PARALLEL DISTRIBUTED COMPUTATION OF POWER SYSTEM TRANSIENT STABILITY PROBLEMS

By

Sallehhudin Bin Yusof, BSc, MEE

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of Doctor of Philosophy,
McMaster University

PARALLEL DISTRIBUTED COMPUTATION OF POWER SYSTEM TRANSIENT STABILITY PROBLEMS

DOCTOR OF PHILOSOPHY (1993)
(Electrical Engineering)

McMaster University
Hamilton, Ontario, Canada

TITLE: Parallel Distributed Computation of Power System Transient Stability Problems

AUTHOR: Sallehhudin B. Yusof, BSc (Southampton), MEE (UTM, Malaysia)

SUPERVISORS: Robert T. H. Alden, BASc, MASc, PhD (Toronto), PEng
Professor of Electrical and Computer Engineering,
Power Research Laboratory, McMaster University,
Hamilton, Ontario, Canada.

Graham J. Rogers, BSc (Southampton), PEng
Associate Professor (Part-time)
Power System Planning Division, Ontario Hydro,
Toronto, Ontario, Canada.

NUMBER OF PAGES: xxi, 223.

ABSTRACT

Transient stability analysis, a necessary tool for planning and operating electrical power systems, is most time consuming to use because of its massive computational burden. Therefore, the solution of this problem on multiprocessor systems has attracted a lot of attention and holds the promise of eventual real-time dynamic security assessment.

In this thesis, methods to speedup the computation of power systems transient stability simulations using parallel distributed processing are examined. The approach of the thesis is twofold. Firstly, practical implementation must consider the use of current widely available networked computers in the industry. Secondly, the parallel distributed algorithms to be developed in this thesis must as far as possible retain algorithmic advances that have been made on its serial counterpart because of their many advantages.

Serial transient stability algorithms are studied and all major algorithmic advances are investigated and implemented. State-of-the-art parallel transient stability calculations are reviewed and for each algorithm the parallelism that has been exploited is identified. The choice of parallel processing hardware and software is of major concern to power utilities. These issues are discussed and justifications for using networked workstation are given.

The two approaches mentioned above lead naturally to coarse-grained partitioning of the transient stability problem. A slow coherency network partitioning that divides the network into several coherent areas along naturally occuring weak links is developed in this thesis. Following network partitioning, computational tasks can be distributed to several computers and then executed simultaneously.

Solution of large, sparse algebraic linear systems is required in most

power system analysis. In transient stability analysis, the solution of the linear transmission network is the major obstacle to real-time simulations. In this thesis, both direct and iterative methods for solving blocked linear algebraic systems associated with the power network are developed and implemented on a cluster of workstations. RPC (Remote Procedure Calls) techniques together with multiprocessing primitives are used for client-server communications. These aspects and techniques are fully discussed in this thesis.

Based on the network partitioning and solution methods developed, a transient stability algorithm is parallelized. The process of parallelization is carefully demonstrated. Both shared memory multiprocessor and RPC based - distributed memory - versions are developed. The corresponding speedup due to these methods are analyzed and discussed. It is shown that techniques developed in this thesis can be applied to production grade transient stability programs.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my supervisor, Dr. R.T.H. Alden for his invaluable guidance and constant encouragement throughout the course of this work. This work would not have been possible without the initial and persistent encouragement, guidance and ideas of Mr. G.J. Rogers, my co-supervisor to whom I am greatly indebted. I specially express my thanks to Dr. B. Szabados for his guidance in parallel distributed processing.

Some parts of the work were carried out at Ontario Hydro. My special thanks to Dr. P. Kundur, the Manager of Analytical and Specialized Methods Department of the Power System Planning Division for allowing the use of computer facilities and providing guidance and encouragement. I must also express my sincere thanks to other engineers at Hydro especially Mr. Sainath Moorthy, Dr. Lei Wang and Mr. Dave Wong with whom I have many fruitful discussions..

My work at the Power Research Laboratory at McMaster University has been made very pleasant by colleagues who are both helpful and friendly. I would like to thank all of them and specially Ming Chan - my C guru. Not only have I had many academic discussions with them but also political and social debates that have made me more understanding of Canadian culture and humour.

My special appreciation goes to my beloved wife, Dr. Nor Aidah, who has patiently endured the long separation and continues to give her support and encouragement. My thanks to my two sons - Arif and Adib - for their understanding, to whom I promise to make up for the lost time. I am most indebted to my parents who have given me all the encouragement and instilled in me the spirit of endurance.

Finally, I would also like to thank my company, Tenaga Nasional

Berhad (TNB) of Malaysia for financial support. Special thanks to officers of the TNB. Mr. Muhammad Ghazi Hassan, Mr. Tajudin Mohd Arif, Dr. Amir Bashah and Mr. Abdul Aziz Abdullah for their encouragement and support. There are many other colleagues in the company that I have derived inspirations from but I like to specially mention Mr. Ahmad Jaafar, Mr. Masri, Mr. Joon, Mr. Rosman, Mr. Cham, Mr. Adan, and Mr. Yusof Rakob.

TABLE OF CONTENTS

LIST OF PRINCIPAL SYMBOLS

*	denotes complex conjugate
(f)	denotes final value in the Modified Euler method
(o)	denotes previous value in the Modified Euler method
(n)	denotes estimated value in the Modified Euler method
$\Delta a_{ij}(t)$	difference between rotor angles of machine i and j at time t (radians)
ΔI_B	change in the complex current vector (p. u.)
$\Delta\theta$	vector of change in the voltage angle (radians)
$\Delta\|V\|$	vector of change in voltage magnitude (p. u.)
ΔV	change in the complex voltage vector (p. u.)
ΔV_r	vector of change in the real component of V (p. u.)
ΔV_x	vector of change in the imaginary component of V (p. u.)
Δx	change in the state vector (p. u.)
δ	rotor angle (radians)
δ_{ij}	difference in voltage angles between machine i and j (radians)
$\delta_i(t)$	rotor angle for machine i at time t (radians)
δ_0	vector of the initial rotor angles (radians)
ε	a small positive number
λ	an eigenvalue
∞	denotes infinity
ω_0	synchronous frequency - 2πfreq - (radians/sec)
σ	proportion of the total computation to be performed in serial
θ	voltage angle (radians)
θ_{vi}	eigenvector of change in voltage angle
τ_1	execution time of a program using one processor (sec.)
τ_{np}	execution time of a program using n_p processors (sec)
τ_N	number of time steps over a simulation period
A	a general sparse square matrix
$\mathbf{A_D}$	the diagonal part of matrix A
$\mathbf{A_{ii}}$	element ii of the state matrix $\mathbf{A_S}$
$\mathbf{A_{ij}}$	element ij of the state matrix $\mathbf{A_S}$
$\mathbf{A_L}$	lower triangular part of matrix of A
$\mathbf{A_O}$	block bordered part of matrix of A
$\mathbf{A_S}$	the state matrix
$\mathbf{A_U}$	upper triangular part of matrix of A
b	the r. h. s. vector in a linear equation, **Ax = b**

B	rectangular matrix in $\dot{x} = Ax + Bu$
B_O	the bordered part of vector b
B_{ij}	an imaginary component of element ij of the reduced admittance matrix
B_N	imaginary part (matrix) of the complex admittance matrix Y
C	a matrix of constant elements
C_P	communication penalty
d-, q-	machine rotor frame direct- and quadrature axes
D-, Q-	network frame Direct- and Quadrature- axes
D_i	damping constant of machine i
e_j	vector that contain all zeros except for the jth component
E'	voltage behind the transient reactance (p. u.)
E''_d	direct-axis subtransient voltage (p. u.)
E''_q	quadrature-axis subtransient voltage (p. u.)
E_F	field voltage (p. u.)
E_q	machine q-axis voltage (p. u.)
f	function of the state and voltage variables - differential equation
freq	synchronous frequency (Hz)
F	algebraic form of function f
g	function of the state and voltage variables - algebraic equation
g_r	function of the real part of the complex current injection
g_x	function of the imaginary part of the complex current injection
G	algebraic form of g
G_C	matrix of the direction cosines
G_{ii}	real component of element ii in the reduced admittance matrix
G_{ij}	real component of element ij in the reduced admittance matrix
G_N	imaginary part (matrix) of the complex admittance matrix Y
h	integration time step
H	compacted form the combined equations of F and G
I	identity matrix
I_B	vector of the bus currents (p. u.)
I_T	machine terminal current (p. u.)
I_d, I_q	machine rotor d- and q-axis currents (p. u.)
I_D, I_Q	network D- and Q-axis currents (p. u.)
I_r, I_x	the real and imaginary vectors of I_B (p. u.)
J_A, J_B, J_C, J_D	submatrices of the augmented state matrix
J_1, J_2, J_3, J_4	Jacobian submatrices in the Newton iterative method
J_d	diagonal block of the Jacobian matrix
J_O	off-diagonal part of the Jacobian matrix

k	denotes the iteration count
k_c	a constant multiple of integration time step h
K_p, K_q	vectors of initial load constant in the non-linear load model
L	grouping matrix
M	a vector of the machine inertias
M_i	inertia for machine i
n-2	denotes previous two time steps
n-1	denotes previous time step
n	denotes current time step
n+1	denotes next time step
n_b	the number of network bus
n_m	the number of generator bus
n_p	the number of processors
P	vector of active power (p. u.)
P_{ei}	the airgap electrical power for machine i (p. u.)
pf	the non-linear active load model frequency exponent
P_{mi}	the mechanical input power for machine i (p. u.)
P_P	partition penalty
pv	the non-linear active load model voltage exponent
Q	vector of reactive power (p. u.)
qf	the non-linear reactive load model frequency exponent
qv	the non-linear reactive load model voltage exponent
r	denotes number of areas
R_S	stator resistance (p. u.)
S	speedup
S^2	serial slowdown factor
t	denotes time
t_1	starting time (sec)
t_2	ending time (sec)
T_e	electrical torque (p. u.)
T_{dq}	network to rotor frame transformation matrix
T	denotes transpose of a matrix or a vector
T_N	number of time steps
T_P	execution time on a single processor of the partitioned problem (sec)
T_C	execution time of the parallel distributed program including all communications (sec)
T_T	execution time of the parallel distributed program when all communications are neglected (sec)
u	vector of stator feedback quantities
u_i	right eigenvector associated with eigenvalue λ_i
u_{vi}	extended right eigenvector associated with eigenvalue λ_i
U	eigenbasis matrix
U_1	first r rows of U
U_2	r + 1 to n_b rows of U
U_V	extended eigenbasis matrix

U_θ	angular component of U_V
V	complex voltage vector (p. u.)
V_d, V_q	rotor frame direct and quadrature axes voltage (p. u.)
V_D, V_Q	network frame direct and quadrature axes voltage (p. u.)
V_i	voltage at bus i (p. u.)
V_j	voltage at bus j (p. u.)
V_{ar}	complex voltage vector for area r
V_0	vector of the initial complex voltage (p. u.)
V_r, V_x	real and imaginary components of V (p. u.)
w_i, w_j	row i and j of the extended right eigenvector in V-θ coordinates
x	state vector in the differential equations
X_d	direct-axis synchronous reactance (p. u.)
X'_d	direct-axis transient reactance (p. u.)
X''_d	direct axis subtransient reactance (p. u.)
X''_q	quadrature axis subtransient reactance (p. u.)
x_{ar}	state vector of area r
y	state vector in the algebraic equations
Y	network admittance matrix (p. u.)
Y_D	diagonal part of the admittance matrix
Y_O	off-diagonal part of the admittance matrix
z	a vector due to $A_D A_U x$
Z_{dq}	generator source impedance matrix (p. u.)

LIST OF FIGURES

LIST OF TABLES

LIST OF TEXT BOXES

LIST OF ALGORITHMS

CHAPTER 1
INTRODUCTION

Electric power systems are large and complex. The systems comprise of generating stations, transmission networks, substations, distribution lines and electrical loads. The generating stations are interconnected through the transmission systems and may cover vast geographical areas. Transmission voltages are transformed at substations before the electric energy is distributed to customers. Substations vary in size and are mostly located near load centres. The complexities of power systems arise from the interactions of these many devices and involve a multitude of variables. A customer, an end-user of electricity, is seldom aware of the complexities involved in operating and maintaining the power systems from which the electricity energy is derived. This indicates that power engineers have been able to operate these complex systems in an acceptably secure manner. However, with continuous expansion of the power system and the growth in electricity demand, power engineers are faced with new and larger problems. Therefore, power engineers are constantly searching for methods to solve these new problems as well as solving the existing problems more efficiently and quickly.

Maintaining system stability is central to a continuous and secure operation of the power system. As emphasized by Dy Liacco [1977], the prime objective of power system operation is to maintain a continuous supply of electric energy to all customers served by the system. Loss of supply following system instability would result in massive economic losses to both the power producers and the customers. Widespread and prolonged interruptions result in disorders to other public amenities such as traffic control, underground

be caused by short circuits on transmission buses, loss of generators and loss of major transmission lines. One of the major objectives of the design and operation of a power system is to ensure that the system remains stable under these various disturbances, large and small.

Kimbark [1948], defined **power system stability as a term applied to alternating-current electric power systems, denoting a condition in which the various synchronous machines of the system remain in synchronism, or "in step", with each other.** This definition has now become a subset of the overall definition of power system stability because instability may also be encountered without the loss of synchronism. As an example, a deficiency in reactive power may progressively bring down the voltage level until voltage collapse occurs.

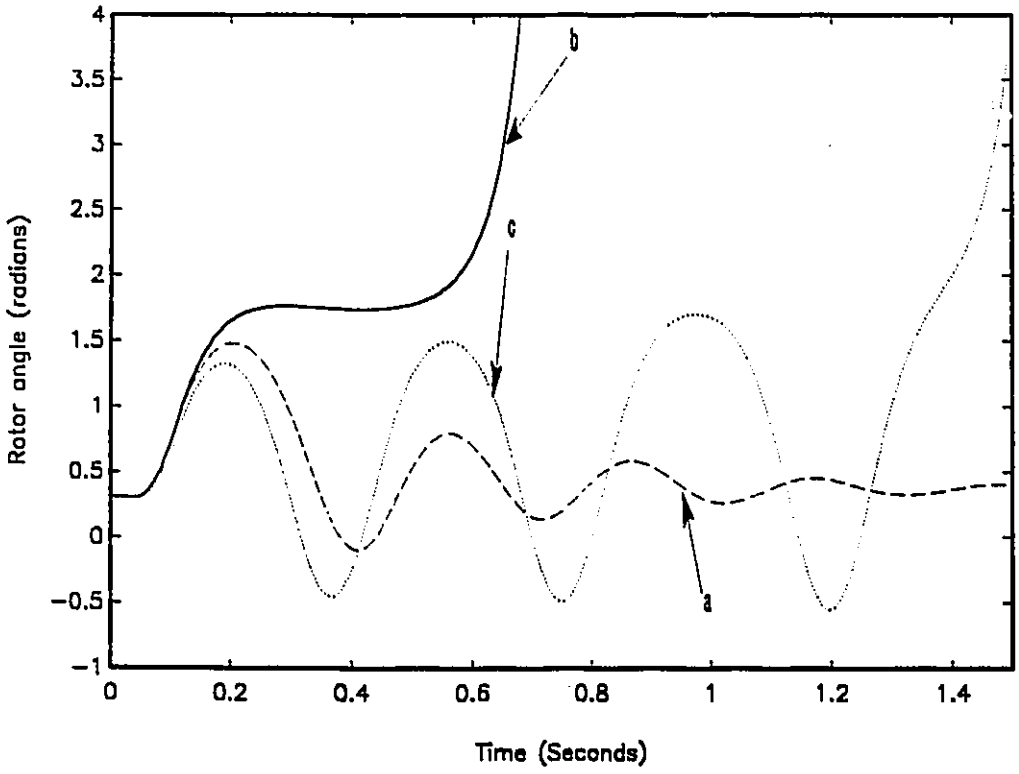

Figure 1.0: Stable and unstable conditions

A more general definition of **power system stability is the ability of a power system to remain in the state of operating equilibrium under normal**

operating conditions and to return to an acceptable state of equilibrium after a disturbance (IEEE Std. 100 [1988]) as depicted in graph (a) of figure 1.0. Power system stability can be broadly classified into (i) rotor angle stability, and (ii) voltage stability (Concordia et. al., [1990]). Rotor angle stability is concerned with maintaining synchronism and balance between the mechanical and electrical torques of a generator. Voltage stability is concerned with maintaining acceptable voltage levels at all buses in the system under normal operating conditions and after a disturbance.

Rotor angle stability can be further divided into (i) transient stability, and (ii) small signal stability (IEEE Task Force on Terms and Definition [1982]). Transient stability is concerned with the response of the power system to severe disturbances, such as a three phase short circuit on a transmission bus or loss of a large generating unit, that are likely to occur during the operation of the system. The transient stability study period is limited to between 1 and 20 seconds. The instability following a severe disturbance is invariably caused by insufficient synchronizing torque to maintain synchronism as shown in graphs (b) and (c) of figure 1.0. The behaviour of the power system is highly non-linear. In transient stability analysis, the non-linear differential equations describing the power systems are solved in a step-by-step manner. This thesis is concerned with improving these solution techniques.

Small signal stability is concerned with the power system response due to small disturbances. A power system is always experiencing small disturbances. In small signal stability analysis, one is interested in the ability of the system to return to a steady state following the disturbance. Normally small signal instability causes growing oscillations which may lead to loss of synchronism as shown in graph (c) of figure 1.0. Because the interactions which cause small signal instability are basically linear in nature, small signal stability analysis is normally done on linearized power system equations by applying linear algebra techniques, such as eigenvalues and eigenvectors (Rogers and Kundur [1989]).

1.2 APPLICATIONS OF TRANSIENT STABILITY SIMULATIONS

Loadflow and transient stability analyses are the two most widely used analytical tools in the planning and operating of power systems. Both of these tools are invaluable in power system analysis. The following sections describe the applications of transient stability analysis in planning and operating the power systems.

1.2.1 Power System Planning

The object of power system planning is to choose between several alternative expansion plans. Each plan meets the forecasted future energy sales and demands over an horizon of interest (usually between 5 to 20 years in the future). Each plan is tested against several technical criteria including steady-state (voltage, thermal limits) and transient stability requirements designed to ensure reliability under normal and contingency conditions. Many simulations have to be performed in order to exhaustively examine the various plans. Economic evaluations and comparisons are then made to choose the least cost plans when all the technical criteria are satisfied. In the present age of growing public concern on the environment, power system planners must also subject their plans to EIA (Environmental Impact Assessment) studies.

In preparing for stability simulations, several load flow conditions, normally extreme cases of high and low demands, are prepared for each alternative plan. The data of the expected generators to be scheduled, together with essential dynamic devices such as the static var compensators (SVC), are also prepared. Transient stability simulations are then performed using several pre-determined contingencies - normally dictated by the planning criteria used by the utility. Depending on the results, further simulations may be performed using new contingencies.

Much time is spent in waiting for the results of the simulations before any assessment can be made. Decisions for modifying any plan depend on the results of the simulations. In many cases, additional simulations are required

to be performed for further examination of the plans. This could be due to the need to change generation scheduling, to reinforce transmission circuits or even to properly tune control devices. In power system planning, improvement in the speed of transient stability simulations would mean that important planning decisions could be made more quickly and that, the productivity of planning engineers could increase. Another way of looking at the benefit of improved simulation speed would be that, for the same allowable time for planning, many more simulations could be performed to thoroughly examine those plans.

1.2.2 Power System Operations

In power system operations, there is a need to employ transient stability simulation for dynamic security assessment. From the perspective of power system operations - maintaining continuous supply - security may be defined as the probability of the system's operating point remaining in a viable state space, given the probabilities of contingencies (Balu et. al., [1992]). However, a more practical definition by The North American Electric Reliability Council defines security in terms of its objective - **prevention of cascading outages when the bulk power system is subjected to severe disturbances**. To avoid cascading outages, security analyses are conducted to ensure that the following conditions are satisfied: (i) all transmission elements are operating within their thermal limits, (ii) voltages on all buses are within a given limit (usually ±5% of nominal), and (iii) the system will recover to an acceptable steady-state operating condition after being subjected to some pre-selected disturbances. The first two conditions are assessed using steady-state analysis, that is, load flow calculation. The third condition requires transient stability simulation.

Presently, in most utilities, dynamic security analyses are conducted by off-line studies using a transient stability analysis program. However, there is increasing demand for fast transient simulation which can be incorporated into the Energy Management System and which can then be used

to determine the critical system limits to contingencies based on the current system conditions. This is the particular application where speed of computation is critical. In such on-line dynamic security analysis, the base case derived from real time conditions is subjected to pre-selected contingencies. The dynamic response of the system is examined, its effects on the system are determined and all expected outages are identified.

1.3 METHODS TO IMPROVE THE SPEED OF STABILITY SIMULATIONS

The speed of transient stability simulations can generally be improved by three methods: (i) using faster computers, (ii) new algorithmic development, and (iii) using parallel, distributed processing. In the first method, one would simply acquire a faster computer and execute the same code. This is the current practice of many power utilities. This option may satisfy the needs of small utilities, but, for others who are operating large interconnected systems, the sluggish computation speed limits the use of transient stability simulation in system operations. While the transient simulation tools have improved considerably over the last 15 years, the improvement has been mainly in modelling complexity and user friendliness rather than in improved algorithms. Mitsche [1993] predicts that the impact of new mathematical methods or algorithms in power system analysis will be at best evolutionary and not revolutionary. The basic algorithm for transient stability simulation remains that of the step-by-step solution of the combined set of differential and algebraic equations (DAE) in the form of:

$$\dot{y} = f(y,x) \tag{1.1}$$

$$0 = g(y,x) \tag{1.2}$$

where y are the state variables in the differential equations and x are the state variables in the algebraic equations. Equation 1.1 describes the dynamics of the power systems and equation 1.2 describes the static network. These equations are now solved serially. Efficient coding and the use of algorithm, such as the Trapezoidal method, has increased the speed of simulation, but the improvement has been small compared to the increase in

the speed of computer processors. The speed of computers will continue to increase and can be augmented by the use of parallel processing. The emergence of parallel, distributed processing has opened a new path for power system engineers to improve the simulation speed of many power system tools. However, the traditional serial algorithms have to be modified or new algorithms have to be developed in order to exploit the use of many processors in parallel.

Parallel processing is the most promising method to achieve substantial improvement in transient stability simulation speed. Transient stability simulation appears to be the main power system problem that will benefit from the use of parallel processing. But as summarized by IEEE Committee Report [1992] in a report on the status of parallel processing in power systems - *in terms of algorithms, there is no clear cut path for the development of production grade tools and in terms of hardware there is too much uncertainty in its choice.* However, Mitsche [1993] suggested that - *distributed computing environments of Unix-based hardware and software promise a simple means to utilize multiple computers in parallel.*

1.4 PARALLEL, DISTRIBUTED PROCESSING

The need to solve problems of ever increasing size and complexity is the main driving force for the development of faster computers (Modi [1988]). Even with high MIPS (Millions of Instructions Per Second) capability, a single computer is not yet able to meet the desired solution time of many existing and new problems in view of their massive computational needs. The only recourse is to have several computers cooperating in solving a problem.

A problem that is a good candidate for parallel, distributed processing must be decomposable. The objective of the decomposition is to divide the problem into smaller sub tasks. An important property to be maintained is that, the interactions between these sub tasks must be minimized.

The term parallel, distributed computation implies the use of MIMD (Multiple Instructions Multiple Data) model rather than SIMD (Single

Instruction Multiple Data) model. The terms SIMD and MIMD were first defined by Flynn [1966]. With SIMD, all processors execute the same instructions in lockstep but on different data. In MIMD machines, all processors may be pursuing the same goal or solving a single problem or they may be solving many problems at the same time. In MIMD machines, each processor has its own memory and can be executing different programs.

Workstations connected in a network are loosely coupled and there is no central control. But the computers can communicate with each other through the computer network. Unlike centralized parallel computers in which the processors can communicate at high speed either through shared memory or high speed buses, communication between workstations in a network is comparatively slow. Therefore, problems that require intensive communications between processors are not suitable to be solved in a network of computers. Problems with highly intensive local computations and little interprocessor communications are therefore most appropriate. In this thesis, the transient stability problem is formulated to have these characteristics.

1.5 OBJECTIVE OF THE THESIS

In the previous sections, transient stability simulations have been shown to be of central importance in the planning and operating of power systems. The needs for improvement in transient simulation speed have also been discussed. ***Therefore, the main objective of this thesis is to investigate the use of parallel, distributed processing to improve the computation speed of transient stability simulations.***

To achieve this objective, several parallel transient stability algorithms have been studied. The parallelisms that have been exploited in each algorithm are examined and their advantages and disadvantages investigated. The partitioning approach used in the thesis is of coarse grain type. Therefore, network partitioning methods are investigated. Their use in parallel, distributed processing of a transient stability problem are described. In this thesis, a more practical implementation approach is adopted. The use

of widely available clusters of workstations is examined and implemented for the simulations. From the results of the simulations, the algorithms and the implementation techniques are shown to meet the objective.

1.6 ORGANIZATION OF THE THESIS

The thesis is made up of eight chapters. Each chapter discusses a particular topic of relevence to the thesis and the contributions made are described. The following paragraphs briefly describe the contents of each chapter of the thesis.

■ **Chapter 1: Introduction** - In this chapter, power system stability and its categories are introduced. Brief descriptions of transient, small signal and voltage stabilities are given. The applications of transient stability analysis in the planning and operation of power systems are discussed. The needs for faster transient stability simulations are justified based on the requirement for secure and economic operations of power systems and for efficient system planning. The desire to improve the speed of transient stability simulation is the main motivation of the work. The objective of the thesis is then described. Finally, the organization of the thesis is given.

■ **Chapter 2: The Transient Stability Problem and its Serial Algorithms** - The purpose of this chapter is to provide the background for parallel, distributed transient stability algorithms given in chapter 3. The transient stability problem is formulated. Several solution methods are described. The advantages and disadvantages of the various algorithms are discussed based on several simulations. The focus is on the Trapezoidal integration method using Newton type iteration. This chapter also introduces a program called DCPS that has been developed for this research.

■ **Chapter 3: Methods in Parallel Transient Stability Calculations** - This chapter begins with a general discussion on parallel processing and encompasses both hardware and software. This chapter also contains the survey of the various parallel transient stability algorithms. In each algorithm, parallelisms that have been exploited are identified. Also,

multiprocessing on a uniprocessor system is introduced in this chapter together with some simulation results. At the end of the chapter, the approach used to exploit parallelism is described.

■ **Chapter 4: Slow Coherency Based Network Partitioning** - In this chapter, the technique used for network partitioning is described. Several methods are discussed. The slow coherency based method that uses an augmented eigenbasis matrix is the focus of this chapter. The derivation of the method is given. Its algorithm and implementation are described. Several results from small to large systems are presented and discussed.

■ **Chapter 5: Using Cluster of Workstations for Parallel, Distributed Computing** - In this chapter, the RPC technique for implementing parallel, distributed programs on a cluster of workstations is introduced. Algorithms for solving linear systems based on block methods are developed. These algorithms are implemented on a cluster of workstations and the results are analyzed.

■ **Chapter 6: Communication Aspects of the RPC Technique for Parallel Distributed Computation** - In this chapter, the communication aspects of the RPC technique is analyzed vis-a-vis scheduling techniques. In this chapter, the parallelization process is discussed and applied.

■ **Chapter 7: Parallel Distributed Transient Stability Algorithms and implementations** - In this chapter, parallel distributed transient stability algorithms are developed through a parallelization process. The corresponding shared memory implementation and simulations are given. The parallel algorithms are also implemented on a cluster of workstations using the RPC technique. The results of simulations and the speedup are analyzed.

■ **Chapter 8: Conclusions** - The contributions of thesis are summarized in this chapter. Several conclusions are presented. Further work on this subject is also proposed.

CHAPTER 2
THE TRANSIENT STABILITY PROBLEM AND IT'S SERIAL ALGORITHMS

2.1 INTRODUCTION

The objective of performing transient stability simulation is to determine the trajectory of power system variables following a disturbance so that stability can be assessed. System variables require time to respond to any change in operating conditions. To determine this response, sets of differential equations can be formulated and solved. The larger the power system, the more will be the number of these equations. However, it is impractical and unnecessary to include all dynamic variables in the analysis and simplified system models are used. The a.c. network responds rapidly to any change in load or network topology. The time constants associated with the network variables are extremely small and can be assumed to be negligible in transient stability analysis without significant loss of accuracy (de Mello [1992]). In rotor angle stability, the concern is electromechanical oscillation, that is, the variation in power output of machines as their rotors oscillate. The time constants associated with the rotors are of the order of 1 to 10 seconds. Therefore, the differential equations that are relevant in this analysis are dominated by those having time constants of this order.

In this chapter, the power system models for transient stability analysis are described. The problem is formulated and solution methods are discussed. It is important to realize that the techniques of transient stability calculations are well established. The applications of sparsity techniques (Tinney, et. al. [1985]) and more stable integration methods (Dommel and Sato

[1972]), the incorporation of complex system device models, and the addition of enhanced user interfaces are some of the major advances that have been made during the last 15 - 20 years. Any attempt to apply new computational techniques to improve the solution speed of this problem such as parallel processing, must take into consideration those earlier advances because of their many advantages. This is the general approach of this thesis. Where applicable, the advances that have been made in serial algorithms are retained. Therefore, before proceeding to discuss parallelization of transient stability computation, a concise review of serial algorithms is required, which is the main objective of this chapter.

Also, in this chapter, the program that has been developed during this research work is introduced. The program has been written for easy extension and development thus making it suitable for future work. The program is referred to several times in later chapters of the thesis.

2.2 BASIC POWER SYSTEM MODEL

Figure 2.0, which is adapted from Stott [1979] outlines the structure of the power system model used in transient stability analysis. It shows a synchronous generator together with its controls and its relationship to the power system network. The diagram clearly separates those components of the power system that are described by differential equations and those by algebraic equations.

Synchronous machine models are based on Park's transformation (IEEE Committee Report [1986]). In this model, the machine electrical behaviour may be represented by equivalent circuits in the rotor direct and quadrature axes (see **Appendix C**). The stator transients are neglected and thus become algebraic in a common reference frame with the network. The machine rotor can be represented by a varying number of direct and quadrature axis equivalent circuits. These circuits are then translated into a set of differential equations. The rotor mechanical equations represent the accelerating torque acting on the machine. An important variable δ, the rotor angle between the

rotor frame (d, q frame) and the synchronously rotating frame (D, Q frame), is included in the rotor mechanical equations. The relationship between the rotor reference frame (d, q frame) and the synchronously rotating frame is illustrated in figure 2.1. Referring to the phasor diagram in figure 2.1, the machine q-axis is determined using:

$$E_q = V + (R_s + jX_q)\,I_T \qquad (2.1)$$

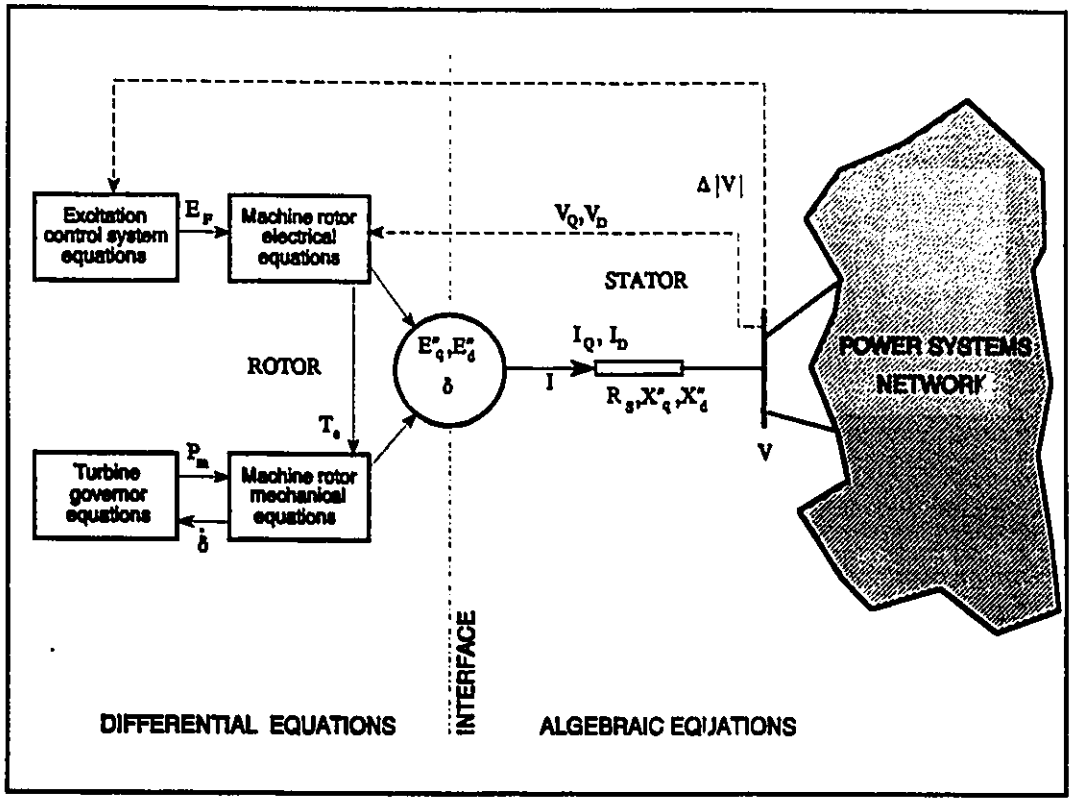

Figure 2.0: Transient model of synchronous generator connected to network

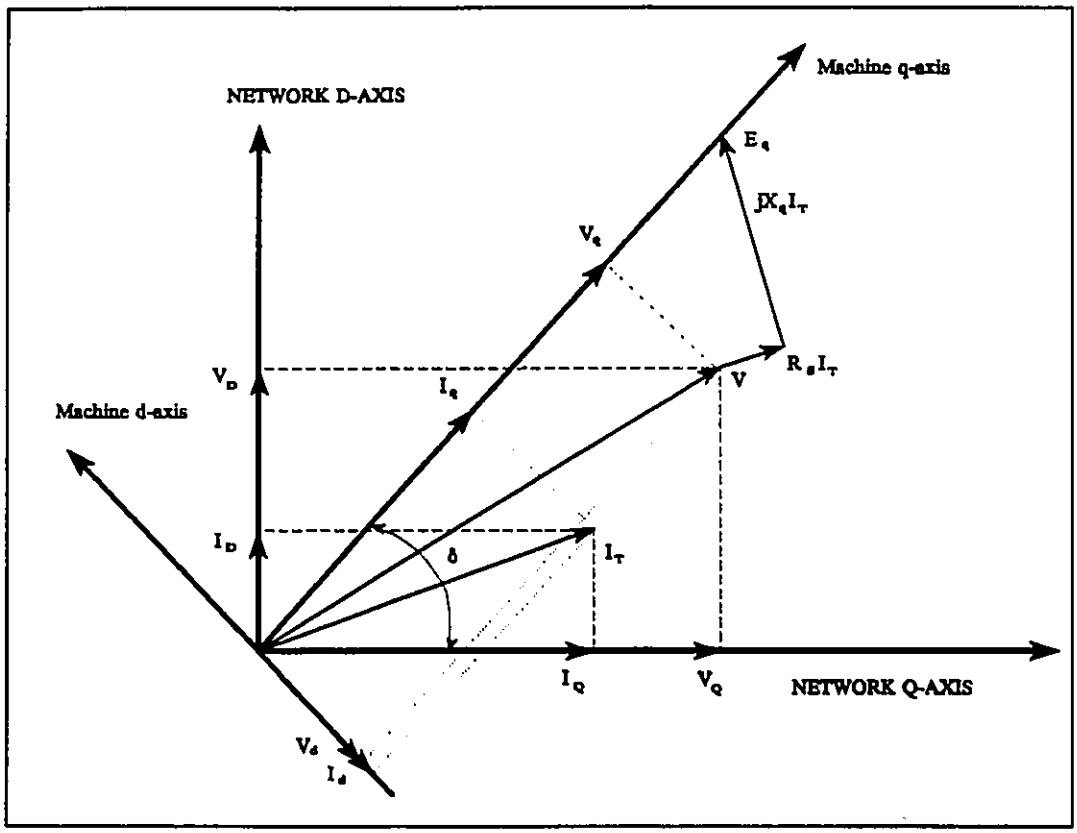

Figure 2.1: Relationship between rotor axis d, q frame and network D, Q frame

The terminal quantities - current and voltage - can now be referred to the machine q-axis as shown in figure 2.1. Terminal voltage referred to the network D-Q frame is transformed to the machine d-q frame by:

$$\begin{bmatrix} V_q \\ V_d \end{bmatrix} = T_{dq} \begin{bmatrix} V_Q \\ V_D \end{bmatrix} \qquad (2.2)$$

where:

$$T_{dq} = \begin{bmatrix} \cos\delta & \sin\delta \\ -\sin\delta & \cos\delta \end{bmatrix} \qquad (2.3)$$

In the machine **d-q** frame, the stator voltage drop is given by:

$$\begin{bmatrix} E''_q - V_q \\ E''_d - V_d \end{bmatrix} = Z_{dq} \begin{bmatrix} I_q \\ I_d \end{bmatrix} \tag{2.4}$$

where:

$$Z_{dq} = \begin{bmatrix} R_S & -X''_d \\ X''_q & R_S \end{bmatrix} \tag{2.5}$$

Thus the current in the **d-q** frame can now be determined from the terminal quantities and the voltage behind transient reactance using:

$$\begin{bmatrix} I_q \\ I_d \end{bmatrix} = Z_{dq}^{-1} \begin{bmatrix} E''_q \\ E''_d \end{bmatrix} - Z_{dq}^{-1} T_{dq}^{-1} \begin{bmatrix} V_Q \\ V_D \end{bmatrix} \tag{2.6}$$

The machine may also be equipped with excitation and turbine governor controls. Furthermore, the excitation systems may be provided with external supplementary control devices. For all these controls, sets of first-order differential equations are formulated. The network is described by algebraic nodal admittance equations. Mathematically, the power systems can be described by a set of non-linear differential-algebraic equations (DAE):

$$YV = I_B(x,V) \tag{2.8}$$

$$\dot{x} = f(x,V) \tag{2.7}$$

where **x** is a vector of state variables, **V** is a vector of bus voltages, $\mathbf{I_B}$ is a vector of current injections and **Y** is the complex admittance matrix.

2.3 SOLUTION OF DIFFERENTIAL-ALGEBRAIC EQUATIONS

The most convenient starting point to discuss this topic is to divide the solution methods into either explicit or implicit numerical integration techniques. Under the explicit technique both Runge-Kutta and Modified

Euler methods have been used extensively. Under the implicit technique, the Trapezoidal method has been the most popular choice in comparison to the Backward Euler Method (Arrillaga, et. al. [1983]).

In this thesis, the Modified Euler method is chosen to represent the explicit technique. A good description of the application of the Modified Euler method to transient stability calculations can be found in Stagg and El-Abiad [1968]. Although the method can become unstable with large time steps, because of its simplicity, it is still being used in many production grade programs (de Mello [1992]).

The Trapezoidal method, which was first used for transient stability by Dommel and Sato [1972], is inherently stable which allows the use of larger time steps. The use of the implicit method also allows the differential equations in (2.7) to be converted to algebraic form which may be combined with the original algebraic equations (2.8), thus allowing the resulting non-linear equations to be solved as a whole. In this thesis, the Trapezoidal method is chosen to represent the implicit technique.

2.3.1 Modified Euler Method

When the Modified Euler Integration method is applied to equation (2.7), the estimate for the state vector is first obtained using:

$$x^{(n)} = x^{(o)} + h\, f(x^{(o)}, V^{(o)}) \tag{2.9}$$

where (n) denotes the estimate, (o) denotes the previous value and h is the integration step. Since V also depends on I_B, which in turn is a function of the state vector, its new estimate is obtained by solving:

$$YV^{(n)} = I_B(x^{(n)}, V^{(o)}) \tag{2.10}$$

The final values of x and V are obtained by solving:

$$x^{(f)} = x^{(o)} + \frac{1}{2}h\,\left(f(x^{(o)}, V^{(o)}) + f(x^{(n)}, V^{(n)})\right) \tag{2.11}$$

$$YV^{(f)} = I_B(x^{(f)}, V^{(n)}) \quad . \qquad (2.12)$$

where (f) denotes final value. The differential and algebraic equations are solved separately. The network equation is usually solved by backward/forward substitution on the LU factor of the Y. The LU factor is only recalculated when there is change in the network topology. The Modified Euler method is given in algorithm 2.1.

Algorithm 2.1: Modified Euler Method

---

for (t = t1; t <= t2; t += h) {

i. Calculate initial estimate of states:

$$x(t)^{(n)} = x(t-h)^{(o)} + f(x(t-h)^{(o)}\ ,\ V(t-h)^{(o)})$$

ii. Calculate bus current:

$$I_B(t)^{(n)} = I_B(x(t)^{(n)}\ ,\ V(t-h)^{(o)})$$

iii. Solve for network voltage:

$$YV(t)^{(n)} = I_B(t)^{(n)}$$

iv. Transform variables from network to rotor frame

v. Calculate final estimate of states:

$$x(t)^{(f)} = x(t-h)^{(o)} + \frac{1}{2}h\,(f(x(t-h)^{(o)}\ ,\ V(t-h)^{(o)}) + f(x(t)^{(n)}\ ,\ V(t)^{(n)}))$$

vi Iterative network solution:

$$YV(t)^{(f)} = I_B(t)^{(f)}$$

(using voltage as convergence criteria)

vii.Transform variables from network to rotor frame

}

In algorithm 2.1, steps **i** and **v** are evaluated for each machine and they are independent of each other. Steps **ii, iv** and **vii** are also evaluated for each bus separately. Steps **iii** and **vi** are computed from the overall admittance matrix of the system. This observation is important when implementing the parallel, distributed version of this algorithm.

2.3.2 Trapezoidal Method

Applying the Trapezoidal integration method to equation (2.7), an algebraic expression is obtained:

$$x_n = x_{n-1} + \frac{1}{2}h[f(x_{n-1},V_{n-1}) + f(x_n,V_n)] \qquad (2.13)$$

The network equation (2.8) becomes:

$$YV_n = I_B(x_n,V_n) \qquad (2.14)$$

where n denotes values at the current time step. Due to the implicit nature of the above equations, an iterative solution technique is required. Basically there are two iterative schemes that can be applied; (i) fixed point iteration, and (ii) Newton type iterative methods.

2.3.2.1 Fixed Point Iterative Scheme

In the fixed point iterative scheme, equations (2.13) and (2.14) are solved directly until convergent. This technique is sometime called the predictor - corrector procedure (Elden, et. al. [1990]). The algorithm is as follows (Algorithm 2.2):

Algorithm 2.2: Trapezoidal Method with Fixed Point Iteration

---

for (t = t1; t <= t2; t += h) {

i. **Calculate states:** $x(t) = x(t-h) + \frac{1}{2}h[f(x(t-h),V(t-h)) + f(x(t),V(t))]$

ii. **Calculate bus current:** $I_B(t) = I_B(x(t),V(t))$

iii. **Solve for network voltage:** $YV(t) = I_B(t)$

iv. **Transform variables from network to rotor frame**

v. **Check convergence of x and V. If not converged goto i, if converged proceed with next time step.**

}

Step i is evaluated for each machine separately. In step ii, current injection is calculated for each bus. Step iii involves the overall network, therefore the equation has to be solved simultaneously. In calculating the states in step i, it is more efficient to divide the equations into two parts as proposed by Arrillaga, et. al. [1983]. The non-integrable part is evaluated only once because it remains constant during the each time step - this variation is sometimes called the Interlaced Alternating Implicit method (Decker, et. al. [1992]). The integrable part is required to be updated during each iteration.

2.3.2.2 Newton Iterative Schemes

Newton's method is a well known iterative technique for solving non-linear equations. The method has quadratic and reliable convergence provided that the initial iterate is sufficiently close to the solution. This method was first used by Gear [1971]. Equations (2.7) and (2.8) can be rewritten as:

$$F(x_n,V_n) = x_n - x_{n-1} - \frac{1}{2}h[f(x_{n-1},V_{n-1}) + f(x_n,V_n)] \qquad (2.15)$$

$$G(x_n,V_n) = YV_n - I_B(x_n,V_n) \qquad (2.16)$$

Applying the Newton-Raphson method to equations (2.15) and (2.16) and dropping all subscripts, the following linear equation is obtained:

$$\begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta V \end{bmatrix} = \begin{bmatrix} -F \\ -G \end{bmatrix} \qquad (2.17)$$

where $J_1 = \frac{\partial F}{\partial x}$, $J_2 = \frac{\partial F}{\partial V}$, $J_3 = \frac{\partial G}{\partial x}$, $J_4 = \frac{\partial G}{\partial V}$ are the elements of the Jacobian matrix. To establish reasonably good starting values for x and V, extrapolation from the previous time step values is performed (Stott [1979]). States are best extrapolated using the Euler method (EPRI EL-2000-CCM-

Project 1208 [1987] and Elden, et. al. [1990]):

$$x_n = x_{n-1} + h\, f(x_{n-1}, V_{n-1}) \tag{2.18}$$

and voltages are extrapolated using the geometric formula:

$$V_n = \frac{V_{n-1} V_{n-1}}{V_{n-2}} \tag{2.19}$$

Here, two alternative solution methods are discussed; namely (i) Full Newton Iterative Scheme and (ii) Decoupled Newton Iterative Scheme (Stott [1979]). Strict applications of Newton's Method requires the LU decomposition of the Jacobian matrix during each iteration. However, this is very expensive. Therefore in practice, the quadratic convergence is partially sacrificed by using the same LU factors over several iterations. A criteria, maximum number of iterations, is set before the LU factor is re-evaluated. This method is generally referred to as the Dishonest Newton's Method. The structure of the Jacobian matrix is shown in figure 2.2.

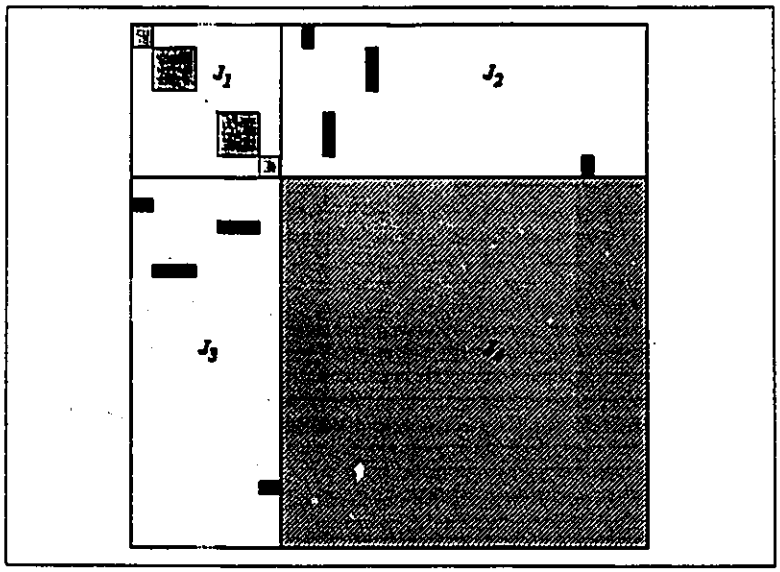

Figure 2.2: Structure of the Jacobian Matrix

Submatrix J_1 consists of independent blocks associated with each generator. These blocks can be formed and factorized separately. Submatrix J_2 follows the structure of J_1, where each block has the same number of rows as of the corresponding block in J_1 and each block has two columns

corresponding to the real and imaginary parts of the machine terminal voltage. Submatrix J_3 is structurally the transpose of J_2 but each block has only two elements with non-zero values. Submatrix J_4 is derived from the network admittance matrix but in expanded form. Applying Gaussian elimination to equation (2.11), we obtain:

$$\begin{bmatrix} J_1 & J_2 \\ 0 & \hat{J}_4 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta V \end{bmatrix} = \begin{bmatrix} -F \\ \hat{G} \end{bmatrix} \tag{2.20}$$

where:

$$\hat{J}_4 = J_4 - J_1^{-1} J_3 J_2 \tag{2.21}$$

$$\hat{G} = -G - J_1^{-1} J_3 F \tag{2.22}$$

The Full Dishonest Newton's method algorithm is as follows:

Algorithm 2.3: Full Dishonest Newton's Method

---

for (t = t1; t <= t2; t += h) {
iteration = 0
if (old_iteration > max_allowable_iteration)

i. Update Jacobian matrix: $\hat{J}_4 = J_4 - J_1^{-1} J_3 J_2$ **and factorize**

end if
ii. iteration = iteration + 1

iii. Compute vector: $\hat{G} = -G - J_1^{-1} J_3 F$

iv. Solve for network using: $\hat{J}_4 \Delta V = \hat{G}$

v. Computes states using: $J_1 \Delta x = -F - J_2 \Delta V$

vi. Check convergence of x and V. If not converged goto ii, if converged goto vii
vii. old_iteration = iteration
}

In the decoupled Newton's method submatrices J_2 and J_3 are neglected from equation (2.21). The Decoupled Newton's method is given in algorithm

2.4.

Algorithm 2.4: Decoupled Dishonest Newton's Method

---

for (t = t1; t <= t2; t += h) {

iteration = 0

if (old_iteration > max_allowable_iteration)

i. Update Jacobian matrix: $\hat{J}_4 = J_4$ **and factorize**

end

ii. **iteration = iteration + 1**

iii. **Compute vector:** $\hat{G} = -G$

iv. **Solve for network using:** $\hat{J}_4 \Delta V = \hat{G}$

v. **Compute states using:** $J_1 \Delta x = -F$

vi. **Check convergence of x and V. If not converged goto ii, if converged goto vii**

vii. **old_iteration = iteration**

}

2.4 INTRODUCTION TO DCPS PROGRAM

During the course of this research work, a general purpose power system simulation program has been developed to implement the algorithms described in this thesis. The program is given an acronym **DCPS** for "**D**ynamic **C**omputation - for **P**ower **S**ystems". There are three program suites in DCPS, namely, (i) **LF** for load flow, (ii) **CG** for coherency grouping and (iii) **TS** for transient stability. Each of the above three programs can be invoked separately but all have the same mode of operations except that some functions may only be available to a particular program suite.

The program is created for the purpose of research. Therefore, sophisticated user-interface facilities are absent. There are two major capabilities/features that are incorporated - (i) the ability to handle systems up to 1000 buses and 250 generators, and (ii) the capability for easy extension. For the program to accommodate the first feature and still be fast, advanced

programming techniques including sparsity[1] were used. When handling large systems, simple facilities for managing data are also required and included in the program. The second feature is important because expandability is the main objective of creating the program in the first place. A modular approach has been used. All routines in DCPS were developed in-house.

The load flow program, LF is important since it is used as a pre-requisite to transient stability simulation. The CG program is used for coherency grouping which will be discussed in more detail in chapter 4. The TS program is the core of this research work. Many algorithms developed in this thesis are implemented in the TS program which will be discussed further in following chapters. In this section the general characteristics of the program are described.

2.4.1 DCPS Environment

The philosophy used when writing the program was to allow for easy extension. Therefore, modularity is a key feature. As soon as the program is invoked, a cursor appears on the screen and awaits instruction from the user (see **Appendix A** for more detail on running the program) . In the DCPS environment, one can then write a *command word* and in some cases followed by several *command parameters* separated by commas for performing a particular task. The *Command word* normally has the same name as the actual C or FORTRAN function of the program. *Command parameters* are usually arguments to the function specified by the user. Furthermore, the series of *command words* and *parameters* may be read from a text file (in ASCII format) and executed automatically rather than being entered manually. Lines of commands as shown in text box 2.1 are used to perform load flow calculations.

---

[1] In implementing sparsity techniques, the choice of storage strategy is crucial. DCPS uses both collection-of-vectors and linklist methods for storage. As an example, the admittance matrix uses linklist storage and the LU factors use collection of vectors. After choosing the storage technique all codes are then written so that the elements are accessed efficiently.

```
% dnlf.com: Run-file for decoupled Newton Load Flow
chco,0.03            % load flow tolerance 3 MVA total
                     % mismatch
redn,etp97.sol,ieee % read network data, format - ieee
ordl;                % ordering buses for minimum fill-ins
ybus;                % admittance matrix, rectangular form
ybpo;                % admittance matrix to polar form
vbpo;                % bus voltages to polar form
bpop,0               % calculate bus power
dnlf,20            % perform load flow maximum iteration 20
```

Text box 2.1: Sample command file - load flow calculations

Details of all command words and parameters are given in Appendix A of this thesis. If the above text is saved to a text file say *dnlf.com.* then while in the DCPS environment the user can execute the whole series commands by typing *runf,dnlf.com* and then pressing the [ENTER] key.

2.4.2 Adding a DCPS Function

Each of the DCPS programs (LF, CG and TS) has a driver procedure (main()) which is used to call one or more functions. Adding a function requires two major steps: (i) placing a module in the driver procedure so that the function can be called by user command, and (ii) development of the actual procedure. In some cases, if new global variables are to be introduced, the header files may need to be changed. The program can then be recompiled using the given *makefile*. Appendix B illustrates a technique of adding a function to TS program.

2.5 LOAD FLOW CALCULATIONS USING DCPS

Load flow calculation is an important prerequisite to transient stability simulation. Only after the system load flow is properly solved would one attempt transient calculations. An incorrect load flow solution may result in instability even without the application of any disturbance. In DCPS, three load flow calculation method are implemented - (i) Gauss-Seidel, (ii) Decoupled Newton, and (iii) Fast Decoupled methods (Stott [1974]). From experience in

using the DCPS, the Decoupled Newton method has been found to perform satisfactorily on all ranges of system size.

In the following example, a load flow case for the 840-bus system given in IEEE common format (IEEE Committee Report [1973]) is solved. When the command file similar to the one found in text box 2.1 is executed, the resulting display is as in text box 2.2. The load flow converged after 14 iterations. One main drawback of the IEEE common format is the limited decimal places allowed for numbers. As a result of this limitation, a solved load flow when rewritten to an IEEE common format will have values truncated and therefore needs to be solved again. To overcome this drawback, a different format is used in DCPS to maintain accuracy of the numbers thus avoiding repeated load flow. The DCPS formats for network and generator data are explained in Appendix C.

In text box 2.2, the network data is first read using `redn` command. Note that the network data is specified in IEEE common format. The network buses are ordered using `ordl` and the admittance matrix in rectangular form is constructed using `ybus` function. The admittance matrix is converted to polar form using `ybpo` command. Similarly voltages are converted to polar form by `vbpo` command. The Decoupled Newton load flow is then invoked by `dnlf` command that carries two parameters - (i) maximum iteration, 20 and (ii) interval to skip the formation of Decoupled Jacobian matrices and its factorisation, 1. The total mismatch obtained at the end of 14th iteration is comparable to the one obtained using production grade software - using PTI's PSSE load flow at Ontario Hydro.

```
lf>runf,dnlf.com
lf>chco,0.03,,,,,,,
lf>redn,etp97.iee,ieee
   840 buses & 1502 branches read
 no. of machine       =  219
 no. of load buses    =  621
 no. of lines         = 1028
 no. of transformers =  474
lf>ord1;
lf>ybus;
 ybus formed: nb = 840 ne = 2824
lf>ybpo;
lf>vbpo;
lf>bpop,0
lf>dnlf,20,1
Iteration/total mismatch:  1 / 2721.2004
Iteration/total mismatch:  2 /  586.3867
Iteration/total mismatch:  3 /  651.9052
Iteration/total mismatch:  4 /  582.6203
Iteration/total mismatch:  5 /  419.4422
Iteration/total mismatch:  6 /  262.8091
Iteration/total mismatch:  7 /  154.0052
Iteration/total mismatch:  8 /   86.7367
Iteration/total mismatch:  9 /   47.4132
Iteration/total mismatch: 10 /   26.4771
Iteration/total mismatch: 11 /   14.1975
Iteration/total mismatch: 12 /    8.2712
Iteration/total mismatch: 13 /    5.0774
Iteration/total mismatch: 14 /    2.6787
 Load flow converged
lf>
```

Text box 2.2: Load flow calculations

2.6 TRANSIENT STABILITY SIMULATIONS USING DCPS

The most convenient way to use DCPS is to write all commands to an ASCII file and execute them automatically using the runf command. In the following example, the nine-bus power system of Anderson and Fouad [1977] as shown in figure 2.3 is used. The objective of this simulation is to compare the effects of using large time step on both the Modified Euler method and the Trapezoidal method. The field voltage, E_{fd} of the exciters of the generators are captured for plotting. The command file used is shown in text box 2.3.

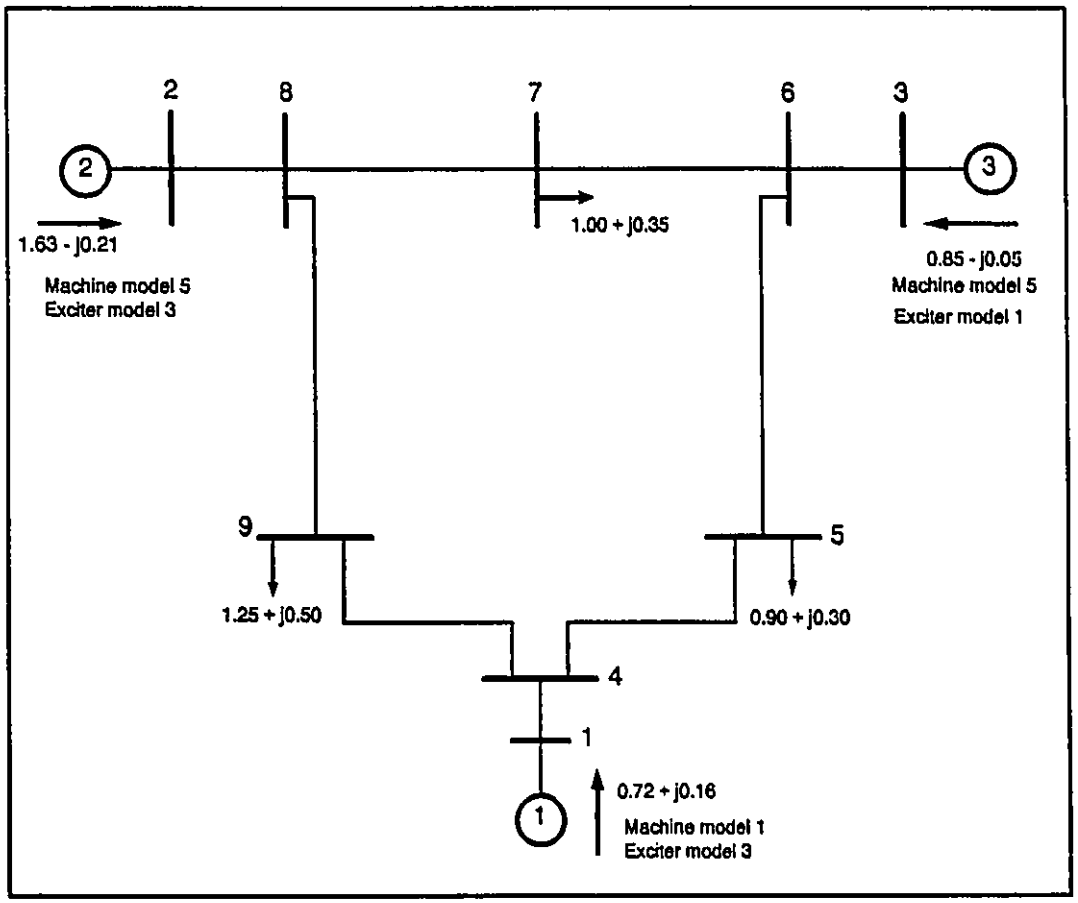

Figure 2.3: Nine-bus power system

```
chco,1.2,1.5,1.5,0.001,0.0005,0.0005,0.01,0.01
redn,nine.sol
redm,nine.mac
orde;
ybus;
bpow;
iniv;
inie;
moym;
cinj;
ginj;
fac2,syb
caps,2,1,3
caps,2,2,3
caps,2,3,3
tsol,2,0.0,0.06,0.01,1
asht,syb,7,0.0,1000.0
fac2,syb
nsol;
dqtr;
tsol,2,0.07,0.17,0.01,1
rsht,syb,7,0.0
fac2,syb
nsol;
dqtr;
tsol,2,0.18,2.1,0.01,1
```

Text box 2.3: Sample DCPS Command File for Transient Simulation

In the command file shown text box 2.3, the network data `nine.sol` and the machine data `nine.mac` are first read. The network buses are ordered using function `orde` which is based on Tinney's [1967] scheme no. 1. The initial values of all machines and their exciters are calculated using the `iniv` and `inie` functions, which can only be executed after the admittance matrix is formed by function `ybus` and the bus powers are calculated by function `bpow`. Function `moym` is used to append the equivalent source admittance of all generators into the admittance matrix. Function `cinj` appends the admittance matrix with the initial equivalent load admittance. To account for non-linear load characteristics in subsequent calculations, current injection is introduced at each load bus. In DCPS, the dependency of load on voltage and frequency is calculated using (Arrillaga, et. al. [1983]):

$$P_i = K_{pi}(V_i)^{pv}(freq)^{pf} \tag{2.23}$$

$$Q_i = K_{qi}(V_i)^{qv}(freq)^{qf} \tag{2.24}$$

Constants K_p and K_q which depend on the nominal value of P_i (active power p. u.) and Q_i (reactive power p. u.) of each individual bus are calculated in function `cinj`. The current injection of all dynamic devices and those due to non-linear load characteristics are evaluated using function `ginj`. The admittance matrix (indicated by parameter `syb`) is factorized by function `fac2`. For output purposes, function `caps` is used to capture the desired states or controls. In the command file of text box 2.3, the field voltage of the three generators are to be captured.

Transient solution is performed using function `tsol`. The first parameter of this function is the solution method, where in the above example, a value of 2 corresponds to the Trapezoidal method with fixed point iteration (Algorithm 2.2). As indicated in the command file, function `tsol` is called in three stages. The first call performs the computation up to 0.06 seconds without any disturbance. Following that, function `asht` is used to affect a short circuit at bus no. 7. The change in the admittance matrix necessitates its refactorization and iterative solution of the network using function `nsol`. Function `dqtr` performs transformation of the voltages and current from the network frame (D-Q) to the rotor frame (d-q). The simulation continues with the removal of the short circuit by function `rsht` and then the final call to function `tsol`.

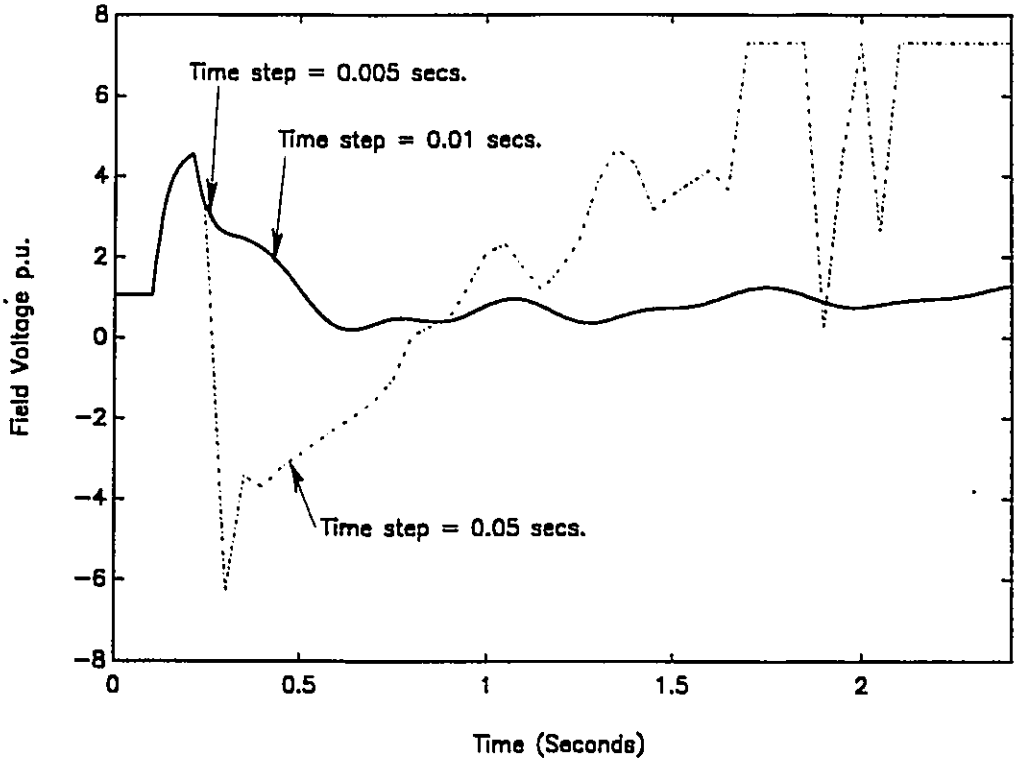

Figure 2.4: Modified Euler Method with time steps of 0.005, 0.01 and 0.05 seconds

Figure 2.4 shows the response of the generator field voltage of a generator using the modified Euler method. The response is typical of a high initial response excitation system. The response with time step of 0.01 seconds is as accurate as the response with a smaller time step of 0.005 seconds. But as the time step is increased to 0.05 seconds, numerical inaccuracy occurred. In using explicit integration methods, numerical inaccuracy - that often lead to numerical instability - can be avoided by using a small time step, usually guided by the smallest subtransient rotor time constant. The main advantage of the modified Euler method is its simplicity, making it easy to code for high efficiency. It is for this reason that the integration technique is still widely used, even in some of the most popular production grade programs (de Mello [1992]).

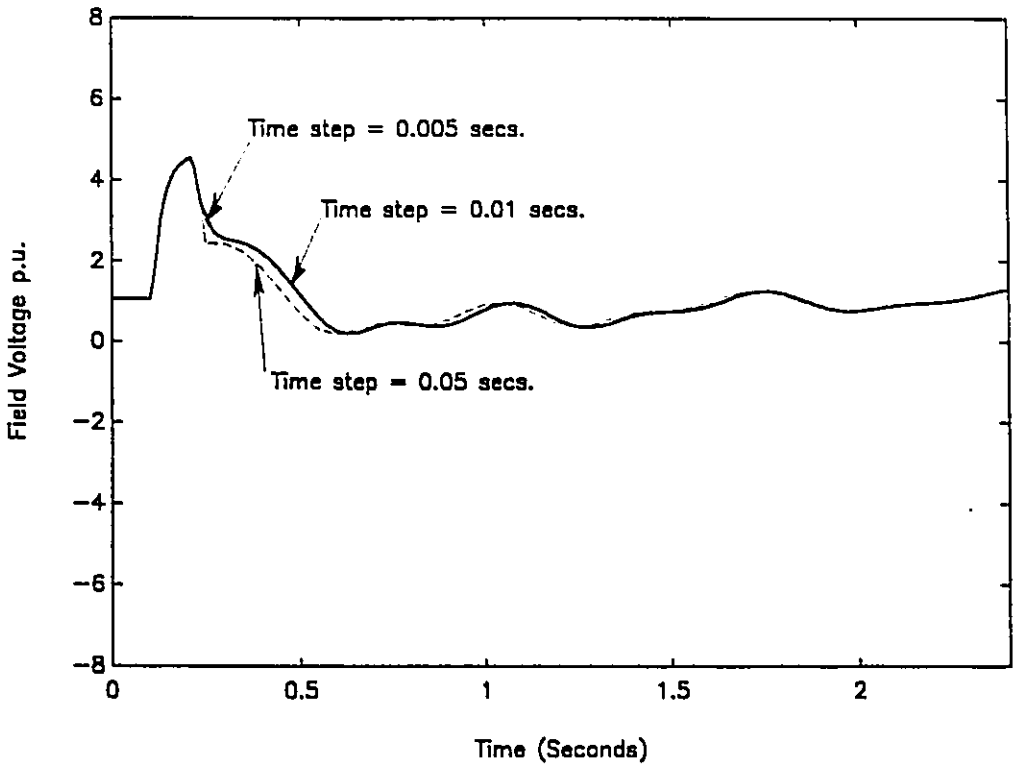

Figure 2.5: Trapezoidal Method with time steps of 0.005, 0.01 and 0.05 seconds

Figure 2.5 shows the same response as in figure 2.4 but using the Trapezoidal method. In contrast to the modified Euler method, the response using the Trapezoidal method with a time step of 0.05 seconds follows the response of those with smaller time steps - 0.005 and 0.01 seconds. The implicit integration method eliminates the small time constant numerical instability problem - the main advantage of the implicit method. This property can be used to speed up the simulation by using larger time step. However, there is a limit to the size of the time step beyond which the fidelity of high frequency response is lost. Also, with a large time step the number of iterations increases and this may offset the advantage of a large time step. This aspect will be discussed further in section 2.7.

2.7 COMPONENT MODELS IN DCPS

In DCPS a generator can be modeled as a classical or as detailed (1.0, 1.1 or 2.2 models - IEEE-PES Committee Report [1990], also see **Appendix C** for the q- and d-axis circuit diagrams. Using the DCPS format, these models are given identifiers 0, 1, 2, 5 respectively. Three exciter models are provided, those are AC4, IEEE Type 1 and Controlled-rectifier type (IEEE Committee Report [1981]). One power system stabilizer model is available - IEEE type 1. These control models are sufficient to make the study cases in this thesis similar to those encountered in practice. The block diagrams of the exciters and PSS are also given in **Appendix C**.

2.8 COMPARISON OF SIMULATION SPEEDS

In this section comparison is made on the simulation speeds of the various transient stability algorithms presented in the previous sections. In particular, the execution speeds of the implementations of algorithms 2.1, 2.2, 2.3 and 2.4 are compared using the same contingency on the IEEE 118-bus test system (see figure 4.4). The contingency and simulation period are the same as those reported in Chai, et. al. [1991]. All generators are modeled in detail and provided with exciters. The command file used is shown in text box 2.4.

```
%=====================================
% TS118b.com: Transient Stability
% Runfile
%=====================================
chco,1.2,1.5,1.5,0.01,0.001,0.001,0.0
redn,s118b.sol
redm,s118bf.mac
orde;
ybus;
bpow;
iniv;
inie;
moym;
cinj;
ginj;
fac2,syb
caps,2,4,4
caps,2,26,4
caps,2,69,4
caps,2,80,4
caps,2,113,4
tsol,4,0.0,.04,.01,0
asht,syb,69,0.0,1000.0
fac2,syb
nsol;
dqtr;
tsol,4,0.05,0.15,.01,1
rsht,syb,69,0.0
fac2,syb
nsol;
dqtr;
tsol,4,0.16,1.12,.01,1
```

Text box 2.4: DCPS command for Simulation speed test

The second and third parameters of function `chco` are voltage exponents for the non-linear active and reactive loads respectively. The fourth and fifth parameters of function `chco` are the convergence tolerance used for states and voltage respectively. The state tolerance is specified to be 1% and voltage tolerance is 10^{-3}. A short circuit is applied at bus 69 for a duration of 100 ms as indicated by functions `asht` and `rsht` in text box 2.4.

Figure 2.6 shows the rotor angle response of machines 26 and 69 with different simulation methods - algorithms 2.1, 2.2, 2.3 and 2.4. As expected, all algorithms produced the same response.

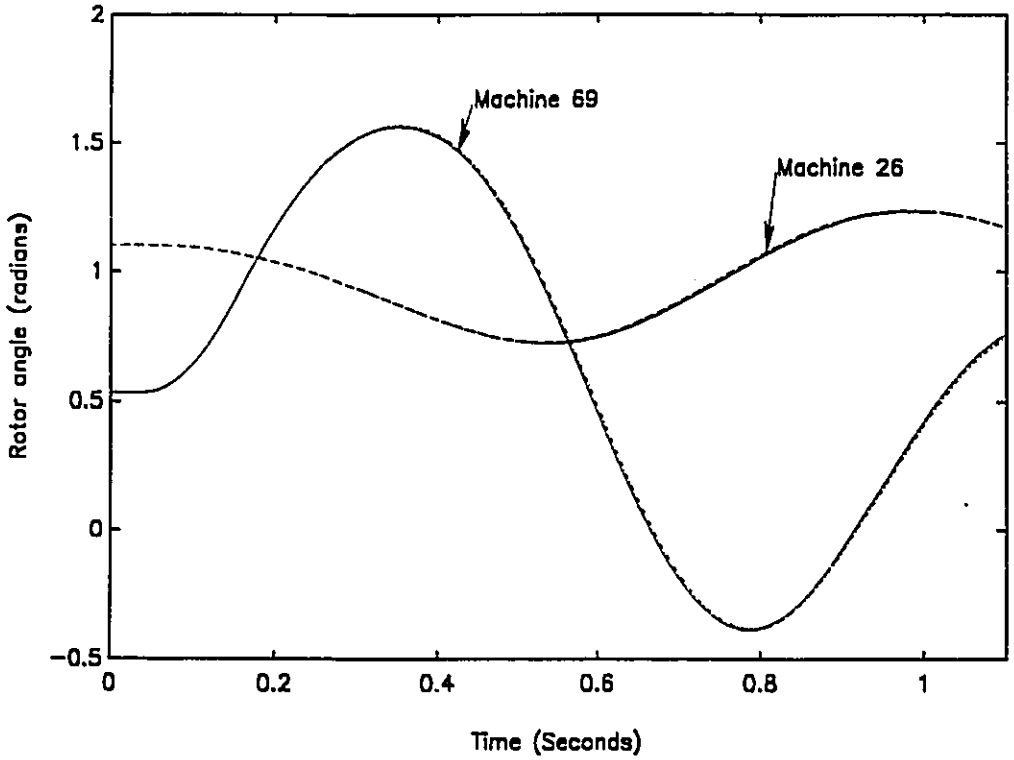

Figure 2.6: Response for machine 26 and 69 using algorithms 2.1, 2.2, 2.3 & 2.4.

Figure 2.7 shows exciter response for machine 69 and 26. A portion of the curves from 0.8 to 1.1 seconds is shown in figure 2.8. As the time increases the response due to the modified Euler method deviates from that of the Trapezoidal based methods. This is due the propagation of the rounding and truncation error in Euler method (Golub and Ortega [1992]).

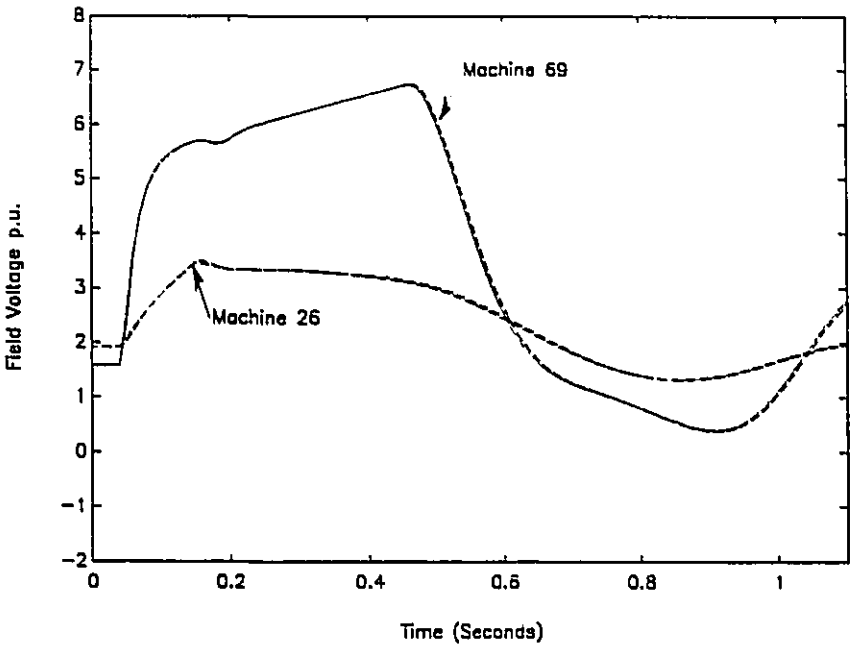

Figure 2.7: Field voltage response for machine 26 and 69 using algorithms 2.1, 2.2, 2.3 & 2.4.

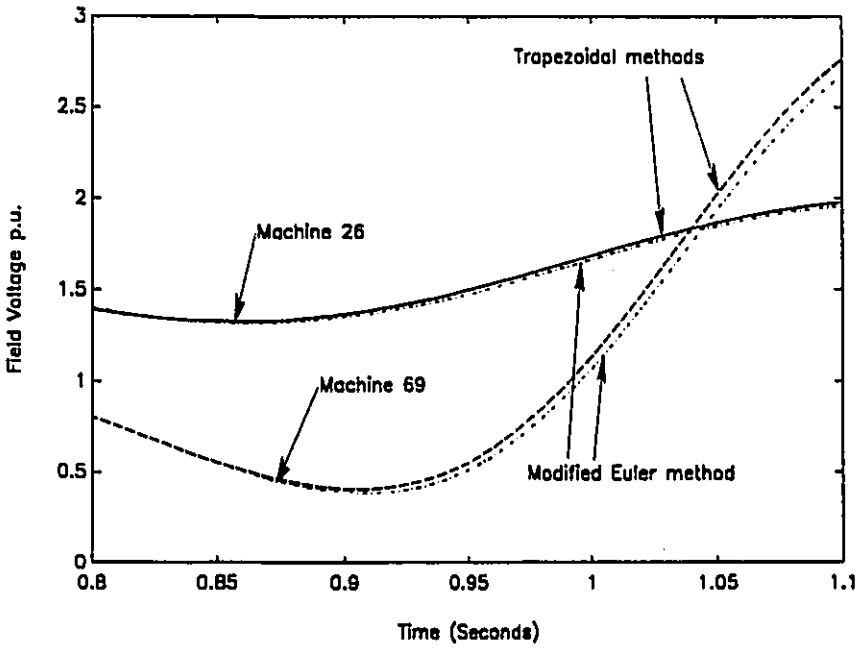

Figure 2.8: Field voltage response for machine 26 and 69 using algorithms 2.1, 2.2, 2.3 & 2.4. (0.8 - 1.1 sec)

Methods	Time step 0.01 sec.	Time step 0.02 sec.	Time step 0.04 sec.
Modified Euler	6.77	unstable	unstable
Trapezoidal	8.55	not converged	not converged
Decoupled Newton	8.75	5.22	3.35
Full Newton Method	8.43	4.98	3.25

Note: All times in seconds

Table 2.1: Speed of various transient stability algorithms with different time steps

The execution times of the four algorithms with various time steps are given in table 2.1. The modified Euler method executed this particular simulation in the fastest time when a time step of 0.01 seconds is used. But the modified Euler method becomes unstable with time steps of 0.02 and 0.04 seconds. The Trapezoidal method with fixed point iteration encountered difficulty in converging when time steps of 0.02 and 0.04 seconds are used. Both the Newton based methods are superior to the first two methods. They remain stable and accurate with time steps of 0.02 and 0.04 seconds. In particular the Full Newton method - with dishonest iteration - is the fastest method when using time steps of 0.02 and 0.04 seconds. In other simulations in this thesis, a time step of 0.01 seconds is used so that all four methods can be compared. It must be noted that for all integration methods, the response using a smaller time step is more accurate than the one with a larger time step. For example, in the implicit methods, the effect of the local truncation error will become more prominent when a large time step is used. However, in transient stability simulations, we are more interested in the outcome of the overall response - stable or unstable - and therefore, as long as reasonable

accuracy is maintained, a larger time step is preferred so that the computation will be faster. The simulations described above were performed on a Sun Sparc Workstation - 16 MB of RAM, 64 kB of cache and CPU clock rate of 25 MHz.

2.9 CONCLUSIONS

In this chapter, the transient stability problem and its solution algorithms have been described. A simulation program, DCPS having three major applications - load flow, coherency grouping and transient stability - has been introduced. Using the DCPS the advantages and disadvantages of implicit and explicit integration methods were evaluated. Using a particular contingency on a test system, the speed of the various methods for transient simulation were compared. The simulations indicated that the trapezoidal method using the Full Newton iterative procedure is the fastest technique. The integration methods that have been used in thesis are of first-order - the modified Euler method, and second-order - the trapezoidal method. In transient stability simulations, higher order explicit integration methods, such as the fourth-order Runge-Kutta, have been used (EPRI EL-2000-CCM-Project 1208[1987]). Although the trapezoidal method is less accurate than the fourth-order Runge-Kutta method, due to its A-stable property, large time steps can be used, so that a faster solution can be achieved.

In the DCPS one can easily specify the desired time step for a given interval. Using manual control of the time steps requires some judgement on the part of the engineer - accurary and rate of convergence have to be considered. For example, a small time step - say 0.01 seconds - could be used immediately after switching. Subsequently, as the response becomes less rapid, the time step could be increased. However, a better time step control is to use the truncation error as the criterion (Stott [1979]).

CHAPTER 3
METHODS IN PARALLEL TRANSIENT STABILITY CALCULATIONS

3.1 INTRODUCTION

In chapter 2, traditional transient stability algorithms and their implementations are discussed. As mentioned in the last chapter, the approach of this thesis is to retain, as far as possible, all algorithmic advances that have been made over the last 10-15 years. Some of these advances have been described in chapter 2 and implemented in the DCPS program. This chapter has two main objectives - firstly to review the state-of-the-art in parallel transient stability calculations and secondly to outline the approach of this thesis in exploiting parallelism in the transient stability problem.

The chapter will first discuss the general subject of parallel, distributed processing. Review of parallel transient stability calculations begins with the discussion on the approaches to exploit parallelism in the problem. Before closing the chapter, the approach used in this thesis to exploit parallelism in the problem will be described. Simulation results using a uniprocessor system will also be presented.

3.2 SERIAL AND PARALLEL PROCESSING

A serial computer with one central processing unit (CPU) has a single source of control that determines the next instruction to be executed. During the execution of each instruction, the data to be operated upon are fetched from a global memory one at a time. Apart from its limitation to single instruction execution at any time, the computation can be made more sluggish by slow memory access and slow input-output devices. In most processors

such bottlenecks are alleviated by the use of cache memory and pipelining. Cache memory operates at high speed which is usually much faster than the processor itself. The processor keeps the most recently used data and instructions in the cache. The information is fetched immediately from the cache by the processor in much faster time than it can be accessed from the main memory. In pipelining, certain parts of the CPU which are responsible for separate functions (fetching operands, arithmetic operation, outputting) can be instructed to operate simultaneously. Supercomputers are built making extensive use of pipelining and intelligent memory organizations. They can perform vector operations at speeds comparable to scalar operations. However, fundamental limitations, such as overheating with compact circuits impose limit on the achievable speed of serial computers.

Through the advances in hardware technology from the 50's, we have witnessed a twenty-fold speedup of serial computers for every decade as shown in figure 3.1 (Modi [1988]). However, to overcome three major obstacles to further improvement in performance - speed limit of serial computers, increasing computational demand and memory bottle-neck - parallel processing is the main alternative.

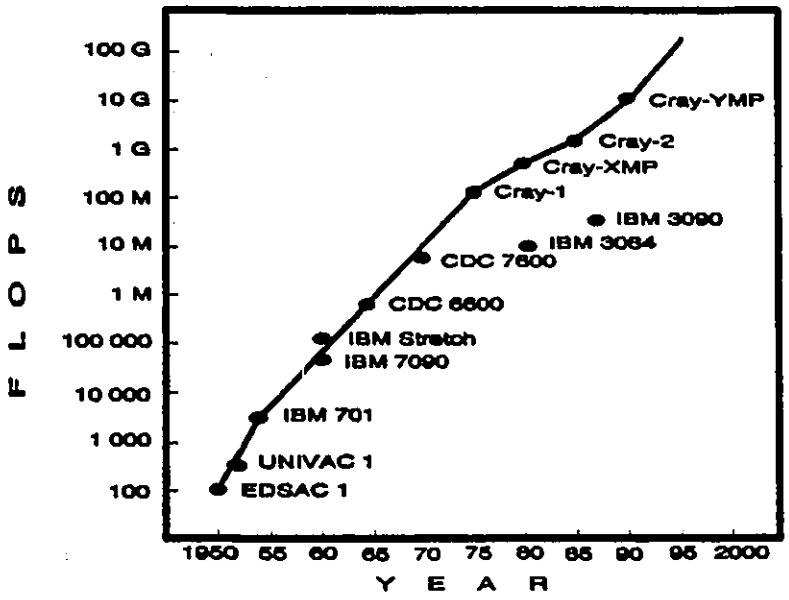

Figure 3.1: Peak computer performance 1950-1995

In parallel processing, instead of using a single CPU, many processors work simultaneously to solve a common problem. An IEEE Committee report [1992] gives a concise definition of parallel processing as: **parallel processing is a form of information processing in which two or more processors together with some form of interprocessor communications system, co-operate on the solution of a problem.** In parallel computers, the single CPU is replaced by many processors, which, even if individually somewhat slower, accelerate the speed of processing by operating in parallel. Expensive high-speed intermediate memory such as the cache can be omitted entirely. This development has created some new challenges. These challenges/issues are discussed in the following sections.

3.3 ISSUES IN PARALLEL PROCESSING

Issues in parallel processing can be broadly divided into two main groups, namely: (i) architecture, (ii) software and algorithms (IEEE Committee Report [1992]). These issues are discussed separately.

3.3.1 Classifications of Parallel Processors/Architectures

Parallel computers have a variety of models. Early parallel computers were built to solve specific problems. A typical one which is suited to solving spatially decomposable problems such as occur in PDEs (Partial Differential Equations) and image processing consists of one- or two-dimensional array of processors, with nearest neighbour interconnections. A processor is used to oversee the states of all processors. For more general purpose computation, loosely coupled processors, termed as multiprocessors are used in which each processor has more control of its own computation. Systolic arrays[1] are designed for special purpose computation such as solving linear systems (Ortega and Voigt [1985]). In this instant, data movement is regular and each

[1] Systolic refers to a distributed memory computation that is highly regimented and synchronous. During a tick of a global clock, each processor communicates with its neighbours and then performs some local computation (Golub and Van Loan [1989]).

processing unit has a built-in computational unit specifically to solve the given problem.

Early literature on the subject of parallel processing invariably referred to the classification made by Flynn [1966]. The parallel architectures are classified into two: SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data). With SIMD, all processors execute the same instruction (or no instruction) in lockstep, but on different data. Since processors are always synchronized, problem of data contention does not arise. The SIMD model works best on a certain class of problems, example, image processing. For general purpose computation the MIMD model is more suitable. In MIMD systems, the instruction may differ across the processors which need not operate synchronously. The processors may be solving a single problem or may be solving many problems simultaneously. Each processor of a MIMD machine, has its own memory - distributed - and can therefore execute a different program. In another model, the symmetric multiprocessor, a global shared memory is added which is accessible by all processors. Using shared memory, the number of processors is however limited. Therefore, massively parallel processors (MPP) are distributed memory machines.

Gurd [1988] reclassified parallel processors into broad categories of either distributed or shared memory type. This reclassification is a refinement of the earlier taxonomy by Flynn [1966]. In distributed memory (integrated) multiprocessor, each processing element (PE) has its own local memory and program. The PE's are connected in a network topology such as mesh, ring or hypercube. The processors communicate with each other by sending and receiving messages. The communication in shared memory systems is achieved by reading and writing to shared variables that reside in common memory.

Duncan [1990] proposed another taxonomy in an attempt to overcome the problem of Flynn's categorization, that is, its inability to include synchronous type architectures such as the systolic array. Furthermore the parallel architectures are categorized from a high-level framework, that is, the

perspective of parallel programming development. Figure 3.2 shows Duncan's taxonomy.

Referring to figure 3.2, synchronous and MIMD paradigm architectures are more specialized in their applications in comparison to the MIMD category. Commercially, distributed and shared memory models of the MIMD architectures are more widely available than the other two categories. Issues of parallel architectures will continue to dominate the parallel processing world. However, to a certain extent this problem may be circumvented by adopting a top-down approach. That is, to develop suitable algorithms and implement them and then determine the appropriate architecture later.

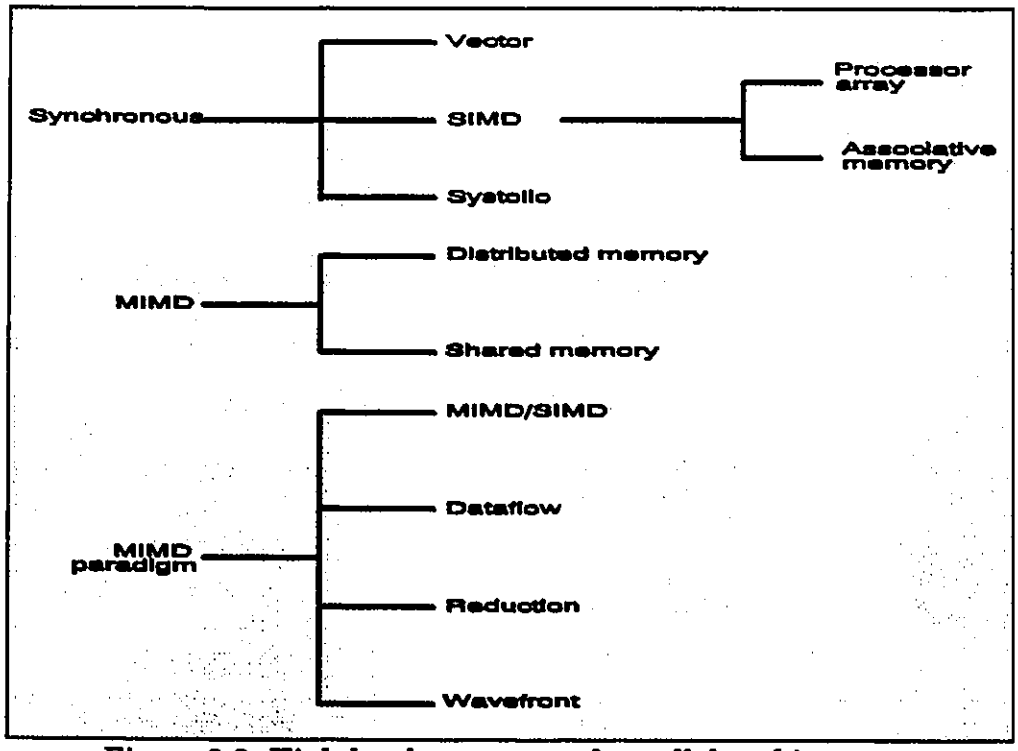

Figure 3.2: High-level taxonomy of parallel architectures

3.3.2 Software and algorithms

There may be as many supporting compilers as there are parallel computer vendors. The immaturity of parallel system software has slowed down its use in the industry. Lack of programmers, applications and the non-

portability of code are some of the major current drawbacks of parallel processing.

There are a number of issues that always emerge concerning parallel software algorithms:

i. **Speedup.** This is defined as the ratio of two programs execution times. If τ_1 is the time to run a program on one processor, and τ_{np} is the time to run it on n_p processors, then the speedup is $S = \tau_1 / \tau_{np}$. It is important that a comparison be made against the fastest serial program executing on the same computer.

ii. **Amdahl's Law.** This law states that, if σ is the proportion of the calculation that is serial, and $1 - \sigma$ is the portion that can be parallelized; then the speedup that can be achieved with n_p processors is:

$$S = \lim_{n_p \to \infty} \frac{1}{\sigma + \frac{1 - \sigma}{n_p}} = \frac{1}{\sigma}$$

Having 10% of the code that must be executed in serial, maximum speedup obtainable is 10 with any number of processors. Therefore, when developing a parallel program, the serial part must be minimized

iii. **Granularity.** This is the size of the task done by a processor between communication events. In coarse-grain, the computation-to-communication ratio is large. Whereas in fine-grain, the computation-to-communication ratio is small. A coarse -grain type problem is normally solved using few processors, 4 - 10. Whilst a fine-grain problem is solved on systems with a large number of processors - MPP (Massively Parallel Processors).

iv. **Serial Slowdown (S^2) factor.** The ratio of the execution times of two programs on a single processor. The first program is the conventional serial program - the fastest. The second program is the parallel program that solves the same problem running on the single processor. In this thesis, this factor is sometimes called partitioning penalty, P_p.

There are two approaches to parallelizing serial algorithms - using a

parallelizing compiler on an existing serial program, and the development of a parallel algorithm and program. Development of parallel programs is very time intensive. Using parallelizing compilers, serial programs can now be parallelized without much effort. But, as stated in IEEE Committee Report [1992] on Parallel Processing - *since these codes are often not written with parallelism in mind, little of it can be parallelized leaving parallel gains to be only slightly greater than 1.0 regardless of the number of processors used.* As rightly put by Dr. John Ross[2] - *an efficient parallel program must be developed and not converted.*

3.4 PARALLEL PROCESSING IN THE POWER INDUSTRY

As pointed out by Tinney [1977], there are four main factors that contribute to the computational burden of power system simulations. These four factors are discussed here in light of the role of parallel, distributed processing.

(i) **Size of network.** Over the years, size of the power system network has steadily increased, mainly due to interconnections of previously isolated systems. To reduce the computational burden, thus making execution time more tolerable, equivalencing of remote systems has been used extensively, however, sometimes at the expense of accuracy. By using faster computers and/or parallel processing, the speed of computation will remain at least tolerable - if not being improved considerably - even when modelling the full system.

(ii) **Details of modelling.** The prime objective in simulation has always been to model the physical system as accurately as possible. However, due to the additional computational burden imposed by detailed models, simplifications are generally made. With the use of faster computers, coupled with parallel processing, this limitation will become secondary.

(iii) **Sophistication of Applications.** Current trends in application software

---

[2] Dr. John Ross of High Performance Research Computing, University of Toronto at OUCC'93 Conference, McMaster University.

include the use of sophisticated graphical user interfaces. Graphical user interfaces, until recently have been extensively used in engineering workstations, but, are now becoming popular on personal computers (PC). This demand for sophistication by computer users is expected to continue (Mitsche [1992]).

(iv) **Time period of dynamic simulations.** The power system simulation period has now been extended into several minutes to study phenomena involving voltage collapse and islanding. Until now, the time domain simulation approach has provided the most accurate replication of the actual dynamics involving voltage instability (Morison, et. al. [1992]). These simulations are very time consuming due to the longer simulation period. Other simulations that require substantial improvement in computation speed include post-mortem analysis and operator training. Therefore, methods to improve computational speed are very much required to accommodate this extension.

3.4.1 Solution of linear systems

In power system analysis, most problems require the solution of linear algebraic systems in the form:

$$Ax = b \tag{3.0}$$

In most power system problems, it is advantageous to solve equation 3.0 using the direct method. The highly sparse matrix A is factorized into lower triangular, diagonal and upper triangular matrices such that:

$$A = A_L A_D A_U \tag{3.1}$$

Following the factorization, x can be obtained by solving:

$$A_L z = b \tag{3.2}$$

$$A_D A_U x = z \tag{3.3}$$

using forward and backward substitutions. Most parallel algorithms that have been developed focus on factorization (equation 3.1), forward solution (equation 3.2) and backward solution (equation 3.3). Since algorithms to solve these equations are distinct, they are normally handled separately. A comprehensive survey of parallel algorithms for sparse linear systems can be found in Heath, et. al., [1990]. Basically, the implementation on parallel architectures has caused no fundamental change in the overall approach to solving sparse linear systems - ordering, symbolic factorization, numeric factorization and triangular solutions. However, additional steps are now required to be performed - decomposition into subtasks and mapping onto processors.

As observed by Donggara [1993], there are three platforms emerging in scientific high-performance computing:

(1) ***Traditional, general purpose supercomputers,***

(2) ***Highly parallel computers (state-of-the-art), and***

(3) ***Clusters of workstations.***

In this thesis, we have chosen the third platform. The third approach is likely to be more cost effective than the other two. In chapter 5 of this thesis, solution of linear systems on a cluster of workstations is discussed. The approach is to use coarse-grain partitioning and static scheduling with the same procedures used in serial algorithms applied in each processor.

3.5 APPROACHES TO PARALLEL STABILITY CALCULATIONS

There is no agreed categorization of approaches to parallel transient stability algorithms. In this thesis, the algorithms are categorized into the following three main approaches based on how parallelism can be exploited:

(i) ***Parallel-in-space***

(ii) ***Parallel-in-time***

(iii) ***Parallel in both time and space simultaneously***

In all of the approaches above, parallelism can be exploited from task or job level or both. The basic algorithm can either be any of the traditional

methods commonly in used now or some non-traditional methods such as the Waveform Relaxation Method (Lelarasmee, et. al. [1982]).

3.5.1 Parallel-in-Space Approach

The Parallel-in-space approach entails the division of the problem into loosely coupled or independent parts. In the transient stability problem, the most obvious parts that can be executed independently are the differential equations describing each machine. However, dividing the network into loosely coupled parts is not straight forward and, therefore, much attention has been focused on this area which will be evident from the range of work described in this section.

The most significant early work on parallel solution of the transient stability problem is the feasibility study described by Hatcher, et. al. [1977]. Using the Trapezoidal integration method, the differential-algebraic equations were reduced to algebraic equations. The authors then focused on the solution of the algebraic equations. Equations (2.11), (2.12), (2.13) and (2.14) described earlier in chapter 2 are repeated here for easy reference.

$$\begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta V \end{bmatrix} = \begin{bmatrix} -F \\ -G \end{bmatrix} \tag{3.1}$$

Applying Gaussian elimination to equations (3.1), the resulting equation is:

$$\begin{bmatrix} J_1 & J_2 \\ 0 & \hat{J}_4 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta V \end{bmatrix} = \begin{bmatrix} -F \\ \hat{G} \end{bmatrix} \tag{3.2}$$

where:

$$\hat{J}_4 = J_4 - J_1^{-1} J_3 J_2 \tag{3.3}$$

$$\hat{G} = -G - J_1^{-1} J_3 F \tag{3.4}$$

Using the decoupled Newton method the iterative scheme for solving equation (3.2) is:

$$\Delta V^{(k+1)} = -J_4^{-1(k)} G^{(k)} \tag{3.5}$$

$$\Delta x^{(k+1)} = -J_1^{-1(k)} F^{(k)} \tag{3.6}$$

In solving equation (3.5) the authors arranged matrix J_4 into block diagonals and off-diagonal matrices:

$$J_4 = J_d + J_O \tag{3.7}$$

The matrix form in equation (3.7) can be realized by using either the NBDF (Near Block Diagonal Form) or the BBDF (Block Bordered Diagonal Form) ordering techniques. The forms of NBDF and BBDF on a 840-bus system are shown in figures 3.3 and 3.4 respectively (using function `ylis` in DCPS). An iterative scheme, called the Newton-SOR (Successive Over-Relaxation), is then applied to equation (3.5) using equation (3.7):

$$\Delta V^{(k+1)} = -J_d^{-1(k)} (G^{(k)} + J_O \Delta V^{(k)}) \tag{3.8}$$

During each iteration, the diagonal blocks are solved in parallel. In this approach, the most important part is the decomposition of the network into block forms. As emphasized by the authors, the goal of the decomposition is to divide the network into almost equal size blocks and minimize interconnections between blocks. In chapter 4, network partitioning will be discussed in more detail.

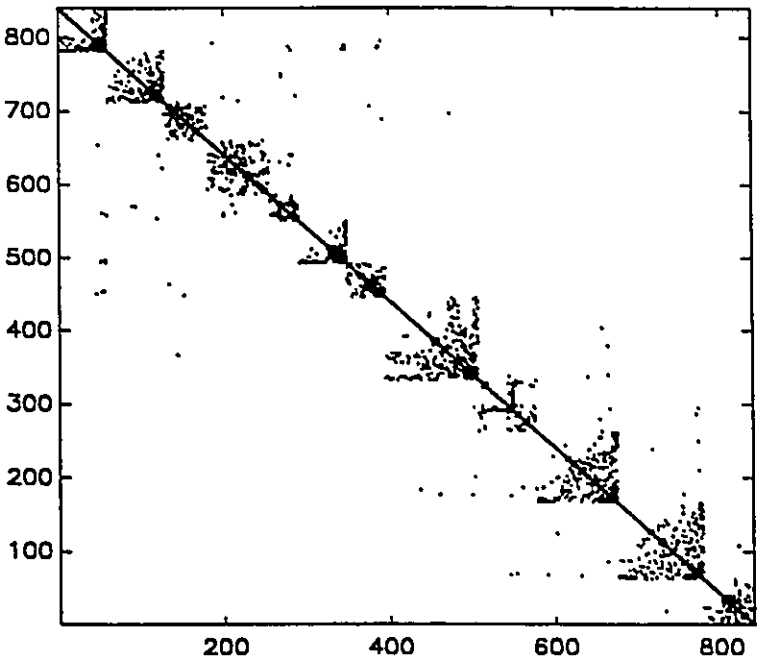

Figure 3.3: NBDF form for the 840-bus system

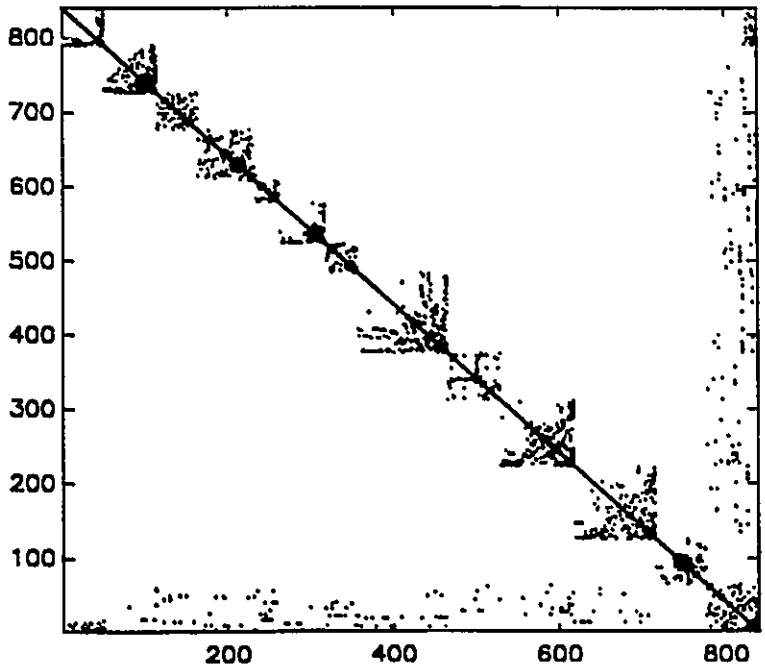

Figure 3.4: BBDF form for the 840-bus system

As observed by the authors, the BBDF ordering performs better than the NBDF ordering. The authors also proposed the architectures of the parallel processors to suit the algorithms. The work of Fong and Pottle [1978] follows the same strategy as that of Hatcher, et. al., [1977] but envisaged the use of microcomputer structures.

All traditional solution approaches can be parallelized in space. One simple technique was demonstrated in Lee, et. al. [1989]. In their paper, the authors used classical machine models whose algebraic differential equations are given by:

$$M_i\ddot{\delta}_i = P_{mi} - P_{ei} - D_i\dot{\delta}_i \tag{3.9}$$

$$I_B = YV \tag{3.10}$$

where M_i is the inertia, D_i is the damping constant, P_{mi} is the mechanical power, P_{ei} is the electrical power and Y is the admittance matrix. Using the method described in Pai [1981], the electrical power term in equation (3.9) is expressed in terms of the elements of the reduced admittance matrix thus combining equations (3.9) and (3.10) into a single differential equation:

$$M_i\ddot{\delta}_i = P_{mi} - V_i^2 G_{ii} - \sum_{j=1, j\neq i}^{n_m} (V_i V_j B_{ij}\sin\delta_{ij} + V_i V_j G_{ij}\cos\delta_{ij}) \tag{3.11}$$

where G_{ij} and B_{ij} are real and imaginary components of element ij of the reduced admittance matrix, V_i and V_j are machine terminal voltage magnitudes, δ_{ij} is the difference of the voltage angles between buses i and j, and n_m is the number of generators. In this technique, there are two main computational procedures, (i) formation of the reduced admittance matrix during the initial stage and every-time after switching, and (ii) solving the differential equations in (3.11). Although the problem demonstrated in the paper has very little practical use, it does provide some insight into the issues of parallel computation, including load balancing - to equally distribute tasks amongst processors - and communications. When forming the reduced

admittance matrix, matrix Y has to be factorized. Parallel Gaussian elimination is used by the authors for factorization. In the paper, the load balancing problem during factorization is reduced by combining the methods of *block division* and *modulo division* to distribute rows of the matrix among the processors in a hypercube. To ensure load balancing in solving equation (3.11), all processors are assigned equal number of equations.

The most recent work that exploits space parallelism to solve the transient stability problem is that by Decker, et. al. [1992]. The basic algorithm used is called the Interlaced Alternating Implicit (IAI) method. This method is similar to the Partitioned scheme described by Stott [1979] - Trapezoidal integration with fixed-point iteration. Equations describing the power system in (2.1) and (2.2) can be rewritten as:

$$\dot{x} = f(x,u) = Ax + Bu(x,V) \tag{3.12}$$

$$YV = I_B(x,V) \tag{3.13}$$

where x is the vector of the generator states, u is a vector interfacing quantities between stator and rotor, V is the vector of complex voltages, I_B is the vector of nodal current injection and Y is the admittance matrix. Applying an implicit integration method to equation (3.12) and re-arranging equation (3.13), the following non-linear equations are obtained:

$$F = x - k_c hf(x,u(x,V)) - C \tag{3.14}$$

$$G = I_B(x,V) - YV \tag{3.15}$$

In the IAI method, equations (3.14) and (3.15) are solved alternately and iteratively. One reasonable simplification made - provided a small time step is used - in the algorithm is that the network equation (3.15) is solved in only one iteration during each time step except after a switching operation. Since the equations in (3.14) describing each machine are independent, they are easily organised in parallel. To solve the network equation (3.15) in parallel,

it is first decomposed into subnetworks by analyzing the one-line diagram of the system. The admittance matrix in BBDF form comprises of r+1 blocks. The network solution is divided into two phases - (i) the solution of the first r diagonal blocks and (ii) solution of the block bordered submatrices. Phase (i) is inherently parallel. To solve phase (ii) in parallel, the Conjugate Gradient Method (CGM) with preconditioning is used. The algorithm was implemented on a transputer system and substantial speedup of 6 times with 8 processors was reported.

3.5.2 Parallel-in-Time Approach

The most significant work to use the parallel-in-time approach is that by Alvarado [1979]. Although the method has not been implemented for practical use, Alvarado's is a new way of looking at the transient stability problem. In the method, trapezoidal integration is applied to equation (3.12) which is reorganized into a form:

$$\dot{x} = Ax + f(t) \tag{3.16}$$

that results in algebraic equation:

$$x^{(n)} = x^{(n-1)} + \frac{h}{2}[Ax^{(n-1)} + f^{(n-1)} + Ax^{(n)} + f^{(n)}] \tag{3.17}$$

Equation (3.17) is then rearranged as:

$$(I - \frac{h}{2}A)x^{(n)} = (I + \frac{h}{2}A)x^{(n-1)} + \frac{h}{2}(f^{(n-1)} + f^{(n)}) \tag{3.18}$$

Since **f** is an explicit function of time, the whole of the r.h.s of equation (3.18) can be evaluated and $x^{(n)}$ can be determined by solving a linear system. Instead of doing this, the author proposed a method to solve for the **x** vectors at T_N time steps simultaneously. T_N is chosen such that, $T_N = 2^{\tau_N}$, where τ_N is an integer. A new vector x^1 is defined as:

$$x^t = (x_1, x_2, \ldots, x_{T_N}) \tag{3.19}$$

A set of T_N equations can then be written as:

$$(I - \frac{h}{2}A)x_1 - (I + \frac{h}{2}A)x_0 = \frac{h}{2}(f_o + f_1) \tag{3.20}$$

$$\begin{aligned} (I - \frac{h}{2}A)x_2 - (I + \frac{h}{2}A)x_1 &= \frac{h}{2}(f_1 + f_2) \\ &\vdots \end{aligned} \tag{3.21}$$

$$(I - \frac{h}{2}A)x_{T_N} - (I + \frac{h}{2}A)x_{T_N-1} = \frac{h}{2}(f_{T_N-1} + f_{T_N}) \tag{3.22}$$

or in more compact form:

$$Ax = b \tag{3.23}$$

Therefore T_N time steps are solved simultaneously. Since matrix A in equation (3.23) comprises of T_N diagonal blocks, these blocks can be solved in parallel. Although the author was very optimistic in using the technique for full-blown transient stability solution, the inclusion of network equations complicates the technique. However, this paper has provided many incentives for researchers to probe further into the possible exploitation of parallel-in-time elements.

3.5.3 Parallel both -in-Time and -in-Space Approach

The waveform relaxation method which was first used by LeLarasmee, et. al. [1982] and later adopted for power systems analysis by Ilic'-Spong, et. al. [1987] and Crow, et. al. [1990] is considered to be the first attempt to exploit both space and time parallelism in transient stability problem. According to the method, equations (3.12) and (3.13) are decomposed into r subsystems. Equations (3.12) and (3.13) are first rewritten as:

$$\dot{x} = f(x,y) \tag{3.24}$$

$$0 = g(x,y) \tag{3.25}$$

As a result of the decomposition, r sets of differential-algebraic equations are obtained:

$$\dot{x}_1 = f(x_1,x_2,\ldots,x_r\ ,\ y_1,y_2,\ldots,y_r) \qquad x_1(0) = x_{10} \tag{3.26}$$

$$0 = g_1(x_1,x_1,\ldots,x_r\ ,\ y_1,y_2,\ldots,y_r) \qquad y_1(0) = y_{10} \tag{3.27}$$

$$\dot{x}_2 = f(x_1,x_2,\ldots,x_r\ ,\ y_1,y_2,\ldots,y_r) \qquad x_2(0) = x_{20} \tag{3.28}$$

$$\begin{aligned} 0 = g_2(x_1,x_1,\ldots,x_r\ ,\ y_1,y_2,\ldots,y_r) \qquad & y_2(0) = y_{20} \\ & \vdots \end{aligned} \tag{3.29}$$

$$\dot{x}_r = f(x_1,x_2,\ldots,x_r\ ,\ y_1,y_2,\ldots,y_r) \qquad x_r(0) = x_{r0} \tag{3.30}$$

$$0 = g_r(x_1,x_1,\ldots,x_r\ ,\ y_1,y_2,\ldots,y_r) \qquad y_r(0) = y_{r0} \tag{3.31}$$

The total solution time is then divided into several windows, $0\text{-}T_1, T_1\text{-}T_2, \ldots, T_{N-1}\text{-}T_N$. The solution over each time window comprises of several iterations. During each iteration each subsystem is solved independently. At the end of each iteration, tie-line voltage waveforms are exchanged and iteration continues. This process continues until convergent. The method exploits time parallelism since over a window period, subsystems are solved simultaneously. The space parallelism is present due to the decomposition of the governing equations in (3.24) and (3.25). The refinement in this method includes the use of slow-coherency properties in partitioning. The power system community was at first very enthusiastic about this unconventional technique. However, the method converges slowly and no additional development has been done.

The work of La Scala, et. al. [1989, 1990a, 1990b, 1991] also attempt to exploit space and time parallelism using the Gauss-Jacobi Block Newton method. The parallel-in-time method for solving the transient stability problem over an interval $[0,T_N]$ is formulated using:

$$H(y) = 0 \tag{3.32}$$

where H is the compacted form of equations (3.14) and (3.15) and $y = [x\ V]^T$. Equation (3.32) can be decomposed and written as:

$$\hat{H}(\hat{y}) = 0 \tag{3.33}$$

where:

$$\hat{y} = \left[y_1^T\ y_2^T\ .\ .\ y_t^T\ .\ .\ y_{T_N}^T\right]^T \tag{3.34}$$

$$\hat{H} = \left[H_1^T\ H_2^T\ .\ .\ H_t^T\ .\ .\ H_{T_N}^T\right]^T \tag{3.35}$$

The iterative algorithm that follows solves (3.34) and (3.35) at each time step and then relaxes the solution on all the time steps concurrently. The Gauss-Jacobi Block Newton iteration is given by:

$$y_n^k = y_n^{k-1} - \left[\frac{\partial H_n}{\partial y_n}\right]^{-1}_{y_n^{k-1}, y_{n-1}^{k-1}} H_t(y_n^{k-1}, y_{n-1}^{k-1}) \tag{3.36}$$

There are T_N independent processes that can be solved in parallel during each iteration. Time windowing similar to the waveform relaxation method is also applied to improve convergence. The most attractive feature of this method is the massive parallelism it possesses. The major drawback of the algorithm is the large number of iterations required for convergence - 60 iterations is quoted in the paper to solve for 0-1 second window.

Another relaxation algorithm introduced by La Scala, et. al. [1991] is based on the method of successive substitution. The admittance matrix Y in equation (3.13) is split into diagonal and off-diagonal parts as follows:

$$Y(x_n) = Y_D(x_n) + Y_O \tag{3.37}$$

where Y_D is the 2×2 block diagonal part of Y (expanded form) and Y_O is the 2×2 block of off-diagonal parts. The voltage V_n can then be solved using:

$$V_n = Y_D^{-1}(x_n) I_B(x_n, V_n) - Y_D^{-1}(x_n) Y_O \tag{3.38}$$

Using the Trapezoidal integration method on the quasi-linear form of equation (3.12), the iterative scheme that follows is:

$$x_n^{k+1} = x_{n-1}^{k} + \frac{h}{2}[A(x_n^k + x_{n-1}^k) + B(u(x_n^k, V_n^k) + u(x_{n-1}^k, V_{n-1}^k))] \tag{3.39}$$

$$V_n^{k+1} = Y_D^{-1}(x_n^k) I(x_n^k, V_n^k) - Y_D^{-1}(x_n^k) Y_O V_n^k \tag{3.40}$$

where $t = 1{:}t_N$ and k is the iteration index for the relaxation method. This method has significantly improved the convergence rate as compared to the earlier method. However, both methods were never implemented on a parallel computer.

Chai, et. al. [1991] presented a parallel version of the Very Dishonest Newton (VDHN) method. Using equation (3.35) and (3.36), the following relaxation scheme is defined:

$$\Delta V_n^{(k+1)} = -[\hat{J}_4^{-1}]^{(0)} \hat{G}(x_t^k, x_{n-1}^k, V_t^k, V_{n-1}^k) \tag{3.41}$$

$$\Delta x_n^{(k+1)} = -J_1^{-1(k)}(F_{1n}^k + J_{2n}^k \Delta V_n^{k+1}) \tag{3.42}$$

Superscript (0) denotes that J_4 is not updated unless the number of iterations is greater than a threshold value. During each iteration, solution of (3.41) and (3.42) for all time steps can be carried out in parallel. The same paper also described the SOR-Newton algorithm using an approximated Jacobian matrix containing only diagonal elements. The algorithms were implemented on both shared memory and distributed memory parallel computers and significant speedup - maximum of 7.4 - was obtained.

3.5.4 Discussions

As rightly pointed out by the IEEE Committee Report [1992]:

"*Most of the results although encouraging, do not yet indicate clear cut paths for the development of production grade engineering tools. Probably the largest uncertainty today is the evolution of hardware. Although several parallel machines are commercially available, they are largely being used for research purposes and their acceptability to industry and long term viability are very unpredictable.*"

In this thesis, to overcome the current dilemma of the industry with respect to the choice of hardware, the use of a network of workstations is proposed. Although a network of workstations represents a large amount of computing power, a single user often cannot utilize this power for his applications (Singh, et., al., [1991]). There are basically three main reasons to the under-utilization of the network computing power: (i) heterogenous nature of the available computers in the network that has varying capabilities and capacities, (ii) the high cost of communications may restrict the range of problems and the exploitable parallelism that can be implemented on these computer systems, and (iii) the difficulty in programming for communications. To overcome the first problem, homogenous workstations are used in this thesis. This choice is practical, since a utility will normally prefer to acquire one type of workstation rather than have a mixture of makes. Fortunately the third problem can be overcome by using software (compiler) that allows one to write code for network communications that hides all the details and intricacies associated with low-level programming language.

The second problem is more difficult to overcome. When using a network of workstations for parallel, distributed processing, communication overheads must be minimized. To achieve minimum communication the problem has to be partitioned into large parts - coarse-grained - rather than small tasks - fine-grained. Algorithms proposed by Hatcher, et. al. [1977], Decker, et. al. [1992] and Crow, et. al. [1990] are all based on coarse-grained partitioning. These algorithms are therefore potential candidates for

implementation in a network of workstations.

3.6 PROPOSED METHOD TO EXPLOIT PHYSICAL PARALLELISM

The methods described by Hatcher, et. al. [1977], Decker, et. al. [1992] and Crow, et. al. [1990] require system decomposition for the purpose of partitioning. For partitioning, the first two workers used a network topology based technique and in the third work a slow-coherency based method together with a heuristic technique to include load buses. The slow-coherency partitioning technique first developed by Chow [1982] for use in power system analysis, divides the network into several coherent areas. This technique is further developed in this thesis and will be discussed in chapter 4. Following a disturbance, generators in the same coherent area tend to oscillate together. This behaviour is used to decouple the coherent areas. By such grouping, submatrix $\mathbf{J}_4$ in equation (3.1) will comprise of a block diagonal matrix $\mathbf{J}_d$ and off-diagonal matrix $\mathbf{J}_O$:

$$\mathbf{J}_4 = \mathbf{J}_d + \mathbf{J}_O \tag{3.43}$$

The number of blocks, r depends on the number of coherent areas. The blocks are normally of different sizes that depend on the coherent characteristic of the system. Each block is also sparse. Using the Decoupled Newton method of equations (3.5) and (3.6), an iterative scheme for a particular block i that is similar to the one by Hatcher, et. al. [1977] but now includes the machine equations, becomes:

$$\Delta \mathbf{V}_i^{(k+1)} = -\mathbf{J}_{di}^{-1(k)}(\mathbf{G}_i^{(k)} + \mathbf{J}_{Oi}\Delta \mathbf{V}_{\neq i}^{(k)}) \tag{3.44}$$

$$\Delta \mathbf{x}_i^{(k+1)} = -\mathbf{J}_{1i}^{-1(k)}\mathbf{F}_i^{(k)} \tag{3.45}$$

where $\neq$i denotes not area i. Since $\mathbf{J}_1$ consists of independent machine blocks, they can be arranged to be associated with the blocks in $\mathbf{J}_4$. The overall Gauss-Jacobi iterative scheme is given in the following algorithm (Algorithm 3.1).

Algorithm 3.1: Gauss-Jacobi iterative scheme

(i) Increase time:
(ii) for i = 1:r

(iii) solve: $\Delta V_i^{(k+1)} = -J_{di}^{-1(k)}(G_i^{(k)} + J_{oi}\Delta V_{*i}^{(k)})$

$$\Delta x_i^{(k+1)} = -J_{11}^{-1(k)} F_i^{(k)}$$

(iv) Update x_i and V_i and check convergence
(v) If not converged goto (iii)
end for
(vi) goto (i)

Part (ii) to (v) of algorithm 3.1 can be executed in parallel, that is, calculations for each area. In the above relaxation method, variables $V_{*\,i}^{(k)}$ are relaxed whilst performing calculations on block i during each time step. This algorithm only exploits space parallelism.

3.7 MULTIPROCESSOR SIMULATIONS ON A UNIPROCESSOR SYSTEM

Algorithm 3.1 was implemented on a uniprocessor computer as reported in Yusof, et. al. [1992]. In this section, the implementation is briefly described. Under the Unix operating system, it is possible to create child processes[3] from a single parent process. Each newly created process will obtain its own share of time-slice and operate under its own control. This, combined with the facility to create a shared memory segment[4], enables a uniprocessor system to simulate a multiprocessing environment. These aspects

[3] Under a Unix operating system, whenever the user executes a program, a process is created. The process contains the program as well as other information required by the operating system - page tables, information on open files, the stack. A child process is an exact copy of the original process - the parent (See Brawer [1989]).

[4] A shared memory facility allows variables to be shared amongst the many processes (See Valley [1991]).

- child processes, shared memory and semaphores[5] - will be discussed in more detail in chapter 5. Good sources of reference on this subject can be found in Brawer [1989], Bauer [1992] and Valley [1991].

The implementation of algorithm 3.1 is best described using the corresponding DCPS command file shown in text box 3.1. Some new commands in the file deserve further explanation. The ordering function `rena` is used instead of `orde` or `ord1` so that the admittance matrix is suitably ordered into NBDF form (see figure 3.3). The admittance matrix is copied to a working matrix using function `copm`. The diagonal blocks and off-diagonal parts are separated using function `dblk`. The off-diagonals are sparsely stored in collection of vectors for efficient use in equation (3.5). Factorization of diagonal blocks is performed using function `fac5` that factorizes each block independently. The first parameter of the `tsol` function is 7 which indicates the use of Gauss-Jacobi iterative method.

[5] A semaphore is a counter typically used to access to a limited pool of shared resources (See Valley [1991]).

```
%=============================================================
% Tgj118.com: Block Iteration Transient Stability
%=============================================================
chco,0.001,0.001,0.01,1.0,,,,
redn,s118b.sol,dcps                    'load network data file
redm,s118b.mac                         'load machine data file
rena;                                  'renumber buses by
areas
ybus;                                  'form admittance matrix
bpow,0                                 'calculate bus powers
iniv;                                  'initialise states
moym;                                  'add machine admittance
cinj;                                  'add in equivalent load
ginj;                                  'calculate current
copm;                                  'copy ybus to slu
dblk;                                  'extract diagonal block
fac5;                                  'factorise slu
caps,2,12,4                            'mark states to capture
caps,2,10,4
caps,2,27,4
caps,2,26,4
caps,2,49,4
caps,2,59,4
caps,2,69,4
caps,2,73,4
caps,2,89,4
caps,2,91,4
caps,2,107,4
caps,2,100,4
tsol,7,0.0,0.02,.02,0                   'time solution
asht,69,0.0,1000.0                      'add shunt
copm;
dblk;
fac5;
tsol,7,0.04,0.10,.02,0                 'continue solution time
rsht;                                  'remove shunt
copm;
dblk;
fac5;
tsol,7,0.12,1.12,.02,0                 'continue solution time
svou;                                  'save sates
```

Text box 3.1: Command file for implementing algorithm 3.1

3.8 RESULTS OF SIMULATIONS

Algorithm 3.1 was applied to four power systems - the IEEE 39-bus test system, the IEEE 118-bus test system, the 348-bus TNB/PUB system and a

662-bus Mid-Western U. S. system. The main objective of the simulation is to estimate the ideal speedup on shared memory parallel computer systems. This is because the simulations are performed based on concepts of shared memory. It is also difficult to estimate the speedup because of the difficulty in assessing the communication overheads due to data contention and also the effects of barriers and data locking. As long as the parallel version has reasonable S^2 factors, the method may be promising on shared memory computers. We note that for the largest system, the S^2 factor is 1.28 which small and thus when executed on actual shared memory computer would tend to gain the most.

Power Systems	Serial (sec.)	Parallel Gauss-Jacobi (sec.)	Parallel Gauss-Seidel (sec.)	No. of Areas	S^2 factor	Ideal speedup
39-bus	1.25	5.53	4.37	6	4.42	1.35
118-bus	8.55	12.21	9.80	6	1.43	4.20
348-bus	11.95	21.57	19.25	5	1.81	2.77
662-bus	44.97	57.42	48.54	9	1.28	7.04

All times in CPU seconds.

Table 3.1: S^2 factor & ideal speedup for algorithm 3.1 on shared memory computer

3.9 CONCLUSIONS

In this chapter the general subject of parallel processing has been discussed. Issues related to power system applications of parallel processing have been described. The chapter also reviewed the state-of-the-art in parallel transient stability calculations. We described our approach to exploit parallelism - a general coarse-grain partition. Important issues related to hardware choice have been discussed and the most cost effective and widely available platform is proposed - a cluster of workstations.

CHAPTER 4
SLOW COHERENCY BASED NETWORK PARTITIONING

4. 1 INTRODUCTION

In parallel, distributed computation, the first requirement is to partition the problem into smaller tasks. These tasks can then be distributed to several processors. Each processor performs the required computation simultaneously in parallel on its partition. Whilst performing calculations, a processor may need to communicate its intermediate results to other processors or vice-versa. Partitioning can be broadly categorized into either fine-grain or coarse-grain- defined in chapter 3. In fine-grain partitioning, the problem is divided into many small tasks. Consequently, the communication bandwidth is large. Whereas with coarse-grain partitioning, the problem is divided into a few large tasks. As a result, there will be fewer communications between the processors. The approach used in this thesis is that of coarse-grain partitioning. The main reasons are: (i) the parallel, distributed algorithms are to be implemented on a cluster of workstations with slow communications, and (ii) to exploit the physical behaviour of the power system through the slow-coherency method. A cluster of workstations has slow communication speed, and coarse-grain partitioning always results in minimum interprocess communications. The exploitation of the physical behaviour of the power system is elaborated further in this chapter.

Traditionally, network partitioning is performed prior to network reduction. Transient stability studies require large amounts of computational effort in terms of time and memory. In practice, for a large power system, it is imperative that system decomposition into a study area and one or more

external areas be carried out to facilitate reduction in this computational burden. The objective is then to automatically decompose the network into several areas. The main characteristic of the decomposition is that elements in the same area are strongly coupled, whereas elements in different areas are weakly coupled. Therefore, it can be assumed that during the occurrence of any disturbance in the study area, the effect of such disturbance on the external areas is quite small. Thus, to reduce computational burden, the external areas can then be represented by simple dynamic equivalents while the study area is represented by its full nonlinear model. By using such system reduction, a large power system can be represented by a smaller system whilst maintaining the essential dynamic behaviour of the original system. As far as parallel, distributed processing is concerned the objective is not to perform network reduction. Instead the partitions are fully retained but distributed over several processors.

The slow coherency technique - first used by Chow [1982] and Chow, et. al. [1989] - has been successfully applied to partitioning of power systems into groups of weakly coupled generators. In this chapter, the development of a method for partitioning all the network buses into several coherent areas based on the slow-coherency technique is described. The development and results of the method described in this chapter is reported in Yusof, et. al. [1992].

In Chow, et. al. [1984], to include the network buses in the decomposition, a machine with a small inertia and without transient reactance is attached to every load bus. Therefore, the linearized electromechanical model of the artificial system included all buses. This results in a characteristic matrix having the same size as the network admittance matrix. However, using this procedure, the network sparsity was retained in the eigencomputation. In the paper, the Lanczos method was used for the sparsity-based eigencomputation.

In the method developed in this thesis, the eigenbasis matrix which is extended to include the network buses is first calculated and then the

algorithm used by Chow [1982] is applied to determine the reference generators. To determine to which reference generator each of the other network buses belongs, the closeness of each bus to all the reference generators is calculated. The closeness between a reference generator i and a bus j is determined by evaluating the cosine of the angle between the two row vectors of the eigenbasis matrix corresponding to buses i and j.

The motivation to include the load buses in the decomposition is to identify the boundaries between coherent areas which are also inherently weak links between partitions. With this information, the decision to retain certain buses and tie-lines in the network reduction program may be based on the physical characteristics of the network rather than being heuristic. In parallel, distributed processing, the boundary buses are required to enable the admittance matrix to be constructed in block form.

In this chapter, several network partitioning techniques are described. The slow-coherency technique for partitioning a power network into several groups of coherent generators is described. The proposed method to include the load buses in the partitioning process is elaborated. The implementation of the program and its usage are described. The applications of the technique will be illustrated using two test systems and two practical systems. The use of the partitioning for studying power system structure is also demonstrated.

4.2 NETWORK PARTITIONING METHODS

In general there are two approaches to power systems network partitioning. The first approach is based on network topology, while the second approach is based on the dynamic behaviour of the power system.

Carré [1966] described a graph-theoretic approach for network partitioning for use in the block iterative solution of linear systems. The basic idea of the approach is to divide the network into several trees because trees are easily ordered optimally. For small systems such as the one shown in figure 4.1, the trees can be determined by inspection. However, for large systems the author described a method of successive contractions. The method

involves a succession of elementary assignment operations in each of which a particular edge and its endpoints are assigned to a tree. When all assignments and corresponding contractions have been performed, the vertices remaining in the contracted graph represent the chosen section graphs, and the remaining edges are the connecting edges. This partitioning technique was then applied by Carré [1968] to load flow solution. Compared with the Gauss-Seidel iterative method, the block iterative method that the author developed executed in faster time and required fewer iterations to converge.

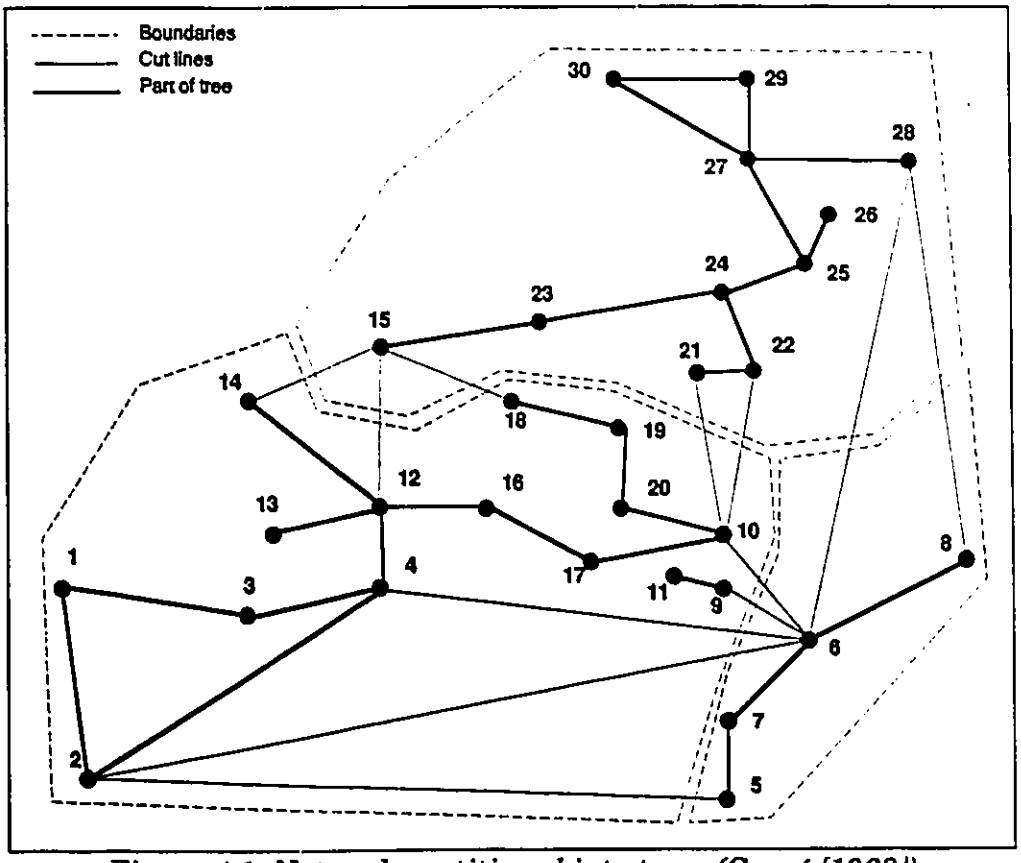

Figure 4.1: Network partitioned into trees (Carré [1968])

In the second approach, coherent behaviour of groups of generators after a disturbance is used. Basically there are two methods that can be used - (i) time domain based simulations, and (ii) eigenvalue analysis. Based on a criteria, for example (Haque [1988]):

$$\delta_i(t) - \delta_j(t) = a_{ij}(t) \pm \epsilon \qquad 0 < t < t_{max} \qquad (4.1)$$

- where $\delta_i(t)$ and $\delta_j(t)$ are rotor angles of generators i and j respectively, a_{ij} is a constant and ϵ is small positive number - groups of coherent generators are determined. A generator pair is said to be perfectly coherent if $\epsilon = 0$. A group of generators are considered as coherent if each pair of generators in the group satisfies the criteria of equation (4.1). Groups of generators are then reduced to a single equivalent machine.

The use of time simulation to determine groups of coherent generators is described by Podmore in an EPRI sponsored research project [1978]. Since the conventional time-domain simulation is too slow for a large system, the author proposed a fast simulation technique that retains the coherent characteristics. Two main assumptions are made - (i) coherent groups are independent of the size of the disturbance, and (ii) coherent behaviour can be retained using classical machine models. Both assumptions are quite valid based on observations of many simulations. The swing equations are first linearized and then the trapezoidal integration method is applied. As a result, a simple algorithm is developed with a constant Jacobian matrix. Rotor angle trajectories of all generators in the system are calculated and the criteria in equation (4.1) is applied for grouping.

Ram Nath, et. al. [1985] also used the linearized system matrix to determine weakly coupled subsystems. However, instead of using the criteria of equation (4.1), the authors used what they termed a 'coupling factor'. The coupling factor between two submatrices - whose elements concentrate on block diagonals -is defined as the ratio of the norms of the off-diagonals to the diagonal submatrices. These coupling factors are evaluated from the linearized state matrix.

Calculating the swing curves over a time period as in the method of Podmore [1978] can become slow as the system grows. This leads to the application of a more direct method. One particular work on the use of a direct method is by Haque [1988]. In this method, the author proposed a direct

method of coherency determination through a combination of faulted and post-fault relative angle measures, and electrical coupling between each pair of generators. The angles are obtained by a simple two-step Taylor series expansion of the faulted and post-fault systems.

All the dynamic-based methods described above focused on grouping of generators while paying little attention to other network buses. However, one technique which includes all buses in the partitioning is the one described by Quintana and Mûller [1991] and Mûller and Quintana [1992] which proposed a method based on the concept of electrical distances between the busbars of a system, independent of its operating condition. Strongly connected busbars - complex voltage and power injections are interdependent - are assigned to the cluster, and weakly connected busbars are assigned to different clusters. The technique is divided into three different stages and requires the computation of a few of the largest eigenpairs of a simplified version of the system admittance matrix. In this chapter, the method used is based on the slow-coherency technique and uses the system eigenvalues and eigenvectors of the linearized system model.

Zaborszky, et. al. [1985] used the so called textured model approach for grouping. Using the textured model to solve nonlinear algebraic equations the nodes of the system are arranged into groups around active inputs. t is assumed that the influence of the active inputs upon the outputs drops below a preset threshold level outside of the group boundaries. However, the difficulty in using this method is in selecting the threshold level.

4.3 SLOW COHERENCY DECOMPOSITION METHOD

Applications of the two timescale method in power systems are given in Chow [1982]. The two timescale method is the basis for the slow coherency technique for partitioning of a power system into groups of coherent generators. The coherent area identification technique is based on the observation that, in transients following a disturbance on a multimachine power system, some generators have the tendency to swing together. For

finding the slow coherent areas, the method requires the calculation of the slow eigenbasis matrix of the electromechanical model of the power system. The r (number of areas) most linearly independent rows of the eigenbasis matrix will become the corresponding reference generators. A grouping algorithm is then applied to group non-reference generators to the reference generators.

The nonlinear model of an n_m-machine, n_b-bus power system in which all machines are represented by classical models is given by:

$$M\ddot{\delta} = f(\delta, V) \tag{4.2}$$

$$0 = g(\delta, V) \tag{4.3}$$

where M is an n × n diagonal matrix of terms involving machine inertias, δ is an n_m-state vector of machine angles, V ($[\ V_r\ V_x\ ]^T$) is a $2n_b$-vector of real and imaginary components of bus voltages, f is the n_m-vector of acceleration power and g is the load flow equation. Linearizing equation (4.2) about an operating point δ_0 and V_0, we obtain:

$$\begin{bmatrix} \Delta\ddot{\delta} \\ 0 \end{bmatrix} = \begin{bmatrix} J_A & J_B \\ J_C & J_D \end{bmatrix} \begin{bmatrix} \Delta\delta \\ \Delta V \end{bmatrix} \tag{4.4}$$

where $\Delta\delta$ is an n_m-vector of machine angle deviation from δ_0, and ΔV is a $2n_b$-vector of the real and imaginary parts of bus voltage deviation from V_0. The terms J_A, J_B, J_C and J_D are Jacobian matrices comprised of the partial derivatives of terms in equations (4.2) and (4.3). Equation (4.4) is also called the augmented system state equation (ASSE) and the coefficient matrix is called the augmented system state matrix. From equation (4.4), we can construct the system state matrix A_S as:

$$A_S = J_A - J_B J_D^{-1} J_C \tag{4.5}$$

Using the system state matrix A_S, the grouping algorithm as found in Chow [1982] is applied as follows (see algorithm 4.1):

Algorithm 4.1: Slow Coherency grouping - generator only

(i) Choose the number of areas, r.
(ii) Compute the eigenvector matrix U of the r smallest eigenvalues.
(iii) Apply Gaussian elimination with complete pivoting on U to obtain r reference machines.
(iv) Order the first r rows of U (called U_1) according to the order found in Step (iii) and solve for L in: $U_1^T L^T = U_2^T$
where U_2 is r+1 to n_m rows of U
(v) Use the L matrix to assign other machines to the coherent areas according to the largest entry in each row of L.

To determine the boundaries of the coherent areas, Chow, et. al. [1989] proposed a heuristic method called the *branching algorithm*. In this algorithm, the boundaries of coherent areas are expanded by one tier of load buses at each step, starting from the generator terminal buses. At the first step, a coherent area consists of the terminal buses of the machines in the area. At each step an area would include additional undesignated load buses that have direct connections to the buses in the area. The *branching algorithm* terminates when all the buses have been designated to one of the coherent areas. This algorithm is therefore based on connectivity and is heuristic in nature. Another algorithm found in Crow, et. al. [1990] is an extension of the branching algorithm. Instead of only using the network connectivity as the criterion, the size of impedance of connecting transmission lines is taken into account.

4.4 PROPOSED METHOD TO INCLUDE LOAD BUSES

The classical machine representation is used in this analysis. The eigenvalue problem involving the system state matrix A_S is:

$$A_S u_i = \lambda_i u_i \tag{4.6}$$

where (λ_i,u_i) is an eigenpair of A_S. Each right eigenvector of A_S, u_i describes how each mode, $\exp(\sqrt{\lambda_i}t)$ is distributed between the states (Rogers and Kundur [1989]). In this case the states are angles of voltages behind the

generator transient reactances. Using the augmented system state matrix in equation (4.4), the eigenvalue problem can be defined as:

$$\begin{bmatrix} J_A - \lambda_i I & J_B \\ J_C & J_D \end{bmatrix} \begin{bmatrix} u_i \\ u_{vi} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4.7}$$

where (λ_i, u_i) is also an eigenpair of A_S as in (4.6). From equation (4.7), corresponding to each right eigenvector u_i there is a voltage vector u_{vi} which describes the mode shape within the system network. In the proposed method, the eigentriple (λ_i, u_i, u_{vi}) is calculated instead of only the eigenpair (λ_i, u_i). The vector u_{vi} can be obtained from:

$$u_{vi} = -J_D^{-1} J_C u_i \tag{4.8}$$

Thus, the eigenbasis matrix [U U_v]^T corresponding to r slowest modes is formed. Matrix U is the eigenbasis matrix of A_S as in equation (4.6) and U_v is the portion extending into the load buses. Since u_{vi} consists of real and imaginary parts u_{vri} and u_{vxi}, and for direct comparison with that of u_i, which is a state vector representing angles of voltage behind transient reactances (E_i'), u_{vi} must be expressed in V-θ coordinates so that it can be used directly with u_i vector. The bus voltage magnitude and angle are given by:

$$|V| = \sqrt{V_r^2 + V_x^2} \tag{4.9}$$

$$\theta = \arctan\left(\frac{V_x}{V_r}\right) \tag{4.10}$$

The voltage magnitude and angle deviations are calculated using:

$$\Delta|V| = \frac{\partial|V|}{\partial V_r}\Delta V_r + \frac{\partial|V|}{\partial V_x}\Delta V_x \tag{4.11}$$

$$\Delta\theta = \frac{\partial\theta}{\partial V_r}\Delta V_r + \frac{\partial\theta}{\partial V_x}\Delta V_x \tag{4.12}$$

Expressing in terms of u_{vi} , we obtain:

$$|u_{vi}| = \frac{V_r u_{vri} + V_x u_{vxi}}{|V|} \tag{4.13}$$

$$\theta_{vi} = \frac{V_x u_{vri} - V_r u_{vxi}}{|V|^2} \tag{4.14}$$

Using (4.14), matrix [U θ_v] is formed for use in the grouping algorithm. The grouping algorithm is similar to one described in the preceding section, except that in step (iv) a matrix of cosines G_C between each reference machine and all other buses is calculated instead of matrix L. If w_i is a reference row vector and w_j is a non-reference row vector of matrix [U θ_v], then G is formed as follows (Chow [1992]):

$$G_C(j,i) = \frac{w_i \, w_j^T}{|w_i| \, |w_j|}, \qquad i=1,r; j=r+1, n_m + n_b - r \tag{4.15}$$

Using the same example found in Chow [1982], the grouping matrices calculated using equation (4.15) and the equation in step (iv) of algorithm 4.1 are compared. We show that the results of grouping using the L and G_C matrices are similar. We use the 48 machine NPCC system as found in Chow [1982]. In both matrices the reference machines are given at the top of each column starting from the second column and the non-reference machines are given in first column. The largest entry in each row is underlined. Each entry in matrix G_C is the cosine of the angle between the row eigenvector of the reference machine and the row eigenvector of the corresponding non-reference machine.

It is interesting to note that using the G_C matrix, machine 33 is grouped under machine 34 whereas using the L matrix, it is grouped under machine 36. Similarly for machine 42, using the G_C matrix, it is grouped under machine 41 and using the L matrix, it is grouped under machine 39.

As stated in Chow [1982], for these machines, where the two largest entries in its row are of the same magnitude, the machines could be grouped under either one of the two reference machines.

REF.	34	36	41	17	29	5	39
10	0.2336	0.8770	-0.0962	0.2095	0.2505	-0.0499	-0.0429
12	-0.1965	0.8794	-0.0870	0.2236	0.2589	-0.0382	-0.0458
13	-0.0798	0.1994	-0.2990	0.8076	0.3995	0.0337	-0.0739
14	-0.0604	0.2231	-0.2729	0.7640	0.4426	0.0856	-0.0910
15	-0.0210	0.0721	-0.3218	0.9716	0.0830	-0.0859	-0.0105
16	0.1135	0.1130	-0.2553	0.9829	-0.0156	-0.1251	0.0211
4	-0.0497	-0.1065	0.0866	-0.1066	0.1188	0.9903	-0.1688
18	0.0811	0.0876	-0.2735	0.9872	-0.0107	-0.1200	0.0152
19	0.1298	0.1396	-0.2355	0.9727	0.0154	-0.1095	0.0174
20	0.1069	0.1147	-0.2554	0.9805	0.0028	-0.1156	0.0132
21	0.0763	0.0982	-0.2731	0.9726	0.0399	-0.0857	-0.0275
22	0.0946	0.1147	-0.2586	0.9710	0.0408	-0.0875	-0.0177
23	0.0012	0.1065	-0.3007	0.9396	0.1760	-0.0362	-0.0403
24	-0.0247	0.1284	-0.2542	0.7387	0.4577	0.1465	-0.1543
25	-0.0753	0.1387	-0.2564	0.6591	0.5494	0.1852	-0.1639
26	-0.0931	0.1560	-0.2877	0.7197	0.5211	0.0926	-0.1219
27	-0.0526	-0.0259	-0.0853	-0.0676	0.9844	0.1221	-0.1778
28	-0.0431	-0.0337	-0.0805	-0.1131	0.9992	0.0163	-0.1455
6	-0.0545	-0.1066	0.0863	-0.1155	0.1305	0.9888	-0.1684
30	-0.0462	-0.0323	-0.0822	-0.1029	0.9975	0.0406	-0.1530
31	0.4435	0.3937	0.0330	0.8042	-0.0568	-0.1390	0.1066
32	0.4798	0.3008	0.7324	0.1725	-0.1876	-0.0556	0.4997
33	0.7133	0.6333	0.3220	0.3080	-0.0956	-0.1171	0.0982
1	-0.0753	-0.0825	0.0221	-0.0981	0.5294	0.8356	-0.1970
35	0.9983	0.1015	0.2045	0.0659	-0.0468	-0.0454	0.0264
2	-0.0729	-0.0864	0.0340	-0.1008	0.4608	0.8766	-0.1942
37	0.3459	0.2200	0.8178	-0.0068	-0.1615	-0.0172	0.5662
38	0.3004	0.1919	0.8100	-0.0496	-0.1644	-0.0215	0.6174
7	-0.0549	-0.1066	0.0855	-0.1154	0.1361	0.9880	-0.1696
40	0.3081	0.1866	0.9159	-0.1251	-0.1605	0.0214	0.4436
3	-0.0524	-0.1078	0.0852	-0.1102	0.1305	0.9887	-0.1696
42	0.2439	0.1446	0.7629	-0.0874	-0.1648	-0.0335	0.7042
43	-0.0395	-0.0107	-0.0449	0.0513	0.5510	0.2997	-0.3355
8	-0.0635	-0.1001	0.0797	-0.0809	0.0684	0.9944	-0.1625
48	-0.0181	0.0397	-0.1562	0.4666	0.3572	0.2397	-0.2938
46	-0.0383	-0.0166	-0.0182	0.0686	0.3274	0.3277	-0.3334
7	0.0883	0.1370	-0.2186	0.8392	0.2002	0.0491	-0.1445
9	-0.0832	-0.0523	0.0159	-0.0104	0.3734	0.9052	-0.1939

Text box 4.1: Matrix G_c

REF.	34	36	41	17	29	5	39
10	-0.0340	0.1277	-0.0090	0.2849	0.4215	0.1810	-0.0017
11	-0.1614	0.6732	-0.0418	0.2334	0.2804	0.0774	-0.0054
12	-0.1299	0.6275	-0.0338	0.2317	0.2706	0.0821	-0.0048
13	-0.0384	0.1228	-0.0114	0.5286	0.3152	0.0979	-0.0033
14	-0.0277	0.1270	-0.0070	0.4732	0.3129	0.1129	-0.0019
15	-0.0263	0.0624	-0.0097	0.7805	0.1689	0.0389	-0.0041
16	0.0365	0.0751	0.0169	0.7621	0.0947	0.0131	0.0046
4	-0.0071	0.0134	-0.0018	0.0342	0.1331	0.8431	-0.0003
18	0.0219	0.0624	0.0107	0.7855	0.0975	0.0136	0.0029
19	0.0383	0.0801	0.0233	0.6717	0.1014	0.0176	0.0127
20	0.0314	0.0723	0.0168	0.7252	0.0990	0.0143	0.0060
21	0.0146	0.0513	0.0052	0.6181	0.0866	0.0098	0.0003
22	0.0218	0.0605	0.0118	0.6228	0.0934	0.0136	0.0040
23	-0.0112	0.0719	-0.0024	0.6324	0.1914	0.0534	-0.0013
24	-0.0099	0.0676	-0.0023	0.3763	0.2357	0.0828	-0.0008
25	-0.0243	0.0806	-0.0081	0.3662	0.2965	0.1093	-0.0024
26	-0.0346	0.0934	-0.0113	0.4292	0.3259	0.0947	-0.0032
27	-0.0055	0.0148	-0.0018	0.0462	0.7278	0.0773	-0.0005
28	-0.0023	0.0039	-0.0009	0.0131	0.9255	0.0217	-0.0002
6	-0.0100	0.0140	-0.0025	0.0282	0.1487	0.8614	-0.0004
30	-0.0036	0.0067	-0.0014	0.0221	0.8749	0.0369	-0.0004
31	0.1323	0.1627	0.0906	0.4632	0.0543	0.0070	0.0537
32	0.0885	0.0694	0.2985	0.1497	-0.0099	0.0008	0.2554
33	0.2793	0.3155	0.1523	0.2412	0.0143	0.0008	0.0411
1	-0.0142	0.0250	-0.0034	0.0604	0.4205	0.5709	-0.0005
35	0.8717	0.0512	0.0099	0.0422	-0.0015	-0.0003	0.0122
2	-0.0139	0.0245	-0.0033	0.0571	0.3794	0.6140	-0.0005
37	0.0497	0.0391	0.3564	0.0804	-0.0096	0.0006	0.3324
38	0.0405	0.0343	0.3704	0.0677	-0.0092	0.0006	0.3866
7	-0.0099	0.0139	-0.0025	0.0282	0.1511	0.8462	-0.0004
40	0.0464	0.0391	0.5096	0.0705	-0.0158	0.0013	0.2932
3	-0.0084	0.0128	-0.0022	0.0323	0.1450	0.8440	-0.0004
42	0.0291	0.0212	0.3607	0.0468	-0.0076	0.0005	0.4760
43	-0.0011	0.0024	0.0014	0.0118	0.1305	0.0094	-0.0006
8	-0.0178	0.0193	-0.0044	0.0551	0.0903	0.8757	-0.0006
48	-0.0024	0.0167	-0.0029	0.1686	0.0649	0.0134	-0.0019
46	0.0000	0.0010	0.0058	0.0099	0.0115	0.0010	0.0006
47	0.0105	0.0369	0.0047	0.3261	0.0748	0.0154	0.0018
9	-0.0203	0.0449	-0.0059	0.1117	0.2948	0.5962	-0.0013

Text box 4.2: Matrix L

Algorithm 4.1 is modified accordingly and given in algorithm 4.2.

Algorithm 4.2: Slow Coherency Grouping - all buses

(i) Choose the number of areas, r.
(ii) Compute augmented eigenbasis matrix $[\ U\ U_v\ ]^T$
(iii) Apply Gaussian elimination with complete pivoting on U to obtain r reference machines.
(iv) Use $[\ U\ U_v\ ]^T$ to form $[\ U\ U_0\ ]^T$
(v) Compute the closeness between reference row vectors and non-reference row vectors using:

$$G_C(j.i) = \frac{w_i\ w_j^T}{|w_i|\ |w_j|}, \qquad i=1,r;j=r+1,n_m+n_b-r$$

(vi) Assign non-reference generators and load buses to reference generators.

The physical interpretation of the proposed technique is as follows. Each row of the eigenvector matrix U_v describes how the r slowest modes are distributed in a particular bus voltage. After finding the reference generator for each area, we calculate the closeness of the reference row eigenvector to each of the load bus row eigenvectors. Throughout the network, any large angle separation between any two load bus row eigenvectors, each with respect to their reference eigenvectors, will constitute a weak link and divide the two coherent areas.

There are two alternatives for calculating the eigenbasis matrix. Equation (4.4) can be used directly, in which case a sparsity based eigencomputation method is required such as the one described by Kundur, et. al. [1990]. By using equation (4.5) any eigencomputation package based on the QR method can be used.

4.5 PROGRAM IMPLEMENTATIONS

The algorithms described in the preceding section are implemented at two locations. The first implementation is at Ontario Hydro where PEALS (Program for Eigenanalysis of Large Systems) supporting procedures are used.

The second one is at the PRL (Power Research Laboratory, McMaster University) implemented on the Silicon Graphics IRIS Workstation under DCPS. In the campus implementation, all routines are developed at the PRL.

4.5.1 Implementation at Ontario Hydro

The formulation of system state matrix A_S (equation (4.5)) utilizes the efficient network solution technique of PEALS (Rogers and Kundur [1989] and Kundur, et. al. [1990]). The network and machine data structures are the same as those used in PEALS. For grouping purposes in the slow coherency method, the program will automatically use classical machine representation although data may be given in detailed form. In the actual implementation, the elements of the characteristic matrix A_S are directly calculated using:

$$A_{ij}=\text{Imag}\{ I_{Bi}^{*}V_i^{j} \}\frac{1}{M_i} \text{ , for } i \neq j$$

$$A_{ii}=A_{ij} + [\text{Imag}\{ E_i'I_{Bi}^{*} \} - \frac{|E_i'|^2}{x_{di}'}\frac{1}{M_i}] \text{ , for } i = j \qquad (4.16)$$

where vector V^j is obtained from the solution of:

$$e_j I_{Bj} = YV^j$$

The vector e_j is a zero vector except the j'th component, which is unity and:

$$I_{Bj} = \frac{E_j'}{x_{di}'}$$

The full eigensolution of A_S is obtained using the QR method. To form the eigenvectors that correspond to the network part, u_{vi}, only r eigenvectors (u_i) associated with the r smallest eigenvalues are used. Forward and backward substitution is performed to obtain u_{vi} in:

$$J_D\, u_{vi} = -J_C\, u_i \qquad (4.17)$$

The r smallest eigenvalues and their corresponding eigenvectors (u_i) can

also be obtained using the implementation of the Modified Arnoldi Method in PEALS (Kundur, et. al. 1990). In this case, the sparse augmented system matrix in equation (4.4) is used directly.

4.5.2 Implementation at the Power Research Laboratory

The implementation at the PRL is best described by referring to the DCPS command file shown in text box 4.3.

```
% Sample Run-Stream to do eigenanalysis for coherency
% grouping
redn,snew.sol,dcps
redm,snew.mac
orde;
ybus;
bpow,0
inic;
cinj;
fybe;
forj;
fstl;
eise;
sor4;
trve;
cohg,4
END
```

Text box 4.3: DCPS-CG command file for coherency grouping

After the load flow and machine data files are read using `redn` and `redm` respectively, the network buses are ordered using `orde`. Bus powers are calculated using `bpow` after the system admittance matrix is constructed using `ybus`. Initial conditions for all classical generators are calculated using function `inic`. The equivalent load admittances are appended to the admittance matrix using `cinj`. In the formulation, real expanded admittance matrix is used. This real matrix is constructed from the complex admittance matrix by using function `fybe`. In DCPS the Jacobian is formed using the following derivations. The swing equation for machine i - neglecting damping is:

$$M_i \ddot{\delta}_i = P_{mi} - P_{ei} \tag{4.18}$$

where P_{mi}, the mechanical power input is assumed constant and the electrical power is given by:

$$P_{ei} = \frac{1}{x'_{di}}(V_{ri}|E_i|\sin\delta_i - V_{x_i}|E_i|\cos\delta_i) \tag{4.19}$$

where $E_i\angle\delta_i$ is the voltage behind subtransient reactance and $V_{ri}+jV_{xi}$ is the machine terminal voltage and x'_{di} is the transient reactance. Substituting equation (4.19) into (4.18), we obtain:

$$M_i \ddot{\delta}_i = P_{mi} - \frac{V_{ri}}{x'_{di}}|E_i|\sin\delta_i + \frac{V_{xi}}{x'_{di}}|E_i|\cos\delta_i = f_i(\delta_i, V_{ri}, V_{xi}) \tag{4.20}$$

Linearizing equation (4.20) about an operating point and writing the equation in a matrix form:

$$M_i \Delta\ddot{\delta}_i = \begin{bmatrix} -(\frac{V_{ri}}{x'_{di}}|E_i|\sin\delta_i + \frac{V_{xi}}{x'_{di}}|E_i|\cos\delta_i) & \frac{|E_i|}{x_{di}}\sin\delta_i & \frac{|E_i|}{x_{di}}\cos\delta_i \end{bmatrix} \begin{bmatrix} \Delta\delta_i \\ \Delta V_{ri} \\ \Delta V_{xi} \end{bmatrix} \tag{4.21}$$

or:

$$\Delta\ddot{\delta} = M^{-1}J_A\Delta\delta + M^{-1}J_B\Delta V \tag{4.22}$$

The real and imaginary components of the current injection at bus i are:

$$I_{ri} = \frac{1}{x'_{di}}(|E_i|\sin\delta_i - V_{xi}) = g_r(\delta_i, V_{ri}, V_{xi}) \tag{4.23}$$

$$I_{ri} = \frac{1}{x'_{di}}(-|E_i|\cos\delta_i + V_{ri}) = g_x(\delta_i, V_{ri}, V_{xi}) \tag{4.24}$$

Linearizing equations (4.23) and (4.24) about an operating point, we obtain:

$$\begin{bmatrix}\Delta I_{ri}\\ \Delta I_{xi}\end{bmatrix} = \begin{bmatrix}\frac{1}{x'_{di}}(|E_i|\cos\delta_i & 0 & -\frac{1}{x'_{di}}\\ \frac{1}{x'_{di}}(|E_i|\sin\delta_i & \frac{1}{x'_{di}} & 0\end{bmatrix}\begin{bmatrix}\Delta\delta_i\\ \Delta V_{ri}\\ \Delta V_{xi}\end{bmatrix} \tag{4.25}$$

or:

$$\Delta I_B = \hat{J}_C \Delta\delta + \hat{J}_D \Delta V \tag{4.26}$$

The network equation in expanded form is:

$$\begin{bmatrix}I_r\\ I_x\end{bmatrix} = \begin{bmatrix}G_N & -B_N\\ B_N & G_N\end{bmatrix}\begin{bmatrix}V_r\\ V_x\end{bmatrix} \tag{4.27}$$

Linearizing equation (4.27) about an operating point results in equation:

$$\begin{bmatrix}\Delta I_r\\ \Delta I_x\end{bmatrix} = \begin{bmatrix}G_N & -B_N\\ B_N & G_N\end{bmatrix}\begin{bmatrix}\Delta V_r\\ \Delta V_x\end{bmatrix} \tag{4.28}$$

or:

$$\Delta I = Y\Delta V \tag{4.29}$$

Using equation (4.28), we eliminate term ΔI_B from equation (4.26) we obtain:

$$(Y - \hat{J}_D)\Delta V - \hat{J}_C \Delta\delta = 0 \tag{4.30}$$

Equations (4.22) and (4.30) can now be written in compact form to obtain the ASSE:

$$\begin{bmatrix}\Delta\tilde{\delta}\\ 0\end{bmatrix} = \begin{bmatrix}J_A & J_B\\ J_C & J_D\end{bmatrix}\begin{bmatrix}\Delta\delta\\ \Delta V\end{bmatrix} \tag{4.31}$$

where $J_C = \hat{J}_C$, $J_D = Y - \hat{J}_D$, $J_A = M^{-1}J_A$ and $J_B = M^{-1}J_B$.

To form the Jacobian matrix as in equation (4.31), function `forj` is used. In the DCPS implementation, state matrix A_s as in equation (4.5) is formed directly using function `fstl`. After the state matrix formed, eigenanalysis can then be performed. The EISPACK routines as found in Smith, et. al. [1976] and Garbow, et. al. [1977] are used for eigenanalysis. Function `eise` determines all eigenvalues and eigenvectors. These eigenvalues and eigenvectors are sorted to ascending order using function `sor4`. To convert elements of the augmented eigenvectors to V-θ coordinate function `trve` is used. Finally function `cohg` with a command parameter - the desired number of areas - is used to perform the grouping according to algorithm 4.2.

4.6 RESULTS OF SIMULATIONS

4.6.1 Test Systems

The two test systems used were the 39-bus NPCC system and the 118-bus IEEE system.

4.6.1.1 39-bus NPCC System

The new grouping method was applied to a 10-machine model of the NPCC (New England) system. For the purpose of illustration, we choose to partition the network into 6 and 3 areas and the results are shown in figures 4.2 and 4.3 respectively . The coherent areas determined by the new method are similar to the one found in Chow, et. al. [1989] where the *branching algorithm* was used. The result is also similar to the one found in Crow, et. al. [1990] where the *branching algorithm* was used that takes into account the line impedances. A sample output of the program for the partitions in figure 4.2 is given in table 4.1. The output of the program provides the number of buses in each coherent area, a list of buses in each area, and all tie-lines between areas. In the DCPS implementation, the `iarea` fields in the load flow data (see Appendix C) can be replaced by the new area numbers as determined by coherent grouping - using function `svar`.

SLOW-COHERENT PARTITIONING OF POWER NETWORK

AREA NO.	REF. MACHINE NO.	NO. OF BUSES IN AREA
1	38	10
2	34	25
3	39	4

BUSES IN AREA 1

30 37 38 2 3 25 26 27 28 29

BUSES IN AREA 2

31 32 33 34 35 36 4 5 6 7 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24

BUSES IN AREA 3

39 1 8 9

LIST OF TIE-LINES BETWEEN AREAS

FROM-BUS/AREA	TO-BUS/AREA
2/1	1/3
4/2	3/1
18/2	3/1
8/3	5/2
8/3	7/2
27/1	17/2

Table 4.1: Sample program output

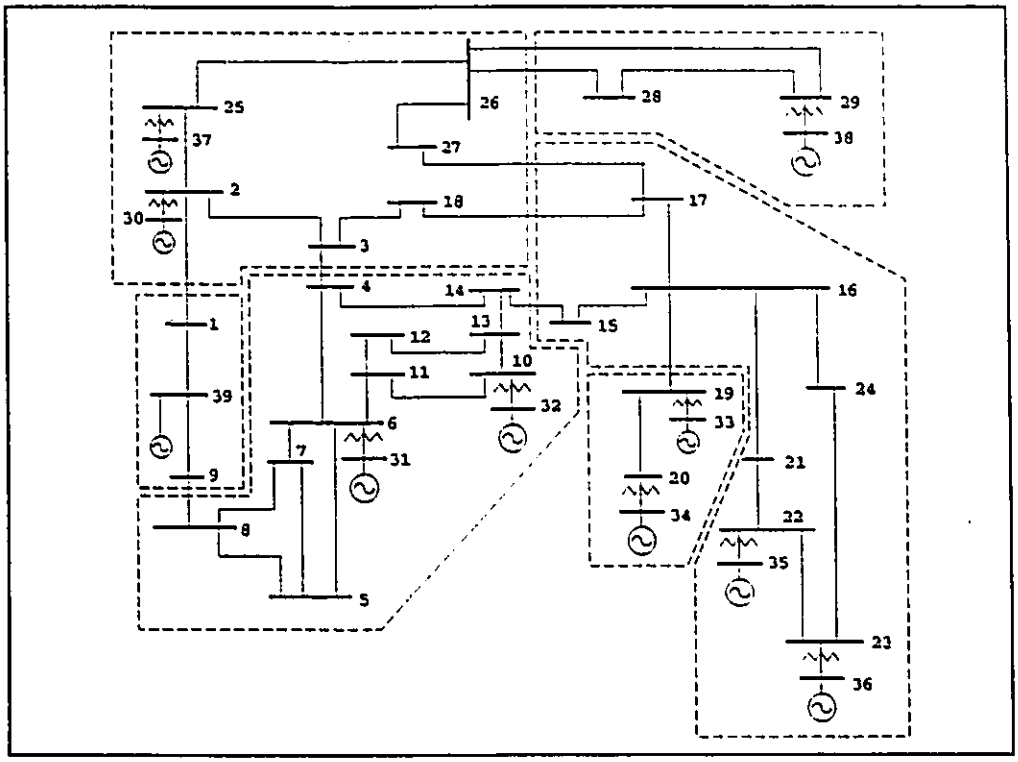

Figure 4.2: 39-bus system, 6-area partitioning

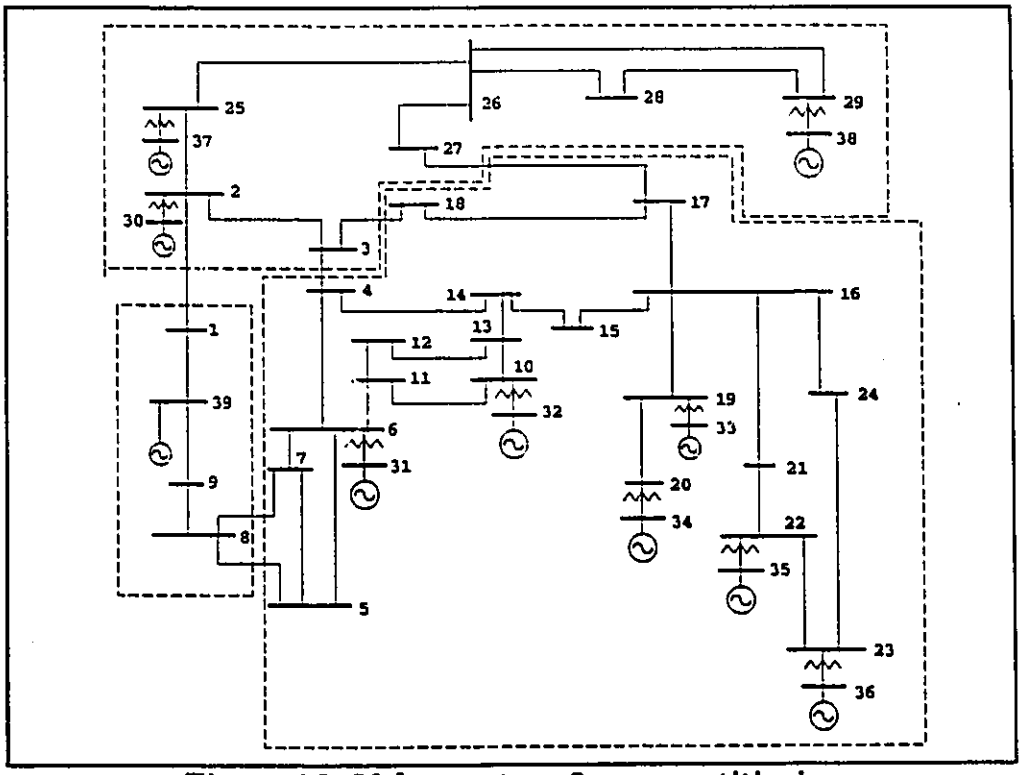

Figure 4.3: 39-bus system, 3-area partitioning

4.6.1.2 118-bus IEEE Test System

The result of the grouping as applied to the 118-bus IEEE test system is shown in figure 4.4. For this system we choose to partition the network into six coherent areas. We compare the partitioning obtained using the proposed method to that of the branching algorithm in Crow, et. al. [1990]. Although the dynamic data used are different, the results of both techniques divided the network generally into the same coherent areas. The partitioning described in Crow, et. al. [1990] was used with the waveform relaxation method for transient stability simulation where substantial improvement in speed was achieved. The partition obtained is also similar to the one obtained by Zaborszky, et. al. [1985] using the textured model approach.

4.6.2 Practical Systems

Two practical systems were used for the simulations: the 348-bus TNB/PUB System and the 2367-bus B.C. Hydro System.

4.6.2.1 348-bus, 77-machine TNB/PUB System

The general layout of the power system is shown in figure 4.5. It is an interconnected power system between the TNB (Tenaga Nasional Berhad) of Malaysia and the PUB (Public Utility Board) of Singapore. The system is characterized by long 275 kV transmission lines stretching from north to south, a north-east-central ring and central-south-east ring. Large generations are located in the east and the central areas and the PUB. Major loads are located on the west coasts of the Northern, Central and Southern areas and the PUB. The geographical areas are indicated by the dashed lines.

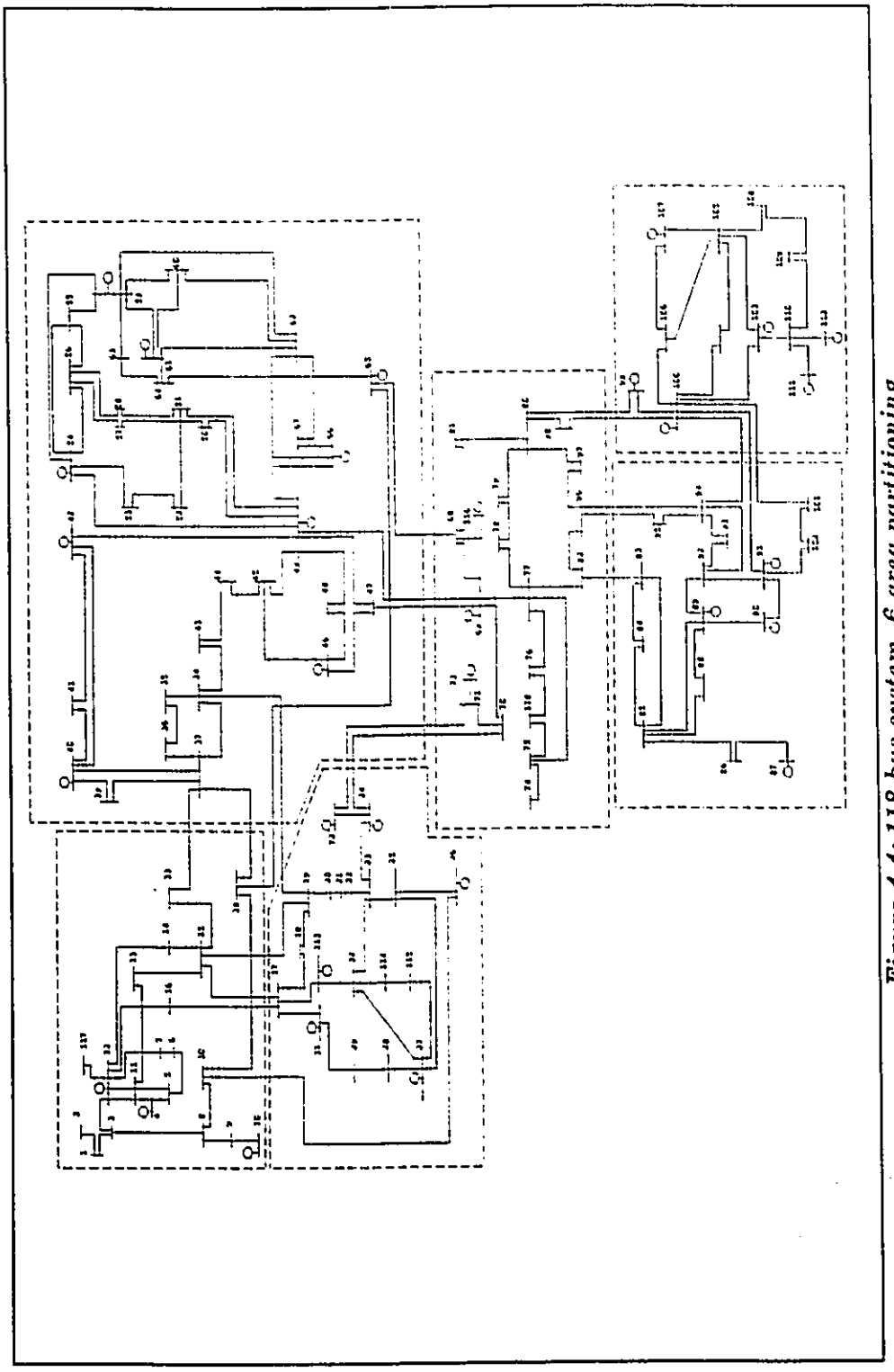

Figure 4.4: 118-bus system, 6-area partitioning

The data used for the simulation is that projected for the peak load in 1995. The result of the simulation clearly showed four coherent areas indicated by the dotted lines. The generators in the Northern and the Central areas belong to the same coherent group. Although the southern tip of Peninsular Malaysia is less than 10 km from Singapore, the Southern and the PUB systems belong to two different coherent groups. This indicates a weak interconnection between the two systems. Using this result we can identify tie-lines critical to interarea oscillations. The four slowest modes are, 0.755, 1.100, 1.200 and 1.284 Hz. It is interesting to note that the lowest frequency of oscillation based on the simulation of the current system is about 0.2 Hz. The planned 275 kV reinforcement, KAWA-NYPG (1994/5) will result in a more tightly coupled system in the future thus increasing the mode frequencies to the values calculated in this study. This partitioning technique was used to study the TNB/PUB system structure with respect to interarea oscillation as described in Yusof, et. al. [1993a]. Lie, et. al. [1993] pointed out the importance of identifying system weak links under various system condition and uses the degree of the weakness of the flow Jacobian. However, the method also makes use of separate coherency technique to identify coherent generators.

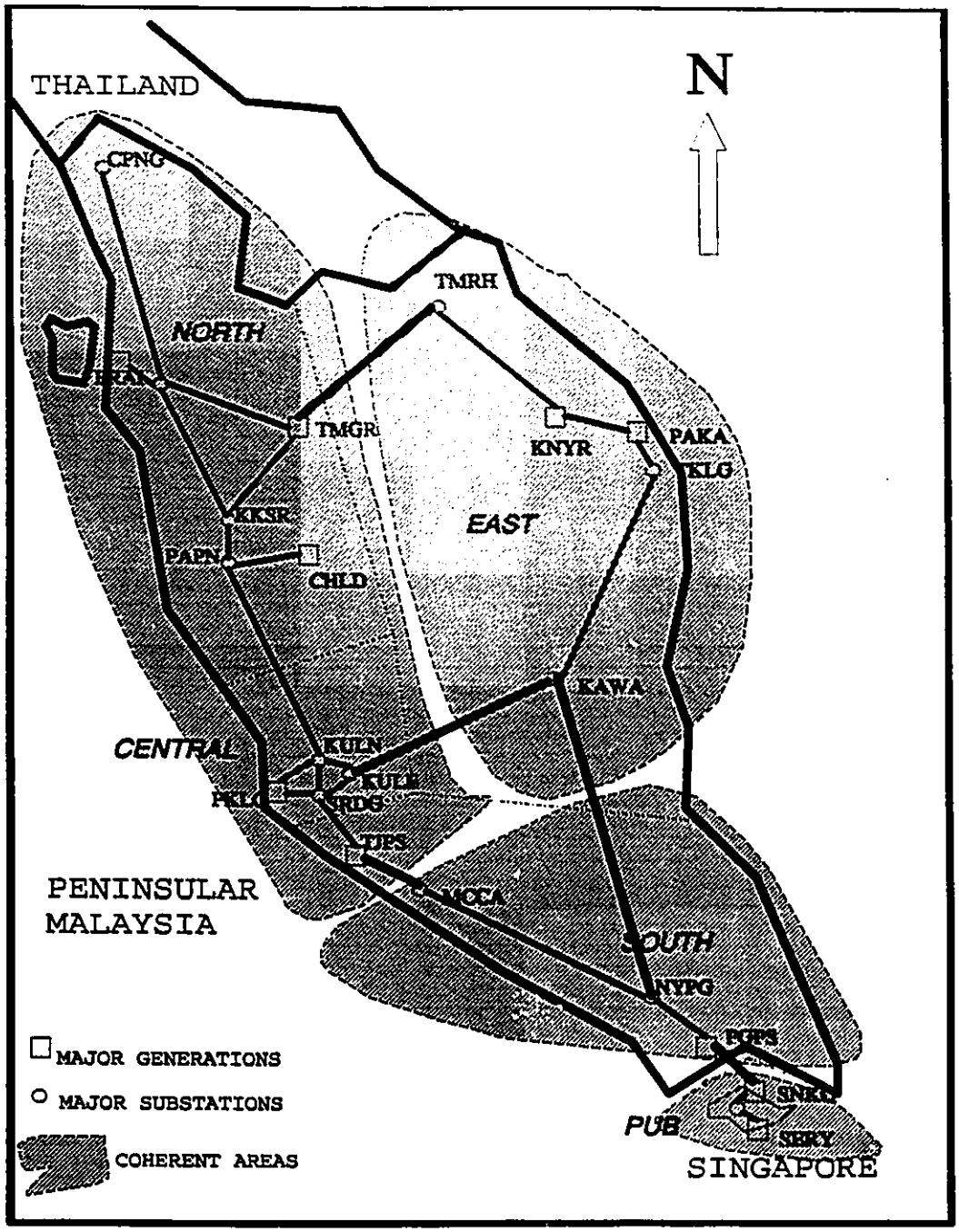

Figure 4.5:. TNB/PUB system, 4-area partitioning

4.6.2.2 2367-bus, 375-machine B.C. Hydro System

The second system that was simulated is the B.C. Hydro System (data available at Ontario Hydro) including some of its neighbouring utilities. Our method clearly partitioned the system into 13 coherent areas as shown in figure 4.6 by the dotted lines. We compare our result with the one found in Chow, et. al. [1984] and it shows that the coherent areas obtained are similar although the data used in that reference was that of 1984. It is interesting to note that the Kemano and Geysers generators are reference machines to two different coherent each having only few buses (8 and 6 respectively) in each area. We observed that the Big Creek and Helms stations are geographically close, but both are reference machines to two different coherent areas. The result also indicates that most parts of the Arizona and New Mexico systems belong the same coherent group. The slowest modes calculated are, 0.343, 0.430, 0.542, 0.728, 0.743, 0.767, 0.816, 0.835, 0.907 and 0.943 Hz. It is interesting to note that the slow modes used for the partitioning are also in the range as found in Mansour [1989], where full nonlinear models were used. The total CPU time required by the program on VAX 8650 computer - at Ontario Hydro - for the B.C. Hydro system simulation was 16.5 minutes.

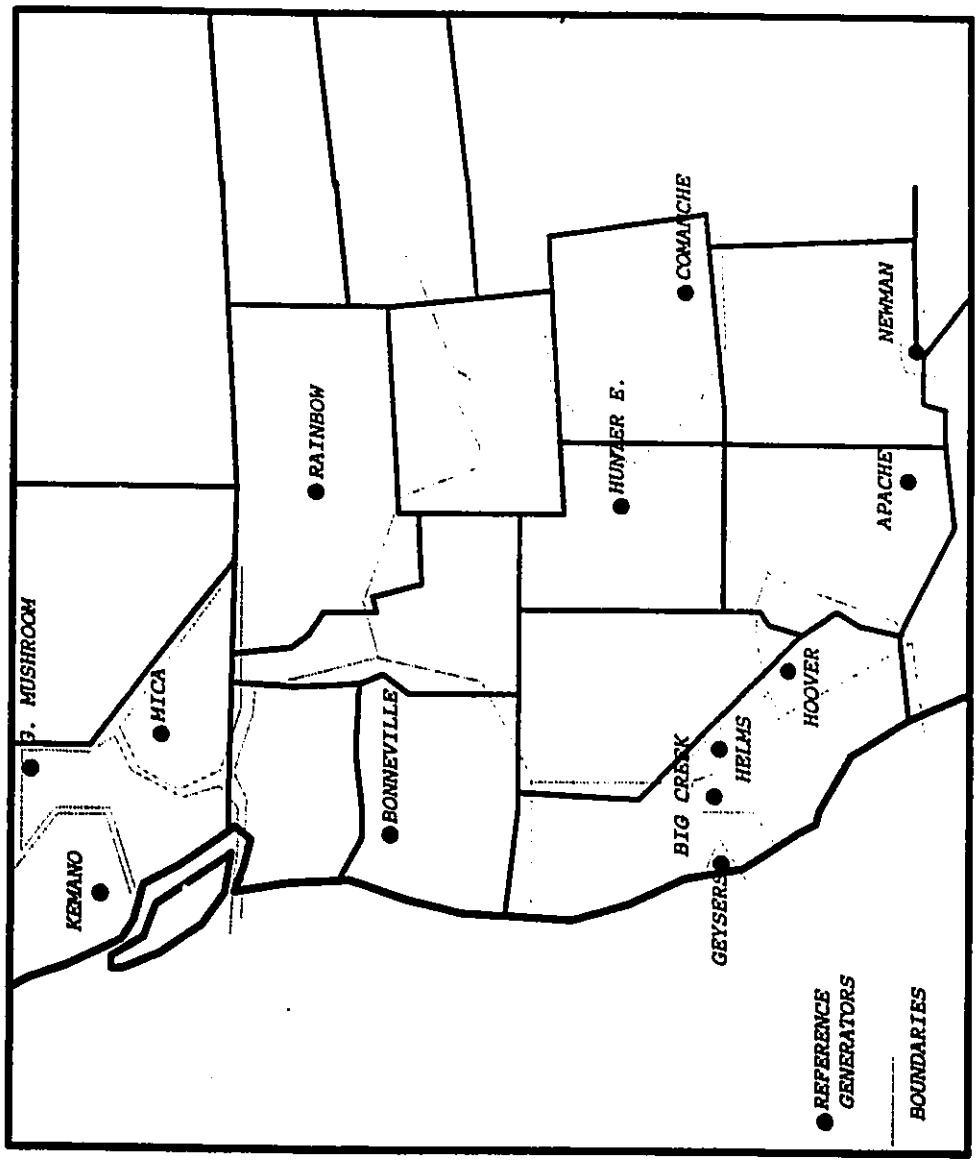

Figure 4.6: B.C. Hydro system partitioning showing locations of reference generators

4.7 CONCLUSIONS

In this chapter, we have presented a method of network partitioning into slow-coherent areas which automatically includes the load buses in the grouping process. In our approach, we used the slow eigenvectors and included the network buses. We extended the physical meaning of the eigenvectors to give mode shapes in the system network buses. The method we used is based on the pertinent physical characteristics of the system and does not require modification of the network model in order to include the load buses in the grouping process. Furthermore, the method is simple to implement. We have also demonstrated in this paper that the generalized eigenvectors are useful parameters to investigate.

The primary objective of the slow coherency based partitioning proposed in this paper is for use in network reduction. In performing the network reduction, one first specifies the coherent areas. Then the boundary buses are identified, all other buses are regarded as internal to the respective areas. Slow coherency based reduction has also been used by McFarlane and Alden [1993][1] in an Artificial Neural Network power system stability assessment. Our proposed method handles both tasks - coherent area and boundary buses identifications - simultaneously. We have demonstrated from the simulations of the two practical systems that the proposed method is viable. We have been able to identify the weak links between areas which are consistent with those of operational experience.

---

[1] This work used the Artificial Neural Network (ANN) approach for transient stability assessment. Due to large number of input variables required to train the ANN even on a moderate size system (100 - 200 buses), the system is first divided into several coherent areas - using the DCPS program. Then each area is reduced to an equivalent aggregated-to-one machine from which the training features are extracted.

CHAPTER 5
USING A CLUSTER OF WORKSTATIONS FOR PARALLEL, DISTRIBUTED COMPUTING

5.1 INTRODUCTION

As discussed in chapter 3, two approaches to implement parallel programs are being used - the first uses state-of-the-art parallel processors and the other uses clusters of workstations. Although many centralized multiprocessor computers are commercially available, being expensive, they are confined to mainly research institutions and universities. However, in the power industry, workstations connected to local area networks (LAN) are widely available. These networked workstations will become more pervasive in the future since many small utilities that are now using mainframe based systems are downsizing to LAN connected computers. Such a system can be viewed as a distributed memory multicomputer system where each computer has its own processor and memory. The computers communicate with each other by passing messages through the network.

A workstation connected to a local area network is normally used for local processing. It sometimes communicates with other workstations for the purposes of sharing data through databases. The processing power of workstations is often under utilized. With the ability to communicate with each other, these networked workstations can be readily exploited for parallel, distributed processing (Bloomer [1991]). Since networked workstations are widely available, there is little additional cost involved in harnessing their computational power. The main task is to select or develop suitable algorithms for solving existing and new problems in an environment in which

the most critical issue is the relatively slow interprocessor communication speed. Using the available hardware for parallel, distributed processing provides excellent opportunities to train and familiarize engineers and programmers with the concepts and problems associated with parallel, distributed programming.

In this chapter, the characteristics of numerical algorithms that are suitable to be solved on a cluster of workstations are discussed. As discussed in chapter 3, the need to solve sparse linear algebraic equations is pervasive in power system analysis. Therefore, in applying parallel, distributed processing, this solution must first be attempted before expanding the technique to other problems. Both block iterative and direct methods for solution of linear systems are described. Iterative and direct algorithms based on the NBDF (Near Block Diagonal Form) and the BBDF (Bordered Block Diagonal Forms) are described. The practical implementation of these algorithms for parallel, distributed computation using the RPC technique is elaborated. The results and the corresponding speedup are analyzed.

5.2 PARALLEL, DISTRIBUTED PROCESSING

A problem that is good candidate for parallel, distributed processing must first be decomposable. The objective of the decomposition is to divide the problem into smaller tasks. Having decomposed the problem, the interactions between these smaller tasks must be minimized. In this chapter, the dynamic behaviour of the power system is used to simultaneously partition the network into several areas that are consequently decoupled along the weak links which ensures minimum interactions - as described in chapter 4.

The term parallel, distributed computation implies the use of the MIMD (Multiple Instruction Multiple Data) model rather than the SIMD (Single Instruction Multiple Data) model (Flynn [1966]). With SIMD, all processors execute the same instructions in lockstep but on different data. In MIMD machines, all processors may be pursuing the same goal or solving a single problem or they may be solving many problems at the same time. In MIMD

machines, each processor has its own memory and can be executing different programs.

A cluster of workstations possesses the characteristics of a MIMD machine. In this chapter, for solving a linear system, the same computations are performed on different computers but on different sets of data. This class of problem is more precisely termed as SPMD (Single Program Multiple Data) - an extension of Flynn's taxonomy (Wilson [1993]). The different sets of data are as a result of the decomposition. Initially, the partitions are distributed to several computers. The computation then starts concurrently. Following this, the subsequent needs for communication between computers depends on the tie-variables between the partitions and whether the algorithm is of synchronous or asynchronous.

Workstations connected to a network are loosely coupled and there is no central control. But the workstations can communicate with each other via the network. The computers are distributed, do not share memory and communicate only by means of message passing (Singhal and Casavant [1991]). Unlike centralized parallel computers in which the processors can communicate at high speed either through shared memory or through high speed buses, communication between networked workstations is relatively slow. Therefore, problems which require intensive communications between processors are unsuitable to be solved in a network system. A problem in which the local computation is much more intensive than the need to communicate with other processors is therefore most appropriate to be solved using a cluster of workstations.

5.3 BLOCK METHOD FOR SOLVING LINEAR SYSTEMS

The linear systems of interest here are those normally encountered in power system analysis. The associated matrices and vectors are highly sparse. Since sparsity based algorithms are very efficient, the sparsity must somehow be preserved in parallel, distributed algorithms. A linear system is given by:

$$Ax = b \tag{5.1}$$

In power system analysis, matrix A is normally associated with the admittance matrix which represent the network connections. Using the method described in Yusof, et. al. [1992], the power network is partitioned into several coherent areas. By suitably ordering the network according to the areas, the resulting matrix A becomes blocked diagonal with highly sparse off-diagonals representing tie-buses between the areas. Matrix A can then be decomposed into two submatrices and is written as:

$$A = A_D + A_O \tag{5.2}$$

where A_D comprises block diagonal matrices and A_O contains the off-diagonal elements. There are two commonly used ordering techniques which determine how the off-diagonals are arranged (Hatcher, et. al. [1977]). In the NBDF (Near Blocked Diagonal Form), the off-diagonal elements are scattered throughout the off-diagonal part. In BBDF (Blocked Bordered Diagonal Form), the elements associated with the tie-buses are arranged to occupy the strips on the right and bottom of the matrices.

The NBDF matrix is conveniently solved using a block iterative scheme. Given a guess solution vector, **x**, the iterative scheme for block i is given by:

$$x_i^{k+1} = A_{Di}^{-1} (b_i - A_{Oi} x_i^k) \tag{5.3}$$

where k denotes the iteration count. The Gauss-Seidel iterative scheme is the preferred algorithm for solving equation (5.3) on a uniprocessor system. The reason is that, recent updates of any element of **x** can be immediately used by other blocks. However, on distributed memory computers, Gauss-Jacobi iterative scheme would be more suitable in order to avoid continuous interprocessor communications. However, it may not be necessary to strictly adhere to the Gauss-Jacobi scheme. Matrix A is often decomposed into unequal sized blocks because the partitioning algorithm is based on the

dynamic behaviour of the physical system. A processor having small size block can update x_i and proceeds with the next iteration without waiting for those larger size blocks to complete their local computations. This asynchronous update method is likely to provide better convergence than strictly updating the solution vector when all blocks complete their computations (Bertsekas and Tsitkilis [1989]).

Using the BBDF Ordering, the linear system can be solved directly. Equation (5.1) can be written as (Hatcher, et. al. [1977], Decker, et. al. [1991]):

$$\begin{bmatrix} A_1 & . & . & . & . & A_{O1} \\ . & A_2 & . & . & . & A_{O2} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & A_m & A_{Om} \\ A_{O1}^T & A_{O2}^T & . & . & A_{Om}^T & A_{OD} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_m \\ x_O \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ . \\ . \\ B_m \\ B_O \end{bmatrix} \tag{5.4}$$

Equation (5.4) can then be solved in two stages. Firstly, x_O is obtained by solving:

$$\hat{A}_{OD} x_O = \hat{B}_O \tag{5.5}$$

where:

$$\hat{A}_{OD} = A_{OD} - \sum_{i=1}^{m} A_{Oi}^T A_i^{-1} A_{Oi} \tag{5.6}$$

$$\hat{B}_O = B_O - \sum_{i=1}^{m} A_{Oi}^T A_i^{-1} B_i \tag{5.7}$$

In the parallel, distributed computation scheme, the LU factor of each block is calculated in its own processor. The right hand parts of the terms in equations (5.6) and (5.7) can also be computed by the individual processor. The whole of equations (5.6) and (5.7) can be evaluated by the same processor that solves equation (5.5). The remaining parts of the solution vector $x_1, x_2, \ldots, x_m$ are obtained using:

$$A_1 x_1 = B_1 - A_{01} x_0 \tag{5.8}$$

5.4 REMOTE PROCEDURE CALL (RPC) CLIENT/MULTIPLE-SERVER MODEL

Before developing the algorithms for solving equations (5.3), (5.5), (5.6) and (5.7), it is useful to describe the parallel, distributed computing model. In this thesis, the RPC technique is used to communicate data and instructions between workstations. Although the idea of remote procedure calls was discussed as early as the middle of 70's, not until 1984 when Birrel and Nelson [1984] described its implementation that the concept became practical and available to programmers. Although there are two main methods for moving data between computers in client-server computing - the RPC and message-passing - the RPC technique is likely to dominate future client-server computing because this technique runs on a variety of computers and has high data integrity (Korzeniowski [1993]).

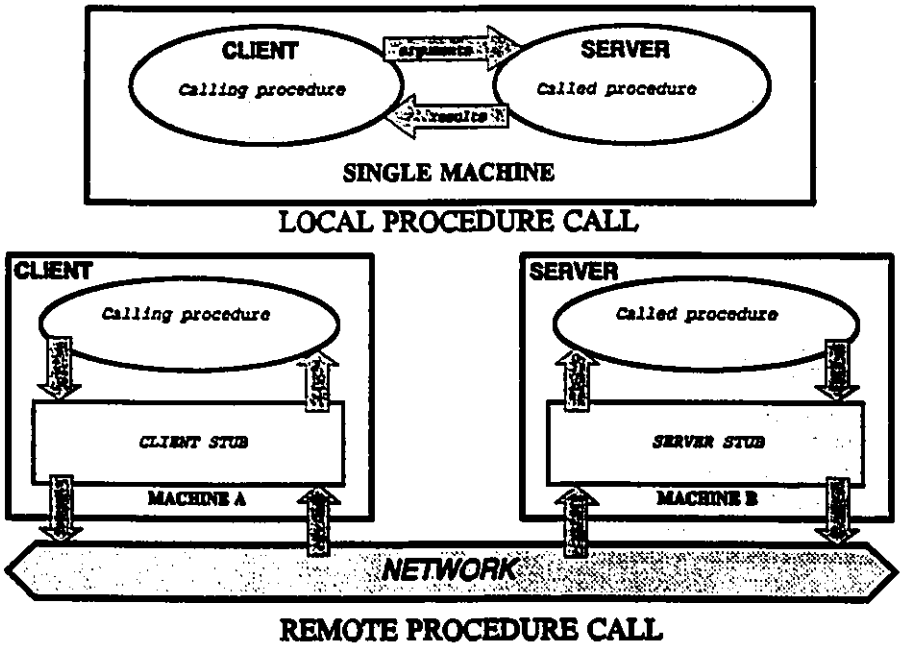

Figure 5.1: Local and remote procedure calls

RPC or Remote Procedure Calling is a mechanism for distributing program tasks to computers in a network and exchanging messages between them (Comer [1991], Comer and Stevens [1993]). Using the RPC, a procedure in a remote computer (server) can be called by a program in a local machine (client). The calling of a remote procedure is like calling a local procedure (see figure 5.1). Programming difficulties and intricacies associated with *network protocol*[1] and data transport are hidden, allowing program designers to focus on the algorithmic problems. The network protocol and data transport are handled by the *client and server stubs* which are codes created when the RPC compiler executes *protocol definition* codes written by the program developer.

The components of the remote procedure call can be described using the ISO - Organization for Standardization - 7 - Layer Reference Model as shown in figure 5.2 (Bloomer [1992]. The *user application*[2] sits at the top layer and uses the data representation - *XDR (External Data Representation)*[3] - to communicate with the RPC library. For transport, the RPC supports *TCP (Transmission Control Protocol)*[4] and *UDP (User Datagram Protocol)*[5] that sit on top of *IP (Internet Protocol)*[6].

---

[1] The IEEE Std 100-1988 gives definitions of protocol as (i) a set of conventions or rules that govern the interaction of processes or applications within a computer system or network, (ii) a set of rules that govern the operation of functional units to achieve communication.

[2] The program written by the user that uses data being communicated through the network.

[3] External Data Representation consists of two parts , (i) XDR Language, (ii) XDR Library functions. The XDR Language is used to specify the RPC messages. The XDR library routines perform encoding and decoding.

[4] Internet Transmission Control Protocol, layered upon IP. The connection is first established - routing, then streams are sent from the sender to the receiver.

[5] UDP - User Datagram Protocol, uses connectionless datagram. The size of each packet is 8kB and it contains the destination address as well as the data. This protocol is less reliable than the TCP.

[6] IP - Internet protocol, an internetwork datagram delivery protocol.

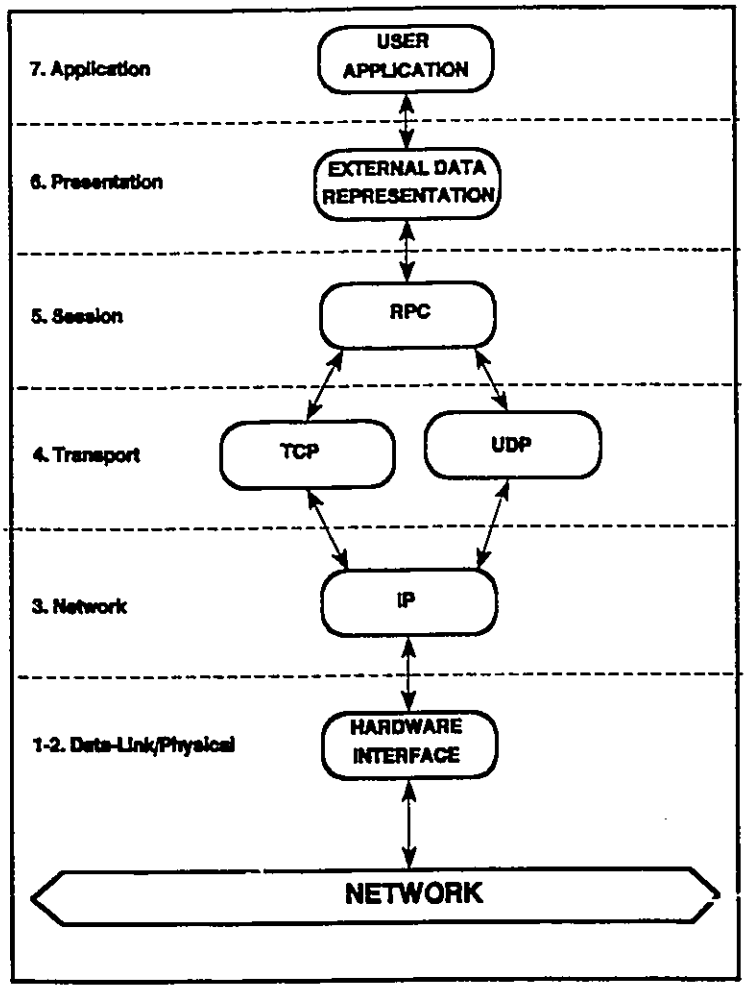

Figure 5.2: RPC system in relation to the ISO 7-Layer Reference model

It is most convenient to use the client-server computing model as depicted in figure 5.1. In a client-server model, there is one client computer and several server computers. As in local procedure call, the calling client will only resume execution after receiving a reply from the server. This limitation must be overcome to allow the client to call several servers at the same time to achieve parallel, distributed processing.

In using the RPC one will first write the *protocol definition*. In the *protocol definition*, types of data and structures to be transported between client and server and vice-versa are specified. It is also necessary to identify all the remote services to be called by the client. The information includes the name of the procedure, its return data type and the procedure number.

Unlike local procedures where many arguments can be specified, an RPC procedure carries only two arguments - the server name and the packet of the data to be sent to the server. The return value of the server procedure is contained in a packet having a structure type as declared in the protocol definition.

5.5 THE PARALLEL, DISTRIBUTED ALGORITHMS

Using the client-server model, as described in the preceding section, the algorithms for solving equations (5.3), (5.5) and (5.7) are now developed. The algorithms are divided into two phases; (i) initialization and (ii) iterative or direct solutions. The purpose of the initialization process is to load each server with its portion of matrix A and vector B and then proceeds with LU decomposition on its given block after which the factors will remain in the memory of the respective server. The second part is the iterative solution given by equation (5.3) or the direct solution as in equations (5.5) and (5.7). In both cases, the main data to be transported during the second phase is the solution vector **x** (see algorithm 5.1).

Algorithm 5.1: NBDF Iterative Method

INITIALIZATION PHASE

(a) Initialization in Client Parent Process:

- **(1) Form A and b**
- **(2) Guess the shared solution vector x**
- **(3) Create n_p processes in client machine**
- **(4) Send to each server its share of A_{0i} and B_i**

(b) Initialization of server:

- **(5) Factorize A_{0i} and keep the factors and B_i in server**

ITERATIVE PHASE

(a) Client machine:

- **(6) Each client child process i send vector x_i to its server**

(b) Servers:

- **(7) Each server solves for x_i**
- **(8) Each server sends solution back to client child i**

(c) Client machine:

- **(9) Each client child process checks convergence of x_i**
- **(10) Each client child process updates x**
- **(11) If converged, process i waits**
- **(12) If not converged, process i goes to (6)**

Algorithm 5.1 is asynchronous because in step (12), process i need not wait for other processes to update the shared vector x. The process simply carries the shared solution vector back to the server processor i to execute next iteration. The BBDF algorithm is as follows (Algorithm 5.2):

Algorithm 5.2: BBDF Direct Method

INITIALIZATION PHASE

(a) Initialization in Client Parent Process:
  (1) Form A and b
  (2) Create n_p processes in client machine
  (3) Send to each server its share of A_i, A_{oi} and B_i

(b) Initialization of servers:
  (4) Factorize A_i and keep the factors and B_i in server
  (5) Each server evaluates its component in the summation part of equation (5.6)
  (6) Send result in (5) to respective client child process

(c) Client machine:
  (7) Evaluate the whole of equation (5.6)

DIRECT SOLUTION PHASE

(a) Servers:
  (8) Each server evaluates its component in the summation part of equation (5.7)
  (9) Send result in (ix) to respective client child

(b) Client machine:
  (10) Evaluate the whole of equation (5.7) and solve for x_o in equation (5.5)
  (11) Send to each server, x_o

(c) Servers:
  (12) Each server solves equation (5.8)

5.6 RPC PROTOCOL DEFINITION

The protocol definition was briefly mentioned in section 5.4. In this section, the protocol definition used for the above algorithms are discussed in more detail.

```
/* irpc.x: XDR Protocol Definition */

/* PART 1 */
const NBMAX = 1000;  /* maximum number of buses */
const NYMAX = 20000; /* size of Y matrix including fill-ins*/
struct volt {
   /* process/area details */
   int   id;
   int   kstr;
   .
   .
   float vx<NBMAX>;
};
struct packet {
   int   nb;            /* number of buses */
   int   iareano;       /* no. of areas */
   int   id;
   float ir<NBMAX>;     /*current injection vectors */
   .
   .
   /* lu factor */
   int   kku;
   int   kkl;
   .
   .
   float yldx<NBMAX>;           /* yldx */
   /* External connections */
   int   iblkstr<NBMAX>;
   int   iexblk<NBMAX>;
   .
   .
   float cexvax<NBMAX>;
};
/* PART 2 */
program IRPCPROG {
  version IRPCVERS {
    void ISVC(packet)   = 1;
    volt WSVC(volt)     = 2;
    void FSVC(packet)   = 3;
    } = 1;
 } = 0x20000001;
```

The sample *protocol definition* shown above comprises of two major parts: definition of variables to be transported across the network and the identification of server procedures. All variables are bundled into a packet or structure. The first structure, *volt* is used in iterative and direct solution stages to communicate voltage and current vectors between client and servers. The *packet* structure contains data that are sent from the client to servers and will reside in server memory throughout the duration of the program. The second part, that is the procedure identification, specifies all server procedures

that may be called from the client. The procedure ISVC load all servers with static data and performs initial computation such as factorization. The WSVC procedure performs either the iterative or the direct solutions. The procedure FSVC is used to release all servers with the static data before server daemons[7] are removed.

5.7 MULTIPROCESSING

Starting of all servers during the first and the second phases must be at the same instance. The approach used here is to create duplicates of the client process in the client machine as described in chapter 3. Each client child process will control each server. Updating of any shared variable will be done by each client child. The relationship between the client, its child processes and the servers is shown in figure 5.3.

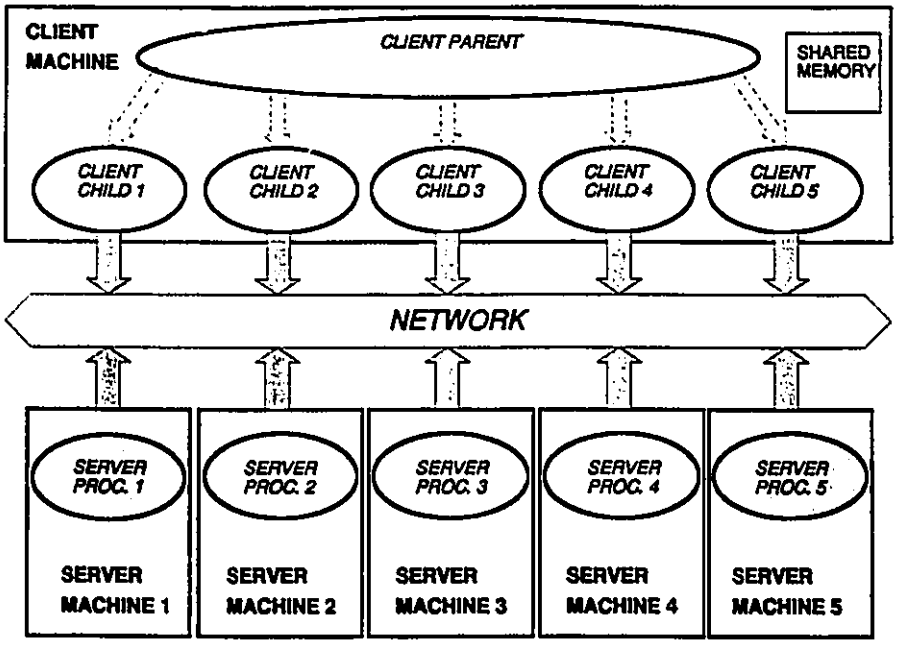

Figure 5.3: Client children and servers relationship

During the initial phase, each client child sends portion of matrix A and

[7] Once started, the server process - daemon - blocks on a receive, busy-waiting for client request. When a request arrives, the server wakes up and executes the relevant procedure.

vector b and other auxiliary data to each respective server. The factorization of all blocks are performed concurrently in each server. The LU factors (static variables) will remain in the server even when the control is brought back to the client. During the second phase, each client child will continue to communicate with its respective server until convergence is achieved in the case of the iterative method.

5.8 INTERPROCESS COMMUNICATIONS (IPC)

There are three mechanisms for interprocess communications in Unix (Valley [1991]) - (i) message passing (ii) shared memory, and (iii) semaphores. In this section, shared memory and semaphores are briefly described since they are used in implementing the above algorithms.

5.8.1 Shared Memory

As shown in figure 5.3, each client child must have access to the solution vector, x. Therefore, the solution vector must be shared amongst all the processes in the client machine. In Unix, shared memory segments are created using `shmget()` function. This function will return the segment identifier called `shmid` which can then be passed to other processes to allow them access to the resource. The next important function is `shmat()` which is used to decide at what address the shared segment identified by `shmid` is to be attached.

A function that can be used for creating shared identifier and consequently attaching it to next available address determined by the operating system is as follows:

```
char *mshared (size, shmid)
int size;
int *shmid;
{
   *shmid=shmget(IPC_PRIVATE, size, IPC_CREAT| IPC_EXCL| 0660);
   if (*shmid == -1) {
     fprintf(stderr, "shmget failed: errno = %d\n", errno);
     perror("       ");
     return;
```

```
    }
    return( (char *) shmat(*shmid, (char *)NULL, 0));
}
```

The above function can be called in a similar manner as `malloc()` memory allocation function. For creating the shared segment of the solution vector x, the function calls are as follows:

```
vreal = (float *) mshared (size, &shmidvr);
xreal = (float *) mshared (size, &shmidxr);
```

To control the shared memory segment, for instance, to remove it, function `shmctl()` is used.

5.8.2 Semaphores

The iterative process in algorithm 1 is asynchronous. Therefore, it is possible that any two client children may be attempting to access the same shared address at the same time. When this happens the two processes are said to be in contention for the same variable. Data contention can also occur in the case of algorithm 2. In evaluating equation (5.7) in step (8), the summation part is updated by each process separately and two processes may be attempting to do the updating at the same time. To avoid data contention, a signal termed semaphore is used.

In the iterative algorithm, if it is only necessary to update the shared vector, the problem of data contention does not arise, since each process is confined to its own part of the shared vector. However, if the shared vector is required to be sent to servers in the subsequent iterations, it is possible that a particular element of the shared vector is being updated by a process while another process requires access the same element. The following illustrates the use of semaphores to ensure that the shared vector can be accessed by only one process at a time.

```
(i)   Copy shared vector x to y
(ii)    Iteration point
(iii) Call server using:  z = WSVC(y, client_handle)
(iv)  sema_lock()
        /* protected area */
          use z to update x
          check convergent
          copy x to y
      sema_unlock
(v)     goto (ii)
```

When a process calls the `sema_lock()` function, a check is made on the status of the lock (`sem_op`) of the semaphore identified by `semid`. If it is unlocked, it will first be locked and the calling process will continue execution in its protected area. If it is already locked, the calling process waits until it becomes unlocked. When a process finishes executing the protected area, the `sema_lock()` function will unlock and makes the protected area available to the next waiting process. The functions to implement `sema_lock()` and `sema_unlock()` are as follows:

```
void sema_lock (semid)
int *semid;
{
   int irtn;
   struct sembuf operations;
   operations.sem_num = 0;
   operations.sem_op  = -1;
   operations.sem_flg = 0;
   irtn = semop(*semid, &operations, 1);
   if (irtn == -1) {
     fprintf(stderr, "semop failed: errno = %d\n", errno);
     perror("       ");
   }
   return;
}

void sema_unlock (semid)
int  *semid;
{
   int     irtn;
   struct sembuf operations;
   operations.sem_num = 0;
   operations.sem_op  = 1;
   operations.sem_flg = 0;
   semop(*semid, &operations, 1);
```

```
   if (irtn == -1) {
     fprintf(stderr, "semop failed: errno = %d\n", errno);
     perror("      ");
   }
   return;
}
```

Functions sema_lock() and sema_unlock() are created using the Unix library function semop(). As in the shared memory mechanism, the semaphore identifier semid is first created using the semget() function using the following function:

```
void create_sema (semid, condition)
int  *semid, condition;
{
   int  control, irtn;
   *semid=semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
   if (*semid == -1) {
   fprintf(stderr, "semget failed: errno = %d\n",errno);
   perror("      ");
   return;
   }
   if ( condition == 0 ) {
      control = 1;
   } else {
      control = 0;
   }
   irtn = semctl(*semid, 0, SETVAL, control);
   if (irtn == -1) {
   fprintf(stderr, "semctl failed: errno = %d\n",errno);
   perror("      ");
   return;
   }
}
```

Apart from the two IPC facilities described above, barriers are also used in implementing parallel, distributed factorisation of matrix A. Barriers are used to ensure that certain part of the algorithm is executed by all processors before proceeding to the next calculations.

5.9 RESULTS AND ANALYSIS

Algorithms 5.1 and 5.2 were implemented on a cluster of Sun Sparc Workstations as shown in figure 5.4. Workstation 1 is chosen as the client, and workstation 2-12 are servers.

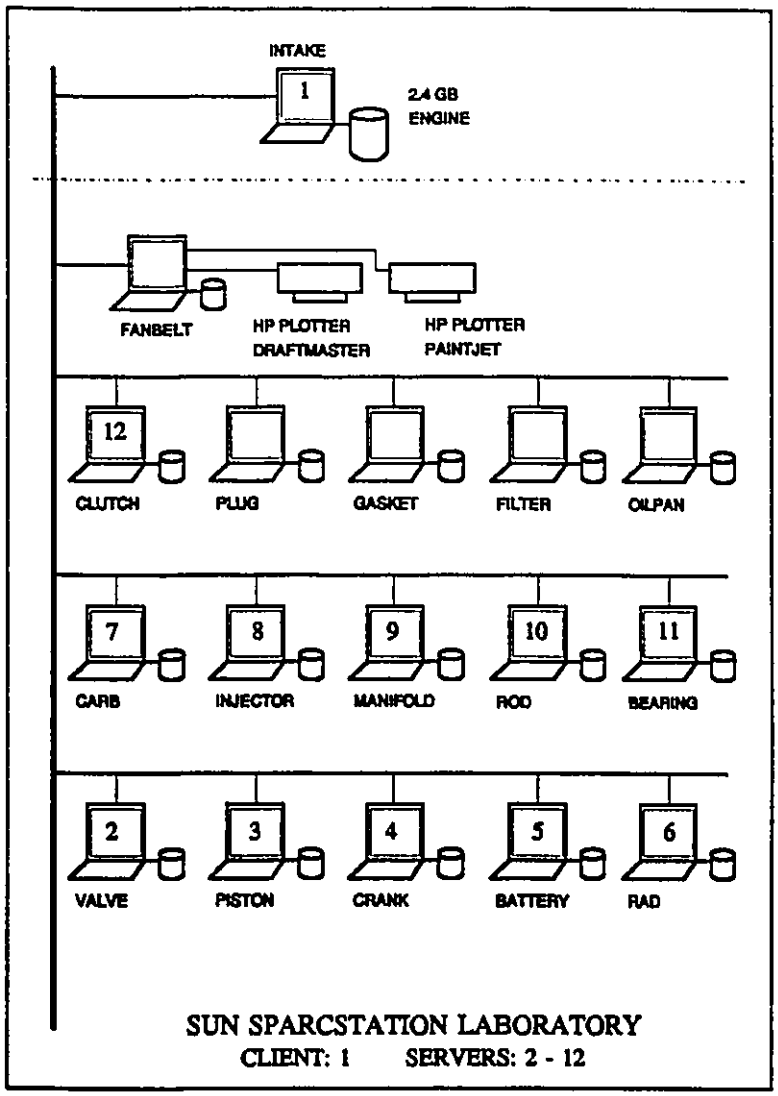

Figure 5.4: Cluster of workstations

Three power systems matrices were tested. The systems are: (i) the 118-bus IEEE test system, (ii) the 348-bus TNB/PUB (Malaysia/Singapore) system and (iii) the 840-bus EGAT/TNB/PUB (Thailand/Malaysia/Singapore) system. To initiate iterative solution, the current injection vector **B** for each

system matrix was slightly perturbed. In the implementation of the algorithms, those portions that manage interprocessor communication form a significant part of the computation time. In cases where the number of workstations is less than the areas, pre-scheduling is recommended (see chapter 6). In pre-scheduling, each server is allocated some areas beforehand thus reducing communication overheads in comparison to self-scheduling method. All the simulations results given in this section were obtained during off-hours.

5.9.1 Factorization

Results of LU factorisations for three serial algorithms executing on a single workstation are given in table 5.1. Although the factorisation speed for the NBDF ordering method on the 118-bus and the 840-bus systems in table 5.1 is faster than the corresponding scheme 1 ordering, the sluggish triangular solution speed of this method makes it uncompetitive to that of the BBDF ordering. The BBDF ordering method results in S^2 (serial slowdown) factors of 1.29, 0.99 and 0.61 respectively. The larger the system, the S^2 factor tends to reduce and in particular for the 840-bus system a speedup of 1.6 is obtained. The results of BBDF ordering factorisation using parallel, distributed algorithms with several workstations are given in table 5.2.

Power Systems	Scheme 1 Ordering		NBDF Ordering		BBDF Ordering	
	Fact	Tria. Soln.	Fact.	*Tria. Soln.	Fact.	Tria. Soln.
118-bus	420	17	250	50	540	35
348-bus	2150	20	2170	190	2130	70
840-bus	18530	90	14970	590	11320	210

*** 5 iterations. Note: all times in milliseconds**

Table 5.1: LU Factorisation, Serial computation speed

Power Systems	BBDF Ordering		
	No. of Workstations	Fact. Time	Speedup
118-bus	2	570	0.74
	4	430	0.98
348-bus	2	1350	1.60
	4	800	2.67
840-bus	2	9230	1.94
	4	5020	3.56
	6	5410	3.30
	8	4980	3.59
	10	4950	3.61
	12	4950	3.61

All times in milliseconds

Table 5.2: LU Factorisation, Parallel, distributed computation speed

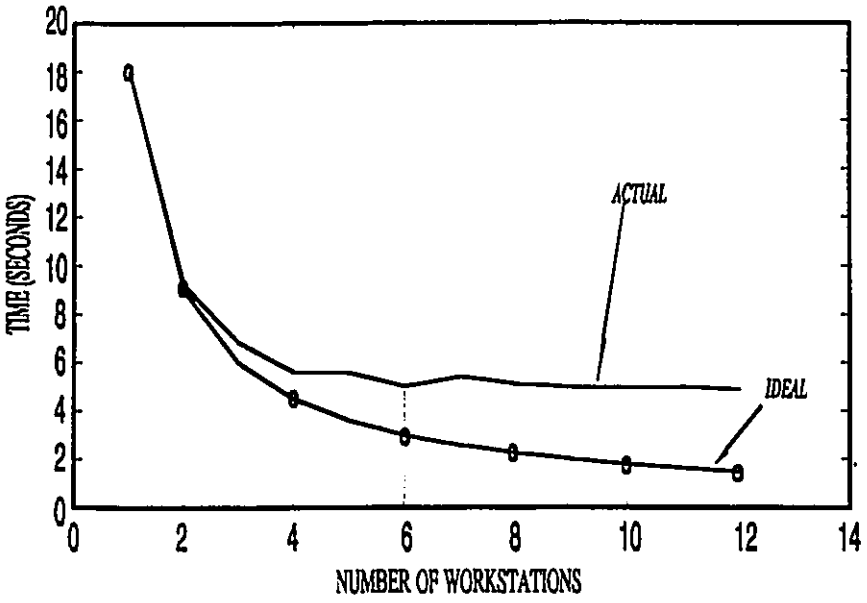

Figure 5.5: Factorisation times for 840-bus system with increasing number of workstations

Figure 5.5 shows how the factorisation time decreases as the number of workstation is increased. The speedup quickly saturates after about 6 workstations. As the number of workstation increases, the time to factorise the diagonal blocks is reduced. But the factorisation time does not improve further due to the factorisation of the bordered matrix performed on the client machine in addition to the increase in communication overheads.

5.9.2 Triangular Solutions

The triangular solution speed for the three methods are also given in table 5.1. For the BBDF ordering method the S^2 factors are 2.06, 3.5, and 2.33 respectively. Table 5.3 shows the results of the parallel, distributed solution using several workstations for the BBDF ordering method. For the 118-bus system no speedup is obtained since for this small system whatever gain is made by distributing tasks will be reduced by the additional communication overheads. Maximum speedup of 2.72 is obtained on the 840-bus system. This value of speedup is small but as long as it is more competitive to that of

scheme 1 solution much has already been gained by the speedup in factorization. Also in table 5.3, speedup against the serial BBDF solution is given in order to observe the effect of increasing the number of workstations as shown in figure 5.6. Maximum speedup for the 840-bus system is obtained with 8 workstations.

Power Systems	No. of Areas	Parallel, distributed soln.		Speed-up
		No. of Servers	Soln. Time	
118-bus 17+	6	2	17	1.00
		4	17	1.00
348-bus 20+	5	2	50	0.40 (1.4)
		4	20	1.00 (3.5)
840-bus 90+	12	2	130	0.70 (1.61)
		4	80	1.13 (2.63)
		6	50	1.80 (4.20)
		8	33	2.72 (6.36)
		10	50	1.80 (4.20)
		12	40	2.25 (5.25)

All times in milliseconds. (In bracket) against serial BBDF method
+ Fastest serial computation speed (From Table 5.1)

Table 5.3: Parallel, distributed BBDF solution

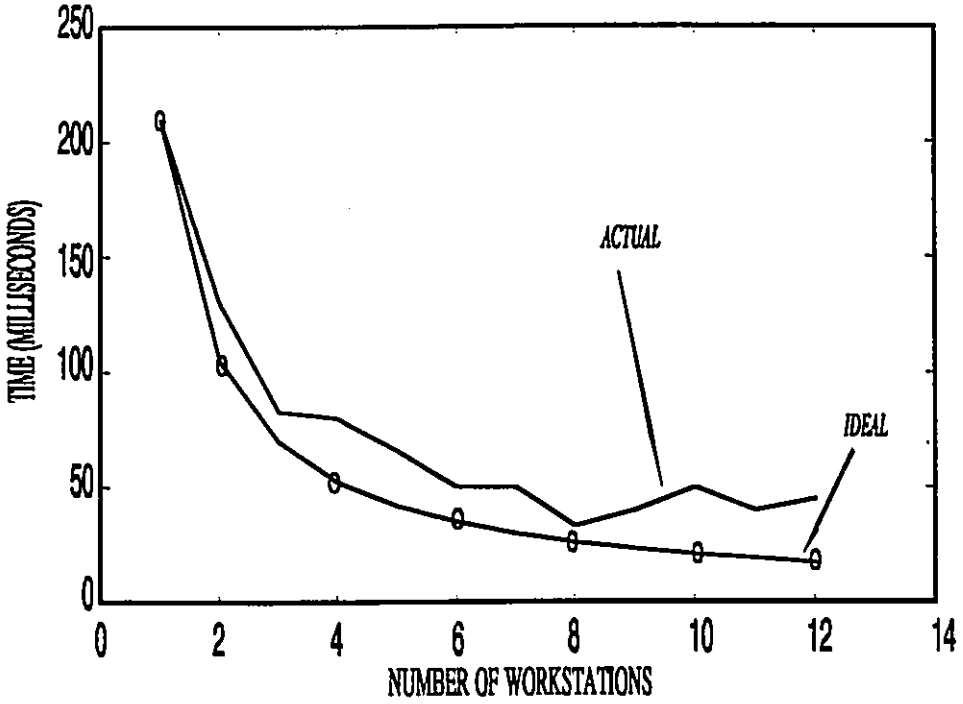

Figure 5.6: Triangular solution speed vs no. of workstations

As observed from figure 5.6, the speed of computation does not improve any further after about 7-8 workstations.

5.10 CONCLUSIONS

In this chapter, parallel distributed solution of linear systems has been implemented on a cluster of workstations. The client/server paradigms implemented using the RPC technique will become pervasive in the future with widespread use of networked workstations. The implementation strategy by distributing the tasks amongst workstations and then executing the computation in parallel have been described. The work described in this chapter is also reported in Yusof, et. al. [1993c].

The results of the computation shows speedup as the number of workstations is increased but that saturation is quickly reached. However the

results clearly indicate the potential applications of networked workstation in power system analysis. The partitioning method together with the algorithms for distributing tasks and parallel execution can be applied to load flow, transient stability and eigenvalue analyses to improve the speed of computation. Future development in high-speed optical networks (Vetter and Du [1993]) will make high-performance computing in a distributed environment more attractive. Furthermore, the use of existing protocol, such as the TCP/IP, requires several layers of software in the ISO 7-layer model that makes data communication very costly. One possible alternative to reduce the communication overhead is to directly link the user application layer to the data-link/physical layer using a specialized/customized network interface card.

CHAPTER 6

COMMUNICATION ASPECTS OF THE RPC TECHNIQUE FOR PARALLEL DISTRIBUTED COMPUTATION

6.1 INTRODUCTION

When using loosely coupled processors for implementing parallel, distributed algorithms, the time spent for interprocessor communications (IPC) is a sizeable fraction of the time needed to solve the problem. This IPC aspect is more predominant when using a cluster of workstations in comparison to using centralized loosely coupled processor systems. In this chapter, the communication aspects are analyzed in relation to task scheduling. In addition, the parallelization process is discussed.

6.2 COMMUNICATION ELEMENTS

The communication penalty, C_P, can be expressed in a ratio (Bertsekas and Tsitsiklis[1989]):

$$C_P = \frac{T_T}{T_C} \tag{6.1}$$

where T_T is the time required by the parallel, distributed program and T_C is the time for computation when all communications are neglected. When using a cluster of workstations and RPC mechanisms, the communication components are readily isolated. The basic idea of using RPC is to transfer a local procedure to a remote machine that can then be called from the local machine. Therefore, when developing an RPC program, one would first implement an algorithm using local procedures. Following that, the local

procedures are transferred to remote machines. For the remote procedures to be successfully called from the local machine, all communication elements must be constructed.

Using the RPC technique, communication elements can generally be divided into the following six parts:

i. ***Packaging time***. This is the time required to prepare data for transmission. In the case of the Sun RPC, the information is packaged into a structure. This part is always within the control of the programmer. If a set of data is to be used repetitively, the *stateful servers*[1] can be initialized in advance. For reliable data transport *connection-oriented*[2] servers are used. However, if all workstations are connected to the same local network then *connectionless* servers may be used.

ii. ***Client Stub time***. This is time required for the program to call the client stub which includes data translation to XDR (External Data Representation) and preparation for transmission by placing the data into a buffer. In the RPC technique, the client stub is a set of codes created by the ***rpcgen*** compiler that performs all the RPC library calls for sending and receiving messages.

iii. ***Transmission time***. This is the time required for transmission of all the bits of the data in the packet into the network. This time is proportional to the size of the packet to be transported.

iv. ***Propagation Time***. This is the time between the end of transmission of the last bit of the packet at the client end and the reception of the last bit of the packet at the server end.

v. ***Server Stub time***. This is the time required by the server to translate the XDR data to the structure type used by the remote procedure.

---

[1] Stateful server as opposed to stateless server - keeps state information - information that the server maintains about states of ongoing interactions with clients.

[2] In layer 4 of the ISO model, one has the choice to either use UDP (User Datagram Protocol) or TCP (Transmission Control Protocol) for data transport. TCP is connection oriented, that is, the client keeps track of the data sent. If collision occurs, the client resends data. But UDP - connectionless - does not keep track of the data sent. TCP is more reliable than UDP.

vi. *Unpackaging time*. The time required for unpacking the structure to its individual contents for direct use by the remote procedure.

The same fragments of times are also incurred on the server when communicating with the client. But this is less important because it is assumed to occur simultaneously on all servers. The transmission and the propagation time can be lumped together under *network time*. The order of magnitude the above times are sampled on an algorithm for power systems network solution based on the BBDF (Block Bordered Diagonal Forms) ordering as described in chapter 5.

6.3 PARALLELIZATION PROCESS

In this section, we discuss the parallelization process to clearly identify the role of communications. Given a serial algorithm, parallelization follows three major steps (see also flowchart in figure 6.1):

(a) ***Partitioning,***

(b) ***Development of serial algorithm that solves the original problem using the partitions found in (a), and***

(c) ***Incorporating IPCs into the algorithm in (b) and dispatching program to other processors.***

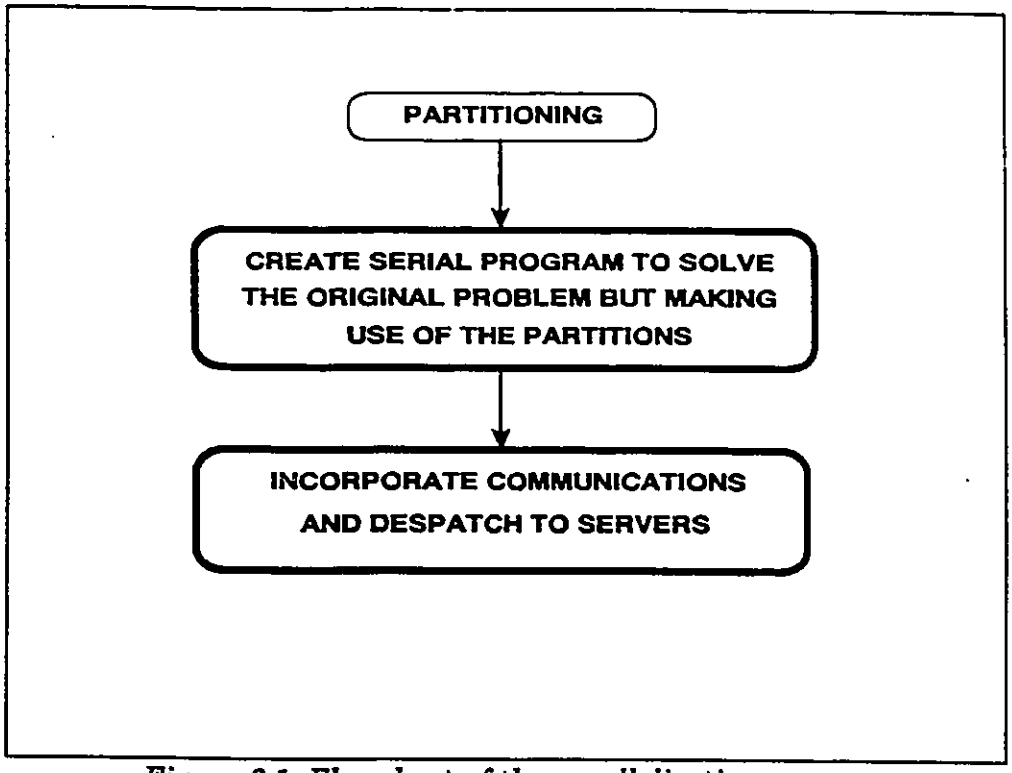

Figure 6.1: Flowchart of the parallelization process

When the RPC paradigm is used on a cluster of workstations, coarse-grain partitioning is usually preferred to fine-grain in order to minimize client-server communications. With coarse-grain partitions, basic serial algorithms that are used to solve the original problem can now be applied to each partition. Therefore, whatever advances and enhancements that have been made on the serial algorithm will be retained in the parallel, distributed version but applied to each partition. In addition, an interface algorithm is required to couple the partitions. The second step is significant because one can use it to assess the partitioning penalty. Partitioning penalty P_P is defined as the ratio of:

$$P_P = \frac{T_P}{T_C} \tag{6.2}$$

where T_P is the execution time on a uniprocessor of the algorithm with the partition and T_C is the same as in equation (6.1). When codes are to be optimized, the objective is then to minimize the partitioning penalty before

incorporating the IPCs. Factor P_p is sometimes called the S^2 (serial slowdown) factor (IEEE Committee Report [1992]). Algorithm 5.2 described in chapter 5 is rewritten into six parts as in algorithm 6.1. The objective is to clearly separate parts that require communications from those parts that are executed serially.

Algorithm 6.1: BBDF network solution

---

i. Pack data for all servers in client.
ii. Client communicates with all servers.
iii. Servers execute phase 1 of the solution and send back results to client
iv. Client executes intermediate solution phase and packs data for all servers
v. Client communicates with all servers.
vi. Server executes final solution phase and sends results to client

---

Algorithm 6.1 was tested on a 840-bus, 219-generator power systems network. The network is partitioned into twelve areas.

6.4 SCHEDULING TECHNIQUES

There are three scheduling methods that can be used in steps (ii) and (v). The *loop-splitting technique* divides the partitions among the processors. There are two implementation strategies: in the loop (i) every time the partition index matches the processor identifier the server is called, and (ii) for a given partition index a server is called that will process several partitions. In case (ii) a server is only called once. In the *self-scheduling technique*, the client continuously checks for free servers. As soon as a server is free it is called upon to work. The *pre-scheduling technique* determines the partitions to be loaded and worked upon by a server. Therefore, each server is called only once. The above scheduling methods are given in pseudo-code forms in text boxes 6.1, 6.2, 6.3, and 6.4.

```
1. Loop-splitting technique - when  the number of processors equals the
   number of areas (partitions)

   for area = 1 to no_area
      if ( pid == area )
         ( work on area )
      end if
   end for
```

Text box 6.1: Loop-splitting technique (a)

```
2. Loop-splitting technique - when the number of processors is less than
the number of areas (partitions)

   for area = pid  to no_area step nproc
         ( work on area )
   end for
```

Text box 6.2: Loop-splitting technique (b)

```
3. Self-scheduling technique

   no_area = m

   start:
   sema_lock
      area = no_area
      no_area = no_area - 1
   sema_unlock

   if ( no_area <= 0  )
      goto finish
   else
      ( work on area )
   end if

   goto start
   finish:
```

Text box 6.3: Self-scheduling technique

```
4. Pre-scheduling technique

   Create schedule vector - sch_vect()

   for area = 1 to no_area
      if ( sch_vect(area) == pid )
         (work on area)
      end if
```

Text box 6.4: Pre-scheduling technique

In the pseudo-codes, `no_area` is the same as the number of blocks in matrix A and pid is the processor identifier number. The available number of processors is denoted by nproc. The schedule vector `sch_vec()` has an index which corresponds to area (block) number and the content of each element is the processor to which the task to be sent. In practical implementation, the codes are executed by each child process concurrently in the client that calls the respective server to perform the work.

6.5 RESULTS AND ANALYSIS

The following results are obtained using the self-scheduling technique. Using the client-single server model as depicted in figure 6.2, the time required for each step of algorithm 6.1 is shown in table 6.1.

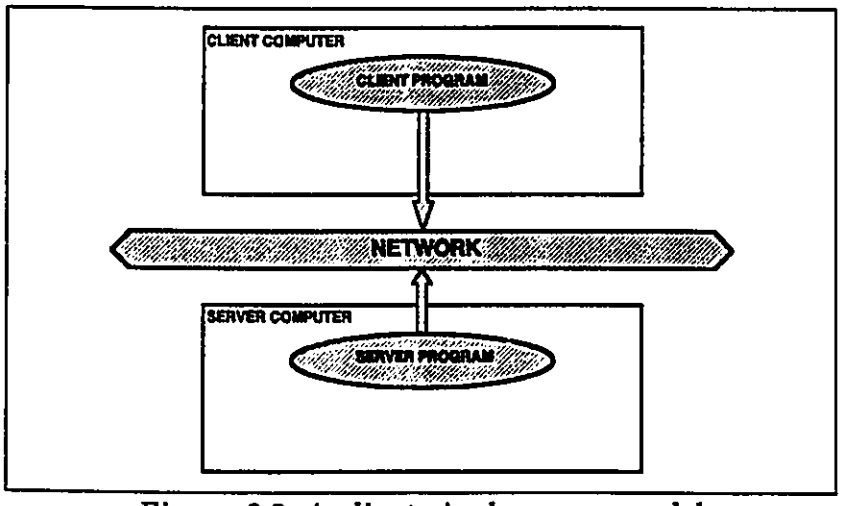

Figure 6.2: A client-single server model

Step	Time (milliseconds)	Percentage of Total time
i	11	1.5
ii and iii	350	47.9
iv	10	1.5
v and vi	360	49.1
Total	731	100.0

Table 6.1: Profile of algorithm 1 - RPC implementation using self-scheduling.

From table 6.1, it can be seen that most of the time is spent on those parts of the program that communicate with the servers. In the self-scheduling technique, when a server is free, the client makes a call. The client also informs the server which portion of the tasks are to be executed. If there are **no_area** partitions, an equal number of server calls are made eventhough the number of servers **nproc** is less than the number of partitions. Whereas in loop-splitting (case (ii)) and pre-scheduling techniques, the number of calls made are the same as the number of servers. The main advantage of self-scheduling is that the client will ensure that all servers are kept busy.

The steps that involve network communications in algorithm 6.1 are ii & iii and v & vi. To determine the communication time involved in results of table 6.1, the same algorithm is executed on a single processor using local procedures as depicted in figure 6.3. This follows the second step of the parallelization process as described in section 6.3. The execution times on the parts equivalent to steps ii & iii and v & vi of algorithm 1 are shown in table 6.2. Using the total time in table 6.1 and 6.2, the communication penalty is 3.97.

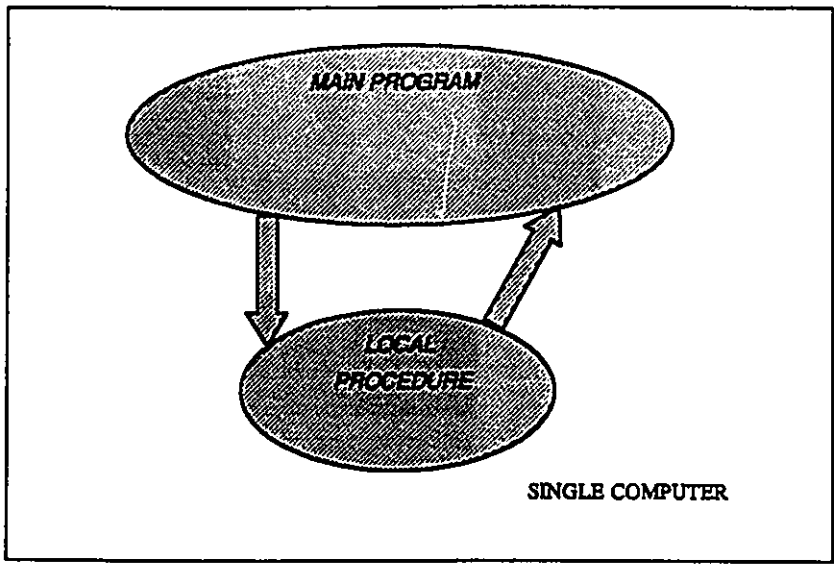

Figure 6.3: A local-procedure model

Step	Time (milliseconds)	Percentage of Total time
i	0	0.00
ii and iii	100	54.35
iv	0	0.00
v and vi	84	45.65
Total	184	100.00

Table 6.2: Profile of algorithm 1 - non-RPC implementation

The equivalent serial algorithm implemented without partitions takes a total of 70 ms to execute. Therefore, the partition penalty is about 2.63 and this is after code optimization has been performed.

To determine whether the *network time* contributes to the times in step ii & iii and v & vi, the RPC program was also executed on a single machine which assumed both the role of client and server at the same time. Therefore, this version eliminates propagation time. The result indicates that there is little difference in the execution speed to that with separate server and client

computers as given in table 6.1.

The communication times for steps ii & iii and v & vi and additional computation in i and iv of algorithm 6.1 implemented using RPC techniques are summarized in table 6.3.

Step	Time (milliseconds)
i	11
ii and iii	250
iv	10
v and vi	276
Total	547

Percentage of communication time = 74.8%

Table 6.3: Summary of communication times - RPC implementation using self-scheduling

Times for data packing and unpacking into structures which become parts of the procedure in both the client and server are small as compared to the RPC library function calls in steps ii & iii and v & vi. The main RPC library functions are `clnt_call()` in the client and `svc_sendreply()` in the server (Bloomer [1991]). Function `clnt_call()` calls the remote procedure. It encodes request arguments and reply results by the specified XDR procedures. Function `svc_sendreply()` is used by service dispatch routines to reply to the client, passing the specified data through the specified XDR encoder.

From the above results, it is clear that for more efficient execution, RPC library calls have to be minimized. The use of the pre-scheduling technique minimizes RPC library calls when the number of servers are less than the number of partitions. However, when the number of servers equals the number of partitions, the number of client RPC calls is the same as the number of servers. Using the pre-scheduling technique the results for the

same program execution as in table 6.1 are shown in table 6.4.

Step	Time (milliseconds)	Percentage of Total time
i	20	16.00
ii and iii	35	28.00
iv	10	8.00
v and vi	60	48.00
Total	125	100.00

Table 6.4: Profile of algorithm 6.1 - RPC implementation using pre-scheduling.

The results in table 6.4 show that by minimizing RPC library calls, the execution speed can be improved. The execution time is comparable to the one with the non-RPC version given in table 6.2. A very important strategy in developing a parallel, distributed program is to ensure that the version which includes communication overheads executes in comparable time to the one without communication, when run on a single server.

Figure 6.4 shows how the execution time using both self- scheduling and pre-scheduling techniques vary with increasing numbers of workstations. Both methods result in similar speedup after 6 workstations are used. The pre-scheduling technique performs better when the number of workstations is small.

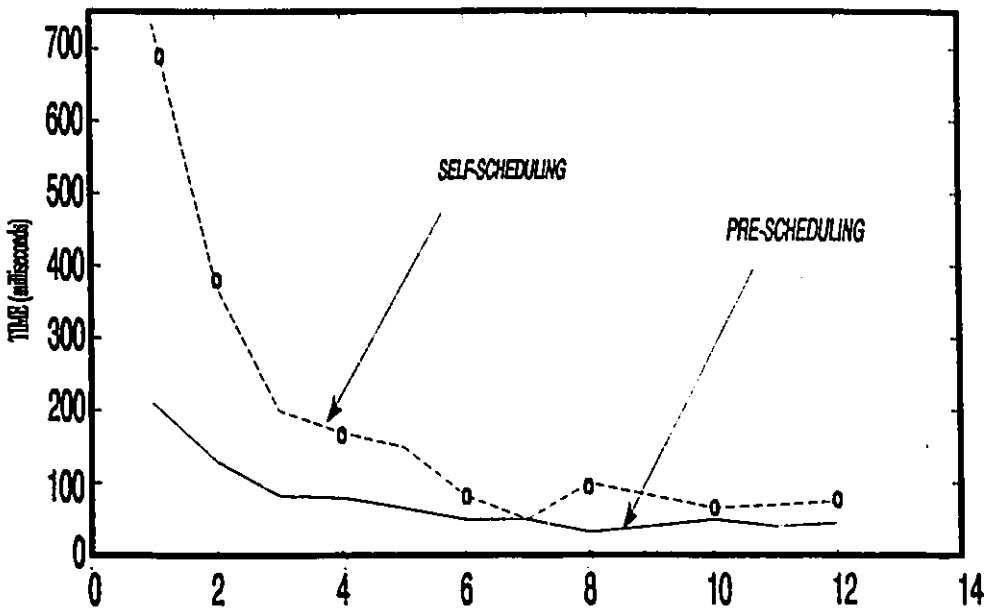

Figure 6.4: Solution of triangular systems for 840-bus system against number of workstations

6.6 CONCLUSIONS

In this chapter, communication factors of a parallel, distributed algorithm implemented using RPC mechanisms have been described. Those factors were measured using a triangular system solution algorithm. Since packing and unpacking portions of the program are within the control of the programmer they can be easily minimized, for example, by using stateful servers. RPC library calls constitute the significant cost of communications. However, by careful scheduling, the number of calls can be minimized. For any application, it is important to make careful choice of the scheduling technique.

CHAPTER 7
PARALLEL DISTRIBUTED TRANSIENT STABILITY ALGORITHMS AND IMPLEMENTATIONS

7.1 INTRODUCTION

In chapter 5, client-server computing using the RPC technique has been applied to the solutions of linear algebraic equations. For serial transient stability algorithms described in chapter 2, the differential equations are easily parallelized because they are independent of each other. However, parallelization of the network solution is less tractable. But using the partitioning method described in chapter 4 and the technique given in chapter 5, the network solution can now be parallelized in the transient stability problem.

In this chapter, algorithms 2.1, 2.2, 2.3 and 2.4 are parallelized. The parallelization process described in chapter 6 is applied to each algorithm. The partition is first applied to each algorithm and then the partitioned problem is solved in serial manner and the S^2 factor determined. Following that, the shared memory implementation of the problem is described. Finally, the RPC implementation is described. The results of simulations on several systems using a cluster of workstations will be discussed and analyzed. To illustrate important segments of the actual program, C syntax is used. These program segments are for illustration only and may have been simplified from the actual implemented code.

7.2 PARTITIONING OF PROBLEMS

The network is to be renumbered so that the resulting admittance

matrix will be in blocked bordered diagonal form (BBDF). As machines are identified with the corresponding network buses, they also belong to the same area number. The sequence of DCPS commands to partition the network matrix is shown in text box 7.1.

```
chco,1.2,1.5,1.5,0.01,0.0005,0.0005,0.01,0.01
redn,tnp95.sol      % read load flow data
redmn,tnp95f.mac    % read machine data
reba;
rene;
ybus;
```

Text box 7.1: DCPS command for BBDF ordering

In the load flow data of the system (`tnp95.sol`), the area number of each bus corresponds to the coherent area number determined using the method described in chapter 4. Function `reba` identifies all boundary buses and changes their area number to a new number. If there are `no_area` coherent areas, the new area is given a number `no_area + 1`. If there are two coherent areas, the resulting admittance matrix will have 3 block diagonals where the last block represents the boundary buses. To illustrate the technique, the nine-bus system of Anderson and Fouad [1977] is modified to have boundary buses as shown in figure 7.1. There are two coherent areas - area 1 and area 2. Four buses are added to represent the boundary buses - 10, 11, 12 and 13 - and placed under area 3. The resulting admittance matrix pattern is shown in figure 7.2.

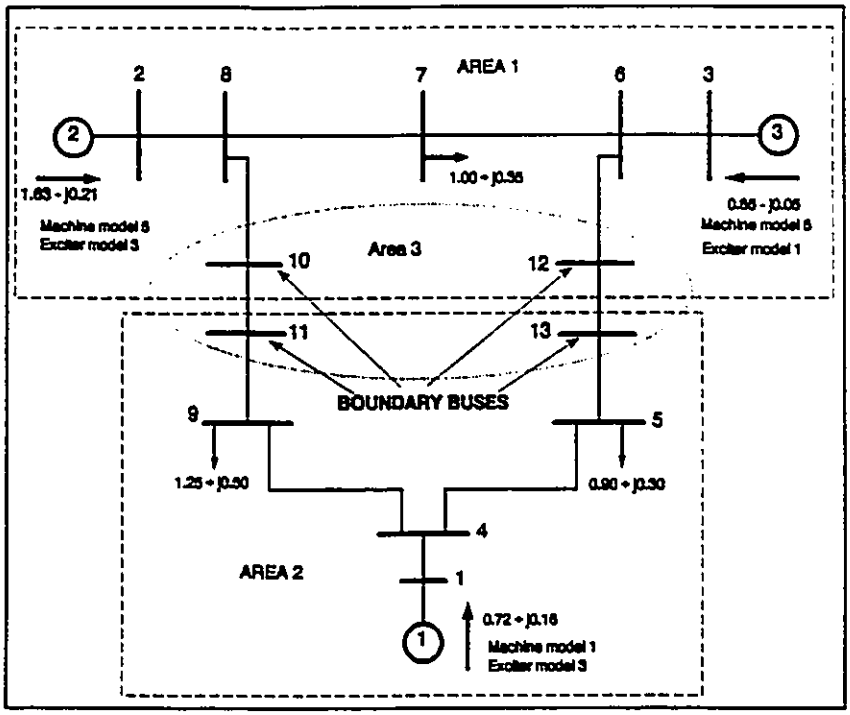

Figure 7.1: Boundary buses for the nine-bus system

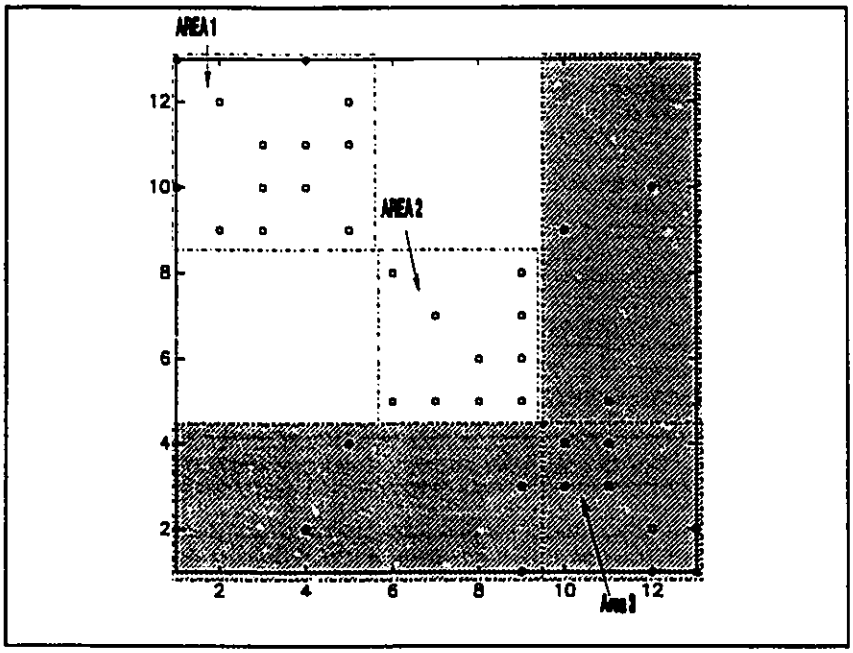

Figure 7.2: Admittance matrix pattern due to partitioning as in figure 7.1

7.3 SOLUTION WITH PARTITIONS

To solve the partitioned problem, all work is divided among areas - the original problem is reformulated by areas. As an example, if the original serial program function `trmv` is called, that operates on the whole system, then function `trmv` has to be modified (say `atrmv`) to operate only on a given area as illustrated in text box 7.2.

```
for (area = 1; area <= iareano; area++)
     atrmv ( area );
```

Text box 7.2: Dividing work amongst areas

In the following sections, the division of work among areas for algorithms 2.1, 2.2, 2.3 and 2.4 is described and illustrated.

7.3.1 Modified Euler Method

Algorithm 2.1 can be written in its code form to illustrate actual function calls and shown in text box 7.3.

```
for ( t = t1; t <= t2; += dt) {
   eul1();          /* initial estimate of states */
   ginjn();         /* initial current injection  */
   sol1();          /* initial network solution   */
   dqtrn();         /* initial transformation     */
   eul2();          /* final estimate of states   */
   ginj();          /* final current injection    */
   sol1();          /* final network solution     */
   dqtr();          /* final transformation       */
}
```

Text box 7.3: Serial Modified Euler method - function calls

There is a slight difference in the implementation of algorithm 2.1 as given in text box 7.3 that the final network solution is performed once during each time step. Dandeno and Kundur [1973] have shown that it is not necessary to perform network iteration unless switching or other large network changes have occurred. In the DCPS implementation, all switching

operations and network changes are performed separately followed by iterative network solution (see example in text box 7.13). After the problem is partitioned, functions are executed by areas as shown in text box 7.4 and the BBDF network solution is incorporated.

```
for (t = t1; t <= t2; t += dt) {
   /* PARALLELIZABLE LOOP 1 */
   for (area = 1; area <= iareano; area++) {
      adqtre ();    /* final d-q transformation          */
      aeul1 ();     /* initial estimate of states        */
      aginjn ();    /* initial current injection         */
      bbsol ();     /* phase 1 of initial network soln.  */
   }

   /* SERIAL PART */
   sol2x ();        /* initial soln. of bordered matrix */

   /* PARALLELIZABLE LOOP 2 */
   for (area = 1 ; area <= iareano; area++) {
      bbso2 ();     /* phase 2 of initial network soln.  */
      adqtrne();    /* initial d-q transformation        */
      aeul2();      /* final estimates of states         */
      aginje ();    /* final current injection           */
      bbsol ();     /* phase 1 of final network soln.    */
   }

   /* SERIAL PART */
   sol2x ();        /* final soln. of bordered matrix    */

   /* PARALLELIZABLE LOOP 3 */
   for (area = 1 ; area <= iareano; area++)
      bbso2 ();     /* phase 2 of final network soln.    */
}
```

Text box 7.4: Modified Euler method - solution of partitioned problem

All functions are called by area except for the solution of the bordered matrix by function `sol2x()` - performs backward/forward solution for equation 5.5. In the parallelizable version, final d-q transformation by function `adqtre` has been moved to the first parallelizable loop.

The objectives at this point are: (i) to arrange so that most functions are in the parallelizable loop, and (ii) to minimize the number of parallelizable

loops. More parallelizable loops means more functions can be parallelized but at the expense of higher communication cost. In table box 7.4, there are three possible parallelizable loops but the third loop with only one function could be executed in serial.

7.3.2 Trapezoidal Method With Fixed Point Iteration

As opposed to the Modified Euler method, the Trapezoidal method has less function calls as shown in text box 7.5.

```
for (t = t1; t <= t2; t += dt) {
   trmc();    /* calculate non-integrable part */
   iteration_point:
   trmv();    /* calculate integrable part     */
   ginj();    /* current injection             */
   soll();    /* network solution              */
   genpow();  /* machine power                 */
   dqtr();    /* d-q transformation            */
   if ( != converge ) goto iteration_point
}
```

Text box 7.5: Serial Trapezoidal method

Function `trmc()` calculates the non-integrable parts that remain constant during each time step and function `trmv()` calculates the integrable part that requires iterative solution. Network solution is performed by `soll()` after the current injection is updated using `ginj()`. The parallelizable version is given in text box 7.6.

```
for (t = t1; t <= t2; t += dt) {
   iter = 0;
   iteration_point:

   /* PARALLELIZABLE LOOP 1 */
   for ( area = 1; area <= iareano; area++ ) {
      adqtr ();        /* transformation                  */
      agenpow ();      /* calculate machine powers        */
      if (iter == 0)
         atrmc ();     /* non-integrable part             */
      atrmv ();        /* integrable part                 */
      aginj ();        /* current injection               */
      bbsol ();        /* phase 1 of network soln.        */
   }

   /* SERIAL PART */
   sol2x ();           /* network bordered matrix soln.   */

   /* PARALLELIZABLE LOOP 2 */
   for ( area = 1; area <= iareano; area++ ) {
      bbso2 ();        /* phase 2 of network soln.        */
      convchk ();      /* convergent check                */
   }
   if ( != converged ) iter++ and goto iteration_point;
}
```

Text box 7.6: Trapezoidal method - partitioned solution

In text box 7.6, there are two parallelizable loops and most functions are in the first loop. Note that functions for d-q transformation `adqtr()` and calculation of terminal power by `agenpow()` are now placed in the first parallelizable loop. Depending on the cost of communications - determined by experiments - the second parallelizable loop may be advantageously executed in serial.

7.3.3 Decoupled Newton Method

The implementation of the Decoupled Newton method is similar to the Trapezoidal method except for the need to form the Jacobian matrix. The parallelizable version is given in text box 7.7.

```
for (t = t1; t <= t2; t += dt) {
   /* SERIAL PART */
   if (olditer > 3)
      anjac1 ();       /* form J1 submatrix          */

   iter = 0;
   iteration_point:
   /* PARALLELIZABLE LOOP 1*/
   for (area = 1; area <= iareano; area++) {
      adqtr ();     /* d-q transformation          */
      agenpow ();   /* machine terminal powers     */
      if (iter == 0)
         atrmc1();  /* non-integrable part         */
      atrmv1 ();    /* integrable part             */
      aginj ();     /* current injection           */
      bbsol ();     /* phase 1 of network solution */
   }

   /* SERIAL PART */
   sol2x ();        /* bordered matrix solution    */

   /* PARALLELIZABLE LOOP 2 */
   for ( area <= iareano; area++) {
      bbso2 ();     /* phase 2 of network solution */
      convchk ();   /* check convergent            */
   }
   if ( != converged ) iter++ goto iteration_point
}
```

Text box 7.7: Decoupled Newton method - parallelizable version

Function anjac1 that forms the Jacobian submatrix J_1 could be parallelized, but, since it is seldom required, executing it in serial does not effect speedup.

7.3.4 Full Newton Method

The parallelizable version of the Full Newton method is similar to the Decoupled method but with additional function anjac2 to form Jacobian submatrices J_2 and J_3 (see table box 7.8). Also the current injection function aginj is slightly modified to account for voltage changes.

```
for (t = t1; t <= t2; t += dt) {
   /* SERIAL PART */
   if (olditer > 3)
      anjac1 ();       /* form J1 submatrix           */
      anjac2 ();       /* form J2 and J3 submatrices */

   iter = 0;
   iteration_point:
   /* PARALLELIZABLE LOOP 1*/
   for (area = 1; area <= iareano; area++) {
      adqtr ();     /* d-q transformation          */
      agenpow ();   /* machine terminal powers     */
      if (iter == 0)
         atrmc1();  /* non-integrable part         */
      atrmv1 ();    /* integrable part             */
      aginj ();     /* current injection           */
      bbso1 ();     /* phase 1 of network solution */
   }

   /* SERIAL PART */
   sol2x ();        /* bordered matrix solution    */

   /* PARALLELIZABLE LOOP 2 */
   for ( area <= iareano; area++) {
      bbso2 ();     /* phase 2 of network solution */
      convchk ();   /* check convergent            */
   }
   if ( != converged ) iter++ goto iteration_point
}
```

Text box 7.8: Full Newton method - parallelizable version

7.3.5 Serial Slowdown Factor

The profile of the parallelizable versions of the four algorithms described in the previous sections are summarized in table 7.1 in terms of percentage of parallelizable parts, maximum theoretical speedup - by applying Amdhal's law - and number of parallelizable loops.

Algorithm	% of parallel part	Maximum theoretical speedup	No. of parallelizable loops
Modified Euler	90	10.0	3
Trapezoidal	92	12.0	2
Decoupled Newton	93 *62	14.0 2.6	2
Full Newton	93 *63	14.0 2.5	2

* Including formation of the Jacobian matrix

Table 7.1: Profile of the parallelizable versions

Table 7.1 shows that all methods have large proportions of parallelizable parts - 90 % or greater. In both Newton-based methods, if a particular iteration requires the formation of the Jacobian matrix, the proportion of the serial part increases, thus reducing the theoretical speedup. Since recalculation of the Jacobian matrix is seldom required, the theoretical speedup is close to those figures given without the reevaluation of the Jacobian matrix. The S^2 factor for each algorithm is determined using the 348-bus system and shown in table 7.2.

Algorithms	Serial version (sec.)	Parallelizable version (sec.)	S^2 factor
Modified Euler	5.75	7.70	1.34
Trapezoidal	6.12	8.65	1.41
Decoupled Newton	8.50	9.90	1.16
Full Newton	8.48	10.18	1.20

Table 7.2: The S^2 factor of the algorithms

The partitioning has a significant effect on the speed of the Trapezoidal method and has the least effect on the Decoupled Newton method. Since the overall effect of partitioning on all methods is small - the S^2 factor is below 2.0 - it indicates that all methods have potential for parallelization (IEEE Committee Report [1992]).

7.4 SHARED MEMORY VERSION

A natural extension to the solution of the partitioned problem is the shared memory multiprocessor implementation. As described in chapter 3, a shared memory multiprocessing environment can be simulated on a uniprocessor system under the UNIX operating system. There are two reasons for having such a simulation. Firstly, much of the earlier and current work on the applications of parallel processing to the transient stability problem has been done on shared memory systems (Chai, et. al. [1991]) and therefore this type of simulation can be used to assess the performance of the method described in this thesis on such multiprocessing systems. Secondly, it is envisaged that the class of parallel processors that power utilities are likely consider acquiring when upgrading their present serial networked computer systems will be the shared memory systems because of they are easier to program then the distributed memory systems. Furthermore, many (Chai, et. al. [1991], La Scala, et. al. [1990, 1991]) have shown that Massively Parallel Processors will be grossly underutilized in solving many power system problems. In this section the shared memory examples are based on the use of C compiler pragmas under IRIX operating systems. But the actual codes for simulations are based on the methods described in chapter 3 and 5.

7.4.1 Pragmas

Compiler pragmas are a mechanism added in C for including compiler directives in the code (Kernighan and Ritchie [1988]). Pragmas do not change the semantic meaning of the code, and any compiler is free to ignore pragmas that are not recognized (Bauer [1992]). The general pragma format is as

follows:

```
#pragma directive [modifier (list ...) ]
{
      statements
}
```

A simple loop:

```
for ( i = 0; i <= max; i++)
      a[i] = b[i] + c[i];
```

can be parallelized using pragmas by:

```
#pragma parallel
#pragma shared (a, b, c)
#pragma byvalue (max)
#pragma local (i)
{
   #pragma pfor iterate (i = 0; max; 1)
      {
         for ( i = 1; i <= max; i++)
               a[i] = b[i] + c[i];
      }
}
```

The statement `#pragma parallel` defines the parallel region - bounded by the curly brackets. The modifier `#pragma shared()` declares all variables in the bracket to be shared by all threads. The `#pragma byvalue()` tells the compiler to both share the variables inside the bracket in all threads and pass by value rather than by reference. To declare variables local to all threads, `#pragma local()` is used. Other pragmas are `#pragma numthreads()` which specifies the number of threads to operate in parallel, `#pragma if (condition)` provides means to specify condition for parallel or serial executions. For regulating parallel loops `#pragma pfor()` is used. The `iterate` modifier with `#pragma pfor` in the example, assigns the initial value, final value and step size of the loop. For more detail on pragma and its applications see Bauer [1992]. In this thesis we use some simple pragmas, to illustrate the shared memory version of the algorithms.

7.4.2 Modified Euler Method

```
#pragma parallel
#pragma share()
#pragma byvalue (t1, t2, dt, maxarea)
#pragma local (area)
{
   for (t = t1; t <= t2; t += dt) {
      #pragma pfor iterate (area=1; maxarea; 1)
      {
         for (area = 1; area <= maxarea; area++) {
            bbso2_0 (area);
            adqtre_0 (area);
            aeul1_0 (area);
            aginjn_0 (area);
            bbso1_0(area);
         }
      }

      #pragma one processor
      {
         sol2x ();
      }

      #pragma pfor iterate (area=1; maxarea; 1)
      {
         for (area = 1; area <= maxarea; area++) {
            bbso2_0 (area);
            adqtrne_0 (area);
            aeul2_0 (area);
            aginje_0 (area);
            bbso1_0(area);
         }
      }

      #pragma one processor
      {
         sol2x ();
      }
   }
}
```

Text box 7.9: Modified Euler method - shared memory version using parallel pragmas

The shared memory version of the Modified Euler method is shown in text box 7.9. To reduce the number of parallelizable loops to two (see text box 7.4), function `bbso2_0` is now located in the first loop. A new pragma, `#pragma one processor` is introduced to temporarily allow serial execution. Most functions are parallelized except the triangular solution of the bordered

matrix. In the for loop bounded by `pfor pragma()` several conditions are observed:

- the order of `area` execution does not change the outcome,
- the order of the function calls must be strictly followed.

7.4.3 Trapezoidal Method

The shared memory version of the Trapezoidal method using fixed point iteration is shown in text box 7.10. In this case only one parallel loop is used.

```
#pragma parallel
#pragma share()
#pragma byvalue (t1, t2, dt, maxarea)
#pragma local (t, area)
{
   for (t = t1; t <= t2; t += dt) {
      iteration_point:

      #pragma pfor iterate (area = 1; area <= maxarea;
1)
      {
         for (area = 1; area <= maxarea; area++) {
            bbso2_0 (area);
            adqtr_0 (area);
            agenpow_0 (area);
            aintup_0 (area);
            atrmc_0 (area);
            atrmv_0(area);
            aginj_0 (area);
            bbso1_0(area);
         }
      }

      sol2x ();
      if (!= converged) goto iteration_point;
   }
}
```

Text box 7.10: Trapezoidal method - shared memory version using parallel pragmas

Also note that for the Trapezoidal method the serial part has been minimized to one bordered matrix solution.

7.4.4 Decoupled Newton Method

```
#pragma parallel
#pragma share()
#pragma byvalue (t1, t2, dt, maxarea)
#pragma local (area)
{
   for (t = t1; t <= t2; t += dt) {
      #pragma one processor
      {
         if (olditer > 3) anjac1 ();
      }

   iteration_point:

      #pragma pfor iterate (area = 1; maxarea; 1)
      {
         for (area = 1; area <= maxarea; area++) {
            bbso2_0 (area);
            adqtr_0 (area);
            agenpow_0 (area);
            aintup_0 (area);
            atrmc1_0 (area);
            atrmv1_0(area);
            aginj_0 (area);
            bbso1_0(iarea);
         }
      }

      #pragma one processor
      {
         sol2x ();
      }
   }
}
```

Text box 7.11: Decoupled Newton method - shared memory version using parallel pragmas

The shared memory version of the Decoupled Newton method (see text box 7.11) is similar to the Trapezoidal method except for an additional serial part that calculates the element of the Jacobian matrix J_1 - `anjac1()`.

To evaluate the effect of each function on the speed of computation when parallelized, a shared memory simulation was performed. The CPU time of the parent process was measured. This time includes all serialized parts and the time spent by the parent process on its share of parallelized parts but does not include time spent by individual child process. Therefore,

this measure can provide the relative effectiveness of each function when parallelized.

Parallelized Functions	Parent process CPU time (sec)	Time Difference (sec)
adqtr	10.80	
adqtr+agenpow	10.62	0.18
adqtr+agenpow+aintup	10.62	0.00
adqtr+agenpow+aintup+atrmc1	10.05	0.57
adqtr+agenpow+aintup+atrmc1+atrmv1	7.67	2.38
adqtr+agenpow+aintup+atrmc1+atrmv1+ aginj	4.88	2.79
adqtr+agenpow+aintup+atrmc1+atrmv1+ aginj+bbso1	3.85	1.03
adqtr+agenpow+aintup+atrmc1+atrmv1+ aginj+bbso1+bbso2	2.95	0.90

Table 7.3: Decoupled Newton method - Effect of parallelizing functions

From table 7.3, there are two functions that contribute to significant speedup when parallelized - `atrmv1` and `aginj`. Function `atrmv1` solves for the states during each iteration that involves solving $J_1 \Delta X = -F$. Function `aginj` is used to calculate current injections that includes evaluating non-linear load variation for each bus. The combined effect of parallelizing network solution functions `bbso1` and `bbso2` is also significant.

7.4.5 Full Newton Method

```
#pragma parallel
#pragma share()
#pragma byvalue (t1, t2, dt, maxarea)
#pragma local (area)
{
   for (t = t1; t <= t2; t += dt) {
      #pragma one processor
      {
         if (olditer > 3){
            anjac1 ();
            anjac2 ();
         }
      }

   iteration_point:

      #pragma pfor iterate (area = 1; maxarea; 1)
      {
         for (area = 1; area <= maxarea; area++) {
            bbso2_0 (area);
            adqtr_0 (area);
            agenpow_0 (area);
            aintup_0 (area);
            atrmc1_0 (area);
            atrmv2_0(area);
            aginj_1 (area);
            bbso1_0(iarea);
         }
      }

      #pragma one processor
      {
         sol2x ();
      }
   }
}
```

Text box 7.12: Full Newton method - shared memory version using parallel pragmas

The shared memory version of the Full Newton method - shown in text box 7.12 - is similar to the Decoupled Newton method except for an additional function `anjac2`, a different function for state solution - `trmv2` - and a slightly modified current injection calculation function `aginj1`. Table 7.12 shows the effectiveness of each function as they are parallelized using shared memory simulation.

Parallelized Functions	Parent process CPU time (sec)	Time Difference (sec)
adqtr	9.37	
adqtr+agenpow	9.37	0.00
adqtr+agenpow+aintup	9.33	0.04
adqtr+agenpow+aintup+atrmc1	9.20	0.13
adqtr+agenpow+aintup+atrmc1+atrmv1	7.85	1.35
adqtr+agenpow+aintup+atrmc1+atrmv1+ aginj+bbso1	3.55	4.30
adqtr+agenpow+aintup+atrmc1+atrmv1+ aginj+bbso1+bbso2	2.60	0.95

Table 7.4: Full Newton method - Effect of parallelizing functions

From table 7.4, the combined effect of functions `atrmv1`, `aginj`, `bbso1` and `bbso2` contribute significantly to speedup.

7.4.6 Shared Memory Simulations

To simulate the shared memory versions on a uniprocessor system, the `#pragma pfor` and `area` loop are replaced by process creation function `process_fork()`. The shared memory simulation is best described by referring to the DCPS command file shown in text box 7.13.

```
%-----------------------------------------
% Shared Memory Simulation
%-----------------------------------------
chco,1.2,1.5,1.5,0.01,0.001,0.001,0.01,0.01
timer,u
redn,tnp95.sol
redmn,tnp95f.mac
reba;
rene;
ybus;
bpow;
iniv;
inie;
moym;
cinj;
ginj;
copm;
dblk;
fac2,slu
caps,2,6981,4
caps,2,8911,4
caps,2,7911,4
bbin;
shvl;
arproc,5
tsol,24,0.0,0.21,0.01,1
asht,slu,8201,0.0,1000.0,0.0
fac2,slu
bbin;
nbso,0.0
shdq;
tsol,24,0.22,0.28,0.01,1
rsht,slu,8201,0.0
fac2,slu
bbin;
nbso,0.0
shdq;
tsol,24,0.29,0.5,0.01,1
```

Text box 7.13: DCPS Command file for shared memory simulations

There are some DCPS functions in text box 7.13 that deserve further explanation. Function `timer` with parameters u specifies that timing probes measure the calling process cpu time and subtract the child cpu times. In profiling the shared memory program on a uniprocessor, the system time is excluded. There are many variables that need to be shared among the processes as well as the need to define semaphores and barriers and this is performed using function `shvl`. Static scheduling is performed by function

arproc with a parameter that specifies the number of processes. For the shared memory Trapezoidal method the pragmas are replaced by a process creation function as shown in text box 7.14.

```
for (t = t1; t <= t2; t += dt) {
   iteration_point:
   pid = process_fork(no_of_processes);
         bbso2_0 (pid);
         adqtr_0 (pid);
         agenpow_0 (pid);
         aintup_0 (pid);
         atrmc_0 (pid);
         atrmv_0(pid);
         aginj_0 (pid);
         bbsol_0(pid);
   process_join (pid, nproc);
   sol2x ();
   if (!= converged) goto iteration_point;
}
```

Text box 7.14: Trapezoidal method - shared memory simulation version

Each function carries an argument pid which is the process identifier number. Function **process_fork()** returns process identifier pid for each child process. Each child process will execute all the functions independently and immediately after its creation. When each child process reaches function **process_join()**, it is killed but the parent or calling process will wait until all child processes are destroyed before executing function **sol2x()**. The shared memory simulation was performed on the 348-bus system and the execution times for the various methods as the number of processes are increased are shown in table 7.5, 7.6, 7.7 and 7.8 (see DCPS command file in text box 7.13).

No. of processes	Execution time (sec.)	Speedup
1	31.27	1.00
2	20.33	1.54
3	15.65	2.00
4	13.63	2.29
5	9.47	3.30

Table 7.5: Modified Euler method - shared memory speedup

No. of processes	Execution time (sec.)	Speedup
1	44.97	1.00
2	29.57	1.52
3	22.80	1.97
4	16.82	2.67
5	13.87	3.24

Table 7.6: Trapezoidal method - shared memory speedup

No. of processes	Execution time (sec.)	Speedup
1	38.25	1.00
2	24.77	1.55
3	18.07	2.12
4	14.37	2.66
5	10.75	3.56

Table 7.7: Decoupled Newton method - shared memory speedup

No. of processes	Execution time (sec.)	Speedup
1	38.13	1.00
2	24.83	1.56
3	18.18	2.09
4	15.33	2.49
5	10.73	3.55

Table 7.8: Full Newton method - shared memory speedup

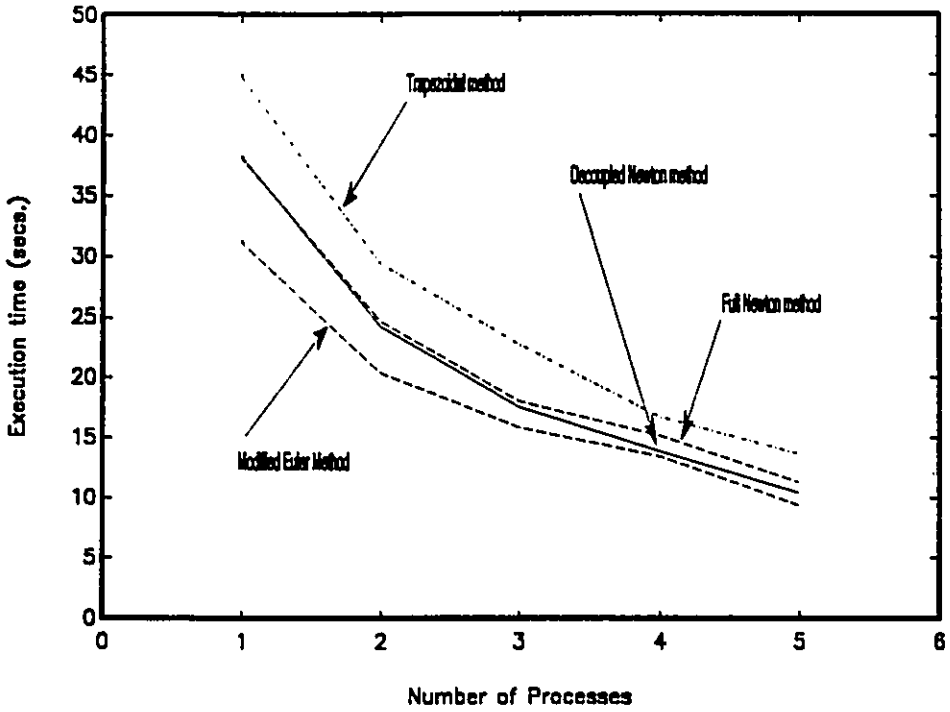

Figure 7.3: Simulated shared memory execution speed vs no. of processes - various parallelized algorithms.

The execution speed of the various methods, as the number of processes is increased, is summarized in figure 7.3. The Euler method, although the fastest, has lower speedup than the Newton based methods. There is very little difference in speed between the Decoupled and Full Newton methods.

7.5 RPC TECHNIQUE ON A CLUSTER OF WORKSTATIONS

In this section, the implementation of the parallel distributed transient stability calculations on a cluster of workstations using the RPC technique is described. The concept is the same as the shared memory version. However, additional tasks are necessary to physically move data between client and servers. To illustrate the process of converting the shared memory to the RPC version, the Modified Euler and the Decoupled Newton methods are used (see text box 7.9 and 7.11). Converting other algorithms will follow the same procedure. It is more convenient to refer to the command file shown in text

box 7.15 to explain the technique.

7.5.1 Initialization of Servers

Before the start of the time-loop, it is more advantages to initialize all stateful servers with some static data. The static data are divided into machine and network categories as follows:

(i) **Machine related**

(a) All states that are already initialized

(b) Machine mechanical and electrical data

(ii) **Network related**

(a) LU factors of the network matrix

(b) Some network data that identify the generator terminals

The initialization task is performed using function `tssvc` immediately after all data structures that need to be transported are properly allocated using `alorpc`. Function `tssvc` is executed in parallel. A function `sendall` is used to send to all servers the new voltage vector after network switching.

```
%=============================================================
% Tr118b.com: BBDF network solution for Transient
% Stability using RPC
% tsol 31 for the Modified Euler method
% tsol 33 for the Decoupled Newton method
%=============================================================
chco,1.2,1.5,1.5,0.01,0.001,0.001,0.01,0.01
timer,us
server,0,intake        %Specify servers
server,1,rod
server,2,filter
redn,s118b.sol
redm,s118bf.mac
reba;
rena;
ybus;
bpow;
iniv;
inie;
moym;
cinj;
ginj;
copm;
dblk;
fac2,slu
caps,2,4,4
caps,2,69,4
caps,2,80,4
bbin;
shvl;
arproc,3
alorpc;                %allocate space for RPC data
tssvc;                 %initializes servers
tsol,31,0.0,0.04,.01,1
asht,slu,69,0.0,1000.0,0.0
fac2,slu
bbin;
nbso,0.0
shdq;
sendall;               %send vectors to servers
tsol,31,0.05,0.15,.01,1
rsht,slu,69,0.0 fac2,slu
bbin;
nbso,0.0
shdq;
sendall;
tsol,31,0.16,1.12,.01,1
```

Text box 7.15: Command file for the RPC version transient stability calculations

7.5.2 Algorithm and Protocol Definition

The required protocol definition is best discussed by studying the parallel distributed algorithms that require client-server communications. Algorithm 7.1 shows the required computation and communications for the Modified Euler method.

Algorithm 7.1: Modified Euler method - RPC technique

(A) INITIALIZATION
(i) Send static data to all servers

(B) TIME SOLUTION
for (time = t_1 ; time <= t_2 ; time =+ h) {

- **(ii) All servers perform initial estimate calculations on their scheduled area**
- **(iii) Client performs initial serial computation for part of the first BBDF network solution**
- **(iv) All servers perform final estimate calculations on their scheduled area**
- **(v) Client performs initial serial computation for part of the second BBDF network solution**
- **(vi) Record the results**

}

In the initialization, a single data structure is required to contain all static and initialized data. In the subsequent communications - during the time solution or after network switching - different structures are used that contain only those data to be exchanged. Referring to algorithm 7.1, during each time step two client-server communications are required - in step (ii) and (iv). During both instances the client has to only send boundary bus voltages to all servers. In step (iii) and (v), the client requires the summation part of equation (5.7) - rewritten in equation 7.1 - from all servers.

$$\hat{B}_0 = B_0 - \sum_{i=1}^{m} A_{0i}^T A_i^{-1} B_i \qquad (7.1)$$

The term B_0 can be calculated in the client or one of the servers. If it is

calculated in a server then it has to be sent to the client where a different data structure is required from the one sent by the client. After the final estimate - step (v) - states are required in the client for the purpose of recording. Therefore, the servers will send a structure that contains final states and voltages. To summarize, the following structures are required:

(a) Boundary bus voltage - from client to server

(b) Boundary bus currents and part of equation 7.1 - from server to client

(c) States, voltages and boundary bus currents - from server to client.

Algorithm 7.2 gives the RPC version of the Decoupled Newton method. In the Modified Euler method two client-server communications are required during each time step but in the Decoupled Newton method the communication frequency is not known but depends on the number of iterations to achieve convergence. The structure of the data to be sent from client to servers during initialization is the same as in the Modified Euler method. After formation of the Jacobian matrix J_1 in step (ii), its LU factors are to be sent to all servers. In step (iii), all servers require boundary bus voltages from the client. To compute the boundary bus voltage in step (iv), the client requires boundary bus currents and the summation part of equation 7.1.

Algorithm 7.2: Decoupled Newton method - RPC technique

(A) INITIALIZATION

(i) Send static data to all servers.

(B) TIME SOLUTION

for (time = t_1 ; time <= t_2 ; time =+ h) {

(ii) Form Jacobian if required and send to all servers
Iteration_point:
(iii) Servers perform computation
(vi) Client performs serial computation
(v) If not converged goto iteration_point.
(v) If converged record states

}

All states and voltages to be recorded are also required by the client in step (iii). The client-server data structures required for the Decoupled Newton method are as follows:

(a) LU factors of the Jacobian matrix J_1 - from client to servers

(b) Boundary bus voltages - from client to servers

(c) States, voltages, currents and part of equation 7.1 - from servers to client

An additional data structure is required for sending voltages, LU factors of the admittance matrix, and currents after the switching operation in the client. A simplified version of the protocol definition is shown below for illustration.

PROTOCOL DEFINITION FOR TRANSIENT SIMULATION

```
/* SIZE OF VECTORS
*/
const NBMAX = 1001;      /* maximum bus no. */
const NMMAX = 251;       /* maximum machine no. */
const NYMAX = 20001;     /* maximum lu factors no. */
const IARMAX = 30;       /* maximum area */
const NJACSZ1 = 3514;    /* Jacobian factors size */
const NJACSZ2 = 35140;

/* MESSAGE STRUCTURE FOR INITIALIZATION OF SERVERS IN
TRANSIENT STABILITY - FROM CLIENT TO SERVER
*/
struct tsinit {
   int     id;              /* server/process id          */
   int     nm;              /* no of machines             */
   int     nb;              /* no of buses                */
   .
   .
   float   pv<1>;           /* real load voltage exponent*/
   float   qv<1>;           /* active load volt. exponent*/

   /* machine data */
   int     imt<NMMAX>;
   int     imn<NMMAX>;
   int     iunit<NMMAX>;
   float   h<NMMAX>;
   float   xd<NMMAX>;

   .
   .
   float   xleak<NMMAX>;
   float   sat1<NMMAX>;
   float   sat2<NMMAX>;
```

```
    /* machine states */
    macstat mc;
    float     eqr<NMMAX>;
    float     eqx<NMMAX>;
    float     vq<NMMAX>;
    .
    .
    float     del<NMMAX>;
    float     cid<NMMAX>;

    /* exciter data */
    int       ibe<NMMAX>;
    int       ety<NMMAX>;
    float     tr<NMMAX>;
    float     ka<NMMAX>;
    .
    .
    float     efdmin<NMMAX>;
    float     efdmax<NMMAX>;

    /* pss data */
    int       ibp<NMMAX>;
    int       ipsty<NMMAX>;
    .
    .
    float     vsmax<NMMAX>;
    float     vsmin<NMMAX>;

    /* exciter + pss states */
    float    devdt<NMMAX>;
    float    defddt<NMMAX>;
    .
    .
    float    vsi<NMMAX>;

    /* lu factor */
    int      iyl<NBMAX>;
    int      iyu<NBMAX>;
    .
    .
    int      ilr<NYMAX>;
    float    yldx<NBMAX>;

    /* Off-diagonals of admittance matrix */
    int      ibst<NBMAX>;
    .
    .
    int      ibmrw<NYMAX>;
    float    bmx<NYMAX>;

    /* schedule vector, voltage and current */
    int      schv<IARMAX>;
    float    vr<NBMAX>;
    float    vx<NBMAX>;
    float    ir<NBMAX>;
    float    ix<NBMAX>;
}
```

```
/* MESSAGE STRUCTURE FOR SENDING BOUNDARY VOLTAGE FROM CLIENT
TO SERVERS - FROM CLIENT TO SERVER
*/
struct tsvol {
   int   id;
   float vr<NBMAX>;
   float vx<NBMAX>;
}

/* MESSAGE STRUCTURE THAT CONTAINS PART OF EQUATION 7.1 AND
   CURRENTS - FROM SERVER TO CLIENT
*/
struct tsborz {
   int id;
   /* borzt */
   float bozrt<NBMAX>;
   float bozxt<NBMAX>;
   /* current */
   float   ir<NBMAX>;
   float   ix<NBMAX>;
}

/* MESSAGE STRUCTURE THAT CONTAINS VOLTAGE, CURRENTS AND
STATES FROM SERVER TO CLIENT
*/
struct tsdat {
   int   id;
   /* voltage */
   float vr<NBMAX>;
   float vx<NBMAX>;

   /* current */
   float   ir<NBMAX>;
   float   ix<NBMAX>;

   /* machine states */
   float    vq<NMMAX>;
   float    vd<NMMAX>;
   .
   .
   float    vxk<NMMAX>;
   float    ciq<NMMAX>;
   float    cid<NMMAX>;

   /* exciter + pss states */
   float   devdt<NMMAX>;
   float   defddt<NMMAX>;
   .
   .
   float   ev<NMMAX>;
   float   eiv<NMMAX>;
};
```

```
/* MESSAGE STRUCTURE THAT CONTAINS FACTOR OF THE JACOBIAN
MATRIX J1 - FROM CLIENT TO SERVER
*/
struct jacfac {
   int      id;
   .
   .
   float    cuvr1<NJACSZ2>;
   float    duvr1<NJACSZ2>;
};

/* MESSAGE STRUCTURE THAT CONTAINS LU FACTOR OF ADMITTANCE
MATRIX - FROM CLIENT TO SERVER
*/
struct lufac {
   /* lu factor */
   int      id;
   .
   .
   float    yuox<NYMAX>;
   float    yudr<NBMAX>;
   float    yudx<NBMAX>;
   float    yldr<NBMAX>;
   float    yldx<NBMAX>;
};


/* MESSAGE STRUCTURE USED BY FUNCTION SENDALL() TO SEND ALL
   NECESSARY VECTORS FROM CLIENT TO SERVER AFTER SWITCHING
*/
struct sndat {
   /* lu factor */
   int      id;
   int      kku;
   int      kkl;
   int      iyl<NBMAX>;
   int      iyu<NBMAX>;
   .
   .
   float    yudx<NBMAX>;
   float    yldr<NBMAX>;
   float    yldx<NBMAX>;

   /* dq vectors */
   float    vq<NMMAX>;
   float    vd<NMMAX>;
   float    ciq<NMMAX>;
   float    cid<NMMAX>;

   /* bus voltage and current */
   float vr<NBMAX>;
   float vx<NBMAX>;
   float ir<NBMAX>;
   float ix<NBMAX>;

   /* Off-diagonals of Ybus */
   int   kkoff;
   int   ibst<NBMAX>;
```

```
   int   ibmrw<NYMAX>;
   int   ibmcl<NYMAX>;
   float bmr<NYMAX>;
   float bmx<NYMAX>;
};

/* SERVER PROCEDURES DECLARATION */
program ATSCPROG {
  version ATSCVERS {
    void   TSSVC(tsinit)        = 13;
    tsdat  TSOL33(tsvol)        = 14;
    void   TSANJAC1(jacfac)     = 16;
    void   SENDALL(sndat)       = 17;
    tsborz TSOL31N(tsvol)       = 18;
    tsdat  TSOL31F(tsvol)       = 19;
   } = 1;
} = 0x20000001;
```

Procedure TSSVC carries structure tsinit and contacts all servers for initialization. In each server, function TSSVC_1 (note that the corresponding procedures in servers are given extension _1) will unpack the structure and load all allocated vectors.

During each time step of the Decoupled Newton method, procedure TSOL33 carries structure tsvol that contains boundary bus voltage and returns structure tsdat. After the Jacobian matrix J_1 is formed in the client, its factors are dispatched to all servers using procedure TSANJAC1. Immediately after the switching operation, an iterative network solution is performed in the client (see text box 7.15). The resulting voltage, LU factors and currents, contained in structure sndat, will be dispatched by procedure SENDALL simultaneously to all servers.

In the Modified Euler method two procedures are used during each time step. To do the initial estimate, procedure TSOL31N that carries structure tsvol and returns structure tsborz is used. In the final estimate, procedure TSOL31F returns tsdat that contains the states and voltages to be recorded.

7.5.3 Modified Euler Method

The RPC codes of the Modified Euler method for the client is given in text box 7.16 and for the server in text box 7.17. As can be observed in both parallelized parts, concurrent server calls are achieved by using heavyweight processing - Unix `fork()` function. All functions within the parallelized parts (see text box 7.9) now reside in the server. Functions `tsol31n_1()` and `tsol31f_1()` calls the server program as shown in text box 7.17.

```
for (t = t1; t <= t2; t += dt) {

      /* INITIAL ESTIMATE */
      /* Parallelized part 1 */
      pid = process_fork(nproc);
         tsol31n (pid);
      process_join(nproc, pid);

      /* Serialized part 1 */
      sol2x();

      /* FINAL ESTIMATE */
      /* Parallelized part 2 */
      pid = process_fork(nproc);
         tsol31f (pid);
      process_join(nproc, pid);

      /* Serialized part 2 */
      sol2x();
}

void tsol31n (pid)
{
   pack_tsvol();
   for ( i = 0; i < nproc; i++) {
      if (pid == i)
         tsborz = tsol31n_1(tsvol, cl[i]);
   }
   unpack_tsborz();
   return;
}

void tsol31f (pid)
{
   pack_tsvol();
   for ( i = 0; i < nproc; i++) {
      if (pid == i)
         tsdat = tsol31f_1(tsvol, cl[i]);
   }
  unpack_tsdat();
}
```

Text box 7.16: Modified Euler method - RPC version - client side

Function pack_tsvol () packs the boundary voltage vector to structure tsvol. Function unpack_tsdat () copies voltages, currents and states that are contained in structure tsdat to corresponding vectors in the client.

```
tsborz *tsol31n_1(tsvol)
{
   unpack_tsvol();

   bbso2_0(pid);
   adqtre_0(pid);
   aeul1_0(pid);
   aginjn_0(pid);
   bbsol_0(pid);

   pack_borz();
   return(borz);
}


tsdat *tsol31f_1(tsvol)
{
   unpack_tsvol();

   bbso2_0 (pid);
   adqtrne_0 (pid);
   aeul2_0 (pid);
   aginje_0 (pid);
   bbsol_0(pid);

   pack_tsdat();
   return(tsdat);
}
```

Text box 7.17: Modified Euler method - RPC version - server side

As shown in text box 7.17, function unpack_tsvol copies the boundary bus voltage vector from structure tsvol. Procedure tsol31n_1 returns structure borz after it is packed using function pack_borz. The procedure that performs final estimate - tsol31f_1 - returns structure tsdat.

7.5.4 Decoupled Newton Method

```
for (t = t1; t <= t2; t += dt) {
      if (olditer > 3) {
            anjac1 ();
            tsanjac1_0();
      }
      iter_point;
      /* === Parallelized part === */
      pid = process_fork(nproc);
         tsol33 (pid);
      process_join(nproc, pid);
      /* === Serialized part by client    === */
      sol2x ();
      if(!= converged) goto iter_point;
}

void tsol33 (pid)
{
      pack_tsvol();
      for ( i = 0; i < nproc; i++) {
      if (pid == i)
         tsborz = tsol33_1(tsvol, cl[i]);
      }
      unpack_tsborz();
      return;
}

void tsanjac1_0()
{
   pack_jacfac();
   pid = process_fork(nserver);
      for (i = 0; i < nproc ; i++) {
         if (pid == i)
            tsanjac1_1(jacfac, cl[i]);
       }
   return;
}
```

Text box 7.18: Decoupled Newton method - RPC version - Client side

In the Decoupled Newton method (see text box 7.18), procedure `tsanjac1_0` is used to despatch factors of the Jacobian matrix J_1 to all servers. As in the Modified Euler method, all parallelized functions are now transferred to the server side as shown in text box 7.19.

```
tsdat *tsol33_1(tsvol)
{
      unpack_tsvol();

      bbso2_0(pid);
      adqtr_0(pid);
      agenpow_0(pid);
      aintup_0(pid);
      atrmc1_0 (pid);
      atrmv1_0(pid);
      aginj_0 (pid);
      bbso1_0(pid);

      pack_tsdat();
      return(tsdat);
}

void *tsanjac1_1(jacfac)
jacfac *setsj;
{
      unpack_jacfac();
      return;
}
```

Text box 7.19: Decoupled Newton method - RPC version - server side

7.5.5 Results of Simulations

In this section, simulation results using the 348-bus and 840-bus systems are used to assess the speedup obtained using the RPC technique with respect to the number of servers.

No. of servers	Execution time (sec.)	Speedup
1	21.03*	1.00
2	18.40	1.14
3	13.85	1.51
4	13.42	1.56
5	14.50	1.45

* Serial algorithm speed

Table 7.9: Modified Euler method - RPC version speedup for 348-bus system

No. of servers	Execution time (sec.)	Speedup
1	33.35*	1.00
2	20.33	1.64
3	12.20	2.73
4	11.73	2.84
5	15.51	2.15

* Serial algorithm speed

Table 7.10: Decoupled Newton method - RPC version speedup for 348-bus system

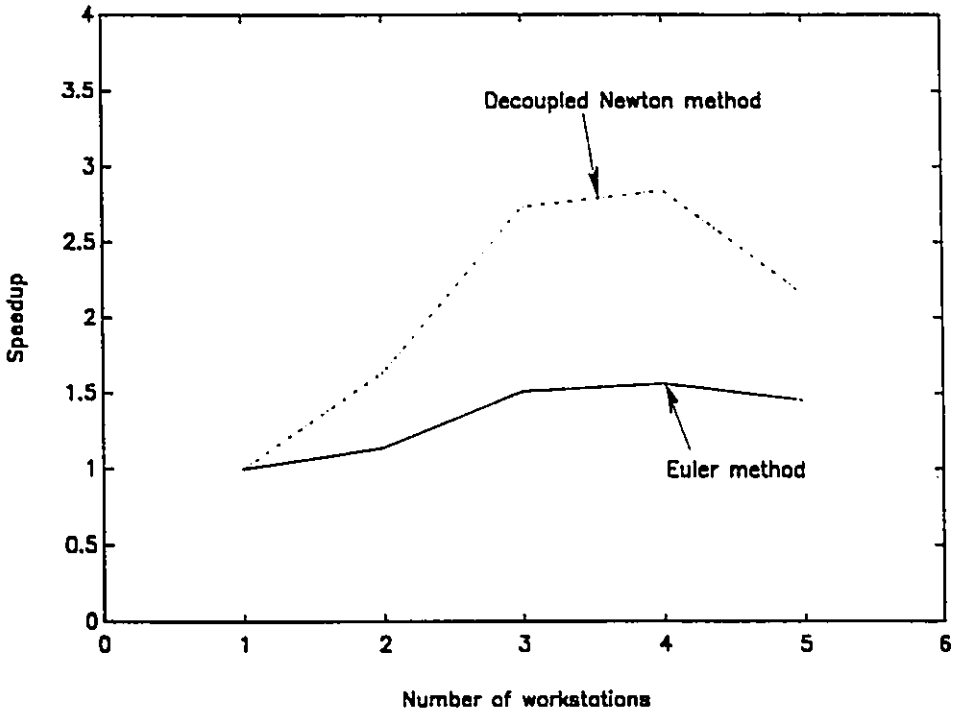

Figure 7.4: Speedup for 384-bus system using the RPC technique

The results of simulations on the 348-bus system using the Modified Euler and the Decoupled Newton methods are given in tables 7.9 and 7.10 respectively. In figure 7.4, the corresponding speedup of both methods are plotted. From the results, the Decoupled Newton method performs better than that of the Modified Euler method in terms of speedup. In both methods, peak speedup occurs with four workstations and when five workstations are used the speedup is reduced. Better performance by the Decoupled Newton method is largely due to a fewer number of communications in comparison to the Modified Euler method. In the Modified Euler method, there are two communications during each time step. In the Decoupled Newton method the number of communications varies from 1 to 4 during each time step. However, due to good convergent characteristics of the Newton-based method, most time steps require only one communication.

No. of servers	Execution time (sec.)	Speedup
1	36.30*	1.00
2	20.20	1.79
3	17.00	2.13
4	15.52	2.34
5	12.40	2.93
6	11.30	3.21
8	12.07	3.00
10	12.40	2.93
12	12.35	2.93

* Serial algorithm speed

Table 7.11: Modified Euler method - RPC version speedup for the 840-bus system

No. of servers	Execution time (sec.)	Speedup
1	52.30*	1.00
2	39.90	1.31
3	27.85	1.88
4	24.30	2.15
5	18.12	2.89
6	14.78	3.54
8	16.40	3.20
10	16.80	3.10
12	16.90	3.10

* Serial algorithm speed

Table 7.12: Decoupled Newton method - RPC version speedup for the 840-bus system

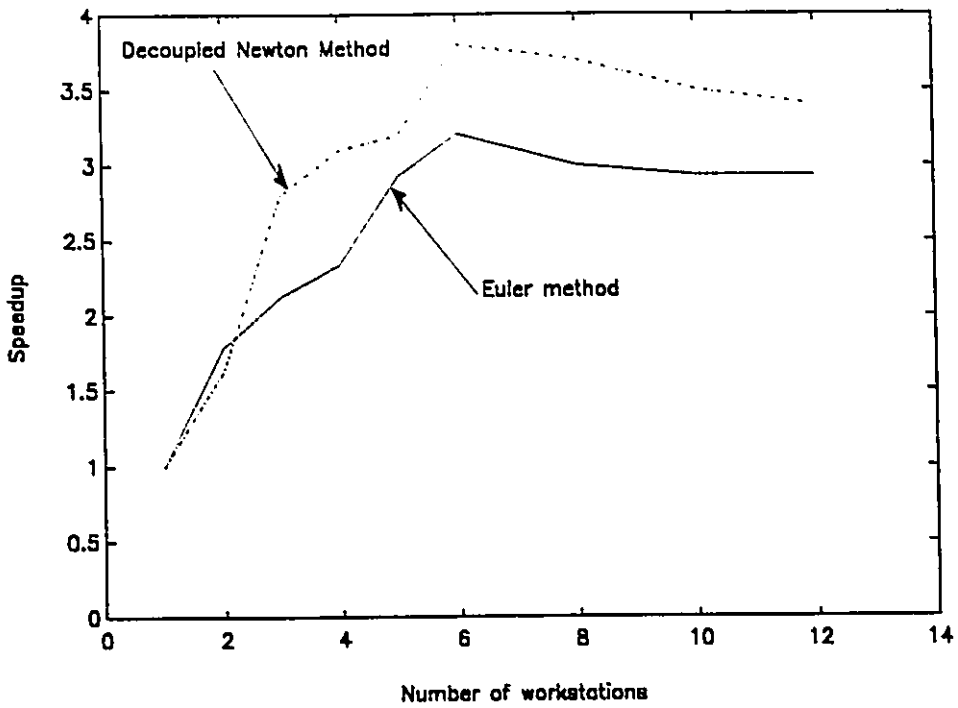

Figure 7.5: Speedup for the 840-bus system using the RPC technique

Tables 7.11 and 7.12 give the results of simulations on the 840-bus system using the Modified Euler and the Decoupled Newton methods respectively. The speedup for both methods are plotted in figure 7.5

The performance of the Modified Euler method improves significantly on the 840-bus system as compared to the one on the 384-bus system. As far as speedup is concerned, the Decoupled Newton method achieves slightly higher maximum speedup. In both methods, the speedup reduces as the number of workstations becomes greater than six. As the number of workstations increases, all gains are counterbalanced by additional communication overhead.

7.6 SELECTIVE BLOCK REFACTORIZATION

A typical stability simulation requires at least two refactorizations of the network admittance matrix immediately after switching operations. The first instant is when the contingency is applied; e. g. short circuit, generator tripping. The second instant is when the applied contingency is removed due to relay action or clearance of the fault. Therefore, a simulation can be divided into five stages as follows:

i. Solution without any contingency (typically from 0 to 0.1 seconds).

ii. Application of contingency (instantaneous).

iii. Solution for the duration of the applied contingency (50 - 200 milliseconds).

iv. Removal of contingency (instantaneous).

v. Solution until the end of the required period (2 - 5 seconds).

The contribution of stage ii and iv to the total solution time for the 348-bus and 840-bus systems are shown in table 7.13.

Power Systems	1.5 sec. run	3.0 sec. run
348-bus	8.0%	3.7%
840-bus	18.0%	10.0%

Table 7.13: Contribution of refactorizations to simulation time

For the 840-bus system, the contribution of stage ii and iv is fairly significant so that it is worthwhile to execute these parts in parallel distributed manner. However for long simulation runs - 10 sec. to 20 sec. ; that are typical in planning studies, the refactorization parts are less significant. But in system operations where assessment of transient stability is required, a 5 second simulation period is typical.

In chapter 5, the block factorization method using the RPC technique has been described. The same method for initial factorization is applied here. But a subsequent change in topology or initiation of short circuit requires only

refactorization of that affected block.

7.7 CONCLUSIONS

In this chapter, parallelization of transient stability algorithms have been described. The transient stability algorithm is rich in parallelisms as indicated by the large percentage of parallelizable parts - more than 90%. Shared memory implementation of the algorithms have also been described and, through uniprocessor simulations, reasonable speedup is obtained. This chapter has also described in detail the corresponding parallel distributed transient stability algorithms and their implementation on a cluster of workstations using the RPC client-server model. The speedup obtained is reasonable in view of the heavy communication overheads in the hardware setup.

CHAPTER 8
CONCLUSIONS

8.1 SUMMARY

In this thesis, state-of-the-art parallel transient stability algorithms and their implementation have been examined. To meet the objective of the thesis - *to speedup transient stability computation using parallel distributed processing* - two approaches have been followed. Firstly, parallel distributed versions of the serial transient stability algorithms have been developed, retaining advanced techniques such as sparsity and the use of implicit integration methods. Secondly, these algorithms have been successfully implemented on computer systems that are widely available in power utilities - clusters of workstations. The choice of workstation clusters for high-performance computing has recently emerged as a cost-effective alternative to supercomputers and state-of-the-art parallel computers (Donggara [1993]). As a result of the second approach, the developed algorithms are also easily adaptable to shared memory multiprocessor systems that are likely the next generation of computers to be acquired by power utilities.

The nature of this thesis is highly practical. Realistic systems of moderate size are used in most simulations. But handling such practical systems requires code sophistication similar to that found in production grade power system simulation programs. For this work, the DCPS program was developed. It is sufficiently general for use in other projects, e. g., McFarlane

[1993][1] - and has the potential for further development.

The parallelizing of transient stability problems is performed in three stages - (i) the problem is first partitioned - in a coarse-grain manner, (ii) algorithms for solving the partitioned problem are then developed, and (iii) the communication elements for implementation on a cluster of workstations are incorporated. This parallelizing process is general and therefore can be applied to any problem that can be partitioned.

This thesis uses the slow-coherency based technique which is particularly suitable for power system simulation. The slow-coherency method was first developed as a structurally based technique for aggregation of generators for dynamic system reduction. In this thesis, the slow-coherency partitioning technique is extended to include load buses. This inherently partitions the network at weak transmission links. These partitions are physically meaningful and lead naturally to parallel distributed computation.

The transient stability problem is rich in parallelism. The sets of differential equations for generators in the system are interconnected only through the transmission network. When the network is suitably ordered into areas that are weakly linked, these areas can be treated more or less independently.

Lack of current implementation of parallel computing is associated with a somewhat slow acceptance of parallel distributed computing technology in the power industry. So far there is no production grade dynamic simulation package that uses multiprocessing. The main reasons are - (i) there is no clear direction in the choice of hardware, and (ii) most current parallel algorithms require drastic modifications to the present serial algorithms and codes. In this thesis, it has been shown that with reorganisation, standard simulation code can be made to run efficiently on a multiprocessing system.

---

[1] In this work, the DCPS environment has been used to develop programs for the Artificial Neural Network (ANN) calculations and to automate simulations involving multiple contingencies used for training the ANN.

8.2 SUMMARY OF CONTRIBUTIONS AND MAIN CONCLUSION

The main contributions of the thesis are summarized as follows:

- A program for power system simulation that can handle up to 1000 buses and 225 generators has been developed (chapter 2). This program can perform load flow, eigenanalysis - with classical machine models - for coherency grouping and transient stability simulations. The program has been designed in modular form and is suitable for use in future research.
- State-of-the-art techniques in parallel transient stability calculations have been investigated (chapter 3). The parallelisms exploited in each method are identified and their advantages and disadvantages are discussed.
- A new slow coherency network partitioning method that includes load buses has been developed and its application for studying power system structure has been demonstrated (chapter 4).
- The partitioning technique (chapter 4) and the RPC (Remote Procedure Call) client-server model (chapter 5) constitute a new parallel distributed solution of power system networks that has been successfully implemented on a cluster of workstations.
- The communication issues in conjunction with the use of RPC techniques and their relationship to task scheduling have been described, analyzed and implemented (chapter 6).
- Parallel distributed transient stability algorithms have been developed and implemented on a cluster of workstations (chapter 7). Both explicit and implicit based transient stability algorithms have been parallelized. Both shared and distributed memory - RPC version - implementations have been described. Speedup in the order of 3 times have been obtained in simulations of practical systems.

The primary achievement of this thesis is the development of tested parallel distributed transient stability algorithms. The algorithms are ready for conversion to production grade for three reasons:

(i) *The algorithms require straight forward modification to the existing serial simulation algorithms.*

(ii) *The algorithms' implementation on widely available networked workstations enables immediate application without costly hardware acquisition.*

(iii) *The algorithms can be easily modified for future shared memory multiprocessor systems.*

8.2 SUGGESTIONS FOR FURTHER WORK

Heavyweight processing (HWP) - client reproducing itself - is used in conjunction with the RPC implementation in this thesis. Lightweight processing (LWP) may be a better choice (Bloomer [1991]). In lightweight processing, another thread of execution starts in the same process address space. However, currently the ONC (Open Network Consortium) RPC does not support LWP. When it is supported, it would be worthwhile to use LWP and compare the results with that of using HWP.

An emerging trend at the moment is the use of coordinating packages in high-performance network computing. PVM (Parallel Virtual Machine) is now being tested by many universities[2]. One of the promising development on the PVM is availability of its version of BLAS (Basic Linear Algebra Software) routines called BLACS. Adapting the algorithms in this thesis to be implemented under the PVM environment would make a very interesting comparison that may prove to be beneficial.

The techniques developed in this thesis are also applicable to other power system problems such as load flow and eigenanalysis. In the DCPS program itself, developing the parallel distributed version of the coherent grouping program would be worthwhile since this task is very computationally intensive.

The main obstacle to further improvement in speedup when using a cluster of workstations for parallel distributed computation is the communication overhead. However, as discussed in chapter 5, many of the

---

[2] At McMaster University, the Computer Information Services (CIS) is testing PVM together with BLACS (Personal communication with Dr. Fred Kus of the CIS [July 1993])

software layers of the ISO 7-layer model can be bypassed through the use of specialized/customized hardware - network interface card (NIC). Apart from using the customized NIC, this work is also expected to involve low-level programming in the application layer. If cost-effective, this technique will assist in exploiting the potential of available networked workstations for parallel distributed computation.

The most exciting current development in computer application is low cost personal desktop symmetric multiprocessors; i.e., shared memory multiprocessor systems. This is brought about by a new operating system known as Windows NT that is likely to supersede the DOS operating system used in personal computers. This new operating system is able to handle multiprocessing based on shared memory interprocessor communications (Udell [1993], Groves [1993]). Implementing parallel distributed algorithms in this thesis on such a shared memory multiprocessor system will require very little additional effort. Results of such implementation could be compared with other published results. At the same time such experience may point to further optimization with respect to the implementation of the algorithms.

The application of the Waveform Relaxation method (WRM) to the transient stability problem developed by Crow and Ilic' [1990] was never implemented on parallel processors. This method which is described in chapter 3 of this thesis is suitable for solution using a distributed memory multiprocessor system such as a cluster of workstations because the problem is partitioned in a coarse-grained manner. In DCPS, the WRM algorithm has been implemented (see **Appendix A**) - using both the Modified Euler and the Trapezoidal methods instead of the Backward Euler method used by Crow and Ilic' [1990] - but arranged for serial execution. It is therefore a worthwhile effort to implement the parallel distributed version of the WRM on a cluster of workstations so that the performance of this technique can be assessed. Such an assessment could provide incentive to further develop the Waveform Relaxation method for transient stability solution.

APPENDIX A

DCPS REFERENCE

This appendix contains detailed description of all DCPS functions. It begins with a description on how to invoke a DCPS program and executing commands. Following that lists of functions and their descriptions common to all program modules and those functions that specifically belong to LF, CG and TS programs are given.

A.0 Running DCPS

While in the directory containing any of the DCPS programs - LF, TS, or CG - one can invoke say the TS program by typing:

```
%ts
```

Then the environment changes to that of the DCPS as shown below.

```
+-----------------------------------+
|                TS                 |
|     Transient Stability Solver    |
+-----------------------------------+

ts>
```

One can then type any command word described in this appendix which will be executed by the program after pressing the [ENTER] key.

A.1 Common functions. These functions are used by all programs suites.

```
bpow     calculates bus powers - rectangular form
chco     changes program constants
clear    clears screen
clem     clears memory
date     shows system date
exit     exits DCPS
fac2     factorizes complex linklist stored matrix
help     displays on screen help
orde     orders network data - Scheme 1
ord1     orders network data - Scheme 1, all machines first
quit     terminates execution of command file
reba     renumbers tie-line buses to the last area
redn     reads network data
rena     orders buses by area using ord1
rene     orders buses by area using orde
runf     runs command file
shf1     shows ascii file
shsc     shows linklist stored matrix
shvo     shows bus information
smcv     displays LU factors
sol1     performs backward/forward solution of YV = I
title    shows DCPS program suite title
ybus     forms admittance matrix
```

Text box A.1: Common functions

A.2 LF functions. These functions are used in LF module for load flow calculations.

bpop	calculates bus powers using voltage and admittance matrix in polar form
chbr	changes branch data
chbs	changes bus data
dnlf	performs decouple newton load flow
dpqv	forms control mismatch vector in decoupled newton load flow
fcb1	factorizes matrices B1 and B2 in the fast decoupled load flow
fdbm	forms matrices B1 and B2 in fast decoupled load flow
fdcv	forms control mismatch vector in fast decoupled load flow
fdlf	performs fast decoupled load flow
flst	initializes bus voltages in flat start
gslf	performs gauss-seidel load flow
init	same as flst
insr	inserts element in real linklist stored matrix
jalf	forms load flow jacobian for decoupled newton method
piqi	calculates some parts of the jacobian - power related
shbr	shows branch data
shbs	shows bus data
shsr	shows real linklist stored matrix
sol2	performs triangular solution using real factors
svlf	saves load flow, i.e, network data
vpo	converts voltage from rectangular to polar forms
ybpo	converts admittance matrix from rectangular to polar forms

Text box A.2: LF functions

A.3 CG functions. These functions are used in CG module for coherency grouping calculations.

```
cohg     performs coherency grouping
eis1     performs eigenanalysis of r chosen eigenvalues on
         state matrix
eise     performs eigenanalysis on state matrix
for1     forms augmented system matrix - method 2
forj     forms augmented system matrix - method 1
fst1     forms state equation - method 2
fste     forms state equation - method 1
fybe     forms expand admittance matrix
inic     calculates initial values on classical machines
sor4     sorts eigenvalues and eigenvectors
svar     saves load flow data including coherent areas
trv1     transforms network eigenvectors to deltaV and
         deltaTheta - method 2
trve     transforms network eigenvectors to deltaV and
         deltaTheta - method 1
```

Text box A.3: CG functions

A.4 TS functions. These functions are used in TS module for transient stability calculations.

alorpc	allocates memory in RPC operation
arproc	static scheduling of servers
asht	adds shunt to a bus
asti	stores all initial values for reiteration in the WRM
bbin	performs stage 1 calculation in BBDF network solution
bbso	performs stage 2 calculation in BBDF network solution
bigs	solves YV = I using gauss-seidel block iterative method
bite	solves YV = I using gauss-jacobi block iterative method
bklp	shared memory version of blocked LU decomposition
bklr	RPC version of blocked LU decomposition - self scheduling
bklrp	RPC version of blocked LU decomposition - static scheduling
bklu	blocked LU decomposition
caps	captures states in transient simulation
cinj	calculates load model constant, appends initial load admittance to the admittance matrix and initialize current injection vector
copb	copy block of admittance matrix
copm	copy admittance matrix
dblk	extracts diagonal block of admittance matrix
dqtr	performs DQ to dq transformation
fac5	factorizes the linklist stored block diagonal admittance matrix
ginj	calculates current injection vector
inie	calculates initial values for exciters
iniv	calculates initial values for machines
iniw	initializes waveform for the first window in the WRM
insb	inserts the factorized block into the LU factor of the admittance matrix
lsnc	factorizes linklist stored blocked form admittance matrix
lsserver	lists all called servers
moym	modifies the admittance matrix to include the machine equivalent source admittance
mstp	sets or changes multiples of time steps by area
nbso	performs iterative block network solution
nsol	performs iterative network solution
redm	reads machine, exciter, pss and governor data
rena	reorders buses by area using Tinney Scheme 1 - generator first
rene	reorders buses by area using Tinney Scheme 1
rsht	removes shunt installed by asht
rsln	restores branch removed by rvln
rvln	removes branch

Text box A.4: TS functions

sendall	sends vectors to servers after switching operation in client
server	declares server to bind
shdq	d-q transformation using shared variables
shed	shows exciter and pss data
shie	shows exciter initial values
shin	shows machine initial values
shmd	shows machine data
shou	shows captured output file
shvl	declares common shared variables
smat	sends admittance matrix to servers
svou	saves captured output to disk file
svcwk	performs BBDF network solution - RPC method with static scheduling
tsol	performs transient solution
wges	initializes states and voltages for current window
wsvc	performs network solution NBDF method using RPC
wsol	performs Waveform Relaxation Method (WRM) transient solution

Text box A.5: TS functions - Continued

A.5 Details of functions and usage

The detail description of each function including examples of usage are given in this section. The following notations are used:

|	or - denotes choice
[*item*]	denote optional item in the square bracket
[;]	placing semicolon after a command suppresses any output
italic	denotes command parameters
command file	refers to any DCPS command file
italic bold	refers to a data field in DCPS data format
bold	command or function name

There are eight examples of command files following the description of each command.

---

alorpc
alorpc[;]

This function allocates space for structures to be transported between client and servers in the RPC transient stability calculations. See example 7.

arproc
arproc,[*no_of_servers*]

This functions produces scheduling vector that allocates tasks amongst the given number of servers. Scheduling is based on loop-splitting technique. See example 7.

asht
asht,*syb*|*slu*,*bus_no*,*r*,*x*,*T*

This function adds a shunt of $r + jx$ p.u. to the given bus. One has option to either add to the original admittance matrix using *syb* or to its copy using *slu*. *T* indicates the time the shunt to be applied. A value of zero means the shunt is to be applied immediately. See example 3.

asti
asht[;]

In the waveform relaxation method, the initial values of states anchor the state and voltage waveforms to the starting time during each iteration. This function stores the initial values of each state and voltage. See example 4.

bbin
bbin[;]

This function performs the initial stage of the network solution in BBDF ordering. See example 7.

bbso
bbso,[*current_offset*]

This function performs the final stage of the network solution in BBDF ordering. *Current_offset* can be used to offset all elements - real and imaginary components - in the current injection vector. See example 8.

bigs
bigs[;]

This function performs network solution using the Gauss-Seidel block iterative method. See example 1.

bite
bite[;]

This function performs network solution using the Gauss-Jacobi block iterative method. See example 1.

bklp
bklp,*slu*|*syb*,*no_processes*

This function factorizes either *slu* - the copy of original admittance matrix - or *syb* - the original admittance matrix - under simulated shared memory environment in a uniprocessor system. The tasks are divided amongst the given number of processes. See example 8.

bklr
bklr[;]

This function factorizes *slu* - the copy of original admittance matrix - in parallel distributed manner using the RPC technique. The method uses self scheduling technique. See example 8.

bklrp
bklrp[;]

This function factorizes *slu* - the copy of original admittance matrix - in parallel distributed manner using the RPC technique. The method uses static scheduling technique based on loop splitting. See example 8.

bklu
bklu,*slu*|*syb*

This function factorizes either *slu* or *syb* in block-by-block manner. See example 8.

bpop
bpop,[;|*0*|*1*]

This function calculates power at each bus when both the admittance matrix and voltage are expressed in polar form. See example 2. Parameter *0* suppresses printout and *1* causes the output to be saved to a disk file. See example 6.

bpow
bpow,[;|*0*|*1*]

This function calculates power at each bus when both the admittance matrix and voltage are expressed in rectangular form. See example 1. Parameter *0* suppresses printout and *1* causes the output to be saved to disk file. See example 5.

caps
caps,*device_no,bus_no,state_no*

This function selects state or control to be captured for plotting in transient stability calculations. *Device_no* 1 is for bus quantities, 2 for basic generator states and 3 for exciter states. See example 3.

chbr
chbr,*from_bus,to_bus,circuit_no*

This function is used to edit the branch data. This is usually used in manual mode.

chbs
chbr,*bus_no*

This function is used to edit the bus data. It is usually used in manual mode.

chco
chco,*acceleration_factor,p_power_exponent,q_power_exponent,tolerance_for_states,tolerance_for_voltage,tolerance_for_waveforms*

This function is used to change programs constants according to the list in the parameters.

cinj
cinj[;]

This function calculates load model constant K_p and K_q. It also appends initial load admittance of each bus to the network admittance matrix and initializes current injection vector. See example 1 and 3.

clear
clear[;]

This function clears the screen. See example 1.

clem
clem[;]
This function clears the memory for restarting the DCPS. It is called automatically by function exit.

cohg
cohg,*no_of_areas*
This function performs slow-coherency grouping specified by the number of areas. See example 5.

copb
copb[;]
This function copies a block of blocked diagonal admittance matrix.

copm
copm[;]
This function copies the original admittance matrix syb which is in blocked diagonal form to a working matrix slu. See example 1 and 4.

date
date[;]
This function displays system date.

dblk
dblk[;]
This function extracts the diagonal block of the NBDF or BBDF ordered admittance matrix and copy it to structure slu. See example 1 and 4.

dnlf
dnlf,*maximum_iteration,skip_interval*
This function performs decoupled Newton load flow. Iteration is limited to *maximum_iteration* and *skip_interval* is the number of iteration before the Jacobian matrices and their refactorization be performed. See example 2.

dpqv
dpqv[;]
This function forms the control mismatch vector in decoupled Newton load flow.

dqtr
dqtr[;]
This function transforms network reference axis quantities - voltage and current - to rotor reference axis. See example 3.

eis1
eis1,*no_of_areas*
This function performs eigenanalysis of *no_of_areas* chosen eigenvalues on the state matrix. See example 5.

eise
eise,[;]
This function performs eigenanalysis of on the whole state matrix. See example 5.

exit
exit[;]
This function exits DCPS. It clears all allocated space, shared memory, semaphores and free all loaded servers. But server daemons must be removed from the system manually.

fac2
fac2,*syb*|*slu*
This function factorizes the admittance matrix. Choosing *syb* as the parameter will factorize the original admittance matrix and using *slu* will factorize the copy of the admittance. See examples 1 and 3.

fac5
fac5[;]
This function factorizes the block diagonal admittance matrix. See example 4.

fcb1
fcb1[;]
This function factorises the matrices B_1 and B_2 in the Fast Decoupled load flow method. See example 6.

fdbm
fdbm[;]
This function forms matrices B_1 and B_2 in the Fast Decoupled load flow. See example 6.

fdcv
fdcv[;]
This function forms the control mismatch vector in the Fast Decoupled load flow. See example 6.

fdlf
fdlf,*maximum_no_of_iterations*
This function performs Fast Decoupled load flow given the maximum number of iterations. See example 6.

flst
flst[;]
This function is used to reset all voltages in load flow for flat start. See example 7.

for1
for1[;]
This function forms the augmented system matrix without explicit inversion of admittance matrix. See example 5.

forj
forj[;]
This function forms the augmented system matrix but with explicit inversion of admittance matrix. See example 5.

fst1
fst1[;]
This function forms the state matrix following **for1**. See example 5.

fste
fste[;]
This function forms the state matrix following **forj**. See example 5.

fybe
fybe[;]
This function converts the complex admittance matrix to real expanded form. See example 5.

ginj
ginj,[*iprst*]
This function calculates current injection for all buses. Print code *iprst* of *0* suppresses printing and any value greater than *1* will cause the current injection vector to be displayed on the screen.

gslf
gslf,[*iterno*]
This function performs Gauss-Seidel iterative load flow calculations with *iterno* as the maximum number of allowable iterations.

help
help,[*function*]
This function provides on-screen help. Without specifying *function*, list of all available commands are displayed. Providing the *function* or *command* name gives detail description of that particular *function*.

inic
inic[;]
Performs machine initialization but using classical machine model. See example 5.

inie
inie[;]
Performs exciters and pss initialization in transient stability calculations. See example 3.

init
init[;]
Performs the same function as flst - flat starting of load flow.

iniv
iniv[;]
Performs initial value calculations for all generators. See example 1.

iniw
iniw[;]
Performs initialization of state and voltage waveforms in the Waveform Relaxation method for transient stability calculations. See example 4.

jalf
jalf[;]
Forms the required Jacobian matrices in the Decoupled Newton load flow method.

lsserver
lsserver,[;]
List servers that have been given client handle using command server.

moym
moym[;]
Modifies the network admittance matrix to include the machine equivalent source admittance. See example 3.

mstp
mstp,*area,multiple*
Sets the *multiple* integration time step for area *area* in the Waveform Relaxation method. See example 4.

nbso
nbso,[*current_offset***]**
Performs iterative network solution in transient stability calculations based on the BBDF ordering. Give a value in *current_offset* to offset current injection of all buses. See example 7.

nsol
nsol,[*current_offset*]
Performs iterative network solution in transient stability calculations. Give a value in *current_offset* to offset current injection of all buses. See example 3.

orde
orde[;]
This function renumbers network buses according to Tinney Scheme 1. See example 3.

ord1
ord1[;]
This function renumbers network buses according to Tinney Scheme 1 but arranges all machine buses the first before the load buses. See example 2.

quit
quit[;]
This function causes the *command file* execution be terminated.

reba
reba[;]
By using this function the boundary buses are grouped into a new area that creates the block bordered submatrix in BBDF ordering. See example 7.

rena
rena[;]
This function renumbers the buses by area using information *iarea* field of the load flow data. The ordering is based on **ord1**. See example 4.

redm
redm,*machine_file*
This function reads the machine data from *machine_file* written in DCPS format. See example 5.

redn
redn,*loadflow_file*,*ieee*|*dcps*|*pti*
This function reads the load flow data from *loadflow_file* written in the given format - IEEE common format, DCPS format or PTI loadflow format. See example 6.

rene
rene[;]
This function renumbers the buses by area using information *iarea* field in the load flow data. The ordering is based on **orde**. See example 7.

runf
runf,*command_file*
This function runs file *command_file* under DCPS environment.

rsht
rsht,*syb*|*slu*,*bus_no*,*T*
This function removes shunt of $r + jx$ p.u. from bus *bus_no* placed by asht. One has choice to either remove from the original admittance matrix using *syb* or from its copy using *slu*. *T* indicates the time the shunt to be removed. A value of zero means the shunt to be removed immediately. See example 3.

rsln
rsln,*from_bus*,*to_bus*,*circuit_no*,*T*
This function restores line removed by **rvln**. Line is specified by *from_bus* no. to *to_bus* no.. *T* is the time the line to be restored. *T* of zero means the line to restored immediately.

rvln
rvln,*from_bus*,*to_bus*,*circuit_no*,*T*
This function removes line specified by *from_bus* no. to *to_bus* no.. *T* is the time the line to be removed. *T* of zero means the line to removed immediately.

sendall
sendall[;]
This function sends new vectors to servers after contingency switching performed in the client. See example 8. See example 7.

server
server,*number*,*server_name*
This function creates client handle for the *server_name* and referred to by the given number. See example 7.

shbr
shbr,*from_bus*,*to_bus*,*circuit_no*
Using this function one can show the information on a branch specified by *from_bus* number to *to_bus* number and the circuit *circuit_no*.

shbs
shbs,*bus_no*
One uses this function to show on the display monitor information on the given bus number. This function is normally used in manual mode.

shdq
shdq[;]
When states are shared amongst processes, one uses this function for d-q transformation instead of function dqtr. See example 7.

shed
shed[;]
This function displays exciter data on the screen.

shfl
shfl,filename
Any ascii file can be displayed on the screen using this function.

shmd
shmd[;]
Use this function to display machine data on the screen.

shou
shou[;]
All captured states in time are kept in the memory and one uses this function to display those states.

shsc
shsc[;]
This function displays linklist stored complex admittance matrix in standard matrix format.

shsr
shsr[;]
This function displays linklist stored real matrix in standard matrix format.

shvo
shvo[;]
This function displays bus information - voltage, power, area etc.

shvl
shvl[;]
In applications using IPC, (interprocess communications) there are common variables that need to be shared amongst processes. This function declares those variables. See example 7.

shie
shie[;]
This function shows initialized exciter states on the screen.

shin
shin[;]
This function shows initialized machine states on the screen.

smat
smat,[*no_of_servers*]
This function send the admittance matrix to the given number of servers before factorization is performed. See example 8.

sol1
sol1[;]
This function performs triangular solution on complex matrix and right hand side vector.

sol2
sol2[;]
This function performs triangular solution on real matrix and right hand side vector.

sor4
sor4[;]
This function sorts eigenvalues and eigenvectors in ascending order. See example 5.

svar
svar,[*filename*]
This function saves the load flow data using the new area number as determined in coherency grouping.

svlf
svlf,[*filename*],[*ieee*|*dcps*|*pti*]
This function saves the load flow files into diskfile *filename* using the specified format.

svou
svou,[*filename*]
This function save the captured states trajectories to diskfile *filename*. This file is saved to *gpf format* that can be directly plotted by gp program.

svcwk
svcwk,[*current_offset*]
This function performs RPC-based BBDF network solution using static scheduling technique. *Current_offset* is used to perturb every element of the current injection vector.

trv1
trv1[;]

This function transforms extended eigenvector - network voltage - to delta-V and delta-ϴ coordinate for a r given smallest eigenvalues.

trve
trv1[;]

This function transforms extended eigenvector - network voltage - to delta-V and delta-ϴ coordinate for all eigenvalues. See example 5.

tsol
tsol,[*method*],[*from_time*],[*to_time*],[*time_step*],[*print_option*]

This function performs transient solution using the given method over the specified time period. The integration step is also specified and one can choose to display the captured states on the screen by specifying *print_option* of 1. The methods are as follows:

1 Serial Modified Euler method
2 Serial Trapezoidal method with linear iteration
3 Serial Decoupled Newton method with dishonest iteration
4 Serial Full Newton method with dishonest iteration
11 Serial Modified Euler method - all calculations area-wise
12 Serial Trapezoidal method with linear iteration - all calculations area-wise
13 Serial Decoupled Newton method with dishonest iteration - all calculations area-wise
14 Serial Full Newton method with dishonest iteration - all calculations area-wise
21 Serial Modified Euler method - shared memory version
22 Serial Trapezoidal method with linear iteration - shared memory version
23 Serial Decoupled Newton method with dishonest iteration - shared memory version
24 Serial Full Newton method with dishonest iteration - shared memory version
31 Serial Modified Euler method - RPC version
32 Serial Trapezoidal method with linear iteration - RPC version
33 Serial Decoupled Newton method with dishonest iteration - RPC version
34 Serial Full Newton method with dishonest iteration - RPC version

See example 3 and example 7.

vbpo
vbpo[;]

This function converts all bus voltages in rectangular from to polar form. See example 2.

wges

wges,[*from_time*],[*to_time*],[*minimum_time_step*]

In the waveform relaxation method, this function provide the guess waveform for each state and voltage. See example 4.

wsvc

svcwk,[*current_offset*]

This function performs RPC-based BBDF network solution using self-scheduling technique. *Current_offset* is used to perturb every element of the current injection vector.

wsol

wsol,[*method*],[*integ_method*],[*start_time*],[*to_time*],[*smallest_time_step*],[*print_option*]

This function performs waveform relaxation method transient solution using the given method over the specified time period. The integration step is also specified and one can choose to display the captured states on the screen by specifying *print_option* of 1. The *methods* are as follows:

1 Gauss-Jacobi iteration

2 Gauss-Seidel iteration

The *integ_methods* (Numerical integration methods) are as follows:

2 Trapezoidal with linear iteration

3 Decoupled Newton method

See example 4.

ybpo

ybpo[;]

This function converts the admittance matrix in rectangular form to polar form. See example 2.

ybus

ybus[;]

This function forms the network admittance matrix, *syb*. All examples use this function.

A.6 Examples of Command Files

```
%=============================================
% Bite.com: NBDF method both on one processor
%=============================================
clear;
redn,etp97.sol,dcps
redmn,aetp97.mac
rene;
ybus;
bpow;
iniv;
moym;
cinj;
ginj;
copm;
dblk;
fac2,slu
bite,0.0002       % bite can be replaced by bigs
```

Example 1: Network solution block iterative method

```
'=============================================
' dnlf.com: Run-file for decoupled Newton Load Flow
'=============================================
chco,0.001,,,,,,,
redn,tnp95.sol,dcps        'read network data
ordl;                      'ordering buses for minimum fill-ins
ybus;                      'form admittance matrix, rectangular form
ybpo;                      'convert admittance matrix to polar form
vbpo;                      'convert bus voltages to polar form
bpop,0                     'calculate bus power
dnlf,10,2
```

Example 2: Decoupled Newton Load flow calculations

```
%=========================================
% TStnp.com: Transient Stability Runfile
%=========================================
chco,1.2,1.5,1.5,0.05,0.001,,,,,,
redn,tnp95.sol
redmn,tnp95f.mac
orde;
ybus;
bpow;
iniv;
inie;
moym;
cinj;
ginj;
fac2,syb
caps,2,6981,4
caps,2,8911,4
caps,2,7911,4
caps,2,9931,4
tsol,2,0.0,.21,0.01,1
asht,syb,8201,0.0,1000.0,0.0
fac2,syb
nsol;
dqtr;
tsol,2,0.22,0.28,0.01,0
rsht,syb,8201,0.0
fac2,syb
nsol;
dqtr;
tsol,2,0.29,2.0,.01,1
```

Example 3: Transient stability solution

```
% wrnew.com: Waveform relaxation method for TS solution
chco,0.001,0.001,0.01,0.95,1.5,1.5,0.005
redn,snew.sol,dcps          %load network data file
redm,snew.mac               %load machine data file
rena;                       %renumber buses by areas
ybus;                       %form admittance matrix
bpow,0                      %calculate buspowers
iniv;                       %initialise states
moym;                         %add machine admittance to admittance
matrix
cinj;                       %add in equivalent load admittance
ginj;                       %calculate current injection vector
copm;                       %copy ybus to slu
dblk;                       %extract diagonal block of slu
fac5;                       %factorise slu
caps,2,20,4                 %mark states to capture
caps,2,2,4
caps,2,3,4
caps,2,22,4
mstp,3,2.0
mstp,4,2.0
asht,syb,9,0.0,1000.0,0.03
rsht,syb,10,0.14
iniw;
%WINDOW 1
asti;                              %store some initial values
wges,0.01,0.20,0.01                %guess states & voltage
wsol,1,3,0.01,0.20,0.01,0          %solve for each area - first window
%WINDOW 2
asti;
wges,0.21,0.40,0.01
wsol,1,3,0.21,0.40,0.01,0
%WINDOW 3
asti;
wges,0.41,0.60,0.01
wsol,1,3,0.41,0.60,0.01,0
%WINDOW 4
asti;
wges,0.61,0.80,0.01
wsol,1,3,0.61,0.80,0.01,0
%WINDOW 5
asti;
wges,0.81,1.0,0.01
wsol,1,3,0.81,1.0,0.01,0
%WINDOW 6
asti;
wges,1.01,1.2,0.01
wsol,1,3,1.01,1.2,0.01,0
svou;
END
```

Example 4: Waveform Relaxation Method for Transient stability solution

```
%============================================================
% Sample Run-Stream to do eigenanalysis for coherency grouping
%============================================================
redn,snew.sol,dcps
redm,snew.mac
orde;
ybus;
bpow,0
inic;
cinj;
fybe;
forj;
fst1;
eise;                    % can be replaced by : eis1,no_of_areas
sor4;
trve;
cohg,4
END
```

Example 5: Coherency grouping

```
%=====================================================
% fdlf.com: Run-file for fast decoupled Load Flow
%=====================================================
redn,tnp95.sol,dcps        'read network data
ord1;                     'ordering buses for minimum fill-ins
ybus;                     'form admittance matrix, rectangular form
fdbm;
fcb1;
ybpo;                     'convert admittance matrix to polar form
vbpo;                     'convert bus voltages to polar form
bpop,0                    'calculate bus power
fdlf,20
```

Example 6: Fast Decoupled Load flow calculations

```
%=============================================================
% Tretp.com: RPC Method for Transient Stability solution
%=============================================================
chco,1.2,1.5,1.5,0.01,0.001,0.001,0.01,0.01
timer,us
server,0,gasket
server,1,piston
server,2,crank
server,3,rad
server,4,carb
server,5,injector
server,6,clutch
server,7,plug
server,8,oilpan
server,9,manifold
server,10,filter
server,11,rod
redn,etp97.sol
redmn,aetp97f.mac
reba;
rene;
ybus;
bpow;
iniv;
inie;
moym;
cinj;
ginj;
copm;
dblk;
fac2,slu
caps,2,83,4
caps,2,269,4
caps,2,425,4
bbin;
shvl;
arproc,5
alorpc;
tssvc;
tsol,33,0.0,0.04,0.01,1
asht,slu,252,0.0,1000.0,0.0
fac2,slu
bbin;
nbso,0.0
shdq;
```

Example 7: RPC-based transient stability calculations

```
%=====================================================
% bklr.com: Factorization methods on blocked form
% admittance matrix
%=====================================================
chco,1.2,1.5,1.5,0.001,0.001,0.0005,0.001,0.001
redn,s118b.sol
redm,s118b.mac
reba;
rene;
ybus;
bpow;
iniv;
moym;
cinj;
ginj;
copm;
dblk;
% normal global factorization
%fac2,slu

% block-by-block factorization
%bklu,slu

% block-by-block factorization - simulated parallel shared
% memory
%bklp,slu,6

% block-by-block factorization - RPC parallel, distributed
server,0,gasket
server,1,crank
server,2,rad
server,3,carb
server,4,injector
server,5,clutch
arproc,1
smat;
%bklr;
bklrp;

% Network solution
bbin;
shvl;
bbso,0.1
```

Example 8: RPC-based LU factorization

APPENDIX B

ADDING FUNCTION TO DCPS

This appendix illustrates the steps required to add a new function to the TS program. The driver procedure (`main()`) of TS program is as follows:

```
/*=====================================================================*
| ts.c: parallel, distributed transient stability calculations.       |
| Program for algorithmic devlopment. Power Reasearch Laboratory      |
| McMaster Univ.                                                      |
| By Sallehhudin Yusof, 1992                                          |
*=====================================================================*/
#include "header.h"

main()
{
   char  cmd[80], comm[80], charzero[2];
   char  nwfl[12], runf[12], iformat[12], mcfl[12], rdfl[12];
   .
   .
   .

   int   nstr, nend, kstr, neaod, naod;
   int   pid, nproc, istrow, iareadiv, instno, procid, ishort, isht;
   float    rvln_time, ybrvr, ybrvx, bl, rsln_time, frtime, totime, dt,
   acfl, pvl, qvl, tolxl, tolvl, tolgl, tolwl, *vro, *vxo, offset,
   coffset, voffset, vrr, vxx, *rwr, *rwx;
   FILE * fl;
   long  foff;
   int   nrec;
   int   isum;
   int   itask;

   char  server_name[12];
   char  *sshared ();
   irredn = 0;
   irord1 = 0;
   irord2 = 0;
   irrena = 0;
   .
   .
   .
   irarproc = 0;
```

```
irsvcwk = 0;

para = ( char **)malloc( 12 * sizeof( char *) );
for ( i = 0; i < 12; i++)
   para[ i ] = ( char *)malloc( 20 * sizeof( char ) );
rmd = ( char **)malloc( 200 * sizeof( char *) );
for ( i = 0; i < 200; i++)
   rmd[ i ] = ( char *)malloc( 82 * sizeof( char ) );
strcpy(erword, "*** Error in executing ");
strcpy(promp, "aats>");
strcpy(charzero, "0");
taats();
acf = 1.2;
pv = 1.5;
qv = 1.5;
tolx = 0.01;
tolv = 0.0001;
tolg = 0.0001;
tolw = 0.01;
locked = 1;
unlocked = 0;
ishort = 0;
itcount = 0;
ioutano = 0;
while (TRUE) {
   nd = 1;
   printf ("aats>");
   cmd[0] = NULL;
   comm[0] = NULL;
   for (i = 0; i < 12; i++)
      para[i][0] = NULL;
   iprtfl[0] = NULL;
   gets(cmd);
   iprtfl[0] = cmd[4];
   iexcute = 0;
   if (cmd[0] != NULL) {
      intrnew(cmd, comm, para);
   }
   if (strcmp(comm, "runf") == 0 ) {
      strcpy(runf, para[1]);
      nd = 0;
      redf(runf, &nd, rmd);
   }
   for (krun = 0; krun < nd; krun ++) {
      if (nd > 1) {
         iexcute = 0;
         cmd[0] = NULL;
         comm[0] = NULL;
         for (i = 0; i < 12; i++)
            para[i][0] = NULL;
         iprtfl[0] = NULL;
         strcpy(cmd, rmd[krun]);
         iprtfl[0] = cmd[4];
         if (cmd[0] != NULL) {
            intrnew(cmd, comm, para);
         }
         if (cmd[0] != '%')
            printf("aats>%s\n", cmd);
      }
```

```
/* Functions Drivers */
if ( strcmp(comm, "help") == 0 ) {
   strcpy(comm, para[1]);
   if (comm[0] == NULL) {
      /* list of command words */
      help();
   } else {
      irhelp = 1;
   }
   iexcute = 1;
}
if (strcmp(comm, "exit") == 0 ) {
   if (irhelp == 0) {
      clem(nm, nb);
      iexcute = 1;
      exit(0);
   } else {
      printf ("exit - exit main() program\n");
      irhelp = 0;
   }
}

.
.
.
.

if ( strcmp(comm, "dblk") == 0 ) {
   if (ircopm == 1 && irhelp == 0) {
      dblk (&nb, &nelu);
      strcpy(pchr, para[1]);
      if (iprtfl[0] == ';') {
         strcpy(pchr, charzero);
      }
      if (pchr[0] == NULL ) {
         shsc(&nb, &nb, slu);
         iprt = 0;
      }
      irdblk = 1;
   } else
    {
      if (irhelp == 1) {
         printf ("dblk - extract diagonal block of slu\n");
         irhelp = 0;
      } else {
         printf("%s dblk: copm\n", erword);
      }
   }
   iexcute = 1;
}

/* A new function driver is added here */
if (strcmp(comm, "shfl") == 0 ) {
   if (irhelp == 0) {
      strcpy(rdfl, para[1]);
      shfl(rdfl);
      iexcute = 1;
      exit(0);
   } else {
      printf ("shfl,filename - display filename\n");
      irhelp = 0;
   }
```

distribution of design D.L. and L.L. F_x has the symmetric pyramid distribution shown in Figure 5.38a. At the ends of the bridge deck the seismically induced forces are approximately the same as the D.L. F_x ($\simeq 0.7 \times 10^6$ lb). At towers, the D.L. F_x is more than 7 times the values of F_x due to other loads.

The seismic induced shear force (Q_z) along the deck is (10-30%) of the D.L. Q_z, and (20-40%) of the L.L. Q_z, depending on the frequency content of the input motion. The locations of maximum combined D.L., L.L. and seismic Q_z are the ends of the bridge deck and the mid main span.

D.L. M_y is almost constant along the deck (2×10^6 lb.ft). L.L. M_y has two peaks, one in the side span (5.6×10^6 lb.ft)and another one in the middle section of the main span (4×10^6 lb.ft). The peak seismic M_y occurs at the same locations and is 65-100% of L.L. M_y in the case of LS1, and 30-50% of L.L. M_y for IS1 and HS1.

The above discussion shows that for 0.1g input the seismic bending moment can be large enough that it is a significant percentage of the dead and live loads. Other response quantities, Q_z and F_x, are dominated by D.L. and L.L. and the seismic component is a much smaller fraction of the total force. However, for ground motions larger than 0.1g, the seismic component will become an even larger percentage of the total force in the deck.

A longitudinally movable deck (MMMM from section 5.5) has a distribution of seismically induced F_x, described in section 5.5.3 and shown in Figure 5.39. In this case the seismic F_x is $\leq 10\%$ of D.L. F_x for the cases of IS1 and HS1, and $\approx$ 50% of D.L. F_x at the deck ends and 10% of D.L. F_x at the tower locations for LS1. Q_z due to LS1

The new function driver can then be placed in the `main()` procedure as shown in the earlier listing. The next step is to write the procedure `shfl.c`. A listing of `shfl.c` is shown below.

```
#include "header2.h"

void shfl(rdfl)
char rdfl[];
{
   /* function to display file on terminal */
   char buf[81],bufl[80],key[1];
   int i,j,k,nd;
   FILE *fileptr;
   fileptr = fopen(rdfl, "r");

   if (fileptr != NULL) {
      i = 1;
      k = 1;
      while ( !feof(fileptr) ) {
         fgets(buf,80,fileptr);
         for (j = 0 ; j <= 75; j++) {
            bufl[j] = buf[j];
         }
         bufl[75] = NULL;
         if (feof(fileptr)) {
            goto keluar;
         }
         printf("%3d %s", i, bufl);
         i++;
         k++;
         if (k >= 24 ) {
            printf(":");
            gets(key);
            if ( strcmp(key,"q") == 0) break;
            k = 1;
         }
      }
   }
   else  {
      printf ("File %s doesnot exist\n",rdfl);
   }
keluar:;
   fclose(fileptr);
   return;
}
```

The final step is to compile the TS program. The new function is added into the list in the `makefile`. The `makefile` is shown below with the new function appended.

```
CC = /usr/5bin/cc
atsc.h atsc_xdr.c atsc_svc.c atsc_clnt.c: atsc.x
   rpcgen atsc.x

objts = ts.o alo2.o alo3.o aloe.o asht.o bpow.o cinj.o clem.o      \
dqtr.o fac2.o fhsc.o ginj.o help.o ibno.o iniv.o ins1.o insc.o     \
insy.o intrnew.o moym.o njac1.o nsol.o ordl.o orde.c reco.o redf.o \
redm.o redn.o rsht.o rsln.o rvln.o shmdf.o shsc.o shsr.o           \
shvo.o smcv.o sol1.o sor1.o sor7.o svou.o titls.o trmc.o trmc1.o   \
trmv.o tsol.o ybus.o ymap.o zero.o fac3.o insbr.o rlo2.o           \
rlo3.o rloe.o euel.o eue2.o eul1.o eul2.o dqtrn.o ginjn.o caps.o   \
inie.o rena.o reba.o copm.o dblk.o insc2.o bbin.o alls.o facl.o    \
solbl.o inscx.o bklp.o lsnc.o copb.o parallel.o rlo4.o alo4.o      \
sreco.o atrmc.o aginj.o abbsol.o sollx.o abbso2.o adqtr.o alo5.o   \
rlo5.o atrmv.o aeul1.o aginjn.o bbso.o bbsol.o bbsc2.o aeul2.o     \
nbso.o adqtrne.o aginje.o adqtre.o sol2x.o shdq.o aintup.c         \
agenpow.o convchk.o rene.o bbinl.o bklu.o isvc.o wksv.o incl.o     \
wsvc.o arproc.o svcwk.o shfl.o                                     \
atsc_clnt.o atsc_xdr.o

ts: ${objts}
   cc -g -o aats ${objts} -lm
```

APPENDIX C

DCPS NETWORK AND GENERATOR DATA FORMAT

In this appendix, the DCPS data format for both the network and generators are described. Although DCPS program can read load flow information in IEEE common data format (IEEE Committee Report [1973]), it is however not suitable for use as prerequisite to transient stability calculations. The basic reason is that, the IEEE common format has limited decimal places for numbers. Consequently, solved load flow bus voltages and powers are truncated when stored again in the format. Therefore, more accurate format is required to overcome this deficiency. As for the generator data, simple format was chosen that separates the generator mechanical/electrical and control components.

C.1 Load Flow Data

The data is made up of three parts - (1) two fields for the number of buses and lines, (2) bus data and (3) line data. A sample data for the nine-bus system is shown in text box C.1. The meaning of each field of the data is explained next.

Text box C.1: DCPS sample of network data for the nine-bus system

```
  9   10
  1    3 1.0400000   0.0000000   0.7169828   0.1588621   0.0000000   0.0000000   0.0000000   0.0000000   2   0.0000000   0.0000000
  1    2 1.0220000   0.1569254   1.6298500  -0.2078590   0.0000000   0.0000000   0.0000000   0.0000000   1   0.0000000   0.0000000
  2    2 1.0120000   0.0814113   0.8500700  -0.0464325   0.0000000   0.0000000   0.0000000   0.0000000   1   0.0000000   0.0000000
  3    0 1.0064377  -0.0404533   0.0000000   0.0000000  -0.0003000  -0.0000900   0.0000000   0.0000000   2   0.0000000   0.0000000
  4    0 0.9910398  -0.0651346   0.0000000   0.0000000   0.6996800   0.3000900   0.0000000   0.0000000   2   0.0000000   0.0000000
  5    0 1.0311998   0.0343761   0.0000000   0.0000000   0.0002300   0.0001100   0.0000000   0.0000000   1   0.0000000   0.0000000
  6    0 1.0352820   0.0125969   0.0000000   0.0000000   1.0002600   0.3498700   0.0000000   0.0000000   1   0.0000000   0.0000000
  7    0 1.0654885   0.0608921   0.0000000   0.0000000  -0.0004500  -0.9337700   0.0000000   0.0000000   1   0.0000000   0.0000000
  8    0 0.9985642  -0.0578535   0.0000000   0.0000000   1.2504801   0.5000600   0.0000000   0.0000000   2   0.0000000   0.0000000
  9
1  1    1    4   0.0000000   0.0576000   0.0000000   1.0250000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
1  1    3    6   0.0000000   0.0586000   0.0000000   0.9850000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
0  1    4    5   0.0170000   0.0920000   0.0790000   0.0000000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
0  1    5    6   0.0390000   0.1700000   0.1790000   0.0000000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
0  1    6    7   0.0119000   0.1008000   0.1045000   0.0000000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
0  1    7    8   0.0085000   0.0720000   0.0745000   0.0000000   0.0000000   0  0   0.0200000   0.9400000   1.0750000   1.0300000
1  1    8    2   0.0000000   0.0625000   0.0000000   1.0250000   0.0000000   8  0   0.0000000   0.0000000   0.0000000   0.0000000
0  1    8    9   0.0320000   0.1610000   0.1530000   0.0000000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
0  2    9    4   0.0100000   0.0850000   0.0880000   0.0000000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
0  1    9    4   0.0100000   0.0850000   0.0880000   0.0000000   0.0000000   0  0   0.0000000   0.0000000   0.0000000   0.0000000
```

PART 1

nb, nl

nb	number of buses
nl	number of lines

PART 2

ibn, ibt, v_m, v_a, p_g, q_g, p_l, q_l, p_s, q_s, *iarea,* q_{min}, q_{max}

ibn	bus no. - limited to four digits
ibt	bus type - 0 for load, 2 for generator, 3 for slack
v_m	voltage magnitude (p.u)
v_a	voltage angle (radians)
p_g	active power generation (p.u)
q_g	reactive power generation (p.u)
p_l	active bus load (p.u)
q_l	reactive bus load (p.u)
p_s	active bus shunt (p.u)
q_s	reactive bus shunt (p.u)
iarea	area no.
q_{min}	minimum reactive power generation (p.u)
q_{max}	maximum reactive power generation (p.u)

PART 3

ilt, icktn, ifb, itb, r, x, b, tpr, tpa, itps, tpsz, tpmn, tpmx, tdvol

ilt	line type - 0 for plain line, 1 for fixed ratio transformer and 2 for variable ratio transformer.
icktn	circuit no.
ifb	the from bus no.
itb	the to bus no.
r	resistance (p.u)
x	reactance (p.u)
b	half line charging (p.u)
tpr	transformer tap ratio (p.u)
tpa	transformer tap angle (radians)
itps	transformer tap side bus no
tpsz	transformer tap step size (p.u)
tpmn	transformer minimum tap (p.u)
tpmx	transformer maximum tap (p.u)
tdvol	the desired voltage magnitude of the controlled bus (p.u)

C.2 Generator Data

A sample of the generator data file is shown in text box C.2. The data comprises of 4 parts - (1) number of generators, (2) machine electrical data that begins with a continuous word 'machine_data', (3) exciter data that begins with a continuous word 'exciter_data' , and (4) power system stabilisers data that begins with a continuous word 'pss_data'. Circuit diagram for the standard second-order generator model is shown in figure C.1. The derivation of the parameters used in the DCPS data - part 2 - can be found in IEEE Standard 115A [19S7] and in Krause [1986]. The diagrams of exciters type 1, 2, 3 and stabilizer type 1 are shown in figure C.2, C.3, C.4 and C.5 respectively. The meaning of each field in the data format is explained next.

PART 1

nm

nm number of generators (optional)

PART 2

imt, imn, iunit, h, x_d, x'_d, x''_d, t'_{do}, t''_{do}, x_q, x'_q, x''_q, t'_{qo}, t''_{qo}, r_a *d,* x_l, s_1, s_2

imt generator model no. - 0, 1, 2, 5

imn generator bus no.

iunit unit no.

h inertia constant (MW-sec / MVA)

x_d direct axis synchronous reactance (p.u)

x'_d direct axis transient reactance (p.u)

x''_d direct axis subtransient reactance (p.u)

t'_{do} direct axis open circuit transient time constant (sec.)

t''_{do} direct axis open circuit subtransient time constant (sec.)

x_q quadrature axis synchronous reactance (p.u)

x'_q quadrature axis transient reactance (p.u)

x''_q quadrature axis subtransient reactance (p.u)

t'_{qo} quadrature axis open circuit transient time constant (sec.)

t''_{qo}	quadrature axis open circuit subtransient time constant (sec.)
r_a	stator resistance (p.u)
d	damping constant
x_l	stator leakage reactance (p.u)
s_1	saturation value 1 at saturation voltage of 1.0
s_2	saturation value 2 at saturation voltage of 1.2

PART 3

ibe, ety, T_R, K_A, T_A, K_E / T_C, T_E / T_B, A_{EX}, B_{EX}, V_{RMAX}, V_{RMIN}, E_{FDMIN}, E_{FDMAX}

ibe	generator bus no.
ety	exciter type
T_R	terminal voltage transducer time constant (sec.)
K_A	voltage regulator gain
T_A	voltage regulator time constant (sec.)
K_E / T_C	K_E self excitation constant for type 1
	T_C lead-lag transfer function time constant for type 3
T_E / T_B	T_E exciter time constant for type 1
	T_B lead-lag transfer function time constant for type 3
A_{EX}	saturation constant 1
B_{EX}	saturation constant 2
V_{RMAX}	voltage regulator maximum output limit (p.u)
V_{RMIN}	voltage regulator maximum output limit (p.u)
E_{FDMIN}	minimum exciter output limit (p.u)
E_{FDMAX}	maximum exciter output limit (p.u)

PART 4

ibp, ipsty, ipsin, K_S, T_5, T_1, T_2, T_3, T_4, V_{SMAX}, V_{SMIN}

ibp	generator bus no.
ipsty	stabiliser type
ipsin	input signal identifier
K_S	stabiliser gain
T_5	reset time constant (sec.)
T_1 / *T2*	first lead-lag transfer function
T_3 / T_4	second lead-lag transfer function
V_{SMAX} / V_{SMIN}	output limits

Text box C.2: DCPS sample of generator data

```
8
machine_data
5    3 1   0.90000 2.08000 0.42700 0.22700 16.00000 0.03900 1.33300 1.06600 0.26700 2.60000 0.23500 0.00227  0.00000 0.00 0.0 0.0
5    2 1   0.60600 4.12700 0.64200 0.42600 12.35000 0.05606 1.96700 1.67200 0.43500 2.00000 0.18660 0.00400  0.00000 0.00 0.0 0.0
5    1 1  13.47000 0.33200 0.05270 0.03900  3.80000 0.05000 0.32200 0.09180 0.04000 0.46000 0.05000 0.00078  0.00000 0.00 0.0 0.0
0   12 1   4.31000 1.04500 0.20300 0.14700  4.30000 0.04840 1.03300 0.66300 0.14700 1.50000 0.21800 0.00231  0.00000 0.00 0.0 0.0
0   16 1   4.81000 1.34600 0.27800 0.21400  5.14000 0.04370 1.00000 0.90000 0.30000 1.50000 0.14100 0.00140  0.00000 0.00 0.0 0.0
5   20 1   9.60000 0.67300 0.13900 0.10700  6.14000 0.04370 0.66400 0.39400 0.10600 1.50000 0.14100 0.00069  0.00000 0.00 0.0 0.0
1   21 1   9.92000 0.59100 0.09610 0.08000  6.00000 0.05000 0.56200 0.33900 0.10000 1.50000 0.05000 0.00000  0.00000 0.00 0.0 0.0
5   22 1   1.68000 2.09300 0.70400 0.63000  8.50000 0.05000 1.25900 1.00700 0.63000 0.90000 0.05000 0.00910  0.00000 0.00 0.0 0.0
exciter_data
 3 1 0.050 100.00 0.055 1.000  0.3600  0.006  1.075  0.125  1.800  5.2000  -5.2000  -7.3000   7.3000
 2 3 0.010 200.00 0.020 1.000 10.0000  0.000  0.000  0.000  0.000  0.2000  -0.2000  -7.3000   7.3000
20 3 0.010 200.00 0.020 1.000 10.0000  0.000  0.000  0.000  0.000  0.2000  -0.2000  -7.3000   7.3000
21 1 0.050 100.00 0.055 1.000  0.3600  0.006  1.075  0.125  1.800  5.2000  -5.2000  -7.3000   7.3000
22 3 0.010  20.00 0.020 1.000 10.0000  0.000  0.000  0.000  0.000  0.2000  -0.2000  -7.3000   7.3000
pss_data
 3  1  1 -5.00  1.5  0.3  0.036  0.3  0.036  0.20  -0.05
 2  1  1 -5.00  1.5  0.3  0.036  0.3  0.036  0.20  -0.05
```

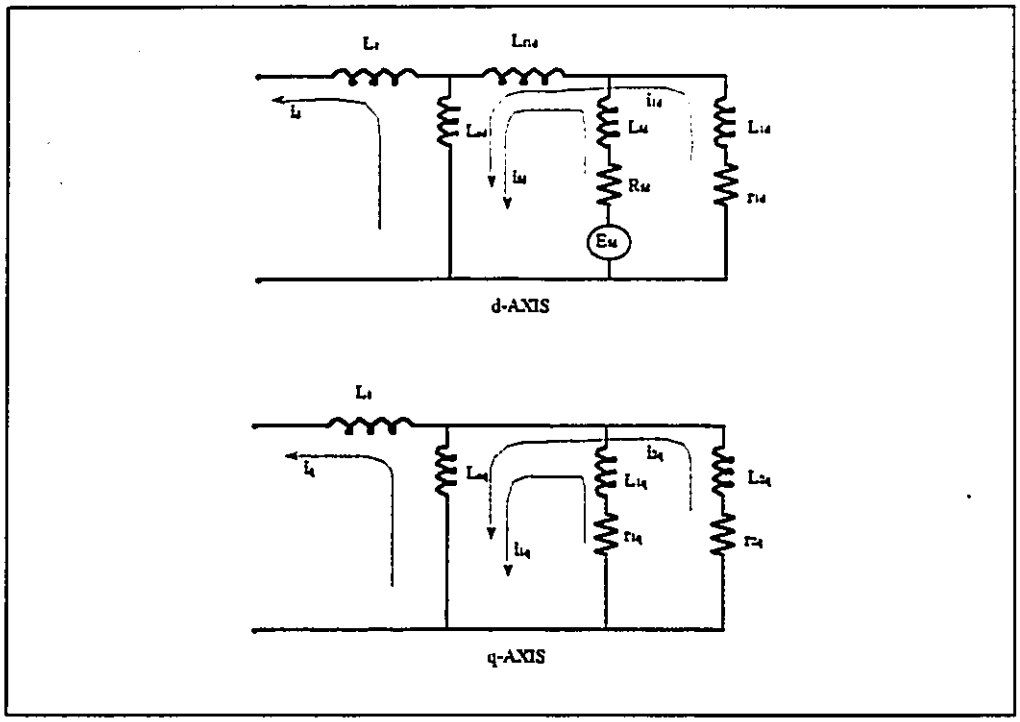

Figure C.1: Second order representation of machine q- and d-axes

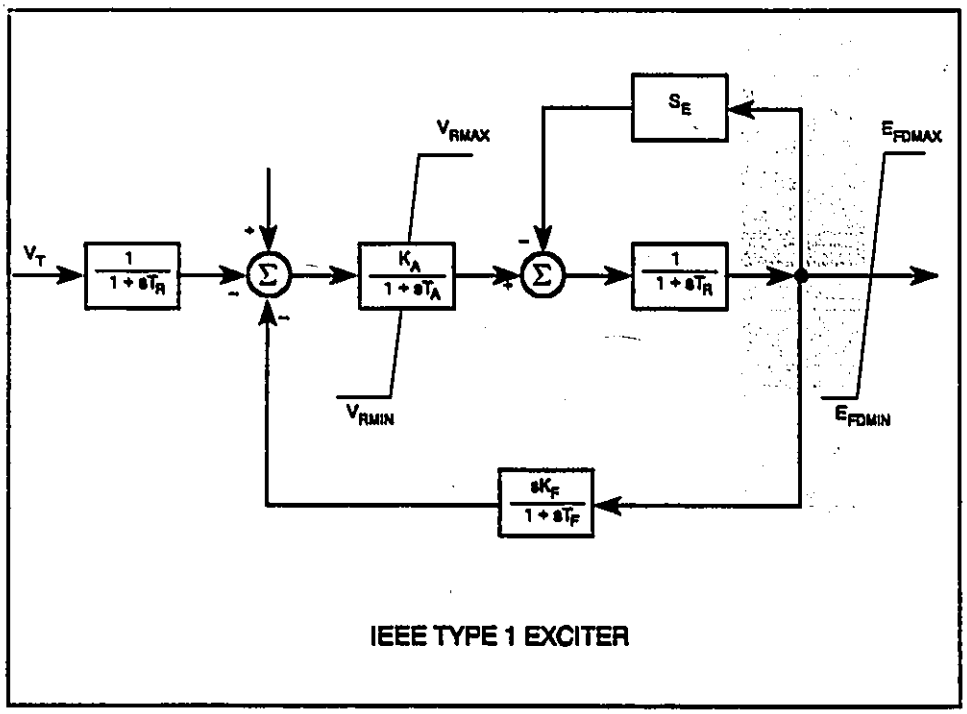

Figure C.2: Exciter - DCPS type 1

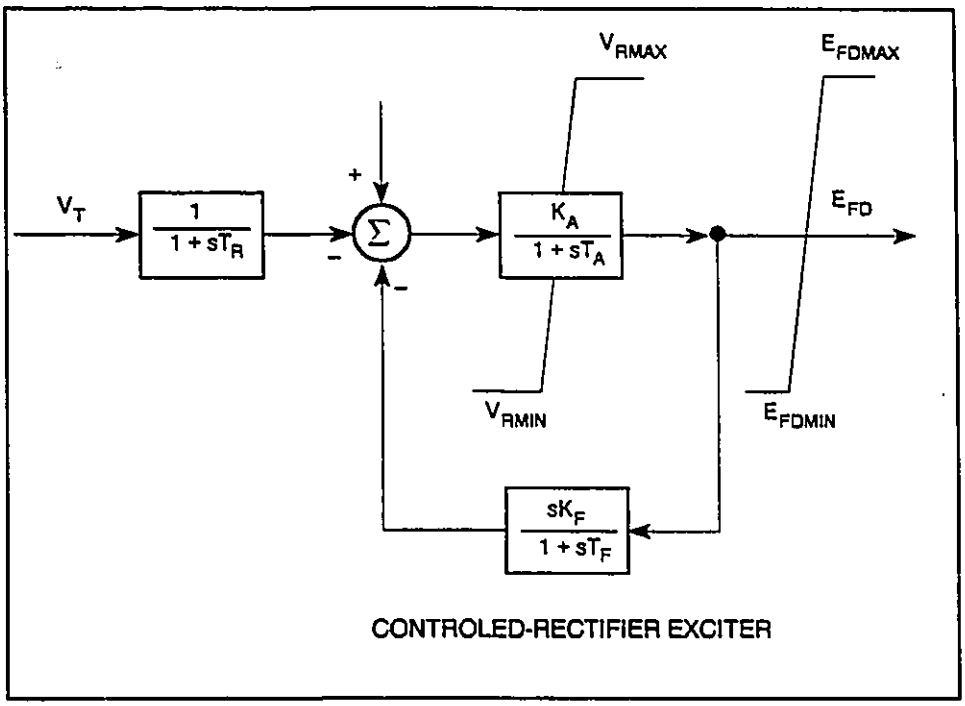

Figure C.3: Exciter - DCPS type 2

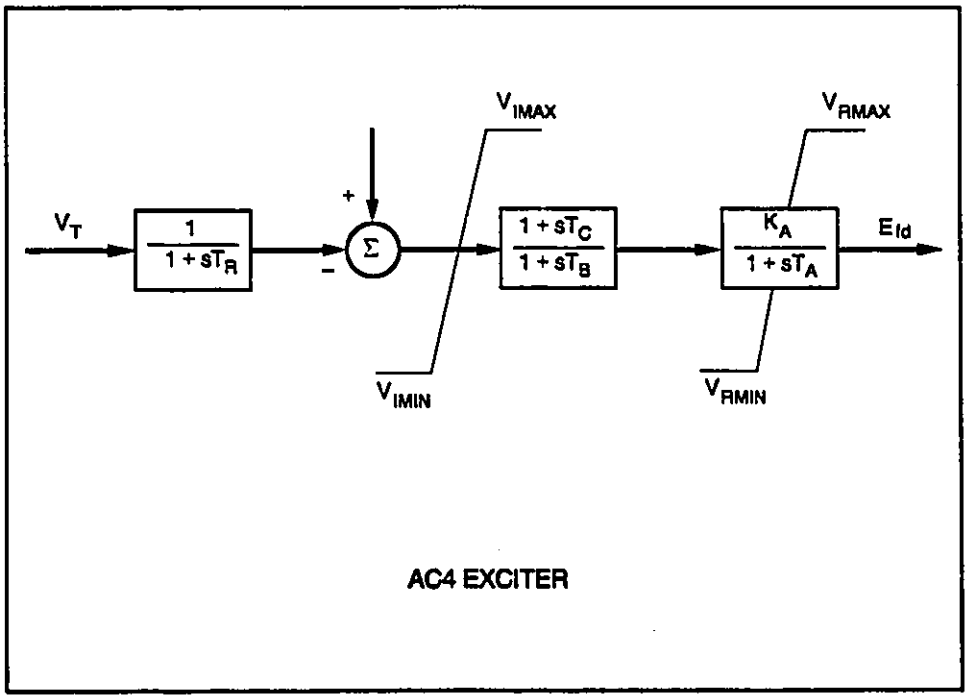

Figure C.4: Exciter - DCPS type 3

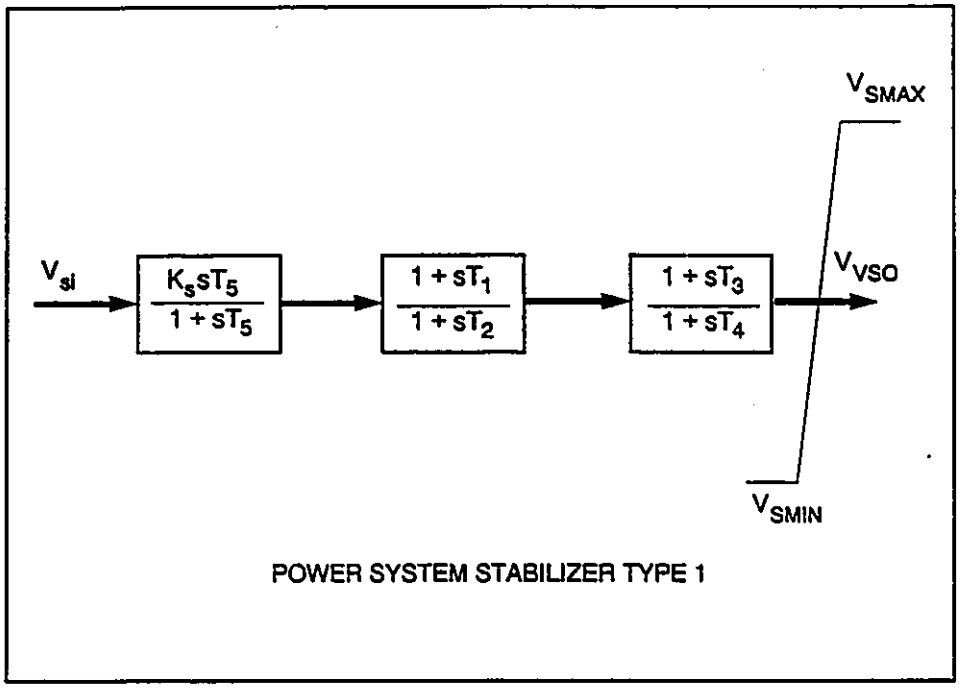

Figure C.5: PSS - DCPS type 1

REFERENCES

F. L. Alvarado [1979], *Parallel Solution of Transient Problems by Trapezoidal Integration*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-98, No. 3, pp. 1080 - 1089, May/June 1979.

P. M. Anderson and A. A. Fouad [1977], **Power System Control and Stability**, Iowa State University.

J. Arrillaga, C. P. Arnold and B. J. Harker [1983], **Computer Modelling of Electrical Power Systems**, John Wiley and Sons Ltd. New York.

N. Balu, T. Bertram, A. Bose, V. Brandwajn, G. Cauley, D. Curtice, A. Fouad, L. Fink, M. G. Lauby, B. F. Wollenberg and J. Wrubel [1992], *On-Line Power System Security Analysis*, Proceedings of the IEEE, Vol. 80, No. 2, pp. 262-280, February 1992.

B. E. Bauer [1992], **Practical Parallel Programming**, Academic Press Inc. 1992.

D. P. Bertsekas and J. Tsitkilis [1989], **Parallel and Distributed Computation**, Prentice Hall Inc., Englewood Cliffs, New Jersey.

A. D. Birrel and B. J. Nelson [1984], *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, Vol. 2, No.1, pp. 39-59, February 1984.

J. Bloomer [1991], **Power Programming with RPC**, O'Reilly & Associates, Inc., Sebastopol, California.

S. Brawer [1989], **Introduction to Parallel Programming**, Academic Press Inc., San Diego, California.

B. A. Carré [1968], *Solution of Load-Flow Problems by Partitioning Systems into Trees*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-87, No. 11, pp. 1931-1968, November 1968.

B. A. Carré [1966], *The Partitioning of Network Equations for Block Iterations*, Computer Journal (London), Vol. 9, pp.84-97, May 1966.

J. S. Chai, N. Zhu, A. Bose and D. J. Tylavsky [1991], *Parallel Newton Type Methods for Power System Stability Analysis Using Local and Shared Memory Multiprocessors*, IEEE Transactions on Power Systems, Vol. 6, No. 4, pp. 1539-1545, November 1991.

J. H. Chow, J. Cullum and R. A. Willoughby [1984], *A Sparsity-Based Technique for Identifying Slow-Coherent Areas in Large Power Systems*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-103, No. 3, pp. 463-473, March 1984.

J. H. Chow [1982], **Time-Scale Modelling of Dynamic Networks with Applications to Power Systems**, Springer-Verlag, New York.

J. H. Chow, R. A. Date, H. Othman and W. W. Price [1989], *Slow Coherency Aggregation of Large Power Systems*, Eigenanalysis and Frequency Domain Methods for System Dynamic Performance, IEEE Publication 90TH0292-3-PWR pp. 50-60.

J. H. Chow and K. W. Cheung [1992], *A Toolbox for Power System Dynamics and Control Engineering Education and Research*, Paper 92 WM 082-8 PWRS, IEEE/PES 1992 Winter Meeting, New York, January 26-30, 1992.

D. E. Comer [1991], **Internetworking with TCP/IP Volume I - 2nd Edition: Principles, Protocols and Architecture**, Prentice Hall Englewood, New Jersey.

D. E. Comer and D. L. Stevens [1993], **Internetworking with TCP/IP Volume III - : Client-Server Programming and Applications**, Prentice Hall Englewood, New Jersey.

C. Concordia, H. Clark and W. Lachs [1990], *Definitions and Concepts*, Voltage Stability of Power Systems: Concepts, Analytical Tools and Industry Experience, IEEE Publication 90TH0358-2-PWR, pp. 1-39.

M. L. Crow and M. Ilic' [1990], *The Parallel Implementation of the Waveform Relaxation Method for Transient Stability Simulations*, IEEE Transactions on Power Systems, Vol. 5, No. 3, pp. 922-932, August 1990.

D. A. Curry [1990], **Using C on the Unix System**, O'Reilly & Associates, Inc., Sebastopol, California.

P. L. Dandeno and P. Kundur [1973], *A Non-Iterative Transient Stability Program Including the Effects of Variable Load-Voltage Characteristics*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-92, pp. 1478-1484, September/October 1973.

F. F. de Mello, J. W. Felts, T. F. Laskowski and L. J. Oppel [1992], *Simulating Fast and Slow Dynamic Effects in Power Systems*, IEEE Computer Applications in Power Systems, pp. 33-38, July 1992.

I. C. Decker, D. M. Falcao and E. Kaszkurewicz [1992], *Parallel Implementation of a Power System Dynamic Simulation Methodology Using the Conjugate Gradient Method*, IEEE Transactions on Power Systems, Vol. 7, No. 1, pp. 458-465, February 1992.

H. W. Dommel and N. Sato [1972], *Fast Transient Stability Solutions*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-91, pp. 1643-1650, July/August 1972.

J. Donggara [1993], *Linear Algebra Libraries for High-Performance Computers: A Personal Perspective*, IEEE Parallel & Distributed Technology, pp. 17-23, February 1993.

R. A. Duncan [1990], A *Survey of Parallel Computer Architecture*, IEEE Computer, pp. 5-16, February 1990.

T. E. Dy Liacco [1977], *Digital Computer Applications in Power System Operations*, Exploring Applications of Parallel Processing to Power System Analysis Problems, EPRI Special Report No. EPRI-EL-566-SR, pp. 13-34, October 1977.

L. Elden and L. Wittmeyer-Koch [1990], **Numerical Analysis - An Introduction**, Academic Press Inc.

EPRI EL-2000-CCM - Project 1208 [1987], **ETMSP (Extended Transient-Midterm Stability Package: Technical Guide for Stability Program**, Jan. 1987.

M. J. Flynn [1966], *Very High Speed Computing Systems*, Proceedings of the IEEE 54(12), pp 1901-1909.

J. Fong and C. Pottle [1978], *Parallel Processing of Power System Analysis Problems via Simple Parallel Microcomputer Structures*, IEEE Transactions on Power Apparatus and Systems, Vol.. PAS-97, No. 5, pp. 1834-1841, September/October 1978.

B. S. Garbow, J. M. Boyle, J. J. Dongarra and C. B. Moler [1977], **Matrix Eigensystem Routines - EISPACK Guide Extension**, Lecture Notes in Computer Science, Springer-Verlag, New York.

G. W. Gear [1971], *Simultaneous Numerical Solutions of Differential-Algebraic Equations*, IEEE Transactions on Circuit Theory, Vol. CT-18, pp. 89-95, January 1971.

G. H. Golub and J. M. Ortega [1992], **Scientific Computing and Differential Equations - An Introduction to Numerical Methods**, Academic Press Ltd .

G. H. Golub and C. F. Van Loan [1989], **Matrix Computation (Second Edition)**, The John Hopkins University Press.

J. Groves [1993], **Windows NT Answer Book**, Microsoft Press.

J. R. Gurd [1988], *A Taxonomy of Parallel Architecture*, Proceedings of International Conference on Designing and Applications of Parallel Digital Processors, Lisbon, Portugal, pp. 57-61.

M. H. Haque and A. H. M. A. Rahim [1988], *An Efficient Method of Identifying Coherent Generators Using Taylor Series Expansion*, IEEE Transactions on Power Systems, Vol. 3, No. 3, pp. 1112-1118, August 1988.

W. L. Hatcher, F. M. Brasch and J. E. Van Ness [1977], *A Feasibility Study for the Solution of Transient Stability Problems by Multiprocessor Structures*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-96, No. 6, pp. 1789-1797, November/December 1977.

M. T. Heath, E. Ng and B. W. Peyton [1990], *Parallel Algorithms for Sparse Linear Systems*, Parallel Algorithms for Matrix Computations, Society for Industrial and Applied Mathematics, Philadelphia.

IEEE Task Force on Terms and Definition [1982], *Proposed Terms & Definitions for Power System Stability*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-101, No. 7, pp. 1894-1897, July 1982.

IEEE Committee Report [1986], *Current Usage & Suggested Practices in Power System Stability Simulations for Synchronous Machines*, IEEE Transactions on Energy Conversion, Vol. EC-1, No. 1, pp. 77-101, March 1986.

IEEE Committee Report [1992], *Parallel Processing in Power Systems Computation*, IEEE Transactions on Power Systems, Vol. 7, No. 3, pp. 629-637, May 1992.

IEEE Committee Report [1981], *Excitation Models for Power System Stability Studies*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-100, No. 2, pp. 494-507, February 1981.

IEEE Committee Report [1973], *Common Format for Exchange of Solved Load Flow Data*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-92, No. 4, pp. 1916-1925, November December 1973.

IEEE-PES Committee Report [1990], **Guide for Synchronous Generator Modelling Practices in Stability Analyses**, IEEE Publication P1110/D11, May 1990.

IEEE Std.100-[1988]: F. Jay (Editor in Chief) [1988], **IEEE Standard Dictionary of Electrical and Electronics Terms**, The Institute of Electrical and Electronic Engineers Inc., New York, NY.

IEEE Standard 115A [1987], **IEEE Standard Procedures for Obtaining Synchronous Machine Parameters by Standstill Frequency Response Testing**, IEEE Inc., New York.

M. Ilic'-Spong, M. L. Crow and M. A. Pai [1987], *Transient Stability Simulation by Waveform Relaxation Methods*, IEEE Transactions on Power Systems, Vol. PWRS-2, No. 4, pp. 943-952, November 1987.

B. W. Kernighan and D. M. Ritchie [1988], **The C Programming Language**, 2nd. Edition, Prentice Hall, Englewood, Cliffs, New Jersey.

E. W. Kimbark [1948], **Power System Stability - Volume 1: Elements of Stability Calculations**, John Wiley and Sons.

P. Korzeniowski [1993], *State of the Art - Make Way for Data*, BYTE Magazine June 1993, pp. 113-115.

P. C. Krause [1986], **Analysis of Electric Machinery**, McGrawHill Book Co.

P. Kundur, G. J. Rogers, D. Y. Wong, L. Wang and M. G. Lauby [1990], *A Comprehensive Computer Program Package for Small Signal Stability Analysis of Power Systems*, IEEE Transactions on Power Systems, Vol. 5, No. 3, pp. 1076-1083, November 1990.

M. La Scala, R. Sbrizzai, F. Torelli [1991], *A Pipelined-in-Timed Parallel Algorithm for Transient Stability Analysis*, IEEE Transactions on Power Systems, Vol. 6, No. 2, pp. 715-722, May 1991.

M. La Scala, A. Bose and D. J. Tylavsky [1989], *A Relaxation Type Multigrid Parallel Algorithm for Power System Stability Analysis*, Proceedings of International Symposium on Circuits and Systems, Portland, Oregon, pp. 1954-1959, May 1989.

M. La Scala, M. Brucoli, F. Torelli and M. Trovato [1990a], *A Gauss-Jacobi-Block-Newton Method for Parallel Transient Stability Analysis*, IEEE Transactions on Power Systems, Vol. 5, No. 2, pp. 1168-1177, November 1990.

M. La Scala, A. Bose, D. J. Tylavsky and J. S. Chai [1990b], *A Highly Parallel Method for Transient Stability Analysis*, IEEE Transactions on Power Systems, Vol. 5, No. 2, pp. 1439-1446, November 1990.

S. Y. Lee, H. D. Chiang, K. G. Lee and B. Y. Ku [1989], *Parallel Power System Transient Stability Analysis on Hypercube Multiprocessors,* Proceedings of Power Industry Computer Applications Conference, Seattle, pp. 400-406, May 1989.

E. LeLarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli [1982], *The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-1, No. 3, pp. 131-145, July 1982.

T. Lie, R. A. Schlueter and P. A. Rusche [1993], *Method of Identifying Weak Transmission Network Stability Boundaries*, IEEE Transactions on Applied Superconductivity, Vol. 3, No. 1, March 1993.

Y. Mansour [1989], *Application of Eigenanalysis to the Western North American Power System,* Eigenanalysis and Frequency Domain Methods for System Dynamic Performance, IEEE Publication 90TH0292-3-PWR pp. 50-60.

A. S. McFarlane and R. T. H. Alden [1993], *Coherent Grouping of Power Systems for Use in Training Artificial Neural Networks*, Paper Presented at The 36th Mid-West Symposium on Circuits and Systems, August 16-18, 1993.

A. S. McFarlane [1993, **Transient Stability Assessment of Power Systems - A Neural Network Approach**, Masters Thesis, McMaster University, Hamilton Canada.

J. V. Mitsche [1993], *Stretching the Limits of Power System Analysis*, IEEE Computer Applications in Power, pp. 16-21, January 1993.

J. J. Modi [1988], **Parallel Algorithms and Matrix Computation**, Clarendon Press, Oxford.

N. Mûller and V. H. Quintana [1992], *A Sparse Eigenvalue-Based Approach for Partitioning Power Networks*, IEEE Transactions on Power Systems, Vol. 7, No. 2, pp. 520-527, May 1992.

J. M. Ortega and R. G. Voigt [1985], **Solution of Partial Differential Equations on Vector and Parallel Computers**, Society for Industrial and Applied Mathematics, Philadelphia.

M. A. Pai [1981], **Power System Stability Analysis by Direct Method of Lyapunov**, North-Holland Publishing Co.

R. Podmore [1978], *Identification of Coherent Generators for Dynamic Equivalents*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-97, No. 4, pp. 1344-1354, July/August 1978.

V. H. Quintana and N. Mûller [1991], *Partitioning of Power Networks and Applications to Security Control*, IEEE Proceedings-C, Vol. 138, No. 6, pp. 535-545, November 1991.

N. Ram, S. S. Lamba, K. S. P. Rao [1985], *Coherency Based System Decomposition into Study and External Areas Using Weak Coupling*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-104, No. 6, pp. 1443-1449, June 1985.

G. J. Rogers and P. Kundur [1989], *Small Signal Stability of Power Systems*, Eigenanalysis and Frequency Domain Methods for System Dynamic Performance, IEEE Publication 90TH0292-3-PWR pp. 50-60.

A. Singh, J. Schaeffer and M. Green [1991], *A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations*, IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 1, pp. 52-67, January 1991.

M. Singhal and T. L. Casavant [1991], *Distributed Computing Systems*, IEEE Computer, pp. 12-15, August 1991.

B. T. Smith, J. M. Boyle, B. S. Garbow, Y. Ikebe, V. C. Klema and C. B. Moler [1976], **Matrix Eigensystem Routines - EISPACK Guide 2nd Edition**, Lecture Notes in Computer Science, Springer-Verlag, New York.

G. W. Stagg and A. H. El-Abiad [1968] , **Computer Methods in Power System Analysis**, McGraw-Hill Inc., New York.

B. Stott [1979], *Power System Dynamic Response Calculations*, Proceedings of The IEEE, Vol. 67., No. 2, pp. 219-241, February 1979.

B. Stott [1974], *Review of Load-Flow Calculation Methods*, Proceedings of The IEEE, Vol. 62., No. 7, pp. 916-929, July 1974.

W. F. Tinney and F. M. Orem [1977], *Power System Computation and Parallel Processing*, Exploring Applications of Parallel Processing to Power System Analysis Problems, EPRI Special Report No. EPRI-EL-566-SR, pp. 1-10, October 1977.

W. F. Tinney and J. W. Walker [1967], *Direct Solution of Network Equations by Optimally Ordered Triangular Factorization*, Proceedings of The IEEE, Vol. 55, No. 7, pp. 1801-1810, November 1967.

W. F. Tinney, V. Brandwajn and S. M. Chan [1985], *Sparse Vector Methods*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-104, No. 2, pp. 295-301, February 1985.

J. Uddel [1993], *Windows, Windows Everywhere*, BYTE pp. 73-94, June 1993.

R. J. Vetter and D. H. C. Du [1993], *Distributed Computing with High-Speed Optical Networks*, IEEE Computer, pp. 8-18, February, 1993.

J. J. Valley [1991], **Unix Programmer's Reference**, Que Corporation.

G. V. Wilson [1993], *A Glossary of Parallel Computing Terminology*, IEEE Parallel & Distributed Technology, pp. 52-67, February 1993.

S. B. Yusof, G. J. Rogers and R. T. H. Alden [1992a], *Slow Coherency Based Network Partitioning Including Load Buses*, IEEE Transactions on Power Systems, Vol. 8, No. 3, pp. 1375-1382, August 1993.

S. B. Yusof, G. J. Rogers and R. T. H. Alden [1992b], *Methods in Parallel Transient Stability Calculations*, Proceedings of The Canadian Conference in Electrical and Computer Engineering 1992, Toronto, Paper TA7.19.1, 13-16 September 1992.

S. B. Yusof, G. J. Rogers, R. T. H. Alden, P. Kundur [1993a], *Small Signal Stability Analysis for Damping Control of TNB/PUB Interconnected Power System*, Proceedings of The International Power Engineering Conference 1993, Singapore, pp. 272-277, March 18-19 1993.

S. B. Yusof, G. J. Rogers, B. Szabados, R. T. H. Alden [1993b], *Communication Aspects of Parallel Distributed Algorithms on a Cluster of Workstations Using the RPC*, Proceedings of The Canadian Conference in Electrical and Computer Engineering 1993, Vancouver, 14-17 September 1993.

S. B. Yusof, G. J. Rogers, B. Szabados, R. T. H. Alden [1993c], *Parallel Distributed Solution of Power System Networks on a Cluster of Workstations*, Paper Submitted to IEEE-PES Winter Power Meeting 1994.

J. Zaborszky, G. Huang and K. W. Lu [1985], *A Textured Model for Computationally Efficient Reactive Power Control and Management*, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-104, No. 7, pp. 1718-1727, July 1985.