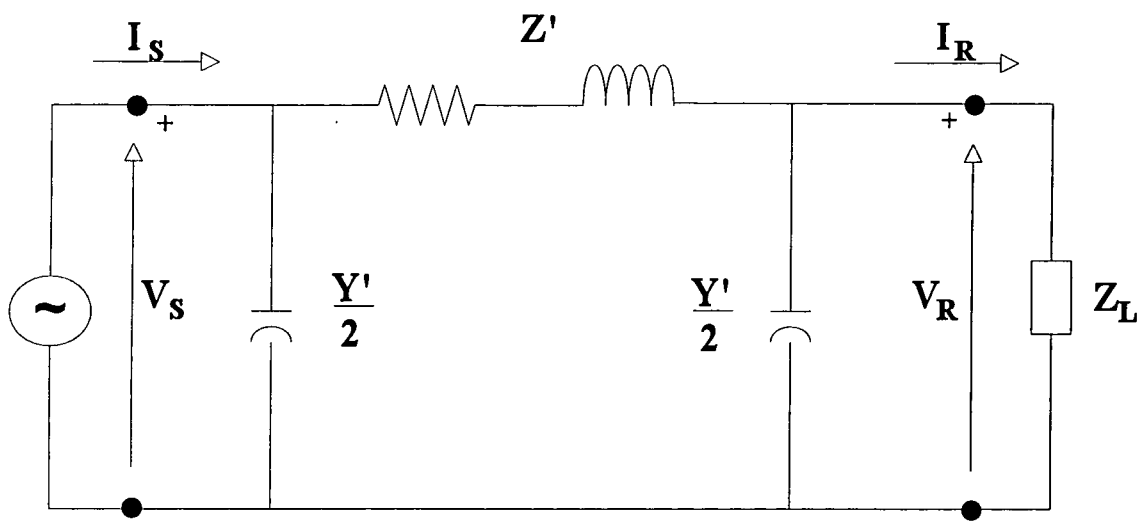


Figure B.6: Single phase π-equivalent circuit of a long transmission line

coefficients of I_R yields

$$Z' = Z_c \sinh \gamma l = \sqrt{\frac{z}{y}} \sinh \gamma l = zl \frac{\sinh \gamma l}{\sqrt{zyl}} \tag{B.25}$$

The total series impedance of the line, Z, is given by $Z = zl$ and thus

$$Z' = \frac{Z \sinh \gamma l}{\gamma l} \tag{B.26}$$

Equating the coefficients of V_R yields

$$\frac{Z'Y'}{2} + 1 = \cosh \gamma l \tag{B.27}$$

Substituting from (B.25)

$$\frac{Y' Z_c \sinh \gamma l}{2} + 1 = \cosh \gamma l \tag{B.28}$$

Hence

$$\frac{Y'}{2} = \frac{1}{Z_c} \cdot \frac{\cosh \gamma l - 1}{\sinh \gamma l} \tag{B.29}$$

Using the identity

$$\tanh \frac{\theta}{2} = \frac{\cosh \theta - 1}{\sinh \theta} \tag{B.30}$$

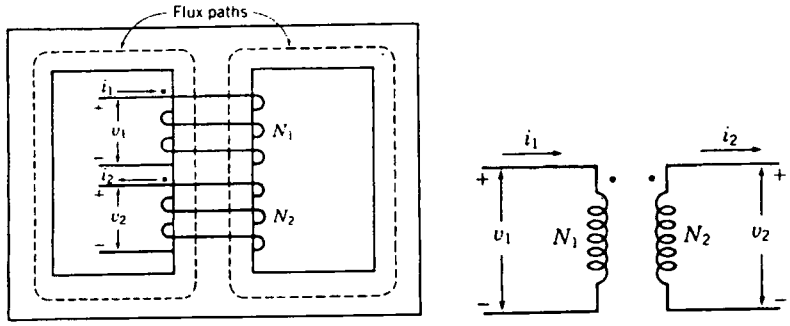


Figure B.7: Two winding transformer and schematic

yields

$$\frac{Y'}{2} = \frac{Y}{2}\frac{\tanh\left(\frac{\gamma l}{2}\right)}{\frac{\gamma l}{2}} \tag{B.31}$$

where $Y = yl$ is the total shunt admittance of the line.

B.3 The Transformer Model

A transformer consists of two or more coils placed such that they are linked by the same flux. The coils are usually wound on to an iron core in order to confine the flux to the coils. The coil which is connected to the load is known as the secondary winding and the other coil is known as the primary winding.

Consider the two winding transformer and its schematic shown in Figure B.7. Assuming the transformer to be ideal (*i.e.* the permeability of the iron core is infinite and the resistance of the windings is zero) the terminal voltages are related by

$$\frac{V_1}{V_2} = \frac{N_1}{N_2} \tag{B.32}$$

Similarly the terminal currents are related by

$$\frac{I_1}{I_2} = \frac{N_2}{N_1} \tag{B.33}$$

The full derivation of these relationships relies on the use of Faraday's Law and Ampere's Law and is given in [104].

Due to the principle of conservation of energy, the power input to the primary winding

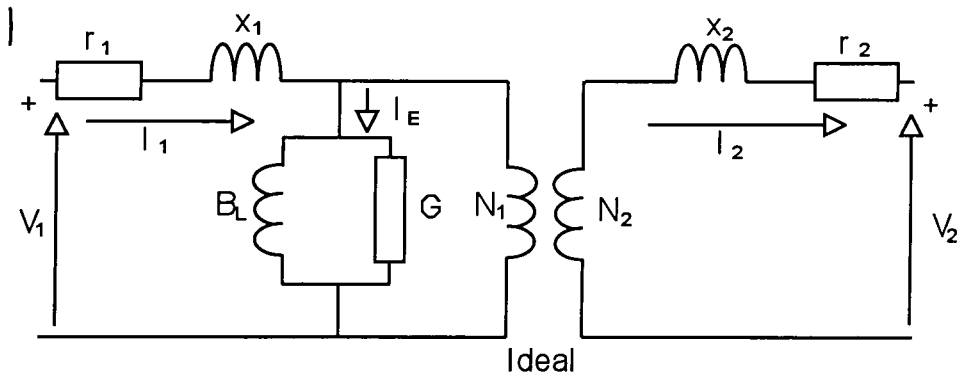


Figure B.8: Modified equivalent circuit of an ideal single phase transformer

must be equal to the power output from the secondary winding.

$$S = V_1 I_1^* = V_2 I_2^* \tag{B.34}$$

If an impedance, Z_2 is connected to the secondary winding then

$$Z_2 = \frac{V_2}{I_2} = \frac{(\frac{N_2}{N_1})V_1}{(\frac{N_1}{N_2})I_1} \tag{B.35}$$

The effective impedance seen at the terminals of the primary winding is

$$Z_2' = \frac{V_1}{I_1} = \left(\frac{N_1}{N_2}\right)^2 Z_2 \tag{B.36}$$

Practical transformers have finite core permeability, winding resistance, losses in the core due to eddy currents and hysteresis and imperfect flux linkage to the coils. These factors must be accounted for in the equivalent circuit of the practical transformer, shown in Figure B.8. This equivalent circuit is derived by adding extra components to account for these effects at the primary and secondary windings of the ideal transformer. Applying a sinusoidal voltage to the primary winding when the secondary winding is open circuit causes a small current to flow in the primary winding. This is the magnetizing current, I_E, and is accounted for in the equivalent circuit model by the inductive susceptance B_L in parallel with a conductance G. The inductive leakage reactance x_1 accounts for flux leakage in the primary winding and x_2 accounts for flux leakage in the secondary winding. r_1 and r_2 are the series resistances of the primary and secondary windings respectively. By referring all quantities to the primary side of the transformer the ideal transformer can be removed from

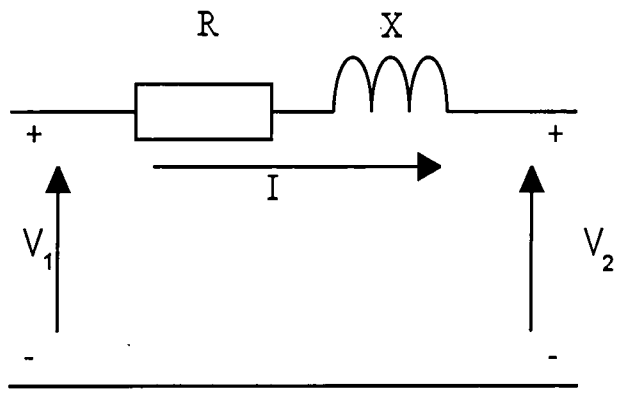


Figure B.9: Equivalent circuit of a practical single phase transformer

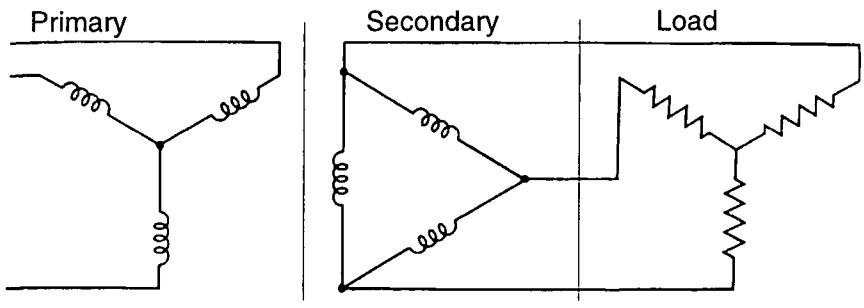


Figure B.10: $Y - \Delta$ connected three phase transformer equivalent circuit

the equivalent circuit. As magnetizing current is small compared to the load current is it often neglected and the equivalent circuit model of a practical single phase transformer is shown in Figure B.9. If there are N_1 turns on the primary and N_2 turns on the secondary it is possible to define $a = \frac{N_1}{N_2}$. The parameters of the model are thus

$$R_1 = r_1 + a^2 r_2 \tag{B.37}$$

$$X_1 = x_1 + a^2 x_2 \tag{B.38}$$

A three phase transformer may be created by connecting a bank of single phase transformers such that the three primary (secondary) windings are Δ connected and the three secondary (primary) windings are Y connected (Figure B.10). The result is the $Y - \Delta$ three phase transformer. Practical three phase transformers are often constructed by winding the three phases onto the same iron core. The equivalent circuit model is constructed from

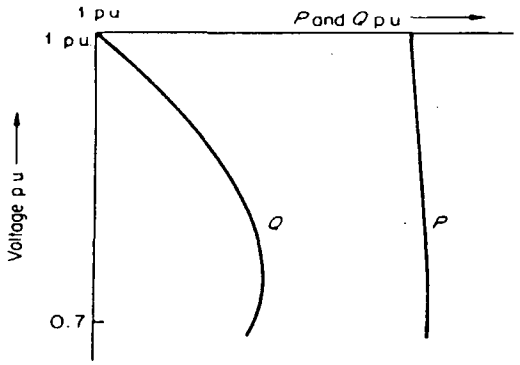


Figure B.11: P-V and Q-V characteristic for a typical synchronous motor

three single phase equivalent circuit models of a practical transformer, connected together in the appropriate manner.

B.4 The Load Model

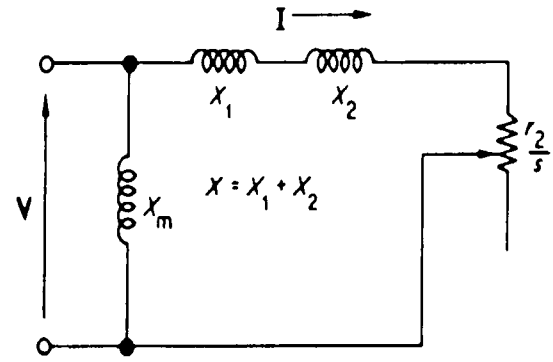
The simulation of a power system requires the loads connected to that system to be accurately modelled. This requires a consideration of how the power and reactive power flows of the load vary with voltage. The individual loads at a bus are usually lumped together to give a composite load for the bus and composite loads typically consist of [105]

Induction motors	50%-70%
Synchronous motors	10%
Heating and lighting	20%-25%
Transmission losses	10%-12%

Heating and lighting loads have well defined characteristics. Lighting consumes no reactive power and the power consumption varies with $(voltage)^{1.6}$. Heating loads also consume no reactive power and maintain a constant resistance as voltage varies and the power consumption varies with $(voltage)^2$.

The power consumed by synchronous motors remains approximately constant. As the voltage drops the reactive power consumption increases. Figure B.11 shows the variation of real and reactive power with voltage for a typical synchronous machine.

Induction motors account for the largest proportion of the composite load and the



X_1 is the stator leakage reactance
X_2 is the rotor leakage reactance
X_m is the magnetizing reactance
r_2 is the rotor resistance
s is the rotor slip

Figure B.12: Equivalent circuit of an induction motor

variation of real and reactive power flow with voltage can be found by considering a suitable equivalent circuit. Figure B.12 shows the equivalent circuit for an induction motor. Assuming that the mechanical loading on the rotor shaft is constant, the electrical power delivered to the rotor is

$$P = 3\frac{I^2 r_2}{s} = constant \tag{B.39}$$

Reactive power consumption is given by

$$Q = \frac{3V^2}{X_m} + 3I^2(X_1 + X_2) \tag{B.40}$$

Real power consumption is given by

$$P = 3\frac{I^2 r_2}{s} = \frac{3V^2}{(\frac{r_2}{s})^2 + (X_1 + X_2)^2} \cdot \frac{r_2}{s} = \frac{3V^2 r_2 s}{R_2^2 + (sX)^2} \tag{B.41}$$

where $X = X_1 + X_2$. Figure B.13 shows the variation of real and reactive power flow with voltage under different mechanical loadings.

From the point of view of a simulation it is the characteristics of the composite load which are of interest rather than the characteristics of its constituents. If the P-V,Q-V characteristics have been measured for each substation then these may be used to model the loads. Unfortunately these characteristics are not readily available. Many analyses

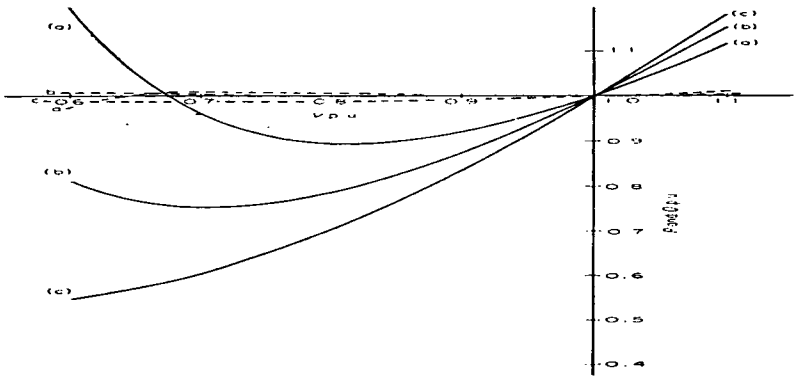


Figure B.13: P-V and Q-V characteristics of an induction motor

represent loads by constant impedances [105] and consequently $P \propto V^2$ and $Q \propto V^2$. If the power consumed by the load is $S = P + jQ$ it is easy to show that the current in the load is given by

$$I^* = \frac{S}{V} \tag{B.42}$$

$$I = \left[\frac{S}{V}\right]^* = \frac{P - jQ}{V} \tag{B.43}$$

From Ohm's Law we have $V = IZ$ and the impedance, Z, used to represent the load is

$$Z = \frac{V}{I} = \frac{V^2}{P - jQ} \tag{B.44}$$

When the network is extensively simplified then the constant impedance model is used although the load that this represents seldom occurs in practice. Other methods of load modelling represent the load with a constant current sink and this is found to give a good approximation to real loads [105].

Appendix C

Deriving the Bus Admittance Matrix

It is necessary to derive the bus admittance matrix for a given system before attempting to calculate the currents and voltages in that system. The bus admittance matrix is determined by performing nodal analysis on the system. Consider the example system of Figure C.1. The nodal admittance analysis method is based simply on Kirchoff's Current law. The consequence of this is that each node, k, in the system obeys the relationship

$$I_k = \sum_{i=1}^{n} y_{ki} V_i \tag{C.1}$$

where n is the number of branches from this node, y_{ki} is the admittance of the branch connecting node k and node i. As this is true for each node in the system we can write the matrix equation of (2.7) where

$$[Y] = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,n-1} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n-1,1} & y_{n-1,2} & \cdots & y_{n-1,n_1} \end{bmatrix} \tag{C.2}$$

The diagonal term $y_{k,k}$ is the sum of all the admittances connected to node k whilst $y_{k,i}$ is the sum of all the admittances between node k and node i. For the example of Figure C.1

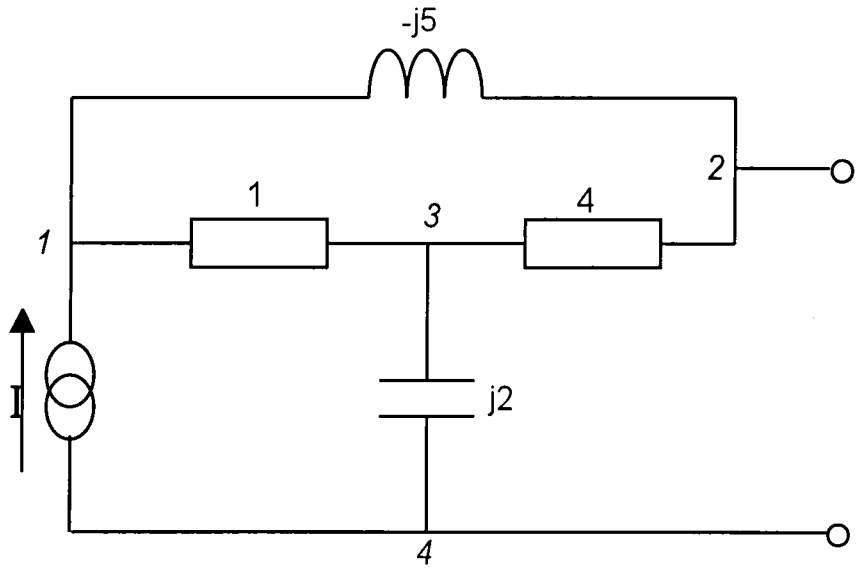


Figure C.1: Example circuit for admittance analysis

the matrix $[Y]$ is

$$[Y] = \begin{bmatrix} 1-j5 & j5 & -1 \\ j5 & 4-j5 & -4 \\ -1 & -4 & 1+4+j2 \end{bmatrix} \tag{C.3}$$

Note that node 4 does not appear in $[Y]$ as one node in the system has to be chosen as a reference node to prevent the creation of a set of dependent equations. Eliminating the equations relating to one node ensures that the remaining set of equations is linearly independent.

When considering power systems and the power flow problem, the bus admittance matrix, $[Y]$, is simply the nodal admittance matrix of the transmission network and can be derived in the same way using nodal admittance analysis. It is usual to select the system slack bus as the reference node.

Consider the simple four bus system shown in Figure C.2.

Each line in the system is a transmission line which has a series impedance of $\bar{z}$ and a shunt admittance of $\frac{\bar{y}}{2}$ connected to each end of the line. Considering the line between buses 1 and 2 :- The series impedance and shunt admittance make contributions to the y_{11} and y_{22} terms of the admittance matrix, according to

$$\bar{y}_{11} = y_{11} + \frac{1}{\bar{z}} + \frac{\bar{y}}{2} \qquad \bar{y}_{22} = y_{22} + \frac{1}{\bar{z}} + \frac{\bar{y}}{2} \tag{C.4}$$

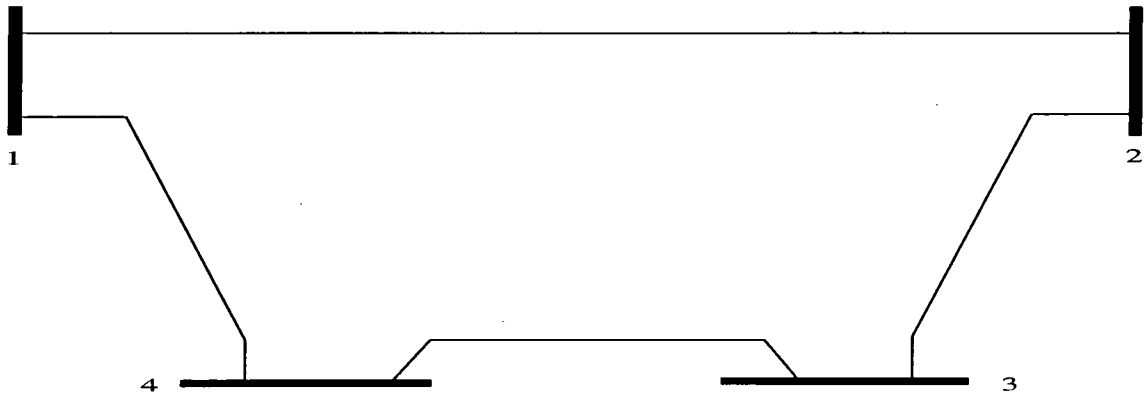


Figure C.2: Simple four bus example system

The off-diagonal terms account only for the series impedance between buses 1 and 2 and hence

$$\bar{y}_{12} = y_{12} - \frac{1}{\bar{z}} \qquad \bar{y}_{21} = y_{21} - \frac{1}{\bar{z}} \tag{C.5}$$

Using the same technique and considering each line in the system in turn, it is possible to derive the complete bus admittance matrix for the system. All that is required is a knowledge of the values of $\bar{z}$ and $\bar{y}$ for each line.

Appendix D

Network Partitioning and Diakoptics

In order to solve the network equations in parallel it is necessary to partition the system network into several smaller, independent subnetworks. The large network is divided by 'tearing' it apart using Kron's method of diakoptics [48]. Each subnetwork is solved independently and the independent solutions are appropriately modified to give the correct solution.

Two methods exist for tearing the network into subnetworks - branch cutting and node tearing. The branch cutting method operates by cutting some of the branches which interconnect the nodes, thereby separating the network into independent subnetworks. The node tearing approach operates by tearing some of the nodes in half to separate the network into a number of subnetworks. The cut branches or the torn nodes give rise to coordination variables in the diakoptic solution. Once the subnetworks have been solved it is these coordination variables which are used to modify the individual subnetwork solutions to give the overall solution. The two methods differ not only in the way that they partition network but also in the way the coordination variables are chosen. The coordination variables used by the branch cutting method are the currents flowing in the branches that are cut. The node tearing method selects the voltages at the torn nodes as the coordination variables. Node tearing usually introduces fewer coordination variables than branch cutting [77] thus making it more computationally efficient. The branch cutting method can be useful when information about the currents flowing in the circuit branches is required. The following sections present the branch cutting and node tearing methods in more detail. The information

presented here is based on that given by Boming et al.[77].

D.1 Node Tearing

Given a network represented by the set of equations $\mathbf{YV} = \mathbf{I}$, it is possible to tear the network into a number of independent subnetworks by choosing appropriate tearing nodes. If these nodes are given larger node numbers (*i.e.* ordered last) then the system of equations will have BBDF form, as below

$$\begin{bmatrix} \mathbf{Y}_{11} & & & \mathbf{Y}_{1c} \\ & \ddots & & \vdots \\ & & \mathbf{Y}_{kk} & \mathbf{Y}_{kc} \\ \mathbf{Y}_{c1} & \cdots & \mathbf{Y}_{ck} & \mathbf{Y}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{V}_1 \\ \vdots \\ \mathbf{V}_k \\ \mathbf{V}_c \end{bmatrix} = \begin{bmatrix} \mathbf{I}_1 \\ \vdots \\ \mathbf{I}_k \\ \mathbf{I}_c \end{bmatrix} \tag{D.1}$$

The voltages, $\mathbf{V}_t$, at the tearing nodes are the coordination variables for this approach and may be solved for by

$$\mathbf{V}_c = (\mathbf{Y}_{cc} - \sum_{i=1}^{k} \mathbf{Y}_{ci}\mathbf{Y}_{ii}^{-1}\mathbf{Y}_{ic})^{-1}(\mathbf{I}_c - \sum_{i=1}^{k} \mathbf{Y}_{ci}\mathbf{Y}_{ii}^{-1}\mathbf{I}_c) \tag{D.2}$$

The solution for the individual subnetworks is given by

$$\mathbf{V}_i = \mathbf{Y}_{ii}^{-1}(\mathbf{I}_i - \mathbf{Y}_{ic}\mathbf{V}_c) \quad i = 1, \ldots, k \tag{D.3}$$

Figure D.1 provides a conceptualization of the node tearing method. The approach can be thought of as tearing nodes apart into pieces. A piece of each node is then connected to the subnetwork to which that node was attached. The voltages, $\mathbf{V}_t$, are the node voltages associated with the tearing nodes but they may be thought of as voltage sources connected with the tearing nodes. In the same way that a part of the tearing node is attached to the subnetwork, a voltage source of the same magnitude as the tearing node voltage is connected to each subnetwork to which that tearing node was attached. As Figure D.1 shows, once the voltage sources (*i.e.* tearing node voltages) are known then the individual subnetworks may be solved independently and in parallel.

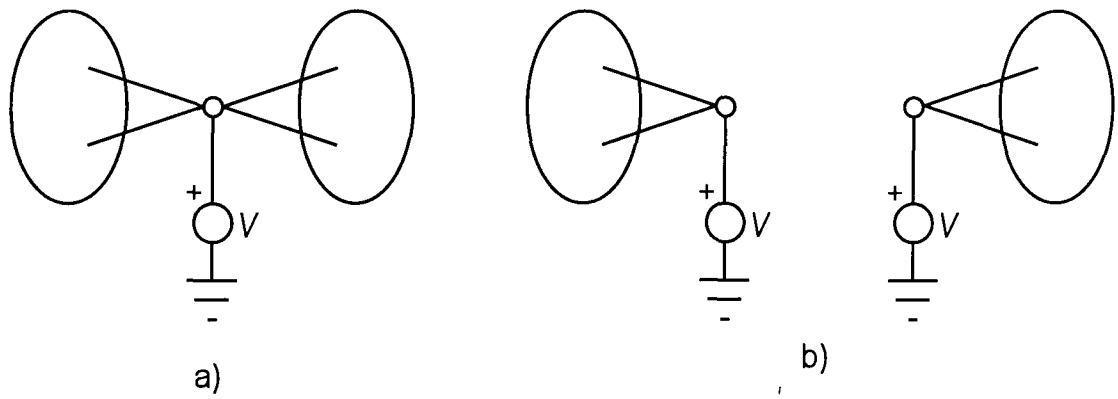


Figure D.1: A conceptual view of node tearing, from Boming et al. a) applying equivalent voltage sources b) tearing the node apart

D.2 Branch Cutting

Given an electrical network consisting of nodes connected by branches, Figure D.2(a), it is possible to replace a number of these branches, L, by current sources. These current sources have the same magnitude as the currents originally flowing in the branches and the network with current sources is electrically equivalent to the original network. The branches which have been replaced by current sources as known as cut branches. Figure D.2(b) and Figure D.2(c) show how the equivalent current sources may be exploited to partition the network into independent subnetworks. Let $\mathbf{Y}$ be the admittance matrix of the network and $\mathbf{Y}_a$ be the admittance network for the network when the cutting branches have been removed. Let $\mathbf{i}_L$ be the currents flowing in the cut branches. Removing the cut branches partitions the network into pieces and the admittance matrix $\mathbf{Y}_a$ becomes block diagonal. Now

$$\mathbf{Y} = \mathbf{Y}_a + \mathbf{Y}_c \tag{D.4}$$

where $\mathbf{Y}_c$ is a a matrix corresponding to the cut branches

$$\mathbf{Y}_c = \mathbf{P}\mathbf{y}\mathbf{P}^T \tag{D.5}$$

where

- $\mathbf{P}$ is a matrix consisting of columns of the incidence matrix of the original admittance matrix which correspond to the cut branches. The elements of each column of this matrix are zero except at the terminal nodes of each branch where the values are ± 1.

- $\mathbf{y}$ is an $L \times L$ diagonal matrix. The elements of this matrix are the admittances of the cut branches.

Substituting (D.4) and (D.5) into $\mathbf{YV} = \mathbf{I}$ yields

$$(\mathbf{Y}_a + \mathbf{PyP}^T)\mathbf{V} = \mathbf{I} \tag{D.6}$$

As $\mathbf{Y}_c$ represents the admittances of the cut branches and $\mathbf{i}_L$ are the currents flowing in them, (D.6) may be rewritten as

$$\mathbf{Y}_a\mathbf{V} + \mathbf{Pi}_L = \mathbf{I} \tag{D.7}$$

by setting $\mathbf{i}_L = \mathbf{yP}^T\mathbf{V}$. (D.6) and (D.7) can thus be expressed in matrix notation as

$$\begin{bmatrix} \mathbf{Y}_a & \mathbf{P} \\ \mathbf{P}^T & -\mathbf{y}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{V} \\ \mathbf{i}_L \end{bmatrix} = \begin{bmatrix} \mathbf{I} \\ 0 \end{bmatrix} \tag{D.8}$$

The coefficient matrix of (D.8) again exhibits BBDF and parallel processing is possible. One additional complication of the branch cutting method is that the admittance submatrices are overdetermined and singular. This problem has to be overcome by defining a reference node in each subnetwork and the effect of this is to remove the overdeterminism from the equations, preventing $\mathbf{Y}_a$ from being singular.

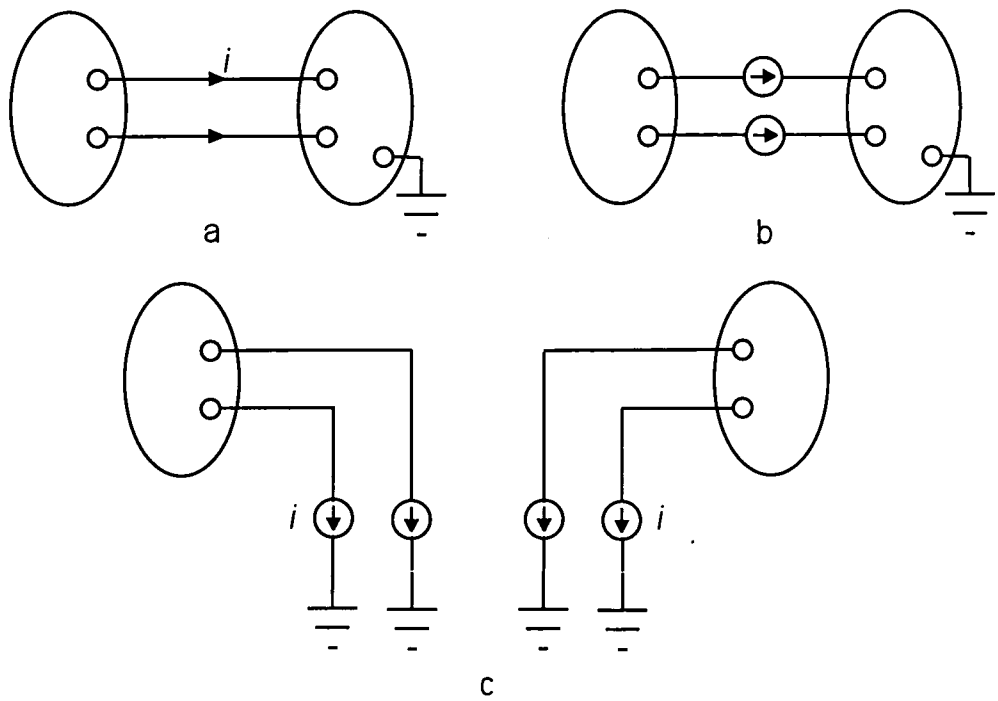


Figure D.2: A conceptual view of branch cutting, from Boming et al. a) original network b) applying equivalent current sources c) cutting the branches

Appendix E

Proof of Liu's Tree Theorems

In Section 4.3.1 two theorems relating to the properties of elimination trees were quoted. These theorems are taken from Liu [52] and the proofs of these theorems are presented below. The proofs are also taken from Liu.

E.1 Notation

The following notation is used in the proofs

- $G(\mathbf{A})$ is the graph of the matrix $\mathbf{A}$
- $T[\mathbf{A}]$ is the elimination tree of the matrix $\mathbf{A}$
- $T[x_j]$ denotes the subtree of the elimination tree rooted at node x_j
- $Adj(v)$ denotes the set of nodes adjacent to v in the graph
- ℓ_{ij} denotes the length of the path connecting nodes x_i and x_j

E.2 Other Theorems Required

Before proving the theorems of Section 4.3.1 it is necessary to state a number of other theorems. The proofs of these theorems are not given but may be found in [52].

Theorem 3 *Let $i > j$. Then $\ell_{ij} \neq 0$ if and only if there exists a path*

$$x_i, x_{p_1}, \cdots, x_{p_t}, x_j$$

in the graph $G(\mathbf{A})$ such that $\{x_{p_1}, \cdots, x_{p_t}\} \subseteq T[x_j]$.

Theorem 4 *Let $i > j$. If $\ell_{ij} \neq 0$, then the node x_i is an ancestor of x_j in the elimination tree.*

Theorem 5 *Let $i > j$. Then $\ell_{ij} \neq 0$ if and only if there exists a path*

$$x_i, x_{p_1}, \cdots, x_{p_t}, x_j$$

in the graph $G(\mathbf{A})$ such that all subscripts in $\{p_1, \cdots, p_t\}$ are less than j.

E.3 Proof of the Tree Theorems

The first theorem used by Liu states that

Theorem 1 *For each node x_j in $G(\mathbf{A})$, the subgraph of $G(\mathbf{A})$ (or $G(\mathbf{F})$) which consists of nodes in the tree $T[x_j]$ is connected, where $T[x_j]$ is the subtree rooted at node x_j.*

The proof of this theorem is derived by induction on the number of nodes t in $T[x_j]$. If there is only one node in $T[x_j]$ then the theorem is obviously true. To prove the general case, assume that the result is true for all subtrees of size less than t, and $t > 1$. Let $x_{s_1}, \cdots, x_{s_m}$ be the child nodes of x_j. Following from the inductive assumption, each subgraph consisting of nodes in $T[x_{s_k}]$, for $1 \leq k \leq m$ has fewer nodes than t and is connected in the graph $G(\mathbf{A})$. For each k, $\{x_j, x_{s_k}\}$ is an edge in the filled graph $G(\mathbf{F})$. By Theorem (3) there exists a path from x_{s_k} to x_j through nodes in $T[x_{s_k}]$. This proves that the subset$T[x_j]$ is a connected subgraph in $G(\mathbf{A})$. Since $G(\mathbf{F})$ is a supergraph of $G(\mathbf{A})$, $T[x_j]$ must also be a connected subgraph in $G(\mathbf{F})$.

Corollary 1 *For each node x_j, the set of nodes in $T[x_j]$ forms a connected component in the subgraph of $G(\mathbf{A})$ $\{G(\mathbf{F})\}$ consisting of all nodes except those in $Adj(T[x_j])$.*

implies that partitioning the tree into disjoint subtrees is the same as partitioning the network(graph) into subnetworks(subgraphs). A second corollary to this theorem is

Corollary 2 *For each node x_j, the set of nodes in $T[x_j]$ forms a connected component in the subgraph of $G(\mathbf{A})$ $\{G(\mathbf{F})\}$ consisting of all nodes except proper ancestors of x_j.*

The second proof derived and used by Liu refers to matrix reorderings. If $\mathbf{A}$ is a given symmetric matrix, the two orderings $\mathbf{P}$ and $\mathbf{Q}$ are said to be *equivalent* if the structures of the filled graphs $\mathbf{PAP}^T$ and $\mathbf{QAQ}^T$ are isomorphic. $\mathbf{P}$ is referred to as an equivalent reordering of the matrix $\mathbf{A}$ if the filled graph of $\mathbf{A}$ and the filled graph of $\mathbf{PAP}^T$ are isomorphic. Given

an initial ordering of the matrix $\mathbf{A}$ and its corresponding elimination tree $T(\mathbf{A})$, let $\mathbf{P}$ be a permutation matrix for $\mathbf{A}$ that corresponds to a topological reordering[1] of the nodes in $T(\mathbf{A})$.

Theorem 2 *Given the matrix,* $\mathbf{A}$, *and an equivalent reordering,* $\mathbf{P}$, *the filled graphs of* $G(\mathbf{A})$ *and* $G(\mathbf{PAP}^T)$ *are isomorphic if they are treated as unlabeled structures.*

The proof of this theorem is derived by letting $\mathbf{F}$ by the filled matrix of $\mathbf{A}$. Set $\bar{\mathbf{A}} = \mathbf{PAP}^T$ and let $\bar{\mathbf{F}}$ be the corresponding filled matrix of $\bar{\mathbf{A}}$. To prove the theorem it must be shown that $G(\mathbf{F})$ and $G(\bar{\mathbf{F}})$ are structurally identical. Let $x_1, x_2, \cdots, x_n$ be the sequence of node eliminations for the matrix $\mathbf{A}$ and let $\bar{x_1}, \bar{x_2}, \cdots, \bar{x_n}$ be the sequence of node eliminations for the matrix $\bar{\mathbf{A}}$. It is sufficient to show that $\{x_i, x_j\}$ is an edge in the filled graph $G(F)$ if and only if $\{\bar{x}_{\bar{r}}, \bar{x}_{\bar{s}}\}$ is and edge in $G(\bar{\mathbf{F}})$.

Assume the $\{x_i, x_j\}$ is an edge in $G(\mathbf{F})$ and $i > j$. By Theorem (4), x_i is a proper ancestor of x_j in the elimination tree $T[\mathbf{A}]$. As $\mathbf{P}$ is a topological ordering, the node $\bar{x}_{\bar{r}}$ is labeled after $\bar{x}_{\bar{s}}$ so that $\bar{r} > \bar{s}$. By Theorem (3), there exists a path in the graph $G(\mathbf{A})$ $\bar{x}_{\bar{r}} = x_i, x_{p_1}, \cdots, x_{p_t}, x_j = \bar{x}_{\bar{s}}$ such that $\{x_{p_1}, \cdots, x_{p_t}\} \subseteq T[x_j]$. Due to the property of the topological ordering $\mathbf{P}$, these nodes $x_{p_1}, \cdots, x_{p_t}$ are labeled before $\bar{x}_{\bar{s}}$ in the matrix $\bar{\mathbf{A}}$. Therefore, by Theorem (5), $\bar{x}_{\bar{r}}, \bar{x}_{\bar{s}}$ is also an edge in the filled graph $G(\mathbf{F})$.

Conversely, let $\{\bar{x}_{\bar{r}}, \bar{x}_{\bar{s}}\}$ be an edge in $G(\mathbf{F})$. For the sake of definiteness, let $\bar{r} > \bar{s}$. Note that x_i does not belong to the subtree $T[x_j]$ otherwise it contradicts the topological ordering property of $\mathbf{P}$. By Theorem (5), there exists a path in $G(\mathbf{A})$ $\bar{x}_{\bar{r}}, \bar{x}_{\bar{p}_1}, \cdots, \bar{x}_{\bar{p}_t}, x_j, \bar{x}_{\bar{s}}$ such that all subscripts $\bar{p}_1, \cdots, \bar{p}_t$ are less than $\bar{s}$. By the property of the topological reordering of $\mathbf{P}$, the nodes in $\{\bar{x}_{\bar{p}_1}, \cdots, \bar{x}_{\bar{p}_t}\}$ cannot be ancestors of x_j. All the nodes on the path $\bar{x}_{\bar{r}}, \bar{x}_{\bar{p}_1}, \cdots, \bar{x}_{\bar{p}_t}, x_j, \bar{x}_{\bar{s}}$ belong to the connected component containing the node x_j in the subgraph of $G(\mathbf{A})$ excluding the set of proper ancestors of x_j in the tree. By Corollary (2) these nodes all belong to the subtree $T[x_j]$. Thus there exists a path in $G(\mathbf{A})$ from $\bar{x}_{\bar{s}} = x_j$ to $\bar{x}_{\bar{r}} = x_i$ through nodes in the subtree $T[x_j]$ and x_i is outside of $T[x_j]$. Again by Corollary (2), this means that x_i is a proper ancestor of x_j so that $i > j$. Therefore, using Theorem (3), $\{x_i, x_j\}$ is also an edge in the filled graph $G(\mathbf{F})$.

Theorem 2 implies that every topological reordering of $\mathbf{A}$ is an equivalent reordering of the matrix $\mathbf{A}$. The corollary to this theorem is that the tree $T[\mathbf{PAP}^T]$ is isomorphic to

[1] A *topological ordering* of a rooted tree is one that numbers the child nodes before their parent nodes. *i.e.*the leaves are numbered first and the root is numbered last.

$T[\mathbf{A}]$ if they are treated as unlabeled structures.

Appendix F

Reducing the Length of Intertask Messages

Consider the communication of update values resulting from factorisation from a sending worker task, T_s, to a receiving worker task, T_r. Before the communication occurs T_s must generate the message it wishes to send as an array of bytes. The Transputer communication primitives send a specified number of bytes, starting at a given address, to the receiving task [103]. Hence the data which makes up the message must lie contiguously in memory. T_s holds the data it wishes to send in the form of sparse matrix linked lists and it must convert this into an array representation before transmission as the list elements are likely to be scattered anywhere in the task's memory space. Recall from Chapter 2 that there is a separate linked list for each row in the matrix. Knowledge of which list is being used is sufficient to uniquely identify the corresponding row in the coefficient matrix. Each list element has three parameters corresponding to column index, real part and imaginary part of the complex value respectively. Knowing the column index and which list is being consulted gives enough information to uniquely identify a single element in the coefficient matrix. These four parameters of each element to be updated by T_r need to be inserted into the message transmitted by T_s. The simplest algorithm for correctly establishing the message array is shown overleaf.

```
set pointer message_ptr to point at start of message
loop i over range of rows which reference subnetwork data held by T_r
   loop j from 1 to length of row i
   if an update in T_r results from element i,j
      Place {i, j, a_{i,j}, jb_{i,j}} into message at position indicated by message_ptr
      Increment message_ptr by four
   end loop j
end loop i
message length = message_ptr - address of start of message
```

Algorithm for generating intertask messages

T_s sends the message by instructing the Transputer to send *message_length* bytes of data located at address *start_of_message* to T_r. T_r receives the message and stores it in an array located at address *head_of_message*. If must then decode the message and add the values contained in the message to the linked list representation of its submatrix. The following algorithm performs the decoding and updating function

```
loop posn from head_of_message to head_of_message+message_length in steps of 4
   extract { row, column, a, jb } from the location posn in the message
   locate linked list for given row
   search for element corresponding to column
   if this element is found
      Add a + jb to the value of this element
   else
      insert a new list element with column index = column, value = a + jb
end posn loop
```

Algorithm for decoding intertask messages

This method of message generation and decoding has the advantage that it is easy to implement. However it is a rather inefficient method as it unnecessarily replicates information about row addresses within the messages. Consider a row in the coefficient matrix of T_s which has entries in columns k, l, m. Updates will be required to elements $(i, k), (i, l)$ and (i, m). Suppose that these elements lie in T_r's submatrix. A message must be sent from T_s

i	k	a_{ik}	jb_{ik}	i	l	a_{il}	jb_{il}	i	m	a_{im}	jb_{im}

Figure F.1: The example three element message

to T_r which contains the values associated with these updates and the message has the contents shown in Figure F.1. Notice that the row address i appears in the message three times and unnecessarily increases the length of the message. Suppose that the location of the first value referencing row i is known (*START*), as is the location of the last value referencing row i (*END*). Suppose also that all values referencing row i lie contiguously between these two locations. It is then no longer necessary to store the row parameter for each element and the message can be reduced to three quarters of its original length. The location of *START* and *END* for each row can be stored in the message header as a partitioning table which shows how to partition the message into its constituent row information. Note that the *END* of row i is immediately adjacent to the *START* of row $i + 1$. Hence the values for i must be located in the range of entries *START(i)* to *START(i+1)*-1. Consequently the partitioning table only needs to store the value of *START* for each row. Provided that the range of rows to be updated is known, the receiving task T_r can update its coefficient matrix using the following algorithm

```
loop i over range of rows to be updated
   loop j from START(i) to START(i+1)-1
   extract { column, a, jb } from position j in the message
   locate linked list for row i
   search for element corresponding to column
   if this element is found
      Add a + jb to the value of this element
   else
      insert a new list element with column index = column, value = a + jb
   end j loop
end i loop
```

Modified algorithm for decoding intertask messages

The format of the modified message is shown in Figure F.2(a) whilst Figure F.2(b) shows the contents of the message from the simple three element example introduced in

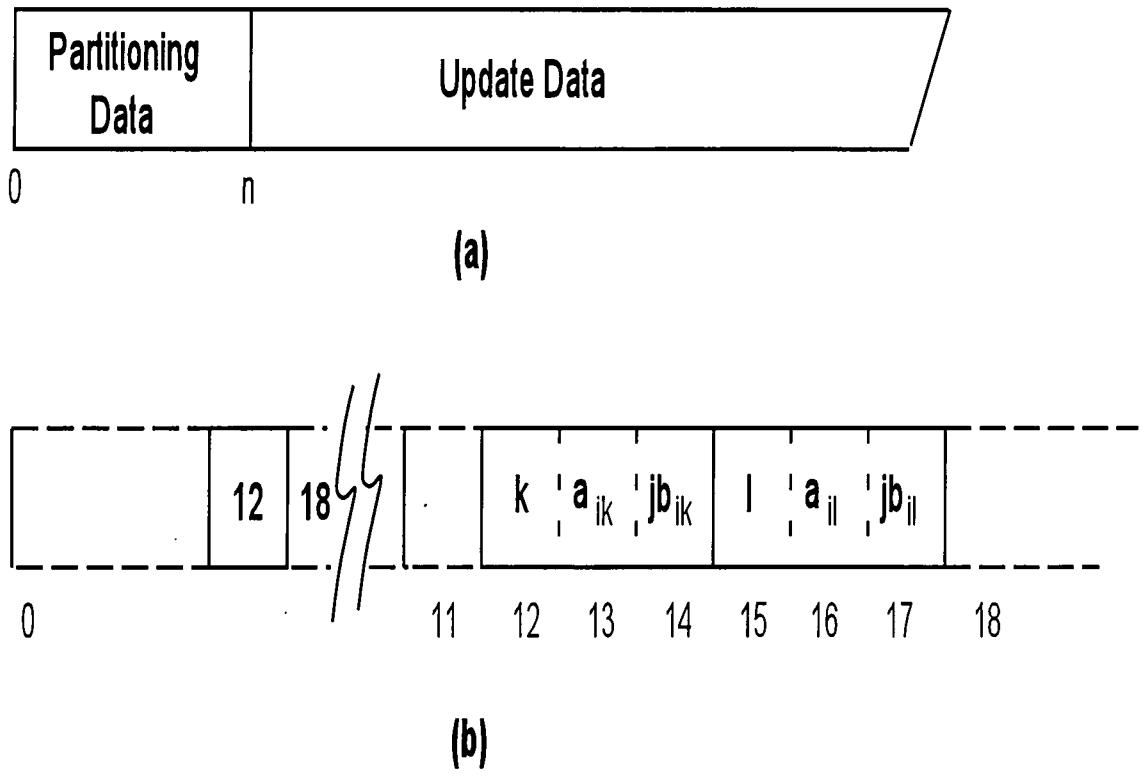


Figure F.2: Modified message structure (a) and three element example message (b)

Figure F.1.

The modified message format does yield significant savings in message length for the lengthy update messages of real systems. The reduction in communication time resulting from this reduction in message length is negligible but the reduction in the amount of time taken to generate and decode messages is quite considerable. A significant increase in performance was observed when the modified message format was implemented in the Transputer-based RP solution. For example, when split into four major and three minor subnetworks, the CEGB 734 node system was processed in 95ms using the modified message format as opposed to 175ms for the unmodified case.

Appendix G

Monitoring the Performance of the Parallel Solution

One of the simplest methods for accurately determining the execution time of a program is to use some external timer hardware that has the desired timing resolution and can be triggered by instructions in the program to be monitored. This sort of external hardware is not sufficiently flexible to allow the computer to poll it for values, making it difficult for the monitored program to associate an accurate timestamp with each event in the program's execution. Fortunately many high level languages provide software implemented timing facilities and these can be used to monitor execution time and provide timestamps. In DOS and Unix environments these timers are usually defined to have a resolution of 1ms, as the languages define a constant 1000 clock ticks per second. Unfortunately this does not portray the true resolution of the timer as timers implemented in software are interrupt driven and the timer value is only incremented every 50ms or so. In practice this means that any action which is performed in less time than the update interval is recorded as taking zero time. For very fast programs this degree of accuracy is insufficient, hence the need to use external timing hardware.

The Transputer programmer is lucky in that two timers are implemented in the hardware of each Transputer and these timers may be polled from software. As they are free running hardware timers they are highly accurate and the slowest of the two timers has a resolution of $1\mu s$ [22], making it useful in almost all applications. A program running on a Transputer may monitor its execution time by polling the timer at the first and last instruction of the program and taking the difference between the two times. A specific action or event

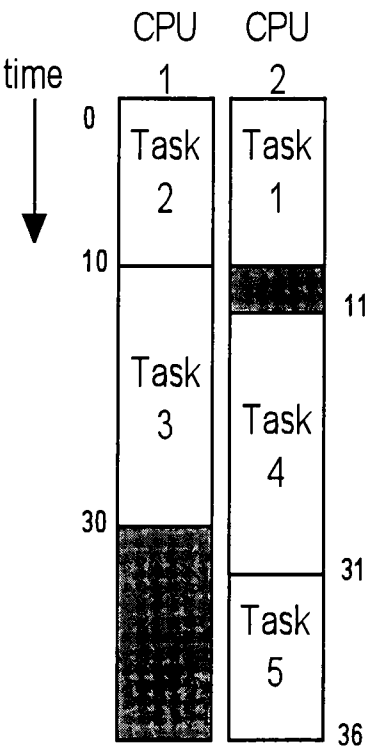


Figure G.1: A Gantt chart used to visualise program operation

within the program may be timestamped by polling the timer immediately prior to that event occurring. The INMOS C language provides numerous functions for performing such temporal manipulation and comparison.

Timestamping the actions of a program can provide information about the efficiency of the components of the program. When parallel programs are considered, timing the execution of each task and timestamping actions within tasks provides comprehensive information illustrating how tasks interact and how the processing resources are used during the execution of the program. Of particular interest is the identification of inefficient idle states. Poring over the numeric results is one way of identifying what is happening within the program but the whole process becomes much more intuitive if some form of visualisation technique is used. The Gantt chart [19, 106] is a useful tool for visualising parallel programs. The chart consists of a list of all the processors in the parallel machine. A list of tasks is allocated to each processor and these tasks are ordered by their start and finish times. Figure G.1 shows a typical Gantt chart. White areas in the chart represent times when the processors are busy performing useful work, grey areas represent times when they are idle. In implementing the Transputer-based RP program the need for a method

of monitoring and visualising program performance was identified. The lack of a suitable development environment made it hard to accurately time the execution of the program and quantify the effect of program modifications. A performance monitoring method was developed out of necessity and its main features are now discussed.

The aim of modifying a program is to refine its operation and alter those parts which will yield the maximum improvement in performance. Identification of the inefficient components is needed and this is where the timestamp information comes in useful [107]. If two actions, A and B occur sequentially within a program, where A commences at time t_A and B commences at time t_B, the execution time of A is clearly

$$t_{eA} = t_B - t_A \tag{G.1}$$

If the commencement of each major program action is timestamped then the difference between adjacent entries in the list of ordered timestamps gives a list of execution times for each of the major program events. Once again it is more intuitive if the information is displayed in a pictorial manner. Adopting the Gantt chart gives a suitable diagrammatic representation. The chart is comprises a list of all the tasks in the program. Each task is allocated a list of all the major actions in that task and these are ordered by their start and finish times. Suitable positioning of timestamping within each task allows the idle states to be identified. Idle states are shown as white regions in the columns of the chart, shaded regions correspond to busy (useful) states. Different shadings correspond to different actions which occur in each task. Figure G.2 shows the chart for a fifteen task implementation of the Recursively Parallel method.

Generating the timing information needed to derive the execution profile charts is not altogether straightforward. The hardware timers on each Transputer are not synchronized and the lack of a global clock makes it difficult to locate actions in absolute time. The solution is to use one task as a time reference and to synchronise the other tasks to this reference. The supervisor task residing on the root processor is the most logical choice of reference task. The simplest method of synchronizing worker task execution with that of the supervisor is to make the first instruction of the worker one which blocks and waits on a start signal from the supervisor task. Polling the hardware timer to record a start time is the second instruction executed by a worker. As Figure G.3 shows there is an inherent problem in this approach. Due to the lack of message broadcasting facilities on

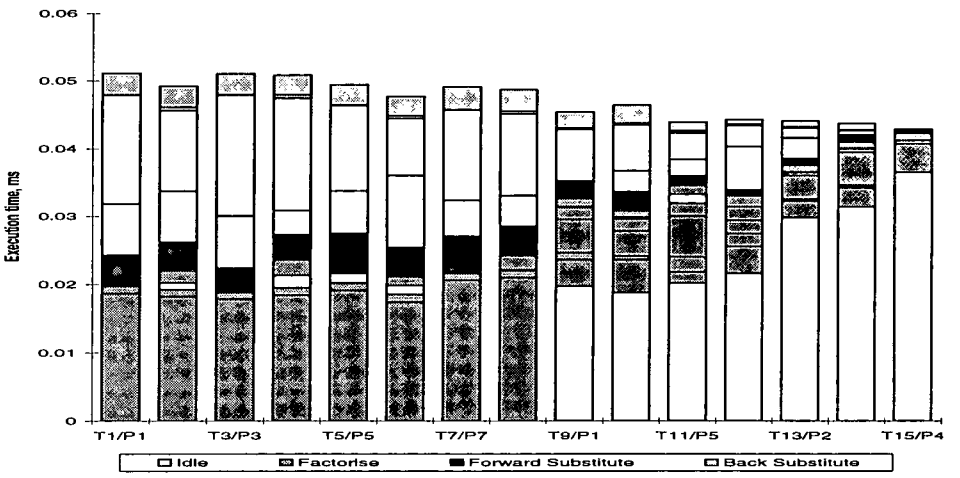


Figure G.2: Visualisation of the RP program execution profile

the Transputer, each worker is started in turn. This causes the starting of the worker tasks to be staggered in time with respect to the absolute time reference and this gives rise to inaccurate timing results. Furthermore the supervisor cannot 'start' until the last worker has been activated. Although the supervisor does no useful processing it can be used to take a stopwatch timing of the solution to validate other timing information. The staggered starting causes the stopwatch timer to be started too late and incorrect results are returned.

This problem has been surmounted by using the block-and-wait strategy in conjunction with variable delay methods. At some point, t_0, the reference task obtains the time from its local hardware timer. It then picks a start time, t_s, where

$$t_s = t_0 + \delta_t \tag{G.2}$$

where δ_t is of the order of 2 seconds. Before t_0 all the worker tasks have blocked and are awaiting communication with the reference (supervisor) task. This task considers all the workers in turn and before communicating with the i^{th} task the reference task polls its local

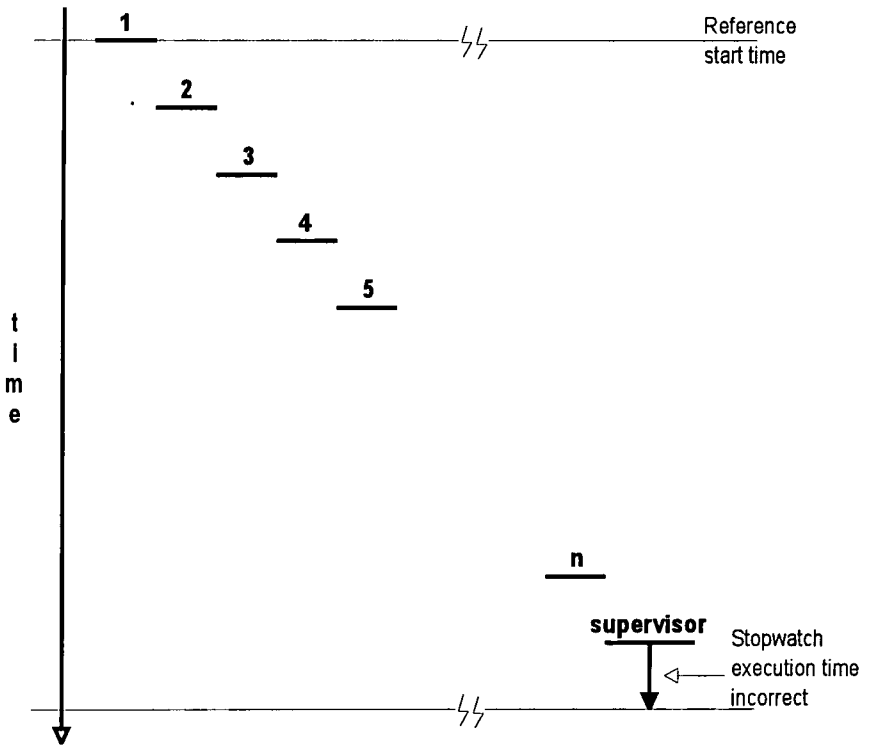


Figure G.3: The staggering effect of block-and-wait synchronisation

timer to read the time t_i, where

$$t_0 \leq t_i < t_s \tag{G.3}$$

A delay, δ_{t_i}, is calculated for the i^{th} task

$$\delta_{t_i} = t_s - t_i \tag{G.4}$$

and this value is communicated to task i. Upon receipt of this delay value task i polls its local timer to obtain the time t_i'. The delay is added to this time to give the start time

$$t_{si} = t_i' + \delta_{t_i} \tag{G.5}$$

and processing cannot commence until after this time. Task i then suspends itself until the local timer value exceeds t_{si}. The approach is shown diagrammatically in Figure G.4. Whilst it does not guarantee that each task will commence execution at exactly the same instant, t_s, in global time it does guarantee that every task will commence within a few processor clock cycles of this time and this is sufficiently accurate.

A staggering in time can still occur due to the delay introduced in sending the value δ_{t_i}

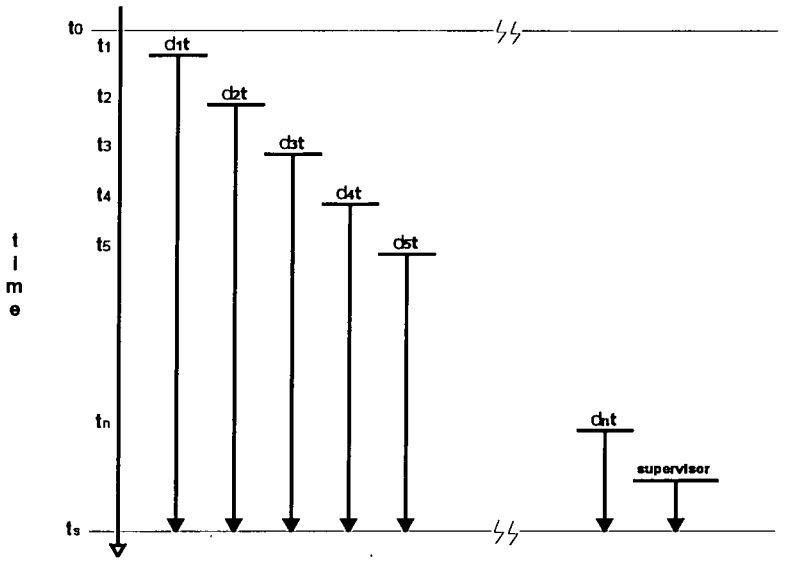


Figure G.4: Synchronisation through variable delays

across the network to the i^{th} task. This can be circumvented by modifying (G.4) such that

$$\delta_{t_i} = t_s - t_i - C(ref \leftarrow i) \tag{G.6}$$

where $C(ref \leftarrow i)$ is the average time taken to transfer a message of the same size as δ_{t_i} between the reference (supervisor) task and task i. Values of $C(ref \leftarrow i)$ can be calculated for $i = 1 \ldots n$ by sending 1000 fixed length messages to each task and monitoring the total transfer time from the sending end. Dividing this by 1000 gives the mean transfer time $C(ref \leftarrow i)$. This analysis of the communication network characteristics is performed before t_0 and before the worker tasks block-and-wait on the reference task. The algorithms below show the implementation of the timing mechanism in the supervisor and worker tasks.

for i=1 to n

for j=1 to 1000

send fixed length message to task i

store transfer time

end j

$C(ref \leftarrow i) = \frac{total\ transfer\ time}{1000}$

end i

t_0 = result of timer poll

δ_t = 2seconds

$t_s = t_0 + \delta_t$

for i = 1 to n

t_i = result of timer poll

$\delta_{t_i} = t_s - t_i - C(ref \leftarrow i)$

send δ_{t_i} to task i

end i

suspend until t_s

(a)

for j=1 to 1000

receive fixed length message

end j

receive δ_{t_i} from supervisor

t'_i = result of timer poll

$t_{si} = t'_i + \delta_{t_i}$

suspend until t_{si}

commence RP solution

(b)

Timing mechanism in the supervisor (a) and worker (b) tasks

The impact of these visualisation and performance monitoring techniques on the optimization of the Transputer-based RP solution cannot be stressed enough. Most of the improvements to the program code arose as direct results of observations on charts of the form of Figure G.2. Accurate calculations of speed-up were only possible after the implementation of the timing and synchronous starting techniques.

Appendix H

Test Systems

The following pages give tree diagrams for the IEEE 118 node network and the reduced CEGB 629 and CEGB 734 node networks used as test systems for the research work described in this thesis. Tree diagrams for the 1624 node representation of the Eastern U.S power system could not be incorporated as they are too large to be printed. However coefficient matrix sparsity plots are given for all four systems. The sparsity plots show the structure of the matrix for a given system after optimal ordering and again after partitioning into the required number of subnetworks. The RBBDF matrix structure is apparent in the sparsity plots for the partitioned systems.

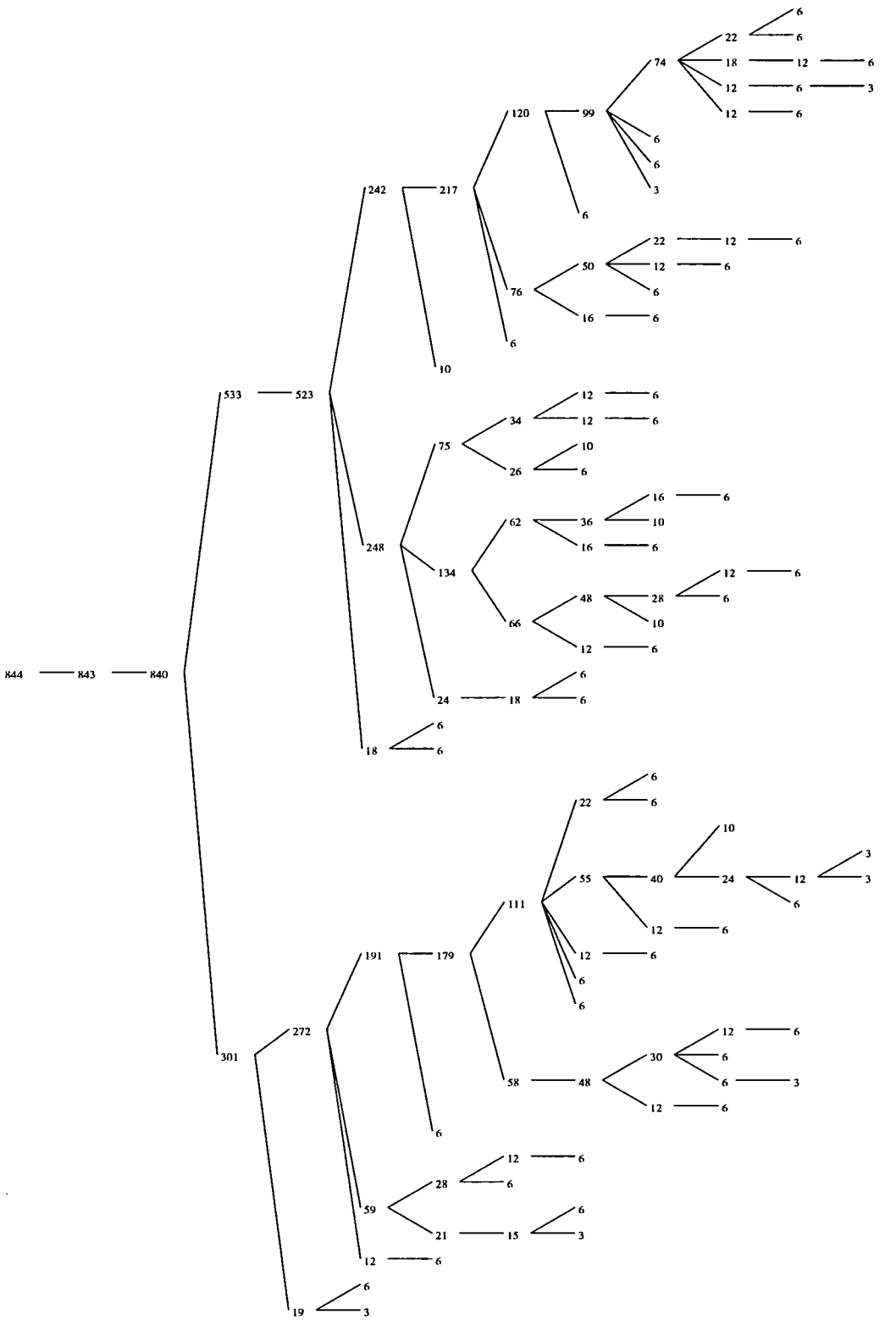


Figure H.1: Weighted elimination tree for MDMLLRU ordered IEEE 118 Node System

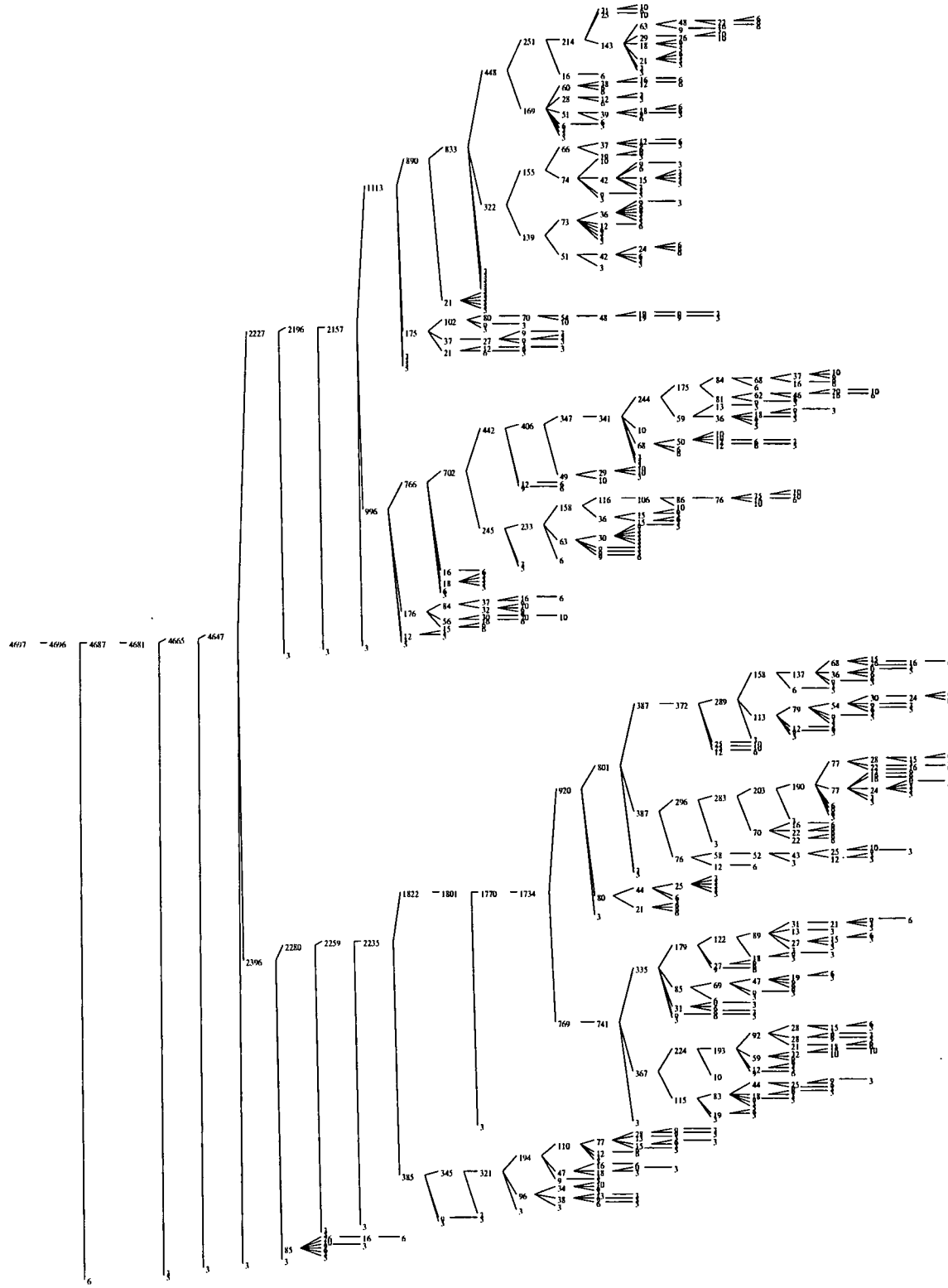


Figure H.2: Weighted elimination tree for MDMLLRU ordered CEGB 629 Node System

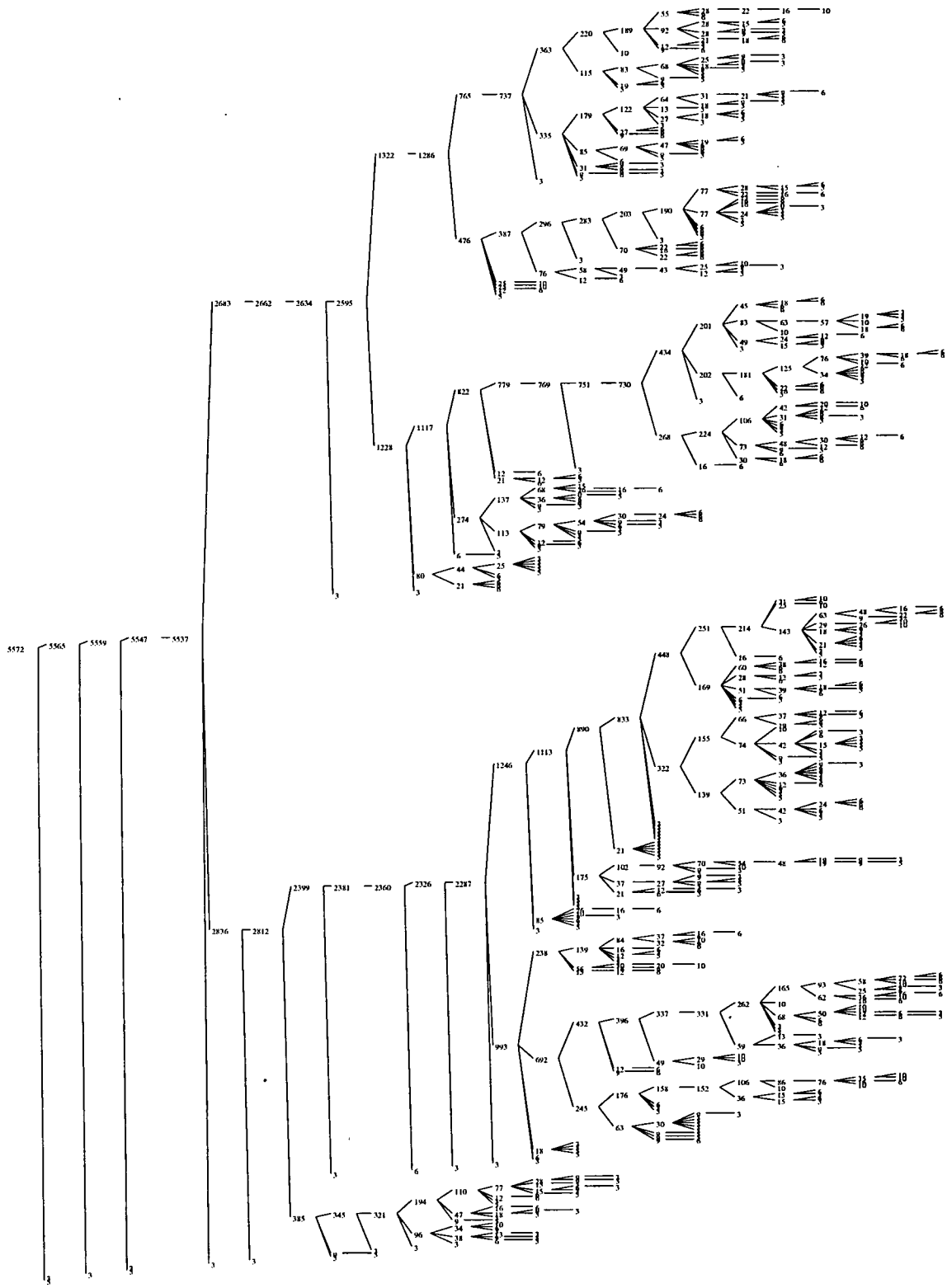


Figure H.3: Weighted elimination tree for MDMLLRU ordered CEGB 734 Node System

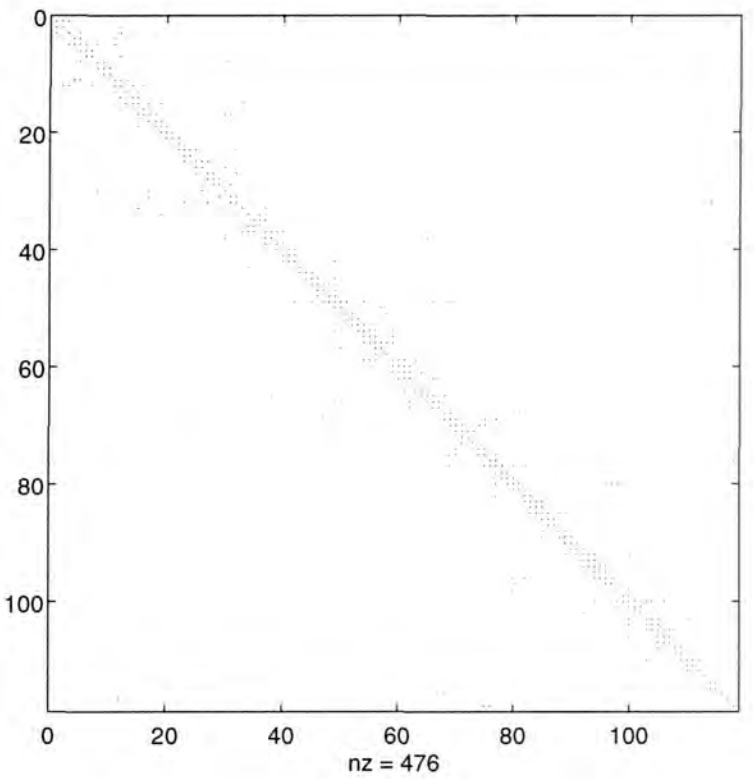


Figure H.4: MDMLLRU Ordered IEEE 118 Node System

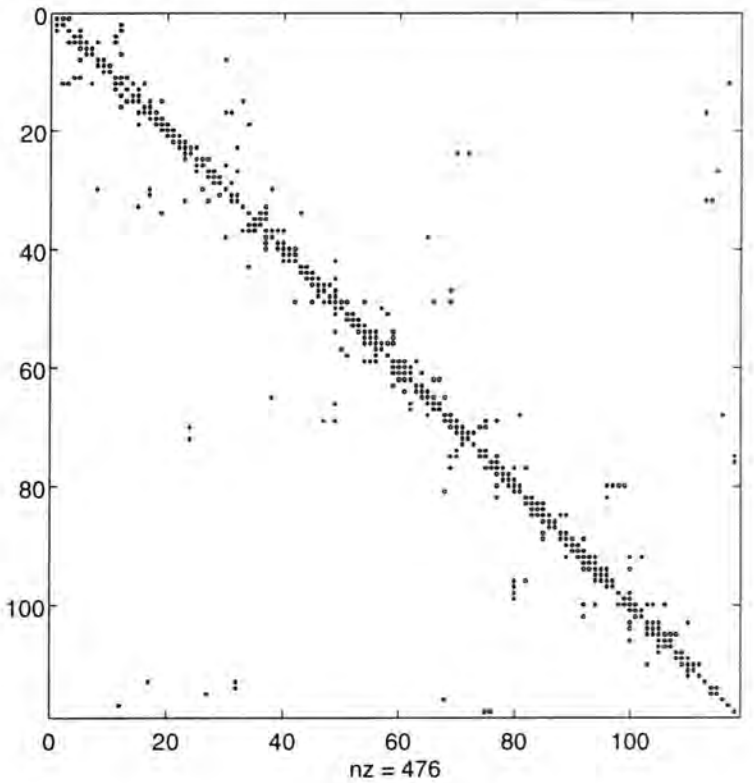


Figure H.5: MDMLLRU Ordered IEEE 118 Node System, partitioned into 2 major and 1 minor subnetworks

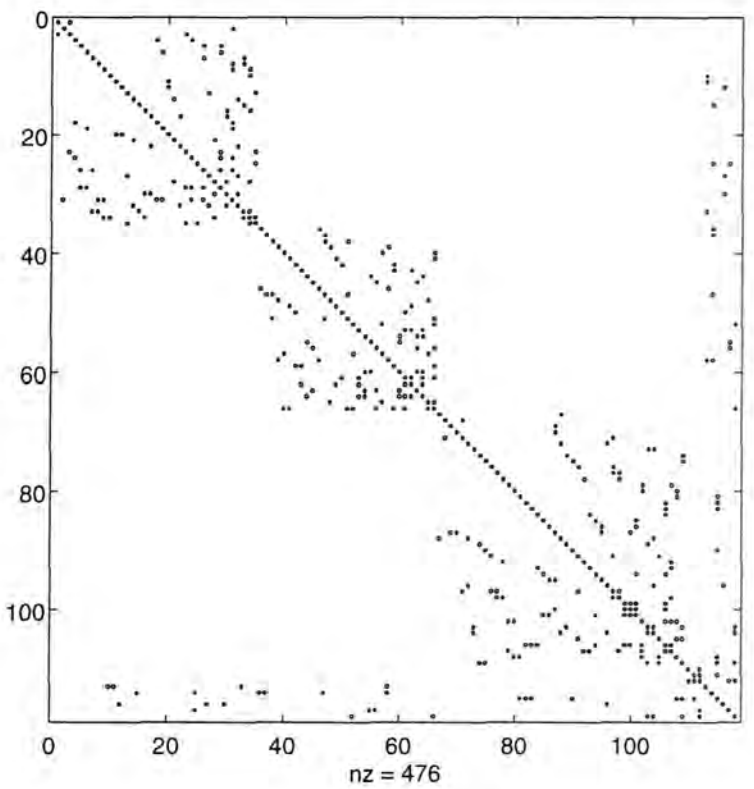


Figure H.6: MDMLLRU Ordered IEEE 118 Node System, partitioned into 4 major and 3 minor subnetworks

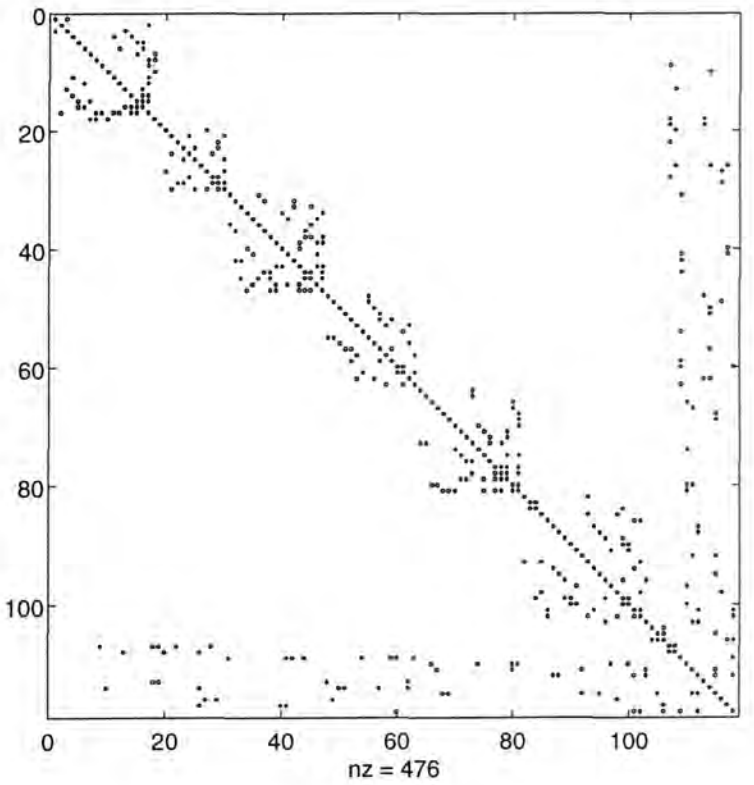


Figure H.7: MDMLLRU Ordered IEEE 118 Node System, partitioned into 8 major and 7 minor subnetworks

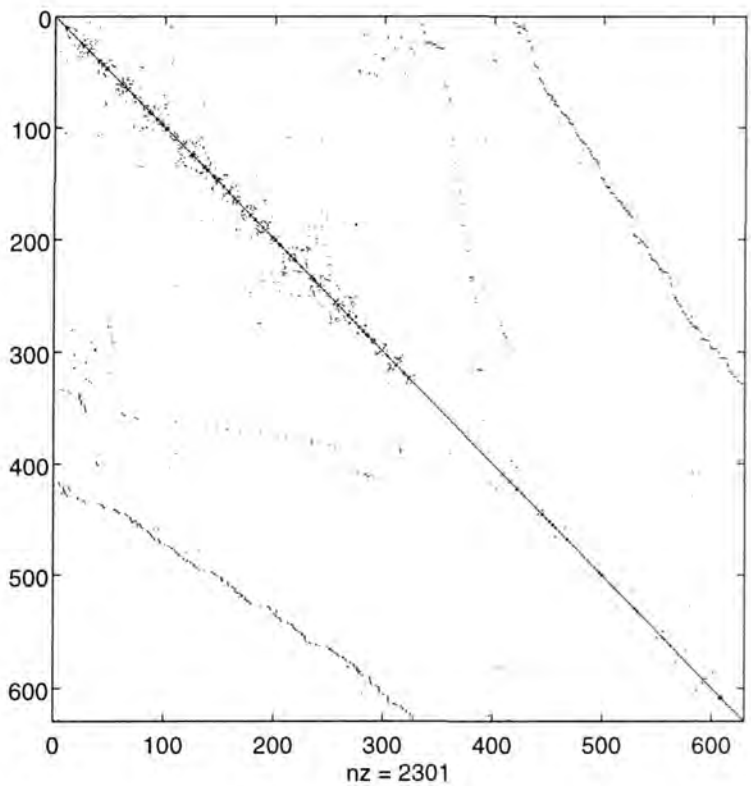


Figure H.8: MDMLLRU Ordered CEGB 629 Node System

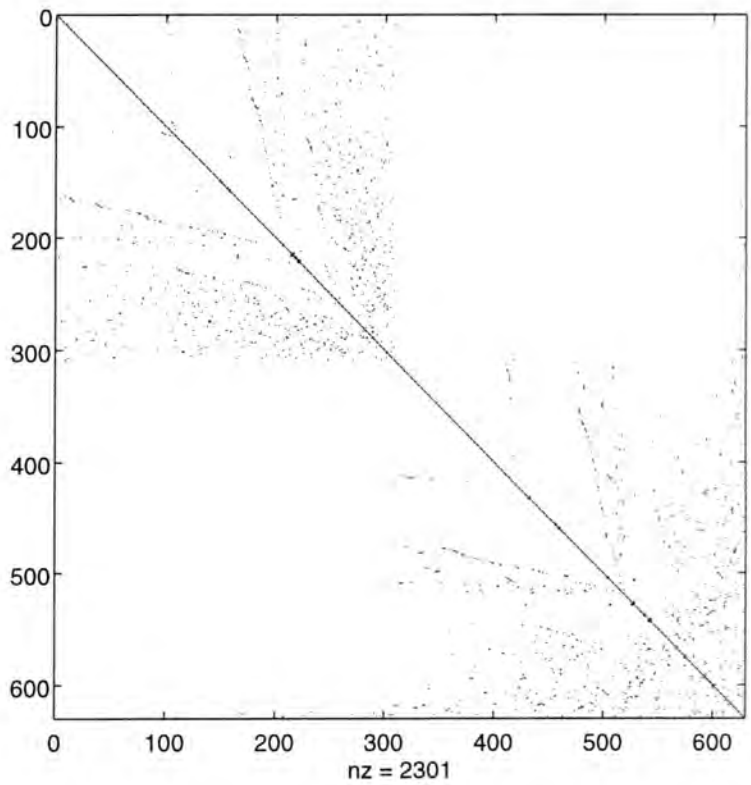


Figure H.9: MDMLLRU Ordered CEGB 629 Node System, partitioned into 2 major and 1 minor subnetworks

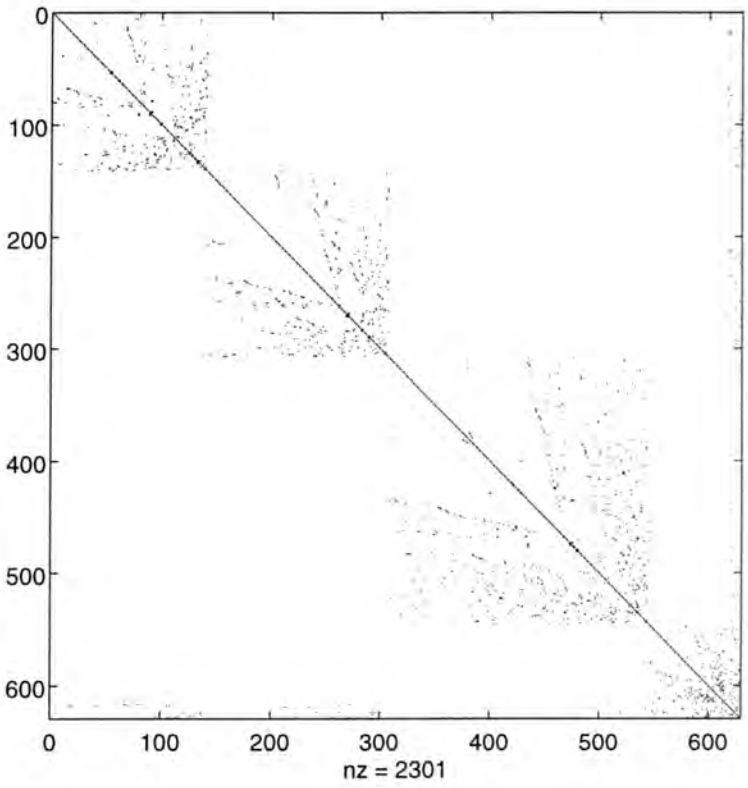


Figure H.10: MDMLLRU Ordered CEGB 629 Node System, partitioned into 4 major and 3 minor subnetworks

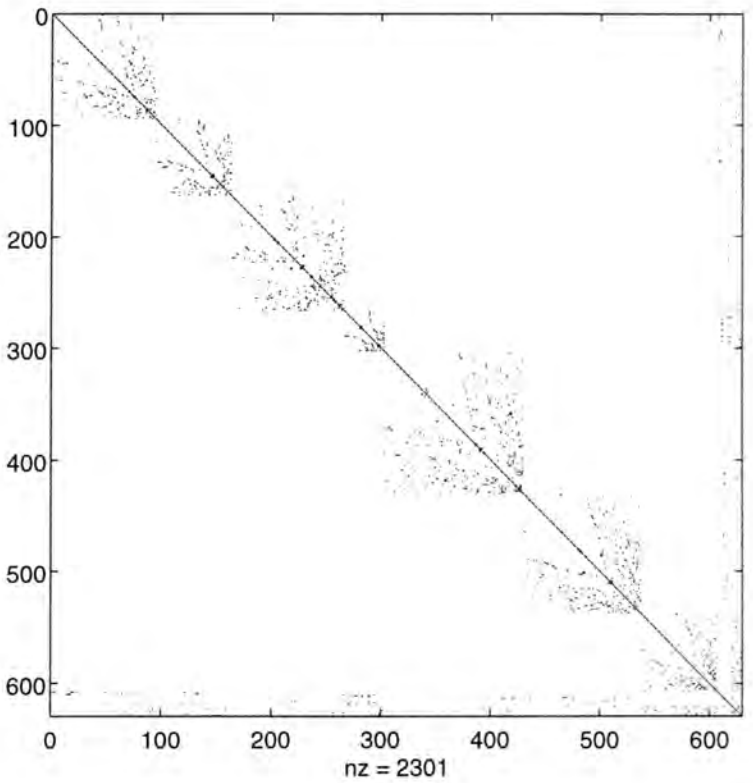


Figure H.11: MDMLLRU Ordered CEGB 629 Node System, partitioned into 8 major and 7 minor subnetworks

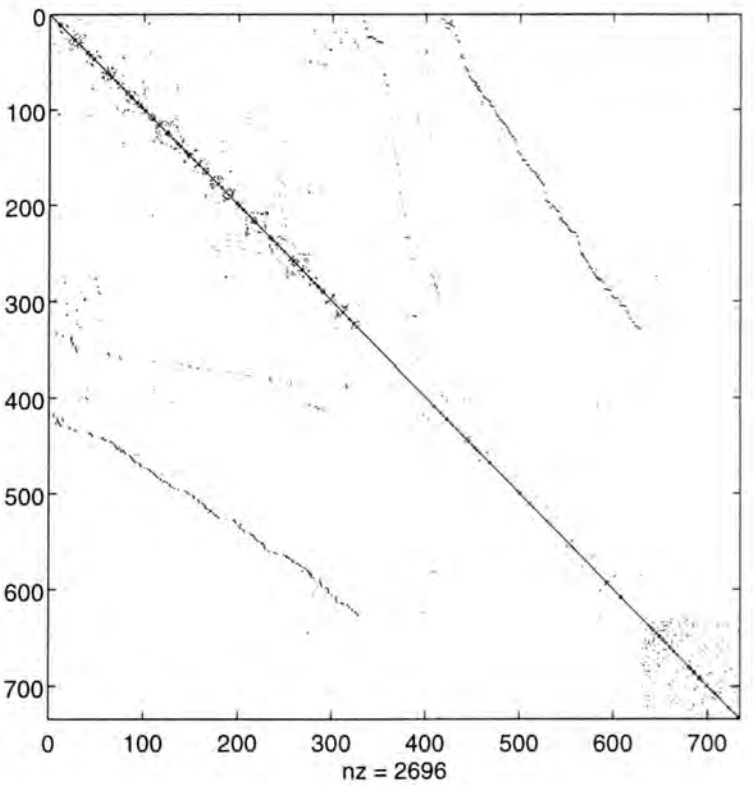


Figure H.12: MDMLLRU Ordered CEGB 734 Node System

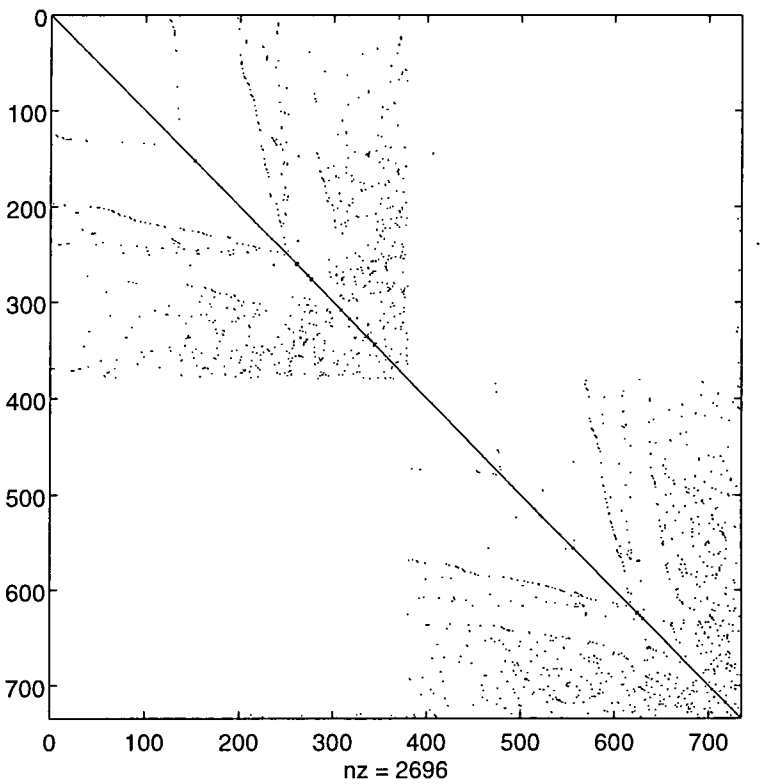


Figure H.13: MDMLLRU Ordered CEGB 734 Node System, partitioned into 2 major and 1 minor subnetworks

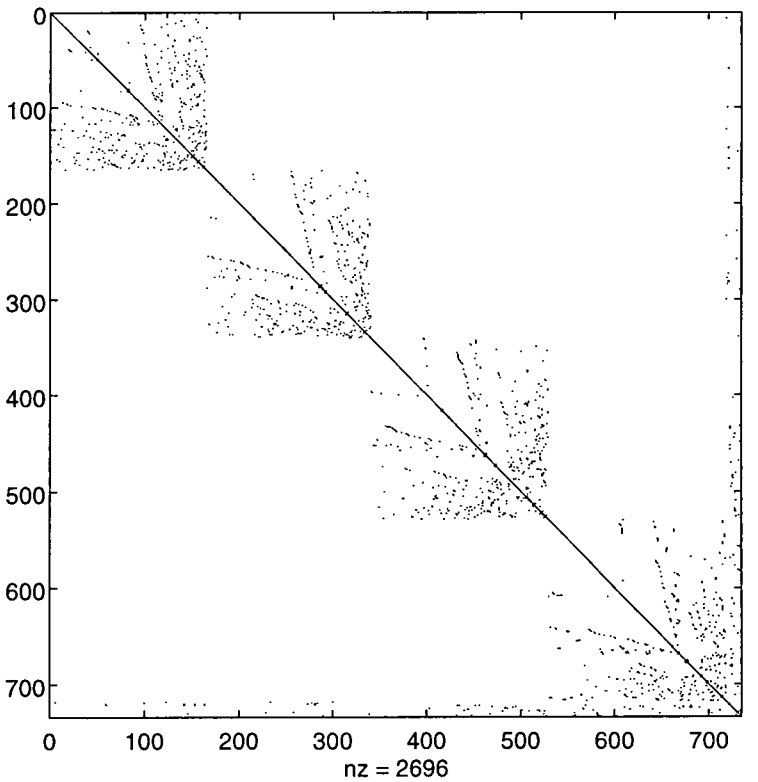


Figure H.14: MDMLLRU Ordered CEGB 734 Node System, partitioned into 4 major and 3 minor subnetworks

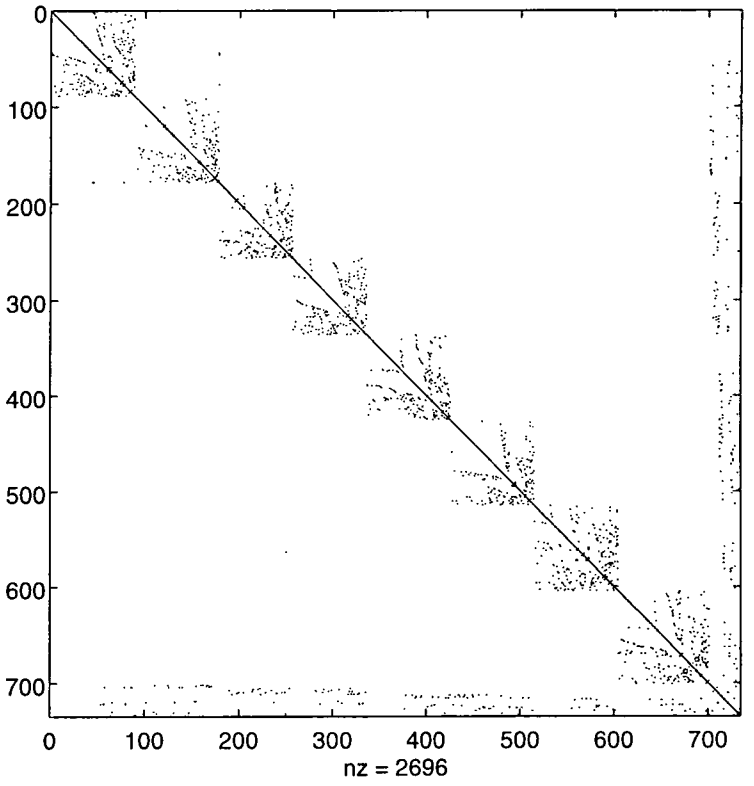


Figure H.15: MDMLLRU Ordered CEGB 734 Node System, partitioned into 8 major and 7 minor subnetworks

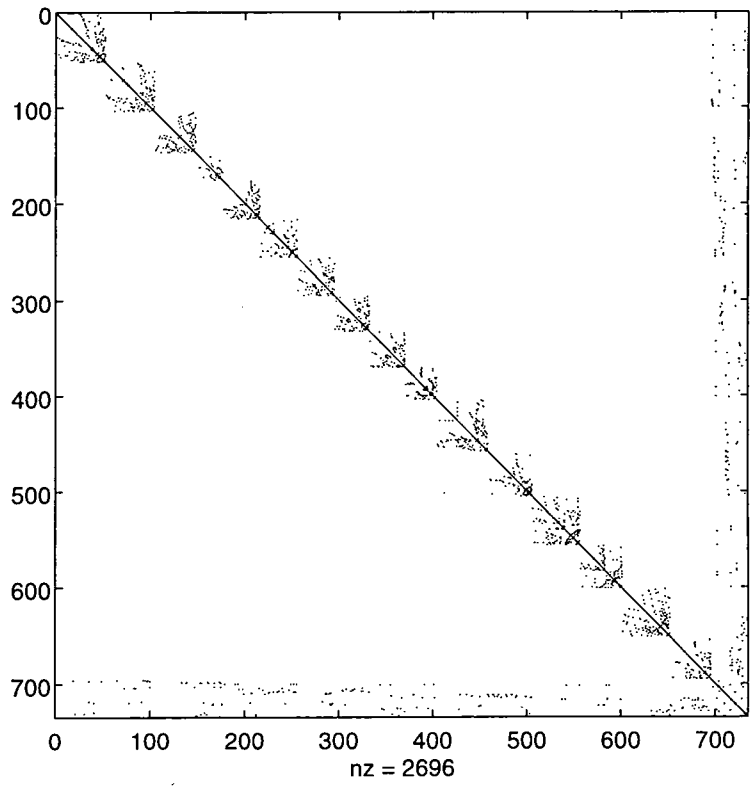


Figure H.16: MDMLLRU Ordered CEGB 734 Node System, partitioned into 16 major and 15 minor subnetworks

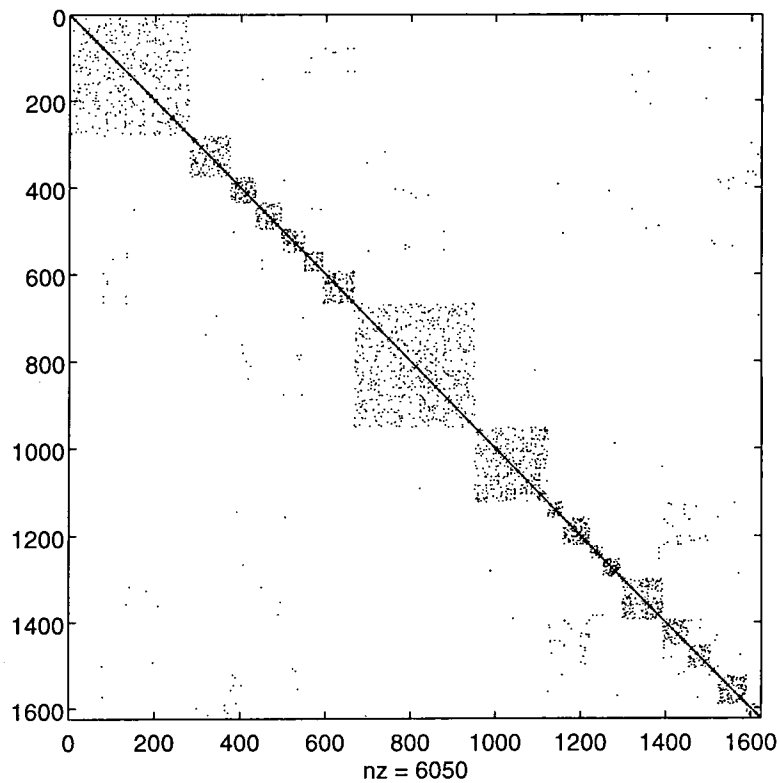


Figure H.17: MDMLLRU Ordered US 1624 Node System

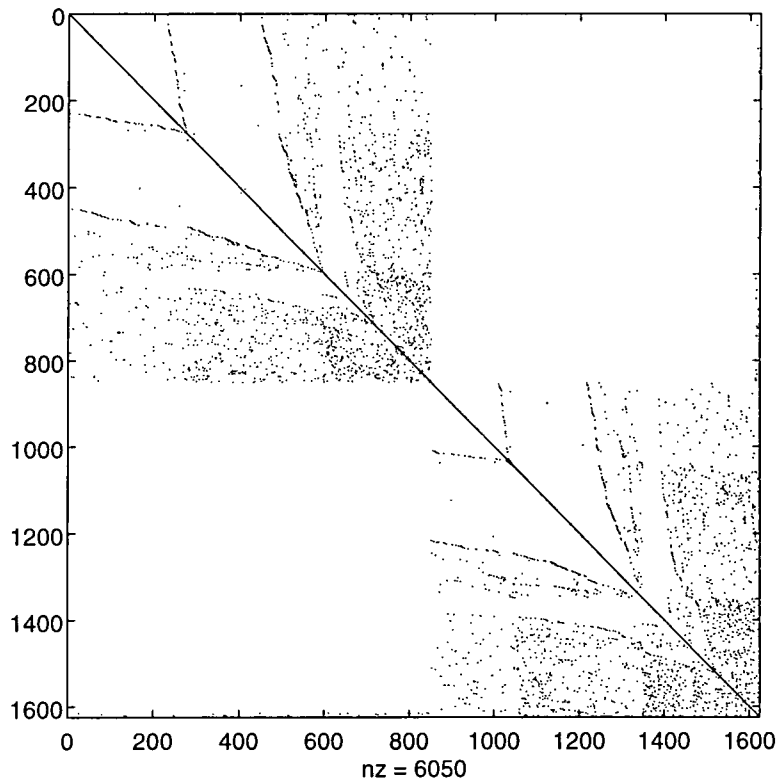


Figure H.18: MDMLLRU Ordered US 1624 Node System, partitioned into 2 major and 1 minor subnetworks

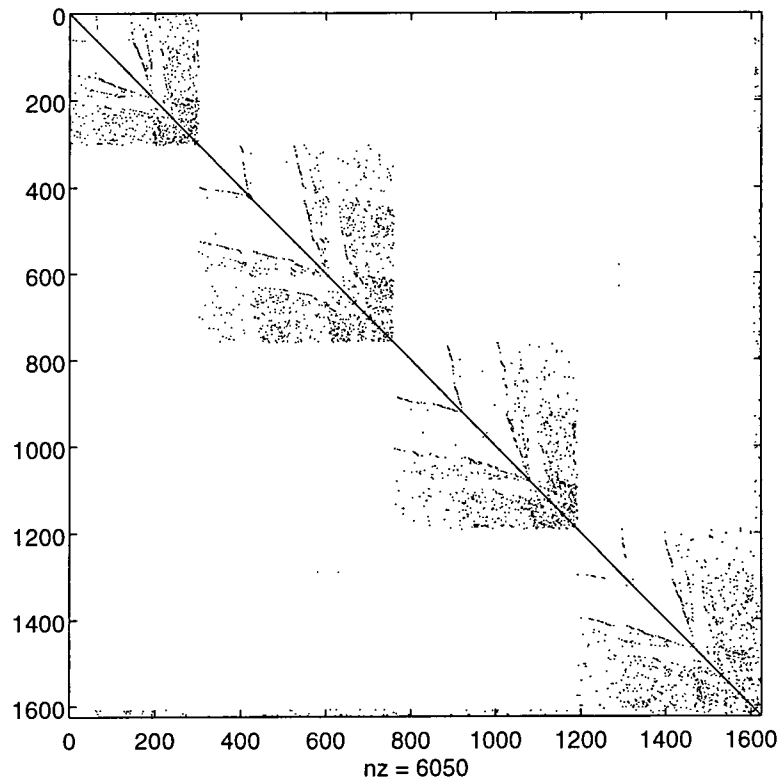


Figure H.19: MDMLLRU Ordered US 1624 Node System, partitioned into 4 major and 3 minor subnetworks

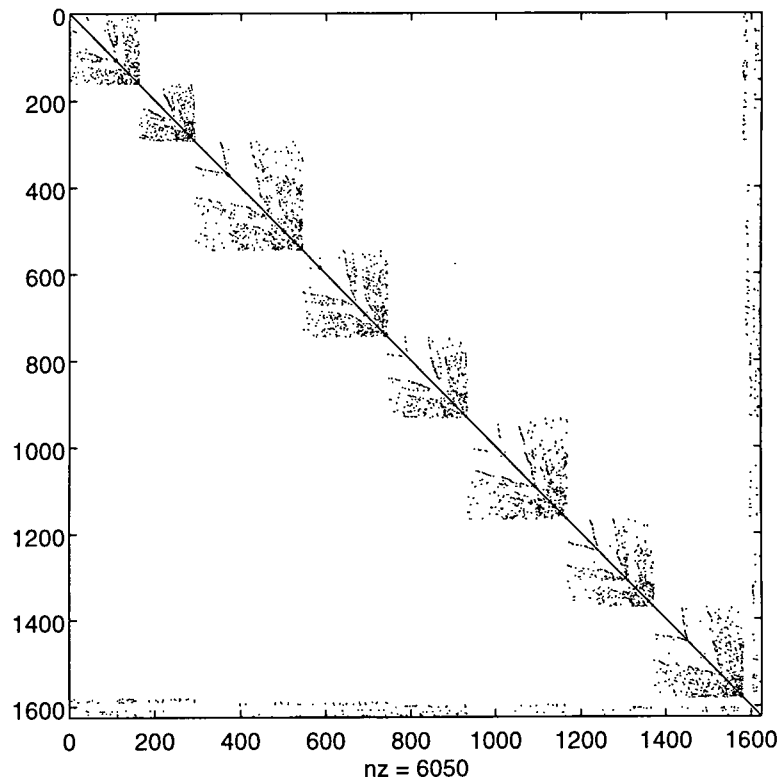


Figure H.20: MDMLLRU Ordered US 1624 Node System, partitioned into 8 major and 7 minor subnetworks

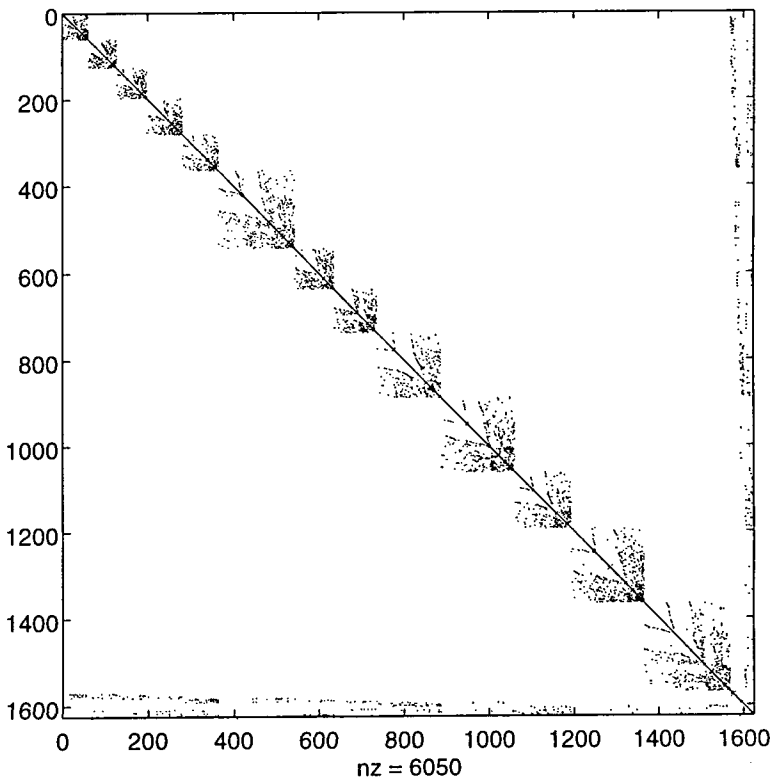


Figure H.21: MDMLLRU Ordered US 1624 Node System, partitioned into 16 major and 15 minor subnetworks