High Speed Transient Stability
Multiprocessing Solutions

By

Wieslaw T. Kwasnicki

A THESIS

Submitted to the Faculty of Graduate Study

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering

The University of Manitoba

Winnipeg, Manitoba, Canada

June 1998

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

HIGH SPEED TRANSIENT STABILITY MULTIPROCESSING SOLUTIONS

BY

WIESLAW T. KWASNICKI

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

DOCTOR OF PHILOSOPHY

Acknowledgements

The author wishes to express his sincere gratitude to his advisor Prof. A. M. Gole for his encouragement, valuable comments and advice during the course of this work.

This research has been supported by the Manitoba HVDC Research Centre as part of a broad research program which led the author to complete the work for this thesis. In particular the author would like to thank Mr. D. Woodford for his inspiration to search for new solution methods, Dr. X. Wang for introducing the LDU, Bus Tearing and Current Compensation methods to this research and for providing the test cases for comparative analyses with other stability programs. The author also appreciates the assistance and great help of Mr. L. Arendt in the work regarding the RTDS implementation and Mr. M. Balcerzak for the Distributed Processing System implementation.

The author appreciates the comments and suggestions received from Prof. A. M. Gole, Prof. D. Meek, and Mr. L. Arendt during the writing of this thesis.

Finally, the author expresses his sincere appreciation to his wife Teresa and his daughters Agnieszka and Natalia for their support, understanding, patience and forgiveness for not being with them when they might need him during the past few years of conducting the research and writing of this thesis.

Abstract

The emergence of parallel processing architectures and fast network computing have opened new opportunities and challenges to apply these recent technologies to solve power system problems. For the transient stability solution applied on parallel processing hardware, a suitable algorithm has been developed to achieve high speed computation for large systems. This new algorithm combines several techniques useful for parallel processing such as : the W–matrix method for solving network equations, Bus Tearing for splitting large network into smaller subsystems, Current Compensation for handling system admittance changes, node re–ordering scheme to improve matrix sparsity, and a load–balancing partitioning scheme for solving one subsystem on many processors operating in parallel.

The multiprocessing algorithm has been incorporated in a stand alone version of a High Speed Transient Stability (HSTS) program. This computer program is aimed at implementations on existing hardware of parallel processing computers such as the Real Time Digital Simulator (RTDS), Distributed Processing Systems (DPS), and other multiprocessing computers (multicomputers).

A mechanism that coordinates the scheduling of interdependent operations of a parallel application is provided to run a program concurrently on separate processors. Although, specific implementations require specialized software to achieve fast communication, the basic mechanism for synchronization is built in the HSTS program and is based on the Message Passing Interface (MPI) software.

Two implementations of the HSTS program have been completed and tested to demonstrate accuracy, efficiency, and computational speed of the proposed multiprocessing solution method. In the RTDS parallel processing implementation, it has been shown that solving the transient stability problem can be performed faster if the system is partitioned for solving on many processors operating in parallel instead of solving by one processor.

For large power systems, processing of large admittance matrices takes most of the computation time. In the Distributed Processing System implementation it has been shown that using the system splitting method computation time can be effectively reduced but at the expense of increasing communication time.

The high communication latency observed in the Local Area Networks may be eliminated or at least significantly improved by the emerging fast network technologies. Alternative software and hardware tools have been designed by other researchers to synthesize groups of computers into a high–performance environment. One such tool is the High Performance Virtual Machine (HPVM) which, according to the report, can deliver a high–performance message communication over high–speed networks with a bandwidth of 80 megabytes per second and a latency under 11 microseconds using the Myrinet interconnect.

It is very probable that with the new technologies of fast network and high performance computing, it will be possible to solve the transient stability problem for large systems in real–time using the HSTS program on a scalable cluster of commodity computers.

Table of Contents

List of Figures

Chapter 2.

Chapter 3.

Chapter 4.

List of Tables

Acronyms and Abbreviations

GS – Gauss–Seidel method

BM – Bergeron method

VDHN – Very DisHonest Newton method

AW – Average Weight partitioning scheme

HSTS – High Speed Transient Stability (HSTS)

DPS – Distributed Processing Systems

RTDS – Real Time Digital Simulator

MPI – Message Passing Interface

LDU – Lower–Diagonal–Upper matrix factorization

LAN – Local Area Network

HPVM – High Performance Virtual Machine

BPA – Boneville Power Administration

Chapter 1

Introduction

Power systems are becoming increasingly complex because of interconnection and fast response of power plants with solid state controllers and the use of larger generating units. These trends result in more productive use of transmission corridors and the system being operated much closer to its stability limits. Transient stability simulation is widely needed for dynamic analysis, but existing off–line transient stability programs require excessive computer time for practical studies. Consequently, fast methods for assessing transient stability limits are required, preferably in real–time or even faster.

The emergence of parallel processing architectures and fast network computing have opened new opportunities and challenges to apply these recent technologies to solve power system problems. For the transient stability solution applied on parallel or distributed processing hardware, suitable algorithms must be developed if high speed is to be achieved for large system sizes. Using those new algorithms implies modifications or even complete rewriting of existing programs in which significant effort has been invested over the years [8]. In order to make the most from the existing solution methods and the new technology, a variety of approaches have been undertaken. One group of methods adapts the fastest sequential algorithms for parallel implementations on special purpose parallel processing computers. Another group of methods applies new algorithms that are written specifically for applications on the existing hardware of parallel processing computers.

There are no obvious parallelisms inherent in the mathematical structure of the power system transient stability problem. Thus, for this specific problem, a parallel (or near–parallel) formulation has to be found that is useful for constructing a parallel algorithm. This solution has to be implemented on a particular multiprocessing computer. The computational ef-

ficiency of such an implementation is dependent on the suitability of the parallel architecture to the parallel algorithm. Therefore, it is no longer meaningful to develop the best parallel algorithm without reference to the target hardware architecture.

There are available various single processor software packages which run the traditional stability program solution at, or near, real time for small or midsize systems. For solving large power systems, one common trend is to adapt these standard solution methods and use the commercially available multiprocessing hardware as their computational tools. High efficiency is usually hard to reach because computation and communication takes too much time during each calculation time–step. For the solution of large scale power system networks, it is possible to substantially reduce the computation time if special purpose parallel processing hardware and parallel programming were used.

There are various types of commercially available parallel processing computers : transputers, shared–memory multi–processor computers, and distributed–memory parallel computers like Hypercube [3] and the Real Time Digital Simulator (RTDS) [6]. The biggest challenge facing the use of parallel processing computers for the power system stability solution in real–time is the communication and data exchange that may be very extensive for large power system models. The RTDS has been successfully applied for real–time electromagnetic transients simulations of power systems and it handles communication problems very effectively. Therefore this parallel processing hardware has been chosen for a prototype implementation of the proposed multiprocessing method for solving the transient stability problem.

The main objective in this research project, is to develop a suitable algorithm for solving power system transient stability problem on many processors operating in parallel. A computer program using such an algorithm is aimed at implementations on existing hardware of parallel processing computers such as the RTDS, Distributed Processing Systems (DPS's), and other multiprocessing computers (multicomputers).

In order to achieve this goal, various solution methods were considered and examined on single–processor workstation, computer networks, and on the RTDS. Those methods included the Gauss–Seidel (GS) and Bergeron (BM) iterative algorithms as well as various sparse techniques for the direct solution of network equations as described in Chapter 2. Those methods were used for developing multiprocessing algorithms which were applied in computer programs, implemented on available hardware, and tested for computational speed.

In the first attempt, the network solution algorithm based on the iterative Gauss–Seidel method was implemented on the RTDS for testing its performance in a parallel architecture. Those tests indicated that the speed of solution significantly reduces as the size of the system grows. Although this method is well suited for parallel processing, it has poor convergence or even divergence (also reported by other researchers [5]) for larger network sizes, so it was abandoned in favour of other methods.

In the second attempt, an alternative solution method was developed based on the concept of a phasor–domain Bergeron line model [5,16]. This new network solution method still possessed the localized benefits of the Gauss–Seidel method but the convergence problems were improved to a certain extent. This method utilizes the concept of travelling waves in transmission lines. However, some prolonged voltage oscillations and spurious overvoltages were observed in system post–fault conditions using this approach. Those effects were viewed as an inherent disadvantage of the Bergeron based iterative algorithm that may potentially be further aggravated in larger systems and thus further work into this method was also suspended.

Since convergence of both the GS and BM iterative techniques was found to take too much time, the search for a suitable method for high speed transient stability solution was re–directed to sparse matrix techniques, which in recent years have gained interest of many research groups. For this reason, a new multiprocessing algorithm has been developed and

incorporated in a stand alone version of the High Speed Transient Stability (HSTS) program written in the 'C' computer language which is described in Chapter 3.

This new algorithm combines several techniques useful for parallel processing applications which include the following methods :

A. LDU–decomposition and LDU–inverse for processing sparse matrices
B. W–matrix method for solving network equations (basic 2–step procedure)
C. Re–ordering scheme to minimize number of fill–ins in the W–matrices
D. Bus Tearing method for splitting large network into smaller subsystems
E. Current Compensation method for handling the changes of system admittances
F. Partitioning scheme for solving one subsystem on many processors operating in parallel.

The network solution applied in the HSTS program has been tested on selected small and medium size test systems against the conventional, commercially available, stability programs such as BPA, PSS/E, and PSDS. Test results on workstations matched very closely the steady–state and post–faults curves of other stability programs. The computational speed was comparable with the speed of conventional stability programs, and promised significant gains when executed in the multi–processor environments.

For the parallel processing implementations of the HSTS program, described in Chapter 4, two multiprocessor environments are considered : the Real Time Digital Simulator (RTDS) parallel processing hardware and the Distributed Processing System (DPS) of the Local Area Network (LAN) at the Manitoba HVDC Research Centre. Those implementations require specific adaptations to the original HSTS program for the specific hardware architectures. The routines for compiling, downloading, initialization, communication and running the program in a multiprocessor environment have to be included for both applications.

The implementation on RTDS required the program C version to be converted to a version employing assembly code for the NEC processors used in the hardware. Also a program on the workstation is required for downloading instruction and data, running, and uploading

the results from the RTDS. In order to perform the transient stability solution on the RTDS hardware for large system sizes, it is necessary to utilize many racks operating in parallel. Due to limitations of the RTDS communication architecture a subsystem not larger than 500 buses can be processed on one rack.

The HSTS program implementation on Distributed Processing Systems was accomplished in more straightforward manner with addition only of the Message Passing Interface (MPI) software required for communications between computers. Since only single–processor computers were present in the local area network (LAN), only the system splitting was applied but not the partitioning scheme which was important for the RTDS implementation. One subsystem was assigned to each computer for collective solution using a cross–network communication. In a cluster of commodity computers one processor is a master synchronizing the program execution on other worker processors.

Both implementations completed in this research work have demonstrated that a high speed transient stability solution can be achieved with the proposed multiprocessing algorithm providing that the hardware allows fast communication links. The HSTS has proven to be a general, scalable, multiprocessing program that can be applied on integrated or distributed parallel processing systems. Communication latency is still a big obstacle in achieving a high performance on regular local area networks. However, it is just a matter of time until a fast network technology becomes commonplace and today's commodity systems will perform as supercomputers at affordable price. Using a High Performance Virtual Machine (HPVM) software, for example, a group of off–the–shelf computers has been synthesized at the University of Illinois to deliver a peak performance of between 100 and 200 billion floating–point operations per second. A high–performance message communication layer can send messages between processors over high–speed Myrinet network delivering bandwidth of 80 megabytes per second and a latency under 11 microseconds as reported by the National Center for Supercomputing at the University of Illinois.

Chapter 2

Transient Stability Problem

2.1 General Characteristic of Power System Problems

Power system stability can be defined as the property of a power system that enables it to remain in a state of operating equilibrium under normal operating conditions and to remain an acceptable state of equilibrium after being subjected to a disturbance [13].

Typically, the stability problem has been one of maintaining synchronous operation of generators and other machines in a power system. Another concern is to maintain the stability of voltages which may collapse even without loss of synchronism. The behavior of the power system can be evaluated when it is subjected to small or large transient disturbances. The system must be able to operate satisfactorily under these conditions and continue to supply the required amount of load.

The power system stability problem involves interaction between the electrical and mechanical systems. Under steady–state conditions, there is equilibrium between the input mechanical torque and the output electrical torque of each machine in the system and their angular speed remains constant. If the system is perturbed this equilibrium is upset, resulting in acceleration or deceleration of the rotors of the machines according to the lows of motion of a rotating body. The stability of the system depends on whether or not the deviations in angular positions of the rotors result in sufficient restoring torques.

The two major categories of power system stability are rotor angle stability and voltage stability. **Rotor angle stability** is the ability of interconnected synchronous machines of a power system to remain in synchronism. **Voltage stability** is the ability of power system to maintain steady acceptable voltages at all buses in the system [13]. The rotor angle stability

phenomenon is usually classified under two categories : small–signal (small–disturbance) stability and transient stability (severe transient disturbances).

The ***small–signal stability*** is mainly caused by insufficient damping of oscillations which may appear in various modes :

a) local modes – (0.6–2.0 Hz) localized at one station or small part of the power system

b) interarea modes – (0.2–0.7 Hz) caused by groups of closely coupled machines being interconnected by a weak ties.

c) control modes – associated with generating units and other controls

d) torsional modes – associated with the turbine–generator shaft system rotational components

In ***Transient stability*** problems the power system is subjected to a severe transient disturbance and the response of power system is observed for selected set of contingencies.

There are many types of power system analyses such as : Power Flow, Transient Stability, Short Circuit Calculations, and Electromagnetic Transients. The interconnected generation and transmission system is inherently large and any power system analysis problem formulation tends to have thousands of equations. Analysis of such systems is one of the most computationally intensive power system problem.

The most common analysis, the **Power Flow**, requires the solution of a large set of non–linear algebraic equations approximately two for each system node. The usual algorithm of an iterative matrix solution exploits the extreme sparsity of the underlying network connectivity to gain speed and conserve storage. Parallel algorithms for handling dense matrices are not competitive with the sequential sparse matrix methods, and since the pattern of sparsity is irregular, parallel sparse matrix methods have been difficult to find [9].

The power flow describes the steady state condition of the power network and thus, the formulation is a subset of several other important problems like the optimal power flow problem or transient stability problem. An effective parallelization of the power flow problem would also help speed up these other solutions. **Transient Stability** requires the solution of

differential equations (2 to 20 for each machine) that represent the dynamics of the rotating machines and other devices such as HVDC converter controls or Static Var Compensators, together with the algebraic equations that represent the connecting network. This set of differential–algebraic equations typically exhibits different types of nonlinearities and various numerical methods can be used to obtain a step–by–step time solution.

It is the size of the above problems and the consequent solution times that encourages the search for parallel processing approaches. Even before parallel computers became a potential solution, the concept of decomposing a large problem to address the time and storage problem in sequential computers has been more or less successfully applied to many of these power system problems. Parallel computers can take advantage of these decomposition/aggregation techniques but usually require a certain amount of adaptation.

Recently there is an on going effort to apply parallel computers to solve specific power system problems. Most of those problems can be parallelized in large portions including the required solution of the linear algebraic equation :

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$

where matrix **A** is large and considerably sparse.

In this research work new methods and algorithms applicable to parallel computers for solving transient stability problem are considered for implementations on RTDS and on Distributed Processing Systems. A significant speed up by parallel processing, in addition to the usual efficiencies, will allow on–line transient stability analysis; a prospect that has spurred research in this area.

2.2 Model of the Transient Stability Problem

For power system stability the behavior of power systems is usually described by two sets of equations. The first, is a set of differential equations defining the dynamics of the devices (loads, generators, exciters, governors, etc.) and the second, is a set of algebraic

equations describing the electrical system through which the dynamic devices and the system loads are connected.

These two equations can be written as :

$$\dot{\mathbf{X}} = F(\mathbf{X},\mathbf{V}) \tag{2.1}$$

$$\mathbf{Y}\cdot\mathbf{V} = \mathbf{I}(\mathbf{X},\mathbf{V}) \tag{2.2}$$

where $\mathbf{X}$ is a vector of state variables describing machine and load dynamics,

$\mathbf{V}$ is a vector of bus voltages,

$\mathbf{I}$ is a vector of bus current injections, and

$\mathbf{Y}$ is a matrix of network admittances

and F is a non–linear function of $\mathbf{X}$ and $\mathbf{V}$.

In general, there are two basic approaches in the development of new algorithms for solving a set of differential equations (2.1) and a set of algebraic equations (2.2) :

A. ***Partitioned approach*** – the set of differential equations (2.1) is solved separately from (2.2) by an integration method and iterated with the solution of the set of algebraic equations (2.2) at every time–step.

B. ***Simultaneous approach*** – set of equations (2.1) is discretized by methods such as the trapezoidal rule and combined with equations (2.2) for solving together at each time–step using some Newton–like method.

Although the size of the problem is bigger in the second approach, the solution on single processor computers can be reached faster because equation (2.1) can often be linearized around the operating point and Jacobian matrix can be held constant unless the system undergoes a significant change [2,9]. One variation of this approach is known as the Very DisHonest Newton (VDHN) method and is used by several research groups as reported in [41,36].

Integration methods for solving equations (2.1) are known as ***explicit*** or ***implicit*** integration methods. In explicit methods (such as Euler, predictor–corrector or Runga–Kutta) the state variable $\mathbf{X}$ at any time is computed from knowledge of variable values from previous

time–steps by evaluating the value of function F(**X**,**V**). Implicit integration methods (such as the trapezoidal rule) use interpolation for evaluating of the function between previous and current points in time. The implicit method is preferred because it provides better numerical stability [2,8].

The length of the integration time–step Δt is restricted by the characteristic of the differential equations. This characteristic is often described by ***stiffness*** which is measured by the ratio of the largest to the smallest time constants of the system. Stiffness in a transient stability simulation increases with modelling detail. In the overall system, not all the time constants may be readily apparent. Thus the stiffness, may be hidden and can only be established by computing the eigenvalues of the linearized system. Explicit integration methods have weak numerical stability and for solving stiff systems they must use very small steps.

The equations are solved on a step–by–step basis to obtain the time response of the system for the specified fault and switching conditions. To find the solution, the differential equation (2.1) can be discretized to transform the differential equations into difference equations. Using the implicit trapezoidal rule,for example, the equation (2.1) will take form of:

$$\mathbf{X}_t = \mathbf{X}_{t-1} + (\Delta t/2)*[\, F(\mathbf{X}_t, \mathbf{V}_t) + F(\mathbf{X}_{t-1}, \mathbf{V}_{t-1})\,]$$

In general, solving the network equations (2.2) is to find the bus voltage vector **V**, which is achieved through the equation :

$$\mathbf{V} = \mathbf{Y}^{-1} \cdot \mathbf{I}(\mathbf{X}, \mathbf{V}) \qquad (2.3)$$

Discretized equations (2.1) can be solved in conjunction with network equations (2.2) or they can be solved separately. In both cases some kind of iterative process is required for solving the two sets of equations because the system state **X** and voltage **V** used in both equations depend on each other.

The step–by–step solution for system voltages and state variables is used in Transient Stability to assess the ability of power systems to maintain synchronism when subjected to severe transient disturbances. The resulting system response is an excursion of generator ro-

tor angles influenced by the non–linear power–angle relationship included in equation (2.1). Following the disturbance, usually a network fault, the state variable **X** cannot change instantly, so the system response is deviation of machine rotor angles which can lead to either an aperiodic drift in rotor angle or to an oscillatory instability. Provided there is enough damping in the system, this perturbation should diminish, and operation at the equilibrium angles be restored.

Typically, for partitioned solution with explicit integration method, the algebraic equations (2.2) are solved first to give system voltages **V**, currents **I** and the corresponding power flows. Using the previous state **X** and and the current value of **V** the time derivative function F(**X**,**V**) in equation (2.1) is computed and the solution of differential equations for new state **X** is obtained using the selected integration method. The process of alternating solutions of algebraic and differential equations is applied successively until the solution at the end of each time–step is reached.

Since the solution of differential equations requires values of state and network variables (**X**,**V**) only from the previous step, the set of differential equations can be solved independently which offers a great amount of programing flexibility important for parallel computations. However, since the network solution takes a considerable amount of solution time, the overall gains achievable are very limited without parallelizing the network solution part.

2.3 Solution Methods for Parallel Processing

A wide range of approaches has been reported in the literature for conventional methods of solving equations (2.1) and (2.2). More recently, various non–conventional methods have also been developed to create better parallelism. Those methods are quite different from the sequential algorithms used today on single processor computers. Thus, adapting any of them for commercial applications requires significant investment in development of software and special hardware for multiprocessing implementations.

For this research project, various solution methods suitable for parallel processing have been examined in order to determine which one is the most practical for implementation on parallel or distributed processing systems. The partitioned approach is used in many production–grade stability programs. For power system simulation in which time–steps are limited by numerical stability, implicit methods like the trapezoidal rule are generally better suited than the explicit methods but they provide variable accuracy.

The partitioned and simultaneous approaches can both be used in parallel computers if proper problem decomposition is applied. Decomposition of the problem for parallel processing can be categorized in two general parallelization types :

A. ***Parallelization in space*** – decomposition of the system variables into smaller groups.

B. ***Parallelization in time*** – several time–steps solved simultaneously [15].

When the set of network equations (2.2) is decomposed, relaxation can be applied to differential equations (2.1). For the method known as the Waveform Relaxation Method [37], it was shown [33] that discrete version of (2.1) together with (2.2) can be decomposed to individual system variables such as bus voltages, and solved simultaneously for all time–steps by Pickard's method. This can provide maximum parallelization in space and time but it requires many iterations and thus convergence is slow.

The Newton–Raphson method is commonly used on single processor computers but a variety of other methods have been developed to solve the network equations. The best known are the Very Dishonest Newton (VDHN) [41], the SOR–Newton[41], the W–matrix[30,38,43], and Coarse Grain Scheduling [42] methods for solving the algebraic equations, and the Waveform Relaxation or the Dynamic Partitioning methods for solving the whole problem.

The methods for solving network algebraic equations are typically one of the two types: ***direct*** or ***iterative*** as described below. Regardless of what type of network solution method is used, another iterative process is required to combine the solution of differential set of equations with the solution of the network algebraic equations.

2.3.1 Direct Methods

The fundamental problem of a network solution algorithm is the solution of a set of linear algebraic equations given in a matrix form by :

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.4}$$

where matrix **A** is assumed to be large and sparse.

If the inverse of matrix **A** exists, then the general solution for an unknown vector **x** is given by :

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b} \tag{2.5}$$

For inverting a matrix, Gauss–Jordan elimination method [11] is about as efficient as any other method. However, because matrix inversion is a computationally intensive process, most of the modern direct methods do not explicitly compute the inverse for finding a solution.

One of the most effective methods for solving equation (2.4) on serial processors is the triangular factorization used along with forward/backward substitution [9] described as follows :

$$\text{triangular factorization} \quad : \quad \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{U} = \mathbf{A} \tag{2.6}$$

$$\text{forward substitution} \quad : \quad \mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.7}$$

$$\text{backward substitution} \quad : \quad \mathbf{D} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.8}$$

where **D** is a diagonal matrix and **L** and **U** are unit–triangular matrices with ones on diagonal.

Fast solution methods are required for triangular factorization and substitutions because typically they are repeatedly computed for a solution on each iteration. Many new algorithms exploit available parallelism through reordering and partitioning of matrix **A**. Those algorithms, however, are not developed for parallel processing so that a great deal of adaptation is required for applications on parallel processing hardware.

The W–matrix method described in Section 2.4, overcomes the poor parallel characteristics of the substitution scheme. This method generalizes the LDU algorithm by converting

the substitution of (2.7) and (2.8) into matrix–vector multiplications without losing significant sparsity. The matrix–vector multiplications applied in this method, are readily parallelizable making a parallel LDU algorithm possible to develop.

2.3.2 Iterative Methods

For high speed simulation on multiprocessor computers, iterative and node–oriented methods have been used. For these methods the matrix splitting concept is applied to solve the network algebraic equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. Matrix **A** is expressed as a sum of **L**, **D**, and **U** matrices:

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U} \tag{2.9}$$

where **D** is a diagonal matrix and **L** and **U** are triangular matrices with zeros on the diagonal.

For the **Jacobi** method, the n–th iteration step is given by :

$$\mathbf{D} \cdot \mathbf{x}_n = -(\mathbf{L} + \mathbf{U}) \cdot \mathbf{x}_{n-1} + \mathbf{b} \tag{2.10}$$

and the solution for $\mathbf{x}_n$ is given by

$$\mathbf{x}_n = -\mathbf{D}^{-1} \cdot [(\mathbf{L} + \mathbf{U}) \cdot \mathbf{x}_{n-1} + \mathbf{b}] = -\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U}) \cdot \mathbf{x}_{n-1} - \mathbf{D}^{-1} \cdot \mathbf{b} \tag{2.11}$$

The matrix $-\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U})$ is the ***iteration matrix*** which, along with the additive term $-\mathbf{D}^{-1} \cdot \mathbf{b}$, maps one set of **x**'s into the next. The eigenvalues of the iteration matrix reflect the factor by which the amplitude of a particular eigenmode of the undesired residual is suppressed during one iteration [11]. For the relaxation to converge, all eigenvalue modulus must be less than 1. The rate of convergence of the method is set by the rate of slowest–decaying eigenmode, i.e., the factor with largest modulus which is called the ***spectral radius*** and is denoted by ρ_s. In principal, the spectral radius ρ_s can be computed analytically for a given iteration matrix and the number of iterations required to reduce the overall error by a given factor is inversely proportional to the value of the spectral radius. Unfortunately, convergence of the Jacobi method is strongly conditioned by the matrix characteristic and usually requires many iterations. It is, however, a classical method dating back to the last century

and although it is not practical because it converges too slowly, it sets the basis for understanding modern iterative methods.

In the **Gauss–Seidel (GS)** method matrix decomposition takes the following form:

$$(\mathbf{L} + \mathbf{D}) \cdot \mathbf{x}_n = -\mathbf{U} \cdot \mathbf{x}_{n-1} + \mathbf{b} \tag{2.12}$$

The presence of the lower triangular matrix **L** on the left–hand side of the equation follows from the updating procedure which is to use the most recent x's as soon as they are computed. The convergence of this method is twice as fast as the Jacobi method, but the large number of iterations [11] leaves the method still impractical on a serial computers. However, because the problem decomposes to single variables, it may still be useful for parallel processing [3].

More recently better algorithms were developed. One of such method is the **Successive Over–relaxation (SOR)**. It is derived from equation (2.12) by solving for $\mathbf{x}_n$ and calculating the rate of change:

$$\begin{aligned} \mathbf{x}_n - \mathbf{x}_{n-1} &= -(\mathbf{L} + \mathbf{D})^{-1} \cdot [\mathbf{U} \cdot \mathbf{x}_{n-1} - \mathbf{b}] - \mathbf{x}_{n-1} \\ &= -(\mathbf{L} + \mathbf{D})^{-1} \cdot [(\mathbf{L} + \mathbf{D} + \mathbf{U}) \cdot \mathbf{x}_{n-1} - \mathbf{b}] \\ &= -(\mathbf{L} + \mathbf{D})^{-1} \cdot [\mathbf{A} \cdot \mathbf{x}_{n-1} - \mathbf{b}] \end{aligned} \tag{2.13}$$

The correction at iteration step n is defined in terms of ***residual vector*** ξ_{n-1} and ***over–relaxation parameter*** ω by :

$$\mathbf{x}_n = \mathbf{x}_{n-1} - \omega \cdot (\mathbf{L} + \mathbf{D})^{-1} \cdot \xi_{n-1} \tag{2.14}$$

where

$$\xi_{n-1} = \mathbf{A} \cdot \mathbf{x}_{n-1} - \mathbf{b} \tag{2.15}$$

and $0 < \omega < 2$ for the method to converge.

It was shown [11] that if ρ_J is a spectral radius of the Jacobi iteration, then the optimal choice for ω for the SOR method is given by :

$$\omega = \frac{2}{1 + \sqrt{1 + \rho^2_J}} \tag{2.16}$$

The SOR method is hundreds or even thousands times faster than the Jacobi method if the overrelaxation parameter is optimally chosen. The weak point of this method is that this optimal value of ω is usually difficult to determine.

2.4 The Bergeron Method

One alternative iterative method developed by the author in this research, is called Bergeron Method (BM) and is based on the Bergeron line model which utilizes the concept of travelling waves in a load flow and stability solution [16]. In its time–domain formulation, this method is widely used in electromagnetic transients programs [1].

The traditionally used nominal π–circuits do not represent the travelling wave nature of transmission lines and this discrepancy becomes larger as the length of line increases. The distributed nature of the line parameters is automatically taken into account in the Bergeron Line Model in which travelling waves are represented by a pair of Norton sources at each line end as shown in Figure 2.1.

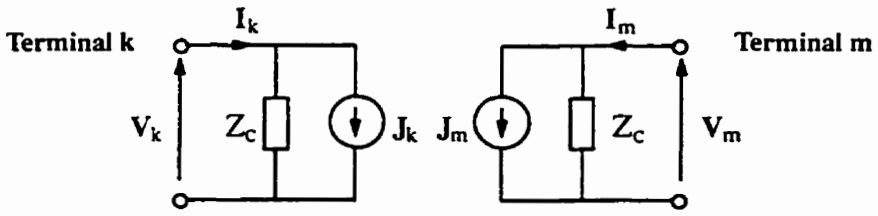


Figure 2.1. Circuit of Bergeron Line Model.

This is the essence of this parallel algorithm. If subsystems of the overall network are connected by transmission lines with travel times exceeding the simulation time–step, a local subsystem is not affected by the happening in the remote subsystem

For a steady–state solution, line parameters are specified by line resistance R, series reactance $X=\omega L$ and shunt susceptance $B=\omega C$. When line length is d, the travel time is :

$$\tau = d / v = \sqrt{L * C} = \sqrt{X * B} / \omega$$

where v is the phase velocity.

For a lossless line (R=0), the impedance Zc in Figure 2.1 is equal to the *surge impedance*:

$$Z_o = \sqrt{L / C} \;\; ; \;\; Y_o = 1 / Z_o \tag{2.17}$$

and the *propagation constant* γ and *electrical length* ϕ are :

$$\gamma = j\omega \sqrt{L*C} = j\beta \;\; ; \tag{2.18}$$
$$\phi = \beta * d$$

From the wave equation in the lossless case we derive the following two equations relating voltage and current phasors at both line ends :

$$\mathbf{V_k} * Y_o \; - \mathbf{I_k} \; = \; \mathbf{F_m} * e^{-j\phi} \tag{2.19}$$

$$\mathbf{V_m} * Y_o \; - \mathbf{I_m} \; = \; \mathbf{F_k} * e^{-j\phi} \tag{2.20}$$

where $\mathbf{F_k}$ and $\mathbf{F_m}$ are forward travelling waves and are defined as

$$\mathbf{F_k} \; = \; \mathbf{V_k} * Y_o + \; \mathbf{I_k} \tag{2.21}$$

$$\mathbf{F_m} \; = \; \mathbf{V_m} * Y_o + \; \mathbf{I_m} \tag{2.22}$$

The above relationship applies not only in the steady–state but also in the transient state when voltage $\mathbf{V_k}(t)$, $\mathbf{V_m}(t)$ and current $\mathbf{I_k}(t)$, $\mathbf{I_m}(t)$ phasors are varying (quasi–statically) in time.

The equivalent current source phasors $\mathbf{J_k}(t)$ and $\mathbf{J_m}(t)$ are determined at time t from the past history at time t–τ. Applying (2.19) and (2.20) to two–port equations at both line ends we obtain :

$$\mathbf{J_k}(t) \; = \; -\mathbf{F_m}(t-\tau) * e^{-j\phi} \tag{2.23}$$

$$\mathbf{J_m}(t) \; = \; -\mathbf{F_k}(t-\tau) * e^{-j\phi} \tag{2.24}$$

For travelling time τ greater than the simulation time–step Δt, forward travelling waves $\mathbf{F_k}$ and $\mathbf{F_m}$ are stored in buffers and are interpolated between discrete points to find appropriate values at time t–τ. When the travelling time is smaller than the time–step, either the line

is treated as a series branch or τ is assumed to be equal to Δt and line susceptance is adjusted to a new value :

$$B_{new} = (\omega * \Delta t)^2 / X \tag{2.25}$$

In this case compensating reactances

$$X_{comp} = 0.5*(B_{new} - B) \tag{2.26}$$

are placed at both line ends so that the overall line impedance is kept unchanged.

For line losses the distributed series resistance R is approximated by lumped resistances and added at both ends of a lossless line (or in few places along the line). When the line is divided in two sections, resistance R/4 is inserted at the end of each half–line section and the line model is defined by the following [1] :

$$Z_C = Z_0 + R / 4 \; ; \; Y_C = 1 / Z_C \tag{2.27}$$

$$\mathbf{J}_k = -(A*\mathbf{F}_k + B*\mathbf{F}_m)*e^{-j\phi} \tag{2.28}$$

$$\mathbf{J}_m = -(A*\mathbf{F}_m + B*\mathbf{F}_k)*e^{-j\phi} \tag{2.29}$$

where

$$\mathbf{F}_k = \mathbf{V}_k*Y_C + H*\mathbf{I}_k \tag{2.30}$$

$$\mathbf{F}_m = \mathbf{V}_m*Y_C + H*\mathbf{I}_m \tag{2.31}$$

$$H = (Z_o - R / 4) / (Z_o + R / 4)$$

$$A = 0.5*(1 - H)$$

$$B = 0.5*(1 + H)$$

Although the line current sources in (2.28) and (2.29) are still defined by voltages from the previous time–steps, as in the Gauss–Seidel method, the delay now represents the actual time for the wave to propagate from one line end to the other.

In the Gauss–Seidel method, because lines are modelled in the form of lumped parameter elements, no specific distinction between branches and lines is made. In the BM method, however, only series branches (such as transformers, series capacitors, reactors or very short lines) and all local loads are grouped in 'clusters' for direct solution by inversion of small

matrices. These clusters and all singular nodes are connected to each other with transmission lines modelled by traveling waves as described above. Iteration in time, according to the traveling wave phenomenon, is applied globally to the system and the network solution for transients is computed until a new steady–state is reached and all standing waves in lines are established. Intermediate time–steps are like iterations in the conventional approach.

The BM algorithm offers a maximum system decoupling by reducing the network admittance matrix **Y** to a 'close–to–diagonal' matrix. Such a reduced matrix consists of only diagonal elements of bus equivalent shunt admittances or small diagonal sub–matrices of cluster admittances. The network solution for bus voltages then becomes a matter of sequential processing of individual nodes and clusters. In addition, this clustering reduces the need for inter–cluster communication, an important aspect for implementation in a parallel processing environment.

2.5 The W–matrix Method

The history of the W–matrix method [30,38,43] is rather short and only a little parallel computer implementation experience has been acquired to fully assess its potential or to reveal its limitations for effective functioning in various parallel environments. One objective for this project was to evaluate computational and communication costs of this method applied on selected parallel and distributed processing hardware.

The largest obstacle to obtain a high speed solution for the stability problem appears to be the repetitive solution of the network equation which is basically the linear algebraic matrix equation (2.4) that with the power system terminology has the following form :

$$\mathbf{Y} \cdot \mathbf{V} = \mathbf{I} \tag{2.32}$$

where : $\mathbf{V}$ – bus voltage vector

$\mathbf{I}$ – bus current injection vector

$\mathbf{Y}$ – network admittance matrix

The objective is to solve the above equation as efficiently as possible. In general, to solve equation (2.32) an inverse of admittance matrix $\mathbf{Y}^{-1}$ is required to calculate bus voltage vector by :

$$\mathbf{V} = \mathbf{Y}^{-1} \cdot \mathbf{I} \tag{2.33}$$

The matrix $\mathbf{Y}$ can be decomposed into factors $\mathbf{L}$, $\mathbf{D}$, and $\mathbf{U}$ as follows :

$$\mathbf{Y} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{U} \tag{2.34}$$

Matrices $\mathbf{L}$ and $\mathbf{U}$ are the lower and upper sparse unit–triangular matrices respectively, and $\mathbf{D}$ is a diagonal matrix. For symmetric admittance matrix, we have $\mathbf{U} = \mathbf{L}^T$.

With the assistance of sparse matrix techniques, the forward/backward substitution method described by equations (2.6) – (2.8) works very efficiently on sequential (single processor) computers. It is, however, very difficult for this algorithm to achieve high efficiency on parallel computers because of the sequential nature of the forward and backward substitutions. The parallel LDU algorithms usually can achieve high efficiency for factorization but are much less powerful on substitutions [8].

The W–method was developed to overcome the poor parallel characteristics of the standard substitution schemes. This method is based on the fact that the inverse of admittance matrix $\mathbf{Y}$ exists as the product of the inverses of factor matrices $\mathbf{U}$, $\mathbf{D}$ and $\mathbf{L}$ according to :

$$\mathbf{Y}^{-1} = \mathbf{U}^{-1} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \tag{2.35}$$

Unlike the inverse of a sparse matrix $\mathbf{Y}$, which is full, the inverses of sparse triangular factors $\mathbf{L}$ and $\mathbf{U}$ are also sparse, though less sparse than the factors themselves.

The solution of (2.32) in this case is obtained by a series of matrix multiplications, instead of substitutions, and in the most primitive form utilizes (2.35) in a two–step process :

$$\mathbf{I}' = \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \cdot \mathbf{I} \tag{2.36}$$

$$\mathbf{V} = \mathbf{U}^{-1} \cdot \mathbf{I}' \tag{2.37}$$

Each of the above matrix–vector multiplications are readily parallelizable. The challenge here is to decompose the above solution process into independent tasks and schedule

them on the processors in such a way as to reduce the communication and synchronization overheads, and to achieve a minimal solution time. It has been recognized that an appropriate reordering and partitioning schemes chosen for a specific hardware architecture can offer significant gains in the computational speed of solving the network equations.

Inverses of **L**, **D** and **U** matrices are found beforehand. Changes in the admittance matrix **Y** resulting from fault, switches, etc., typically require repeated LDU decomposition and LDU inversion of the system admittance matrices. However, admittance changes can also be handled by techniques such as Current Compensation which do not involve expensive matrix reinversions as described in Section 2.6.

For effective computation, some re–ordering method can be applied to **Y** to enhance the sparsity of the inverse matrices $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$. Several algorithms were proposed to minimize the number of elements in the inverse triangular factors [31,32,34]. A simple ordering scheme is chosen for this method as described in Section 2.5.2.

2.5.1 Partitioning in W–matrix Method

The simplest W–matrix solution method for the algebraic equation (2.32) comprises sparsity oriented LDU decomposition of the nxn system admittance matrix $\mathbf{Y} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{U}$ and LDU inverse expressed by $\mathbf{Y}^{-1} = \mathbf{U}^{-1} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1}$. In those expressions, **L** and **U** are lower and upper triangular matrices with unity elements in the diagonals and **D** is a diagonal matrix as illustrated in Figure 2.2.

In order to derive more complex W–matrix solution methods, the unit lower triangle matrix **L** is further decomposed to a product of a series elementary matrices $\mathbf{L}_i$; i = 1, 2,...,n :

$$\mathbf{L} = \mathbf{L}_1 \cdot \mathbf{L}_2 \cdot \ldots \cdot \mathbf{L}_n$$

Each elementary factor matrices $\mathbf{L}_i$ is a modified identity matrix with the i–th column replaced by the i–th column $\boldsymbol{C}_i$ of the factor matrix **L** as illustrated in Figure 2.3.

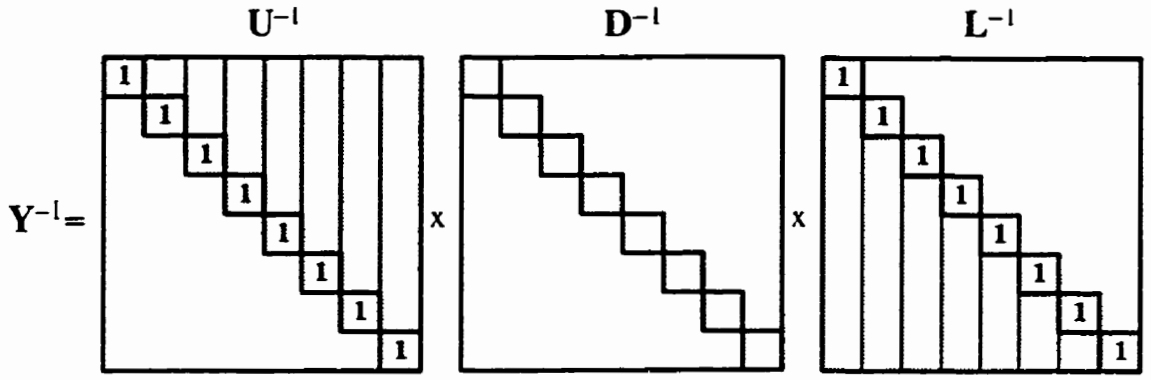


Figure 2.2. Sparsity Oriented LDU Inverse of the Admittance Matrix.

The inverse of lower triangular matrices **L** may then be found as a product of the inverse elementary factor matrices $\mathbf{L}_i^{-1}$:

$$\mathbf{L}^{-1} = \mathbf{L}_n^{-1} \cdot \ldots \cdot \mathbf{L}_2^{-1} \cdot \mathbf{L}_1^{-1} \tag{2.38}$$

The n–th elementary factor matrix is an identity matrix and can be omitted in the above series matrix multiplications.

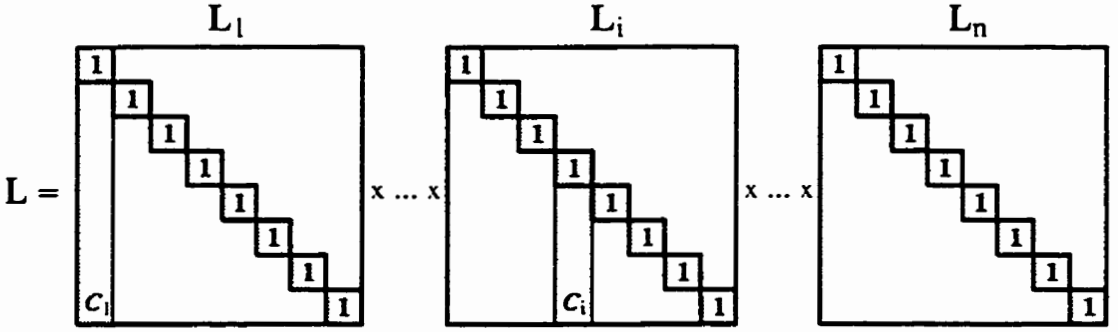


Figure 2.3. Matrix **L** Expanded as a Product of Elementary Factor Matrices.

The inverse elementary factor matrices $\mathbf{L}_i^{-1}$ can indeed be calculated in a very simple manner because they are equal to the original elementary factor matrices $\mathbf{L}_i$ with the off–diagonal entries in column C_i negated. A simple case of 3x3 matrix illustrating the inverse matrix computation by (2.38) is given in the following example :

Example 2.1 :

Let **Y** be a 3x3 symmetric admittance matrix. After factorization, $\mathbf{Y} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{U}$ in which $\mathbf{U} = \mathbf{L}^T$.

The lower triangle matrix **L** and the current injection **I** are given by :

$$\mathbf{L} = \begin{bmatrix} 1 & & \\ a & 1 & \\ c & b & 1 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

L can be expanded as a product of three elementary factor matrices : $\mathbf{L} = \mathbf{L}_1 \cdot \mathbf{L}_2 \cdot \mathbf{L}_3$

$$\mathbf{L} = \begin{bmatrix} 1 & & \\ a & 1 & \\ c & & 1 \end{bmatrix} \times \begin{bmatrix} 1 & & \\ & 1 & \\ & b & 1 \end{bmatrix} \times \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}$$

$$\mathbf{L}_1 \cdot \mathbf{L}_2 \cdot \mathbf{L}_3$$

$\mathbf{L}_1$ = identity matrix + 1–st column of mat

$\mathbf{L}_2$ = identity matrix + 2–nd column of ma

$\mathbf{L}_3$ = identity matrix **I**

The inverses elementary matrices denoted here by $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{W}_3$ are simply computed by reversing signs of the off–diagonal elements in matrices $\mathbf{L}_1$, $\mathbf{L}_2$, and $\mathbf{L}_3$ respectively :

$$\mathbf{W}_1 = \mathbf{L}_1^{-1} = \begin{bmatrix} 1 & & \\ -a & 1 & \\ -c & & 1 \end{bmatrix} \qquad \mathbf{W}_2 = \mathbf{L}_2^{-1} = \begin{bmatrix} 1 & & \\ & 1 & \\ & -b & 1 \end{bmatrix} \qquad \mathbf{W}_3 = \mathbf{L}_3^{-1} = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}$$

The inverse of lower triangular matrix **L** is computed as a product of matrices $\mathbf{W}_i$ by :

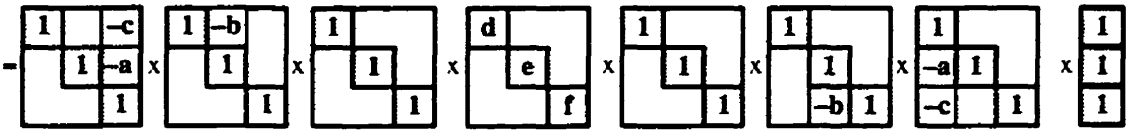
$$\mathbf{L}^{-1} = \mathbf{W}_3 \cdot \mathbf{W}_2 \cdot \mathbf{W}_1 = \begin{bmatrix} 1 & 0 & 0 \\ -a & 1 & 0 \\ ab-c & -b & 1 \end{bmatrix}$$

The above is the right inverse matrix because $\mathbf{L} \cdot \mathbf{L}^{-1}$ is the identity matrix :

$$\mathbf{L} \cdot \mathbf{L}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ a-a & 1 & 0 \\ c-ab+ab-c & b-b & 1 \end{bmatrix} = \mathbf{1}$$

With the inverse factor matrix $\mathbf{L}^{-1}$ computed by a series multiplication of inverse elementary matrices, solution of equation (2.32) requires numerous matrix multiplications which in the above example would be given by :

$$\mathbf{V} = \mathbf{Y}^{-1} \cdot \mathbf{I} = [\, \mathbf{U}^{-1} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \,] \cdot \mathbf{I} = [\, \mathbf{W}_1^{T} \cdot \mathbf{W}_2^{T} \cdot \mathbf{W}_3^{T} \cdot \mathbf{D}^{-1} \cdot \mathbf{W}_3 \cdot \mathbf{W}_2 \cdot \mathbf{W}_1 \,] \cdot \mathbf{I} =$$



The only non–zero elements or fillins used in computation of the inverse matrix $\mathbf{L}^{-1}$ as a product of inverse elementary matrices (2.38) are the negated elements of the original matrix $\mathbf{L}$. Because inverse elementary matrices consist exactly the same number of non–zero elements as their originals, no extra fillins are involved in computation of the inverse triangular matrix $\mathbf{L}^{-1}$. The overall cost of solving equation (2.32), however, is also affected by the number of series matrix multiplications involved in voltage computation.

To reduce the number of series matrix multiplications, groups of consecutive elementary factor matrices $\mathbf{L}_i$ can be pre–multiplied together to form a ***partition***, i.e. the product in equation (2.38) can be broken down into certain number of blocks as shown below :

$$\mathbf{L}^{-1} = \underbrace{[\, \mathbf{L}_n^{-1} \cdot \ldots \cdot \mathbf{L}_{kp}^{-1} \,]}_{\mathbf{W}_p} \cdot \ldots \cdot \underbrace{[\, \mathbf{L}_{(k3-1)}^{-1} \cdot \ldots \cdot \mathbf{L}_{k2}^{-1} \,]}_{\mathbf{W}_2} \cdot \underbrace{[\, \mathbf{L}_{(k2-1)}^{-1} \cdot \ldots \cdot \mathbf{L}_1^{-1} \,]}_{\mathbf{W}_1}$$

A W–matrix is obtained by multiplying all of the elementary matrices within each partition and the inverse matrix can be computed as a product of those W–matrices $\mathbf{W}_j$, $j = 1, \ldots p$:

$$\mathbf{L}^{-1} = \mathbf{W}_p \cdot \ldots \cdot \mathbf{W}_2 \cdot \mathbf{W}_1 \tag{2.39}$$

where

$$\mathbf{W}_p = \mathbf{L}_n^{-1} \cdot \ldots \cdot \mathbf{L}_{kp}^{-1}; \quad \ldots \quad ; \quad \mathbf{W}_2 = \mathbf{L}_{(k3-1)}^{-1} \cdot \ldots \cdot \mathbf{L}_{k2}^{-1}; \quad \mathbf{W}_1 = \mathbf{L}_{(k2-1)}^{-1} \cdot \ldots \cdot \mathbf{L}_1^{-1}$$

This is the most general procedure for factor matrix inversion. In the longest form, it involves all the inverse elementary factor matrices, $\mathbf{W}_i = \mathbf{L}_i^{-1}$ and, although no additional fillins are introduced in this case, it requires the maximum number of series matrix multiplications.

In the shortest form it involves only one W–matrix, $\mathbf{W} = \mathbf{L}^{-1}$, and although the inversion of the whole sparse unit–triangular matrix $\mathbf{L}$ introduces new fillins, no series matrix multiplication is involved.

Partitioning of the upper triangular matrix $\mathbf{U}$ may be performed with W–matrices obtained in a similar fashion. For a symmetric admittance matrix, $\mathbf{U} = \mathbf{L}^T$ and it can be computed by :

$$\mathbf{U}^{-1} = \mathbf{W}_1^T \cdot \mathbf{W}_2^T \cdot \ldots \cdot \mathbf{W}_p^T \tag{2.40}$$

The W–matrices in (2.39) and (2.40) usually remain very sparse, though fillins may be introduced depending on the partitioning scheme used. The network solution can now be expressed by :

$$\begin{aligned} \mathbf{V} &= \mathbf{Y}^{-1} \cdot \mathbf{I} = [\, \mathbf{U}^{-1} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \,] \cdot \mathbf{I} \\ &= [\, \mathbf{W}_1^T \cdot \mathbf{W}_2^T \cdot \ldots \cdot \mathbf{W}_p^T \cdot \mathbf{D}^{-1} \cdot \mathbf{W}_p \cdot \ldots \cdot \mathbf{W}_2 \cdot \mathbf{W}_1 \,] \cdot \mathbf{I} \end{aligned} \tag{2.41}$$

When each matrix $\mathbf{W}_i$ is equal to the inverse of elementary factor matrix $\mathbf{L}_i^{-1}$, then (2.41) is merely an expression of conventional forward and backward substitution. We are free, however, to combine the adjacent $\mathbf{W}$ matrices in any useful way. The W–matrix method generalizes the solution phase of the LDU algorithm which in multiprocessor environments can be utilized to gain computational speed.

Appropriate partitioning methods can maximize sparsity by reducing the number of fillins in W–matrices. In a parallel processing environment, however, this does not necessarily represent a significant saving in processing time because the series matrix multiplications in voltage calculation (2.41) require exchange of results of each multiplication among processors and this consumes a lot of communication time. Effective W–matrix partitioning methods would have to consider not only the amount of computational effort measured by the number of multiplication–addition operations associated with processing the fillins but also take into account other factors such as communication associated with particular implementations of the solution algorithm.

Because series matrix multiplications cost communication time, which is different in different parallel or distributed processing systems, the problem of optimization of processing time in multiprocessor environment strongly depends on what hardware is used for solving the stability problem. The strategy is either to minimize the number of partitions at the expense of increased number of fillins or to minimize the number of fillins at the expense of increased number of partitions.

The factors which may affect the parallel efficiency of the W–matrix method include: size of admittance matrix, number of fillins, number of partitions, structure of the factor matrices and communication time between processors. Those are related issues and the best compromise is the key in determining the parallel efficiency of the W–matrix method. For implementation of the network solution by W–matrix on the RTDS, Distributed Processing Systems (DPS) or other parallel processing hardware the following strategy is proposed :

A) A large system will be split into smaller subsystems, using a system splitting technique described later in Section 2.6, one for each RTDS rack or each single or multiprocessor computer in the DPS.

B) Subsystem admittance matrices will be LDU factorized, and sparsity maximized using a node reordering method. The LDU inverse factor matrices will be computed for each subsytem.

C) A procedure for partitioning the factor matrix will be implemented but only an elementary W–matrix will be applied to each subsystem admittance matrix to reduce the number of communication between processors.

Since at this stage phase–shifting transformers are not considered, the admittance matrix $\mathbf{Y}$ is assumed symmetrical and the W–matrix is equal to the inverse of the subsystem lower triangular factor matrix such that $\mathbf{W} = \mathbf{L}^{-1}$ and $\mathbf{W}^T = \mathbf{U}^{-1} = [\mathbf{L}^T]^{-1}$.

D) Network equations will be solved in a two–step process executed for each subsystem :

$$\mathbf{I}' = \mathbf{D}^{-1} \cdot \mathbf{W} \cdot \mathbf{I} \tag{2.42}$$

$$\mathbf{V} = \mathbf{W}^{T} \cdot \mathbf{I}' \tag{2.43}$$

E) Parallelization of the network solution computation for processors on one RTDS rack or one multicomputer in DSP will be done by assigning a certain number of rows of matrices $\mathbf{W}$ and $\mathbf{W}^{T}$, a partition, to be processed on each processor according to a partitioning scheme which will balance the processors workload.

The following is a brief description of the bus reordering and partitioning methods proposed for high speed transient stability solution.

2.5.2 Node Reordering Scheme

For any sparse matrix $\mathbf{Y} = \mathbf{L} \cdot \mathbf{D} \cdot \mathbf{U}$, the inverse of its factors, $\mathbf{U}^{-1}$ and $\mathbf{L}^{-1}$ usually remains sparse but in addition to the non–zero elements in factors $\mathbf{L}$ and $\mathbf{U}$, new fillins are generated. Fillins are unwelcome since they increase the computation required for the network solution so that any method of minimizing their number is desirable.

Reordering, i.e. pivoting of rows and columns, is an effective tool of reducing the number of fillins in the triangular factors $\mathbf{L}$ and $\mathbf{U}$. Reordering has also been found useful in reducing fillins in W–matrices when combined with proper partitioning scheme. It has been recognized that a smart reordering and partitioning scheme is the key to the success of W–matrix method.

Given the triangular factorization matrix $\mathbf{L}$ or $\mathbf{U}$, new non–zero elements will be created in its inverse $\mathbf{L}^{-1}$ or $\mathbf{U}^{-1}$. The number of new non–zero elements, called "inverse fillins", depends on the ordering of the system nodes. The computation of factor matrices $\mathbf{L}$, $\mathbf{D}$, and $\mathbf{U}$ is thus preceded by node re–ordering or ordering of rows and columns to minimize the number of non zero elements in their inverses.

Several effective schemes have been developed for determining near–optimal orders for inverse fillins in LDU matrices. For this research work, a simple scheme proposed by Tinney [32] has been chosen. This ordering scheme is based on counting the number of branches for

each system node (or counting non–zero elements in rows of admittance matrix $\mathbf{Y}$). Nodes with fewer connections are put at the top of the new order list when the nodes with the most connections (rows with most fillins) are placed at the bottom.

This scheme, although is not fully optimal, still preserves 10–20% sparsity of the inverses matrices $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$ in the most typical cases.

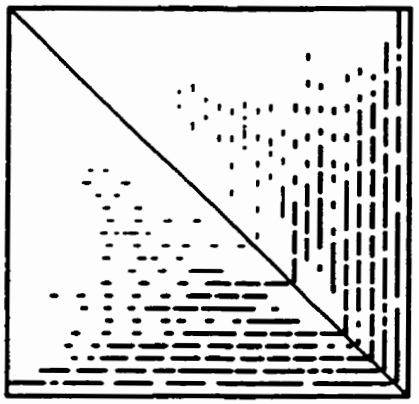


Figure 2.4. Typical Sparse Matrix Density Distribution.

Many of the optimal ordering methods minimize the total number of fillins but the distribution of those fillins is very uneven. A typical example of sparsity distribution is illustrated in Figure 2.4.

Usually there are many buses at the top of the order list that correspond to rows in the admittance matrix with only a few non–zero elements. As we go to the bottom of this list the number of non zero elements increases very quickly and one bus from the bottom of the list can be as expensive as hundreds of buses from the top. This obviously must be taken into account when selecting the partitioning scheme which is supposed to balance loads on all parallel processors involved in the computation.

2.5.3 Balancing Processor Workload – Average Weight Partitioning Scheme

In the proposed W–matrix based solution algorithm, processing of one system node is assumed to be assigned to one processor. It means that all row–times–column multiplications

involved in bus voltage computation as well as solution of all differential equations associated with node generators and loads will be solved on the same processor. Optimal ordering minimizes the total number of fillins but also results in very uneven distribution of matrix sparsity. This uneven sparsity combined with random allocation of system generators and loads results in a very different processing time requirement for the various system buses. Therefore, assigning an equal number of buses to each processor will not be a very efficient way of utilizing the processing time.

A more efficient method is to assign different number of busses to processors by taking into account the bus connectivity, matrix sparsity, and complexity of differential equations. For more optimal balancing of the processor's workload, it is proposed that each system bus i have attributed a weighting factor w_i proportional to :

a) number of fillins in corresponding row of matrices $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$,

b) number and type of generators connected to this bus,

c) number and type of loads connected to this bus,

d) other devices and events such as faults or switchings associated with this bus, and

e) communication time required to exchange data with other processors.

The total time of processing N buses on M processors is proportional to the total weight factor : $\mathbf{w} = \sum_{i=1}^{N} w_i$

which produces an average weight per processor to be equal to :

$$\mathbf{w_a} = \mathbf{w} / \mathrm{M}$$

The average weight $\mathbf{w_a}$ is used as a criterion for partitioning a given list of system buses. This partitioning method groups consecutive nodes in the optimally ordered list of system buses until the sum of their individual weights w_i adds up as close to the average weight $\mathbf{w_a}$ as possible. Assigning each partition for processing on one processor should approximately balance the workload.

This new method is later referred to as the Average Weight (AW) partitioning scheme.

2.6 System Splitting by Bus Tearing Method

The set of algebraic equations describing an electrical system is given by equation (2.32). The size of this matrix equation is often in a range of few thousands. Even with the best sparsity elimination method, the problem size remains large and in addition, a full current injection array and the results of series multiplications by W–matrices must be exchanged between all processors participating in the network solution. The repetitive row–column multiplications, as well as the inter–processor communications, consume the biggest amount of processing time in these cases. In order to achieve a high speed solution of large systems, network splitting is probably unavoidable.

To reduce the size of network matrix equation, the Bus Tearing Method can be applied to split the system into smaller subsystems that are much easier to handle in a multiprocessor environment.

By choosing some of the buses as tearing nodes (also called cut set nodes in some literature [34]), the network can be split into several smaller subsystem. The system admittance matrix Y can be re–arranged so that the tearing buses are located at the bottom as shown in Figure 2.5.

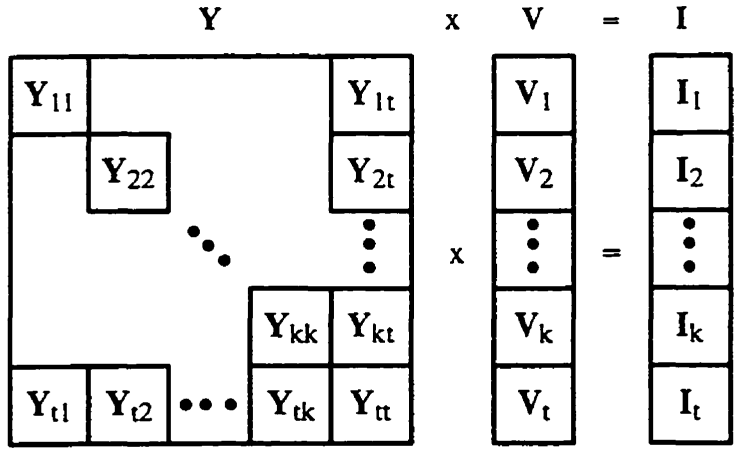


Figure 2.5. Bus Tearing Network Solution Equation.

Here the network is divided into k subsystems represented in the admittance matrix Y by k square matrices Y_{ii}. These subsystems are interconnected through a set of interface

buses which form the tearing subsystem represented by admittance matrix $\mathbf{Y}_{tt}$. Branch connections between subsystems and the tearing buses form the upper and lower border matrices $\mathbf{Y}_{it}$ and $\mathbf{Y}_{ti}$ respectively.

An example of splitting a system of 505 buses into two subsystems grouping 5 system zones is shown in Fig. 2.6. This mechanism for grouping system buses into subsystems is based on the zone information which characterize a physical network. This, however produces a very uneven subsystem size which is not good for balancing the processor's load. A better mechanism developed in this project is based on network connectivity. A desired number subsystems and one interface subsystem are formed by selecting a "seed" bus for each subsystem (which can be from different zones) and the successive adding of neighboring buses connected by branches. The subsystem domains grow like crystals which, after using all system buses, can exchange buses by taking or returning them to the interface group until a balanced set of subsystem with minimized interface subsystem size is reached. The results of this new subsystem splitting method are presented in Table 1 in which the interface subsystem has index 0 and the desired number of subsystems is 1 to 6.

Table 1 : System Splitting for the 505–bus Test System

Number of Subsystems	Subsystem Size						
	0	1	2	3	4	5	6
1	0	505					
2	13	246	246				
3	20	160	165	160			
4	34	118	118	117	118		
5	32	94	93	95	96	95	
6	36	78	79	79	78	77	78

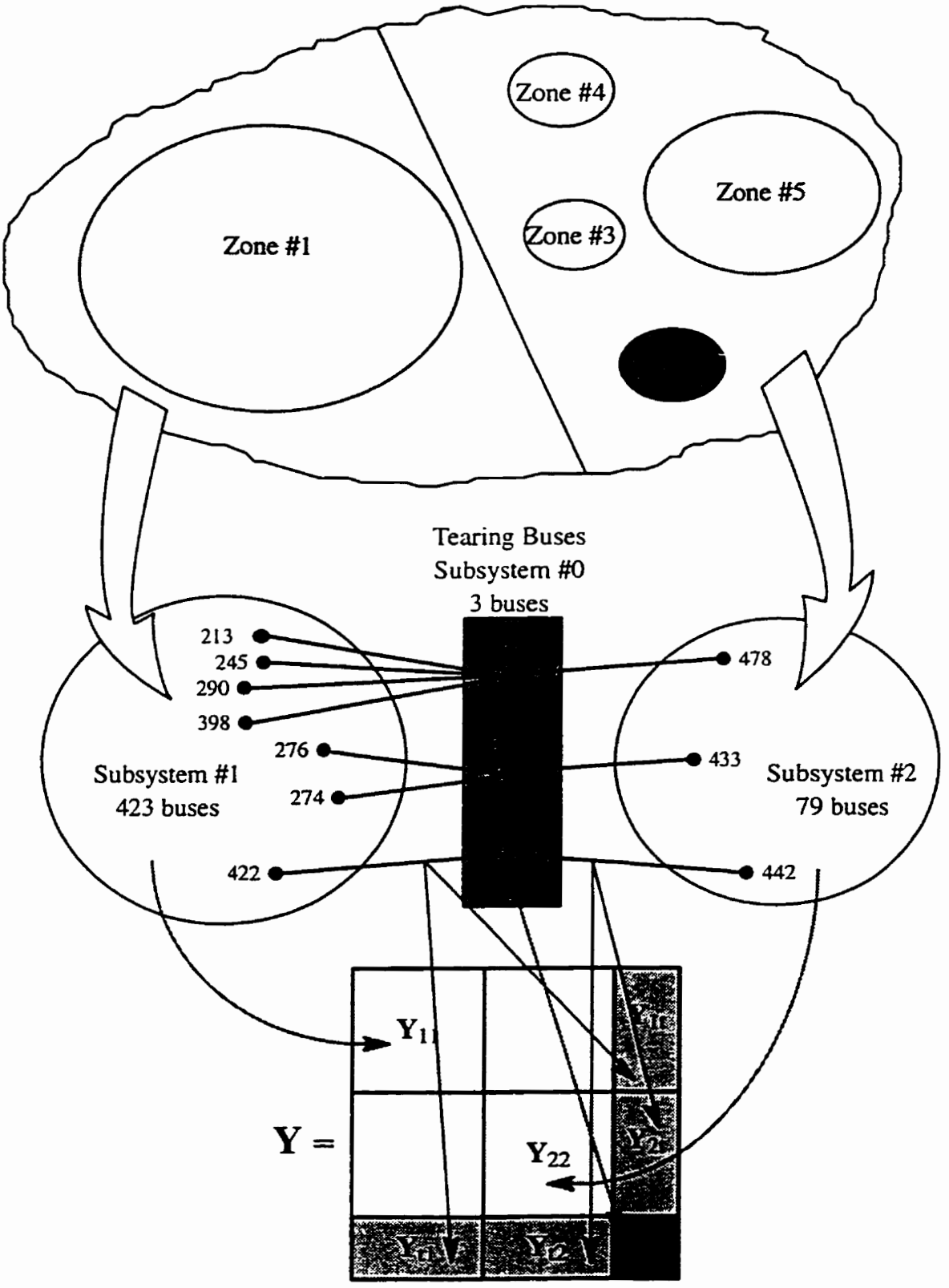
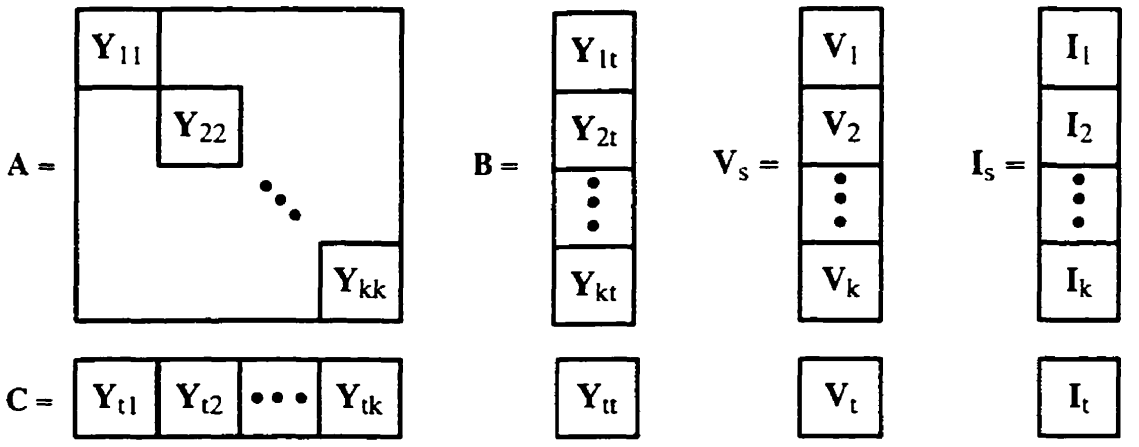


Figure 2.6. Zone Based Network Splitting Method for 505–bus Test System

After splitting system buses into subsystems, the system admittance matrix **Y** is also divided into four parts : the block diagonal matrix **A**, the interface admittance matrix $\mathbf{Y}_{tt}$, and the border matrices as **B** and $\mathbf{B}^T$. Similarly, the voltage and current vectors are divided into two parts corresponding to subsystem and to interface buses :



If the voltages at tearing buses $\mathbf{V}_t$ were known, the bus voltages in a given subsystem would not be directly dependent on current injections in other subsystems and could be solved from the subsystem equation :

$$\mathbf{Y}_{ii} \cdot \mathbf{V}_i = \mathbf{I}_i - \mathbf{Y}_{it} \cdot \mathbf{V}_t \qquad i = 1, 2, \ldots k$$

$$\mathbf{V}_i = [\, \mathbf{Y}_{ii} \,]^{-1} \cdot [\, \mathbf{I}_i - \mathbf{Y}_{it} \cdot \mathbf{V}_t \,] \qquad i = 1, 2, \ldots k \qquad (2.44)$$

The following derivation shows how to find the bus tearing voltage vector $\mathbf{V}_t$ required for solving the subsystem equations (2.44). Using partition matrices defined above, the network equation (2.32) can be re–written as :

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{Y}_{tt} \end{bmatrix} \times \begin{bmatrix} \mathbf{V}_s \\ \mathbf{V}_t \end{bmatrix} = \begin{bmatrix} \mathbf{I}_s \\ \mathbf{I}_t \end{bmatrix}$$

From this we obtain the following two matrix equations :

$$\mathbf{A} \cdot \mathbf{V}_s + \mathbf{B} \cdot \mathbf{V}_t = \mathbf{I}_s \qquad (2.45)$$

$$\mathbf{C} \cdot \mathbf{V}_s + \mathbf{Y}_{tt} \cdot \mathbf{V}_t = \mathbf{I}_t \qquad (2.46)$$

and we can derive $\mathbf{V_s}$ from equation (2.45)

$$\mathbf{V_s} = \mathbf{A^{-1}} \cdot [\, \mathbf{I_s} - \mathbf{B} \cdot \mathbf{V_t} \,] \tag{2.47}$$

Substitute (2.47) into (2.46) produces

$$\mathbf{C} \cdot \mathbf{A^{-1}} \cdot [\, \mathbf{I_s} - \mathbf{B} \cdot \mathbf{V_t} \,] + \mathbf{Y_{tt}} \cdot \mathbf{V_t} = \mathbf{I_t}$$

$$[\, \mathbf{Y_{tt}} - \mathbf{C} \cdot \mathbf{A^{-1}} \cdot \mathbf{B} \,] \cdot \mathbf{V_t} = \mathbf{I_t} - \mathbf{C} \cdot \mathbf{A^{-1}} \cdot \mathbf{I_s}$$

which can be written as a simple matrix equation

$$\mathbf{Y'_{tt}} \cdot \mathbf{V_t} = \mathbf{I'_t} \tag{2.48}$$

with matrix $\mathbf{Y'_{tt}}$ and vector $\mathbf{I'_t}$ defined respectively by :

$$\mathbf{Y'_{tt}} = \mathbf{Y_{tt}} - \mathbf{C} \cdot \mathbf{A^{-1}} \cdot \mathbf{B} = \mathbf{Y_{tt}} - \sum_{i=1}^{k} \mathbf{Y_{ti}} \cdot \mathbf{Y_{ii}^{-1}} \cdot \mathbf{Y^T_{ti}} \tag{2.49}$$

$$\mathbf{I'_t} = \mathbf{I_t} - \mathbf{C} \cdot \mathbf{A^{-1}} \cdot \mathbf{I_s} = \mathbf{I_t} - \sum_{i=1}^{k} \mathbf{Y_{ti}} \cdot \mathbf{Y_{ii}^{-1}} \cdot \mathbf{I_i} \tag{2.50}$$

The **Bus Tearing Procedure** can now be described in the following four steps :

Step 1 : Interface admittance calculation

$$\mathbf{Y'_{tt}} = \mathbf{Y_{tt}} - \sum_{i=1}^{k} \mathbf{Y_{tti}} \tag{2.51}$$

where

$$\mathbf{Y_{tti}} = \mathbf{Y_{ti}} \cdot \mathbf{Y_{ii}^{-1}} \cdot \mathbf{Y^T_{ti}} \qquad i = 1, 2, \ldots k \tag{2.52}$$

Step 2 : Tearing bus current injection calculation

$$\mathbf{I'_t} = \mathbf{I_t} - \sum_{i=1}^{k} \mathbf{I_{ti}} \tag{2.53}$$

where

$$\mathbf{I_{ti}} = \mathbf{Y'_{ti}} \cdot \mathbf{I_i} \tag{2.54}$$

$$\mathbf{Y'_{ti}} = \mathbf{Y_{ti}} \cdot \mathbf{Y_{ii}^{-1}} \qquad i = 1, 2, \ldots k \tag{2.55}$$

Step 3 : Tearing bus voltage calculation

$$\mathbf{V_t} = [\, \mathbf{Y'_{tt}} \,]^{-1} \cdot \mathbf{I'_t} \tag{2.56}$$

Step 4 : Subsystem voltages calculation

$$\mathbf{V}_i = [\ \mathbf{Y}_{ii}\ ]^{-1} \cdot \mathbf{I'}_i \qquad i = 1, 2, \ldots k \tag{2.57}$$

where the subsystem current injections $\mathbf{I'}_i$ are modified by :

$$\mathbf{I'}_i = \mathbf{I}_i - \mathbf{Y}^T_{ti} \cdot \mathbf{V}_t \qquad i = 1, 2, \ldots k \tag{2.58}$$

Using the two–step W–matrix technique for solving equation (2.57) leads to the following modified **Step 4** :

Step 4a : Intermediate serial matrix multiplication

$$\mathbf{I''}_i = \mathbf{D}_i^{-1} \cdot \mathbf{L}_i^{-1} \cdot \mathbf{I'}_i \quad i = 1, 2, \ldots k \tag{2.59}$$

Step 4b : Subsystem voltages calculation

$$\mathbf{V}_i = \mathbf{U}_i^{-1} \cdot \mathbf{I}_i'' \qquad i = 1, 2, \ldots k \tag{2.60}$$

All admittance matrices can be pre–calculated, partitioned and stored in appropriate processor memories. Each subsystem i = 1, 2, ... k is assumed to be solved on one multicomputer (one rack on RTDS). In order to minimize the number of communications, the computations of interface current injections $\mathbf{I'}_t$ and voltages $\mathbf{V}_t$ can be done separately on each processor. Typically, this is a problem of a very small size and the required computation will take less time than the communication between processors required otherwise. When all processors solve the interface voltages by themselves, the only information that needs to be exchanged between multicomputers or RTDS racks are the amounts of the subsytem current injections from subsystems to tearing buses $\mathbf{I}_{ti}$ in Step 2.

Since the interface bus voltages are made locally known on each processor, current injections for the partition buses on each processors $\mathbf{I'}_i$ can be updated. However, for the voltage computation in Step 4 a full subsystem current vector is needed. This requires communication, but only among processors on the same multicomputer / rack. One additional communication within each multicomputer / rack is also required between the Steps 4a and 4b for exchanging the results of serial matrix multiplication.

2.7 Handling Admittance Changes

There are three methods being used or considered for handling admittance changes in the network solution algorithms most commonly applied in stability programs. Those methods are :

1. Re–computing the **Y** matrix inverse by LDU decomposition/inversion whenever an admittance change takes place.
   – this method is used in most off–line commercial stability programs.

2. Pre–calculating all necessary combinations of the inverse of matrix **Y** which may be caused by changes during a simulation run and extracting the appropriate matrices from memory when the change takes place.
   – this method is used in the RTDS simulator for electromagnetic transients.

3. Current Compensation method which converts changes in admittance matrix **Y** into equivalent modifications of bus current injections **I**.
   – this method is reported in [14] and is used in the stability program from CRIEPI [21].

Each of the above methods requires certain amount of storage space and execution time. The first method does not require much storage space but it is computationally very expensive because the decomposition as well as the factor matrix inversion must be recomputed and this process is hard to parallelize. The second method is just the opposite. It requires lots of memory for storing whole inverse matrices for various combination of admittance change but execution time is minimally affected.

The third method requires some extra storage and processing time as described in the following section. Due to the limited size of the DSP memory in the current architecture of the RTDS, it is considered to be the best compromise to allow on–line admittance changes in this parallel implementation of the High Speed Transient Stability program.

2.7.1 Current Compensation Methods

The problem being considered here again is the solution of equation (2.32) for voltage vector **V** when the admittance matrix changes due to switching or a fault so that the new equation is given by :

$$(\mathbf{Y} + \Delta\mathbf{Y}) \cdot \mathbf{V} = \mathbf{I} \tag{2.61}$$

where :

Y – is a sparse n×n network admittance matrix

$\Delta\mathbf{Y}$ – is a modification to it involving one or more network elements, and

I – is the current injection vector.

A direct solution of equation (2.61) normally requires matrix reinversion and is given by :

$$\mathbf{V} = (\mathbf{Y} + \Delta\mathbf{Y})^{-1} \cdot \mathbf{I} \tag{2.62}$$

The objective is to solve the above equation as efficiently as possible, preferably without the necessity of matrix reinversion. Providing that the modification does not involve too many elements and does not need to be permanently incorporated in the network equation, the solution can be obtained more economically without repeating the expensive LDU factorization and factor reinversion.

In order to derive such a method, modifications to the network admittance matrix in equation (2.62) are written in a compact form as :

$$\Delta\mathbf{Y} = \mathbf{M} \cdot \delta\mathbf{y} \cdot \mathbf{M}^{T} \tag{2.63}$$

where :

$\delta\mathbf{y}$ – m×m matrix consisting of the amounts of all admittance changes

M – n×m connection matrix consisting only the integers 0, 1, and –1

For a symmetric change of branch admittance between nodes i and k, an admittance change of Δy is added to Y_{ii} and Y_{kk}, and subtracted from Y_{ik} and Y_{ki} elements of the system admittance matrix **Y**. The modification in this simple case can be expressed in two different

manners : Branch Oriented Modification and Node Oriented Modification as described below.

A. Branch–Oriented Modification

This representation is used only with symmetrical admittance matrices. For a single branch admittance change between nodes i and k the admittance modification matrix $\mathbf{\Delta Y}$ is given by :

$$\Delta Y = \underbrace{\begin{bmatrix} \cdots \\ +1 \\ \cdots \\ -1 \\ \cdots \end{bmatrix}}_{\mathbf{M}} \begin{matrix} \\ i \\ \\ k \\ \\ \end{matrix} \cdot \times \underbrace{\begin{bmatrix} \Delta y \end{bmatrix}}_{\delta y} \cdot \times \underbrace{\begin{bmatrix} \cdots & +1 & \cdots & -1 & \cdots \end{bmatrix}}_{\mathbf{M}^T} \quad (+1 \text{ at column } i,\ -1 \text{ at column } k)$$

When m branches are modified simultaneously, δy becomes an mxm diagonal matrix, and **M** has m columns, each with entries +1 and −1 in the relevant positions. A shunt representing fault would have only the +1 entry.

B. Node–Oriented Modification

This is a more general representation which can also be used for non–symmetrical admittance modifications. In this case a single branch admittance change between nodes i and k in admittance modification matrix $\mathbf{\Delta Y}$ is given by :

$$\Delta Y = \underbrace{\begin{bmatrix} \cdots & \cdots \\ +1 & 0 \\ \cdots & \cdots \\ 0 & +1 \\ \cdots & \cdots \end{bmatrix}}_{\mathbf{M}} \begin{matrix} \\ i \\ \\ k \\ \\ \end{matrix} \cdot \times \underbrace{\begin{bmatrix} \Delta y & -\Delta y \\ -\Delta y & \Delta y \end{bmatrix}}_{\delta y} \cdot \times \underbrace{\begin{bmatrix} \cdots & +1 & 0 & \cdots \\ \cdots & 0 & +1 & \cdots \end{bmatrix}}_{\mathbf{M}^T} \quad (\text{columns } i,\ k)$$

Node–Oriented modification representation is chosen for handling the admittance changes in the high speed transient stability algorithm because it is more general.

Changes of branch and shunt elements which are applied simultaneously, can be combined together, each contributing to one or two elements of the admittance change matrix $\delta\mathbf{y}$.

In order to avoid computationally expensive matrix reinversion a method has been derived which requires only the original inverse matrix $\mathbf{Y}^{-1}$ for solving equation (2.61). This method is known as the **Current Compensation Method** [14] and is fully equivalent to a direct solution given by (2.62). It is derived from this equation by the following matrix transformations :

$$
\begin{aligned}
\mathbf{V} &= (\,\mathbf{Y} + \Delta\mathbf{Y}\,)^{-1}\cdot\mathbf{I} \\
&= [(\mathbf{1} + \Delta\mathbf{Y}\cdot\mathbf{Y}^{-1}\,)\cdot\mathbf{Y}\,]^{-1}\cdot\mathbf{I} \\
&= \mathbf{Y}^{-1}\cdot(\mathbf{1} + \Delta\mathbf{Y}\cdot\mathbf{Y}^{-1})^{-1}\cdot\mathbf{I} \qquad \text{let} \quad \mathbf{X} = \Delta\mathbf{Y}\cdot\mathbf{Y}^{-1} \\
&= \mathbf{Y}^{-1}\cdot(\mathbf{1} + \mathbf{X})^{-1}\cdot\mathbf{I} \\
&= \mathbf{Y}^{-1}\cdot(\mathbf{1} + \mathbf{X})^{-1}\cdot[\mathbf{1} + \mathbf{X} - \mathbf{X}]\cdot\mathbf{I} \qquad \text{where} \quad [\mathbf{1} + \mathbf{X} - \mathbf{X}] = \mathbf{1} \\
&= \mathbf{Y}^{-1}\cdot[\,\mathbf{1} - (\mathbf{1} + \mathbf{X})^{-1}\cdot\mathbf{X}\,]\cdot\mathbf{I} \qquad \text{let} \quad \mathbf{F} = (\mathbf{1} + \mathbf{X})^{-1}\cdot\mathbf{X} \\
&= \mathbf{Y}^{-1}\cdot[\,\mathbf{1} - \mathbf{F}\,]\cdot\mathbf{I}
\end{aligned}
$$

Matrix $\mathbf{F}$ states the amount of current injection $\mathbf{I}$ that must be modified due to the admittance change by $\Delta\mathbf{Y}$. The formula for solving network equation (2.61) is now given by:

$$\mathbf{V} = \mathbf{Y}^{-1}\cdot[\,\mathbf{I} + \Delta\mathbf{I}\,] \tag{2.64}$$

where $\Delta\mathbf{I} = -\mathbf{F}\cdot\mathbf{I}$ and

$$\mathbf{F} = (\mathbf{1} + \mathbf{X})^{-1}\cdot\mathbf{X} = (\mathbf{1} + \Delta\mathbf{Y}\cdot\mathbf{Y}^{-1})^{-1}\cdot\Delta\mathbf{Y}\cdot\mathbf{Y}^{-1}$$

Computation of matrix $\mathbf{F}$ can be further simplified by substituting the definition (2.63) for the admittance modification $\Delta\mathbf{Y}$ and further algebraic rearrangement to obtain :

$$\mathbf{F} = \mathbf{M}\cdot\mathbf{c}\cdot\mathbf{M}^{T}\cdot\mathbf{Y}^{-1} \tag{2.65}$$

where

$$\mathbf{c} = [\,(\delta\mathbf{y})^{-1} + \mathbf{z}\,]^{-1} \tag{2.66}$$

$$\mathbf{z} = \mathbf{M}^{T}\cdot\mathbf{Y}^{-1}\cdot\mathbf{M} \tag{2.67}$$

The general solution of equation (2.61) has now the following form :

$$\mathbf{V} = (\mathbf{Y}^{-1} - \mathbf{Y}^{-1} \cdot \mathbf{M} \cdot \mathbf{c} \cdot \mathbf{M}^{T} \cdot \mathbf{Y}^{-1}) \cdot \mathbf{I} \qquad (2.68)$$

Three main current compensation methods are derived from equation (2.68) by different computational arrangements :

A. Post–compensation Method

$$\mathbf{V} = \{ \mathbf{1} - \mathbf{Y}^{-1} \cdot \mathbf{Mc} \cdot \mathbf{M}^{T} \} \cdot \mathbf{Y}^{-1} \cdot \mathbf{I} \qquad (2.68a)$$

B. Pre–compensation Method

$$\mathbf{V} = \mathbf{Y}^{-1} \cdot \{ \mathbf{1} - \mathbf{M} \cdot \mathbf{c} \cdot \mathbf{M}^{T} \cdot \mathbf{Y}^{-1} \} \cdot \mathbf{I} \qquad (2.68b)$$

C. Mid–compensation Method : $\mathbf{Y}^{-1}$ replaced by factors $\mathbf{U}^{-1} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1}$

$$\mathbf{V} = \mathbf{U}^{-1} \cdot \{ \mathbf{1} - \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \cdot \mathbf{M} \cdot \mathbf{c} \cdot \mathbf{M}^{T} \cdot \mathbf{U}^{-1} \} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \cdot \mathbf{I} \qquad (2.68c)$$

The expression in the parentheses in the above equations is an $n_x n$ matrix representing the compensation. Method B is convenient for use in combination with the W–matrix network solution. First current $\mathbf{I}$ is calculated from dynamic equations and then it is modified by the current compensation method by the amount of $\Delta \mathbf{I}$:

$$\Delta \mathbf{I} = -\mathbf{F} \cdot \mathbf{I} = -\mathbf{M} \cdot \mathbf{c} \cdot \mathbf{M}^{T} \cdot \mathbf{Y}^{-1} \cdot \mathbf{I} \qquad (2.69)$$

The W–matrix network solution is applied to the modified current injection $\mathbf{I} + \Delta \mathbf{I}$ in the described earlier two–step calculation :

$$\mathbf{I}' = \mathbf{D}^{-1} \cdot \mathbf{W} \cdot (\mathbf{I} + \Delta \mathbf{I}) \qquad \text{where } \mathbf{W} = \mathbf{L}^{-1}$$

$$\mathbf{V} = \mathbf{W}^{T} \cdot \mathbf{I}'$$

A variety of fault conditions can be modelled and corresponding matrices $\mathbf{F}$ can be pre–calculated and stored for use during the simulation run. The initial inverse matrix $\mathbf{Y}^{-1}$ is required for computation of matrix $\mathbf{F}$. When a fault at bus i is applied, only i–th row of $\mathbf{Y}^{-1}$ is used and for a branch switching between bus i and bus j, both i–th and j–th rows are used.

When the system admittance matrix is split by the Bus Tearing method, the required impedances have to be collected from all subsystem matrices. In this case it is easier to use the

physical meaning of impedance. In order to find the elements of i–th row of matrix $\mathbf{Z} = \mathbf{Y}^{-1}$, the physical meaning of impedance can be viewed as a system response to a unit injection current at bus i, i.e. elements of i–th row of system impedance matrix is equal to the vector of system bus voltages **V** when the current injection **I** is zero everywhere except for bus i where it is equal to 1.0 :

$$\mathbf{V} = \mathbf{U}^{-1} \cdot \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \cdot \mathbf{I} \qquad \text{where} \qquad \mathbf{I} = [\ 0\ 0 \ldots\ 1 \ldots\ 0\ 0\ 0\ ]^T$$

When the system is split by Bus Tearing Method, the above computation includes also the intermediate steps for bus tearing computation and current injection modifications as described earlier in the Bus Tearing procedure by equations (2.53) – (2.58).

If the admittance change is not known ahead of time, the compensation matrix **F** must be computed on-line during the simulation run. For on-line computation of matrix **F**, a single admittance change would require an amount of computation equivalent to one network solution for system voltages at fixed currents i.e. excluding the dynamic equation. An alternative method for on-line computations of matrix re–inversion is also presented in the following section.

2.7.2. Adjustment of the Inverse Matrix

If the admittance matrix modification involves too many elements or the change is permanent, it may be less expensive to apply a fast method for matrix re–inversion once rather than apply the Current Compensation method continuously for all the following time–steps of a simulation run. One alternative method for calculating the inverse matrix was proposed by Sherman and Morrison [28,29] in 1949.

Computational effort for obtaining the inverse of a matrix would be reduced considerably if the inverse could be transformed in a simple manner, corresponding to some specific change in the original matrix. If one element is changing in the original matrix the resulting changes in the elements of the new inverse can be computed from the old inverse as shown below.

Consider n–th order square matrix **A** and its inverse $\mathbf{B} = \mathbf{A}^{-1}$. Also denote

a_{ij} – elements of matrix **A** ; i,j = 1,2, ..., n

b_{ij} – elements of matrix **B** ; i,j = 1,2, ..., n

Suppose that the element a_{IJ} has changed by an amount of Δa_{IJ} so that the new value is :

$$A_{IJ} = a_{IJ} + \Delta a_{IJ}$$

When the original matrix **A** changes to $\mathbf{A'} = \mathbf{A} + \Delta\mathbf{A}$, the elements of new inverse matrix **B'** can be computed based on the previous elements by :

$$B_{ij} = b_{ij} - b_{iJ} \cdot b_{Ij} \cdot G_{IJ} ; \quad i,j = 1,2, ..., n \tag{2.70}$$

where :

$$G_{IJ} = \Delta a_{IJ} / (1.0 + b_{IJ} \cdot \Delta a_{IJ}) \tag{2.71}$$

providing that $1.0 + b_{IJ} \cdot \Delta a_{IJ}$ is not equal to zero

Equations (2.70) are conveniently subdivided into three groups :

a) i = I : $B_{Ij} = b_{Ij} \cdot H_{IJ} ; \quad j = 1,2, ..., n$ (2.72)

b) j = J : $B_{iJ} = b_{iJ} \cdot H_{IJ} ; \quad i = 1,2, ..., n$ (2.73)

c) all others : $B_{ij} = b_{ij} - B_{iJ} \cdot b_{Ij} \cdot H_{IJ} ; \quad i,j = 1,2, ..., n, \; i \neq I, j \neq J$ (2.74)

where : $H_{IJ} = 1.0 / (1.0 + b_{IJ} \cdot \Delta a_{IJ})$ (2.75)

If two or more elements are to be changed, the new inverse can be found by successive applications of the method.

Single change of one diagonal element :

Suppose that a diagonal element a_{II} has been changed by :

$$A_{II} = a_{II} + \delta$$

Employ Sherman–Morrison to the original inverse **B** requires the following update :

$$B_{ij} = b_{ij} - b_{iI} \cdot b_{Ij} \cdot G_{II} ; \quad i,j = 1,2, ..., n \tag{2.76}$$

where :

$$G_{II} = \delta / (1.0 + b_{II} \cdot \delta) \tag{2.77}$$

Change of one diagonal element by δ and one off–diagonal in the same row by $-\delta$:

Suppose that a diagonal element a_{II} and an off–diagonal element a_{IJ} have been changed by:

$$A_{II} = a_{II} + \delta$$

$$A_{IJ} = a_{IJ} - \delta$$

Employ Sherman–Morrison to original inverse **B** requires the following update :

$$B_{ij} = b_{ij} - (b_{iJ} - b_{iI}) \cdot b_{Ij} \cdot G_{II} ; \qquad i,j = 1,2, ..., n$$

where :

$$G_{II} = \delta \; / \; [\; 1.0 + (b_{II} - b_{IJ}) \cdot \delta \;]$$

Illustrative example :

When branch admittance is changed by Δy, four elements of the admittance matrix are affected. For the branch between nodes k and m those elements are :

diagonal elements :

$$A_{kk} = a_{kk} + \Delta y$$

$$A_{mm} = a_{mm} + \Delta y$$

off–diagonal elements :

$$A_{km} = a_{km} - \Delta y$$

$$A_{mk} = a_{mk} - \Delta y$$

For a change of diagonal element A_{kk} the equation takes the form :

$$B_{ij} = b_{ij} - b_{ik} \cdot b_{kj} \cdot G_{kk} ; \quad i,j = 1,2, ..., n$$

where :

$$G_{kk} = \Delta y \; / \; (\; 1.0 + b_{kk} \cdot \Delta y \;)$$

Applying this formula once more for change of A_{mm} produces :

$$B_{ij} = b_{ij} - b_{ik} \cdot b_{kj} \cdot G_{kk} + b_{im} \cdot b_{mj} \cdot G_{mm} ; \qquad i,j = 1,2, ..., n$$

where :

$$G_{mm} = \Delta y \; / \; (\; 1.0 + b_{mm} \cdot \Delta y \;)$$

Chapter 3

High Speed Transient Stability (HSTS) Program

3.1 HSTS Algorithm and Program Structure

For high speed transient stability solution of large systems a new algorithm has been developed and incorporated in the HSTS program written in the 'C' computer language. The original version of the HSTS program was aimed at running on single processor workstations, but because it was developed with parallel processing in mind, further implementations on the RTDS and the Distributed Processing Systems were implemented with minimal modifications to the original program. This new algorithm combines several methods important for parallel processing applications which include the following techniques as introduced in the previous Chapter:

A. LDU–decomposition and LDU–inverse for processing sparse matrices,
B. W–matrix method for solving network equations (2–step procedure),
C. Re–ordering scheme to minimize number of fill–ins in the W–matrices,
D. Bus Tearing method for system splitting the large network into smaller subsystems,
E. Current Compensation method for handling system admittance changes, and
F. Partitioning scheme for solving one subsystem with many processors operating in parallel.

The decomposition of the network admittance matrix Y, into factors L, D, and U is a purely sequential computation and is very difficult to parallelize as mentioned in the previous chapters. In the proposed method, this decomposition is conducted off–line by the host computer before the actual simulation run begins. For handling the admittance change during

simulation run, the Current Compensation method is applied which does not require matrix decomposition or inversion but instead uses the pre–computed matrix **F** as described in Section 2.7.1.

The network solution, without system splitting, is simply a multiplication of large sparse matrices by a vector. When full matrices are used in computation, this problem by its nature is perfectly straightforward for parallel processing with each row–column multiplication implemented on separate vector processors. However, with large system admittance matrices we have to utilize the sparsity to improve computation efficiency. The LDU–decomposition with a re–ordering scheme to minimize fill–ins in W–matrices produces uneven distribution of sparsity in the admittance matrix i.e. processing of one row is more expensive than the other. In addition, there are also differential equations associated with different system nodes so that the work load may differ even more from one bus to the other. The Average Weight Partitioning scheme described in section 2.5.3 can be applied to balance the processor load.

Since effective parallel processing compilers do not yet exist, the HSTS program is generically structured to allow application on most common parallel or distributing processing systems. This program is modularized and can be easily reconfigured and implemented on different types of parallel processing computers, including the RTDS. Logic is placed in the program so that it can be compiled by the "C" compilers available on single processor computers and executed in a multiprocessor environment either parallel or distributed.

The complete HSTS algorithm with flowcharts is presented below :

HSTS Algorithm :

Step 1 : *Initial Calculation (not parallelized, executed on the host computer)*

The HSTS program residing on a host workstation computer reads and interprets the data file which is assumed to be in the PSS/E stability program data format and performs the following tasks to initialize data for each processor participating in the parallel processing :

1a) Split the system into R subsystems one for each RTDS rack or multicomputer

1b) Form R subsystem admittance matrices $\mathbf{Y}_{ii}$, i = 1, ... R and interface matrices $\mathbf{Y}_{tt}$, $\mathbf{Y}_{ti}$, i = 1, ... R

1c) Apply reordering scheme and LDU decomposition to produce triangular and diagonal matrices $\mathbf{L}_i$, $\mathbf{D}_i$, $\mathbf{U}_i$ for each subsystem i = 1, ... R.

1d) Compute LDU inverses and W–matrices $\mathbf{W}_i = \mathbf{L}_i^{-1}$ and $\mathbf{W}^T = \mathbf{U}_t^{-1}$

1d) Calculate admittance matrices $\mathbf{Y}_{tti}$, $\mathbf{Y'}_{ti}$, i=1, ... R and $\mathbf{Y'}_{tt}$ required for System Splitting by Bus Tearing and defined by :

$$\mathbf{Y}_{tti} = \mathbf{Y}_{ti} \cdot \mathbf{Y}_{ii}^{-1} \cdot \mathbf{Y}^T_{ti} = \mathbf{Y}_{ti} \cdot \mathbf{W}_i^T \cdot \mathbf{D}_i^{-1} \cdot \mathbf{W}_i \cdot \mathbf{Y}^T_{ti} \qquad \text{as of eq. (2.52)}$$

$$\mathbf{Y'}_{ti} = \mathbf{Y}_{ti} \cdot \mathbf{Y}_{ii}^{-1} = \mathbf{Y}_{ti} \cdot \mathbf{W}_i^T \cdot \mathbf{D}_i^{-1} \cdot \mathbf{W}_i \qquad \text{as of eq. (2.55)}$$

$$\mathbf{Y'}_{tt} = \mathbf{Y}_{tt} - \sum_{i=1}^{k} \mathbf{Y}_{tti} \qquad \text{as of eq. (2.51)}$$

1e) Compute $\mathbf{Z}_{tt} = [\ \mathbf{Y'}_{tt}\ ]^{-1} = \mathbf{L}_t^{-1} \cdot \mathbf{D}_t^{-1} \cdot \mathbf{U}_t^{-1}$

1f) Partition sub–systems using Average Weight partitioning scheme to balanced workload to P processors on each RTDS rack or network multicomputer.

1g) Initialize processor Local and Global Data Memories

Step 2 : *Current Injections to Subsystem Buses*

Each processor solves its own partition of subsystem buses and calls dynamic models to integrate the associated differential equations. The resulting current injection for their nodes are then fed into the appropriate buses in the following order :

2a) Solve system differential equations using previous state vector $\mathbf{X}$ and the most recent bus voltages $\mathbf{V}_i$ to obtain new current injections $\mathbf{I}_i$ to system buses,

2b) Update currents for non–linear loads, DC links and other system devices,

2c) Apply Current Compensation using pre–computed matrix $\mathbf{F}$ if system admittance change took place.

Step 3 : *Current Injections to Interface Subsystem*

The original current injections to the interface buses, due to generators and other devices connected to them, must also be updated by the injections due to the overall effect of subsystems that they are connected to.

3a) Compute the amount of current injections $\mathbf{I}_{ti}$ to tearing buses due to the subsystem currents $\mathbf{I}_i$:

$$\mathbf{I}_{ti} = \mathbf{Y'}_{it} \cdot \mathbf{I}_i \qquad \text{as of eq. (2.54)}$$

3b) Transfer the amount of current injections $\mathbf{I}_{ti}$ to each processor solving the subsystem for local computation of interface bus voltages.

Step 4 : *Tearing Bus Voltages*

Each processor computes all tearing bus voltages locally in order to reduce communication costs.

4a) Read the amounts of current injections $\mathbf{I}_{ti}$ from other processor to tearing buses

4b) Update current injections to tearing buses

$$\mathbf{I'}_t = \mathbf{I}_t - \sum_{i=1}^{k} \mathbf{I}_{ti} \qquad \text{as of eq. (2.53)}$$

4c) Compute tearing bus voltages (solving equation 2.48)

$$\mathbf{V}_t = \mathbf{Z}_{tt} \cdot \mathbf{I'}_t \qquad \text{as of eq. (2.56)}$$

Step 5 : *Subsystem Current Injections*

Since voltages $\mathbf{V}_t$ are computed locally by each processor, the system current injections for buses within each processor partition can be updated without communication.

5a) Compute the amount of current injections $\mathbf{I'}_{it}$ to subsystem buses from the interface buses:

$$\mathbf{I'}_{it} = \mathbf{Y}^{T}_{ti} \cdot \mathbf{V}_t \qquad \text{as of eq. (2.58)}$$

5b) Update subsystem current injections :

$$\mathbf{I'}_i = \mathbf{I}_i - \mathbf{I'}_{it} \qquad \text{as of eq. (2.58)}$$

5c) Transfer current injections $\mathbf{I'}_i$ to each processor in the same rack / multicomputer.

Step 6 : *Subsystem Voltage Computation – Part I.*

6a) Compute the intermediate product of series matrix multiplication in W–matrix solution:

$$\mathbf{I''}_i = \mathbf{D}_i^{-1} \cdot \mathbf{W}_i \cdot \mathbf{I'}_i \qquad \text{as of eq. (2.59)}$$

6b) Transfer intermediate product $\mathbf{I''}_i$ to each processor in the same rack / multicomputer.

Step 7 : *Subsystem Voltage Computation – Part II.*

7a) Compute subsystem voltages from the second part of series matrix multiplication :

$$\mathbf{V}_i = \mathbf{W}_i^T \cdot \mathbf{I}_i'' \qquad \text{as of eq. (2.60)}$$

7b) Upload subsytem voltages $\mathbf{V}_i$ to host computer for monitoring

A flow chart for the algorithm implementation in the HSTS program is presented below. Shown in Figure 3.1 is Part A of the HSTS program flowchart. This initialization part is executed before entering the Time Loop and consists of reading and interpreting the input data files as well as forming all matrices and arrays required for the solution algorithm.

For parallel or distributed processing systems this is an integral part of the HSTS Compiler which generates Download File for each processor involved in the stability solution. The format of this file is different for different implementations but in each case it consists mapped memory contents necessary for each processor to perform the assigned task computations.

When running the program on a single processor workstation, the download files are not generated and this part is immediately followed by the solution part performed on the same computer. The solution part consisting of solving all system differential and algebraic equations is embraced by the ***Time Loop***. The convergence of the differential–algebraic network solution is achieved by repeating the computations a few times within the ***Iteration Loop***.

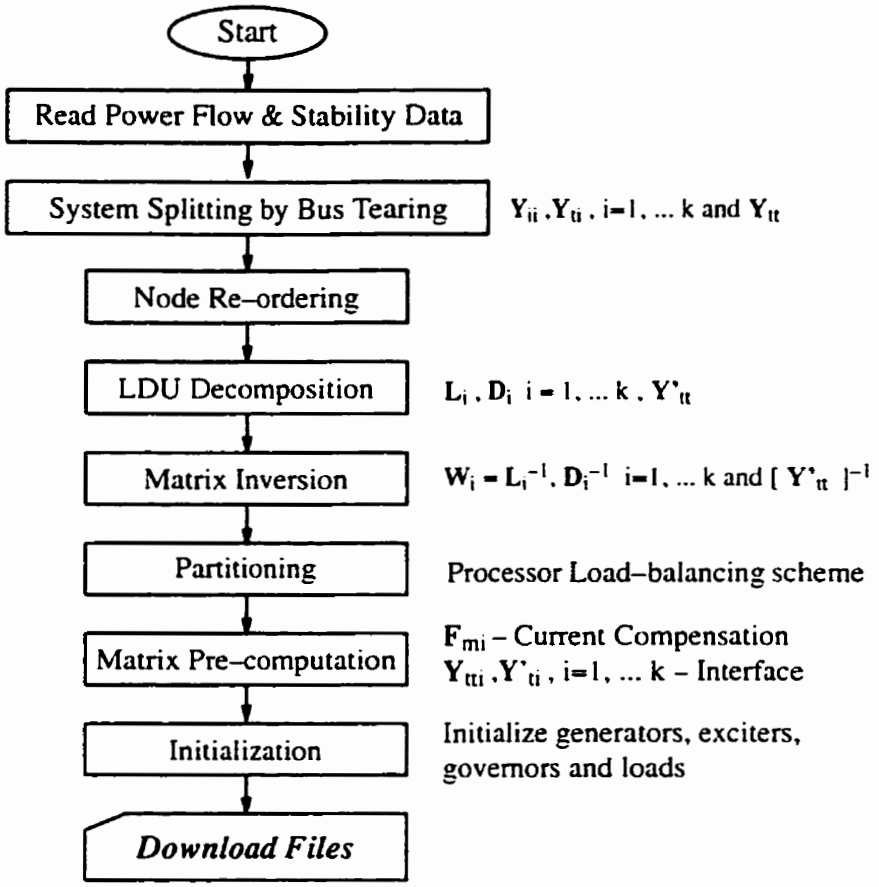


Figure 3.1. High Speed Transient Stability Program – Part A : HSTS Compiler. System Splitting, Node Ordering, LDU Decomposition and Inverse.

There are two types of computations involved in the solution part. One type is performed only occasionally when system events take place and other is the routine computation of currents and voltages. The solution is thus divided into two parts :

Part B : System Admittance Change

Part C : Network Solution Methods

Part B is executed only when events such as faults, switchings or other system operation changes take place. This part is shown in Figure 3.2 One full network solution has to be performed every time system admittance is changed. Current compensation computation is initialized and it is continuously applied in Part C for the duration of an event.

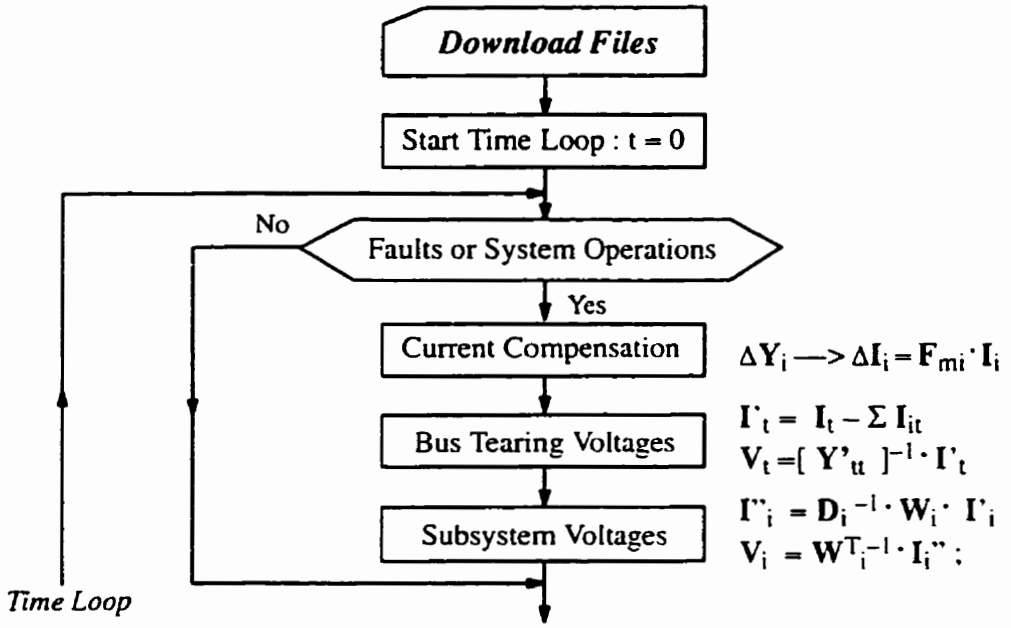


Figure 3.2. High Speed Transient Stability Program – Part B : HSTS Solution. System Admittance Change.

Part C begins with Pre–calculations block which initializes the Iteration Loop. The repetitive solution of the network differential and algebraic equations is performed within the Iteration Loop until either a termination criterion is met or the maximum number of iterations is reached. Since this part is repeated many times during a simulation run, computational efficiency takes the highest priority. The flowchart for this part is shown in Figure 3.3.

One rack of RTDS processors is equivalent to a multicomputer consisting P processors. The looping for processing the tasks by processors j = 1, ..., P on racks i= 1, ...,R shown in the chart, are targets for parallelization in the multiprocessor environment. The number of buses on each processors will be determined by the partitioning scheme which balances the processor workload.

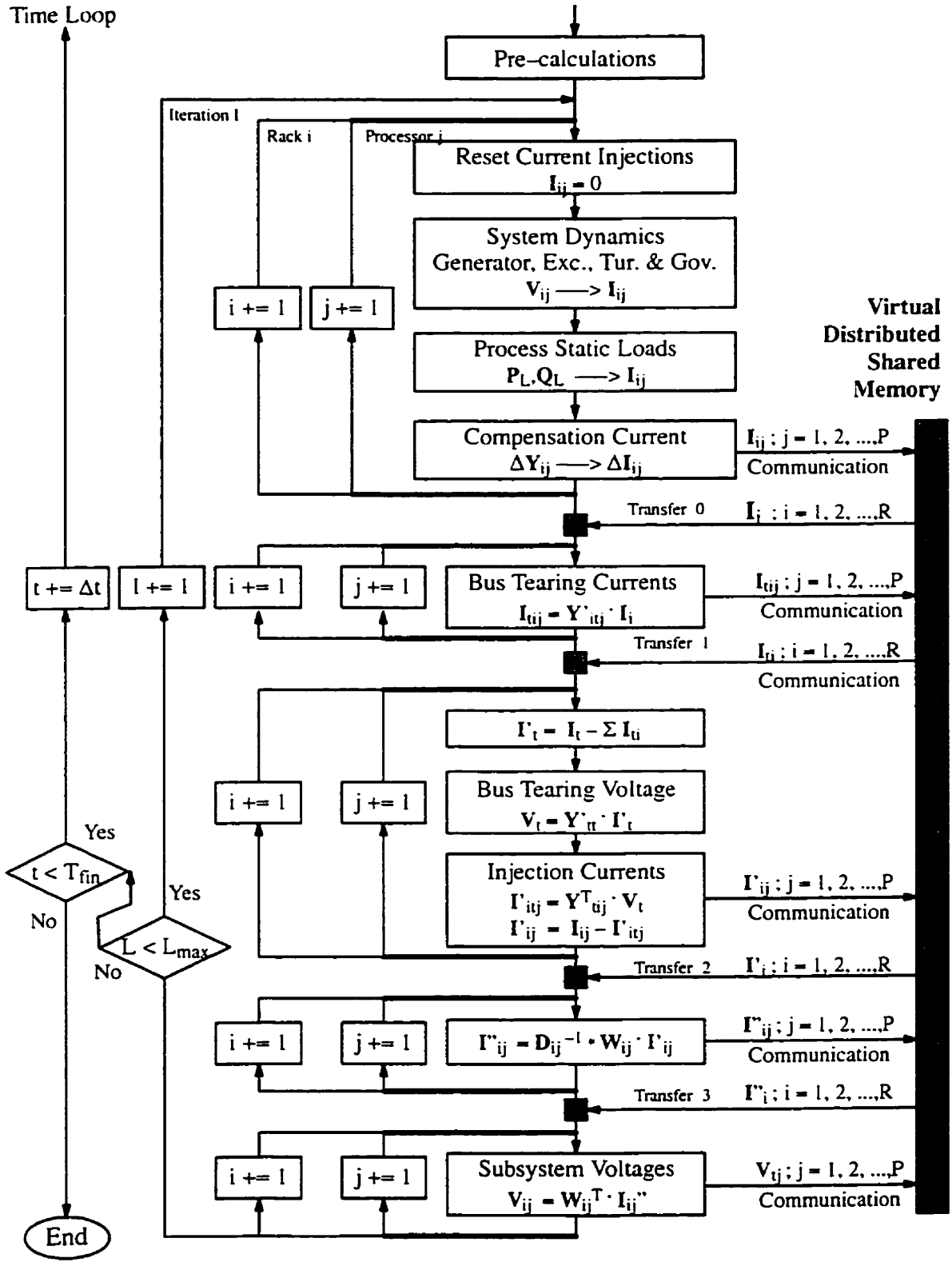


Figure 3.3. High Speed Transient Stability Program – Part C : HSTS Solution.

W–matrix & Current Compensation Network Solution Method.

Communication links are different in nature for the RTDS and the Distributed Processing Systems. For RTDS, the exchange of information among processors on the same rack are made via the Virtual Distributed Shared Memory which is transparent to all processors on the same rack and also its IRC part is also made transparent to processors on other racks. For the Distributed Processing Systems communication between processors requires cross–network links.

3.2 HSTS Validation Tests on Workstation and PC

The network solution applied in the HSTS program has been tested on a selected 505–bus test system against the conventional stability programs such as BPA, PSS/E, and PSDS. The test system consists of 505 busses, 704 branches , 435 transformers, 52 generators, and 126 loads of various types.

In the test, a three phase 2–cycle solid fault was applied to bus 344 with varying damping factor D. The same tests were performed with HSTS and BPA stability programs. The BPA program could be run only on UNIX Workstation, where the HSTS program was run on both the UNIX workstation and the PC computers. System splitting was applied in HSTS cases to observe its effect on execution time.

Simulation results using the HSTS program match closely the steady–state and post–fault curves of other stability programs. Plots for selected 4 bus voltages and for 4 machine angles for the BPA and HSTS cases are shown in Figures 3.4 and 3.5. A very good agreement between the two programs can be observed for system bus voltages. Some differences in the post fault swing curves are observed which are due to differences in dynamic model representations. Similar differences were also observed between BPA and PSS/E or PSDS results and thus they are considered to be acceptable. Since the dynamic modelling was not a primary objective for this research, this matter was not investigated further.

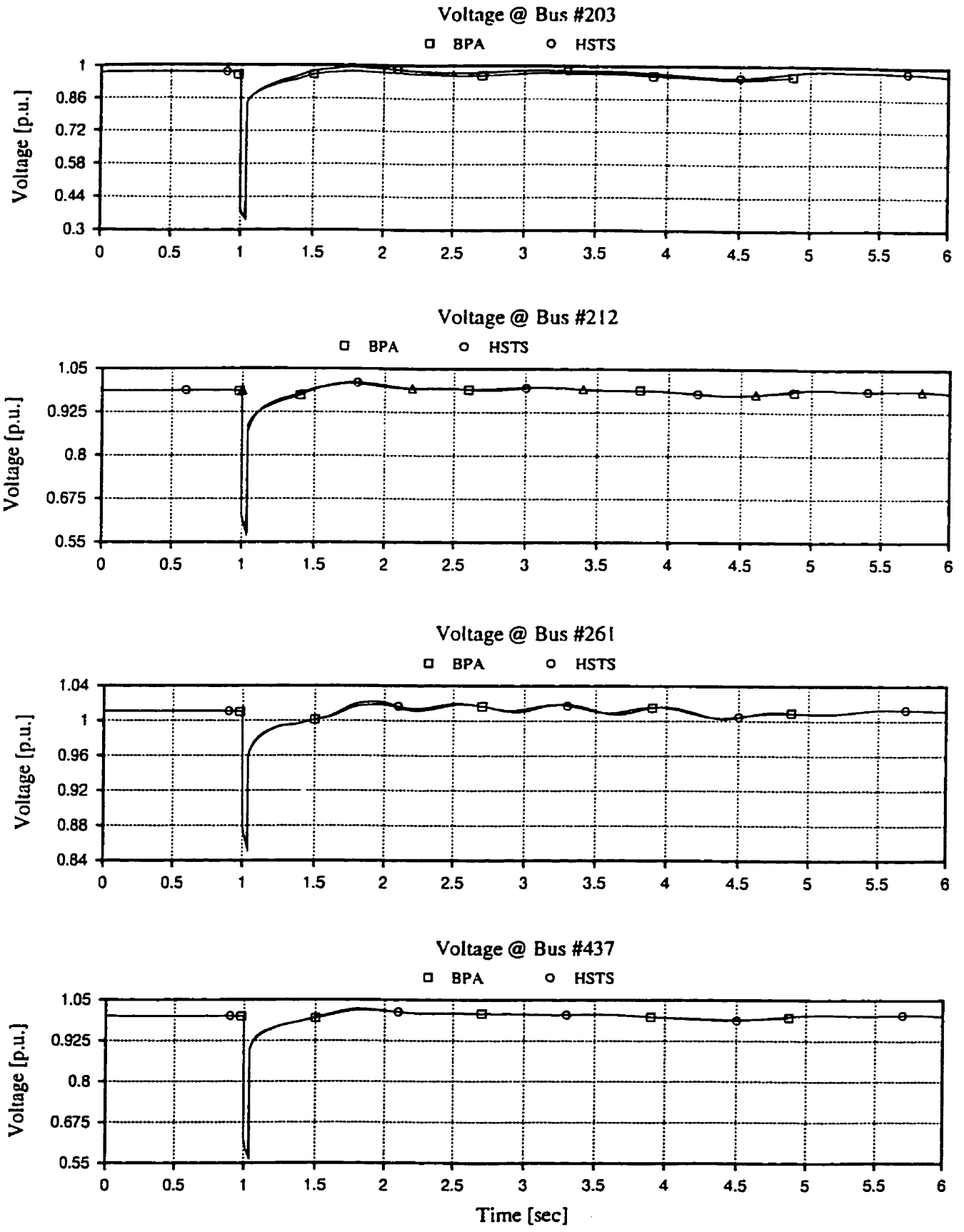


Figure 3.4. BPA and HSTS Comparison – System Bus Voltages.

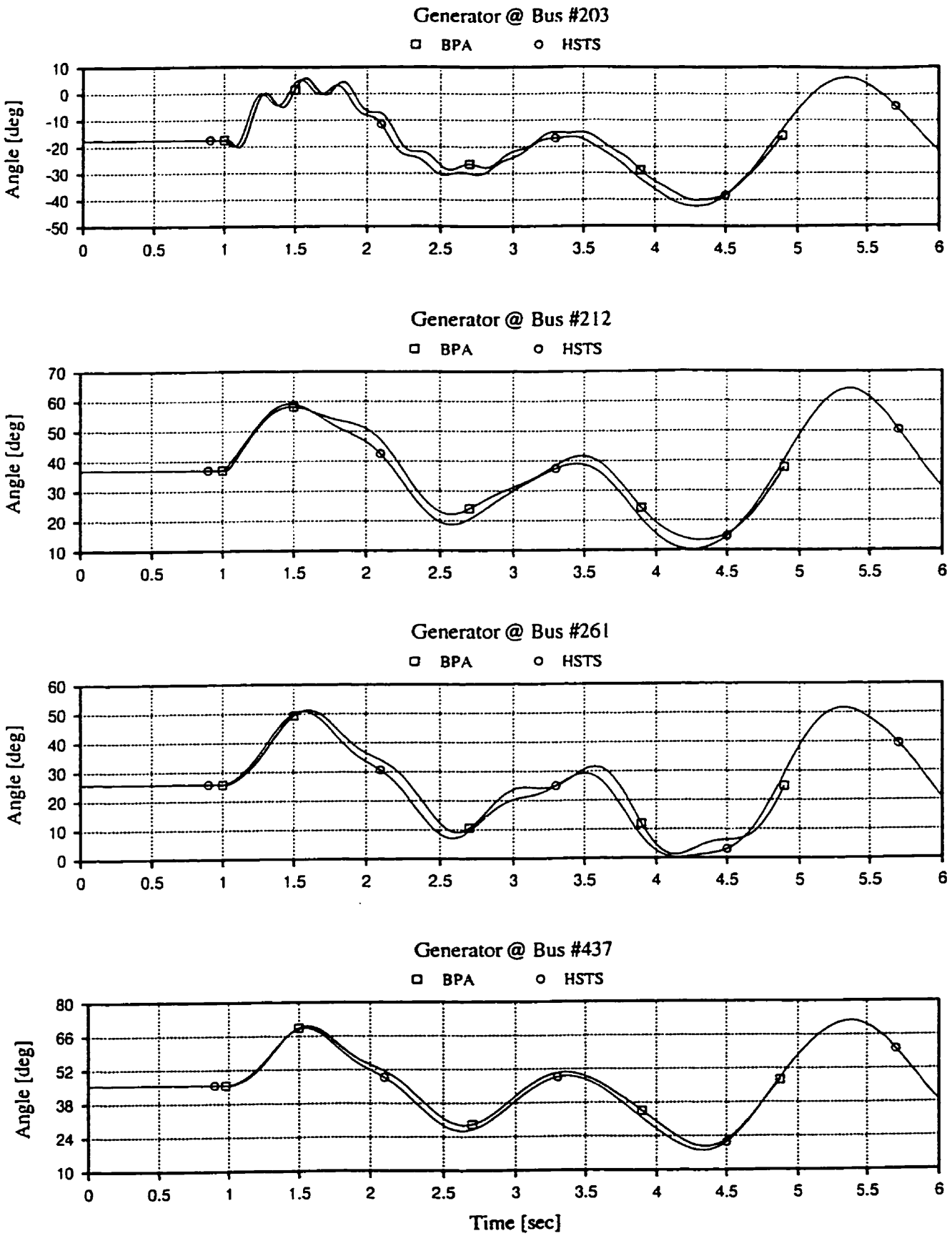


Figure 3.5. BPA and HSTS Comparison – Machine Angles.

Table 2 : BPA Execution Time on UNIX Workstation

BPA Case	Execution Time	
	Average Time–step	Total
1 subsystem	162.5 msec	195 sec

Table 3 : HSTS Execution Time on UNIX workstation

HSTS–UNIX Cases	Execution Time		
	Min. per time–step	Max. per time–step	Total
1 subsystem	23.1 msec	590.2 msec	76.3 sec
2 subsystems	15.4 msec	381.3 msec	50.8 sec
3 subsystems	16.7 msec	416.5 msec	55.0 sec
4 subsystems	16.3msec	409.7 msec	54.3 sec
5 subsystems	15.8 msec	393.0 msec	52.1 sec
6 subsystems	20.9 msec	535.6 msec	68.9 sec

Table 4 : HSTS Execution Time on PC (without MPI communication)

HSTS–PC Cases	Execution Time		
	Min. per time–step	Max. per time–step	Total
1 subsystem	12.2 msec	330.1 msec	43.0 sec
2 subsystems	9.3 msec	247.6 msec	34.8 sec
3 subsystems	8.5 msec	222.9 msec	32.5 sec
4 subsystems	8.9 msec	233.3 msec	33.8 sec
5 subsystems	9.3 msec	244.2 msec	36.0 sec
6 subsystems	10.7 msec	281.8 msec	41.0 sec

Simulation cases : 505–bus test system, 6–second run, time–step $\Delta t = 5$ msec

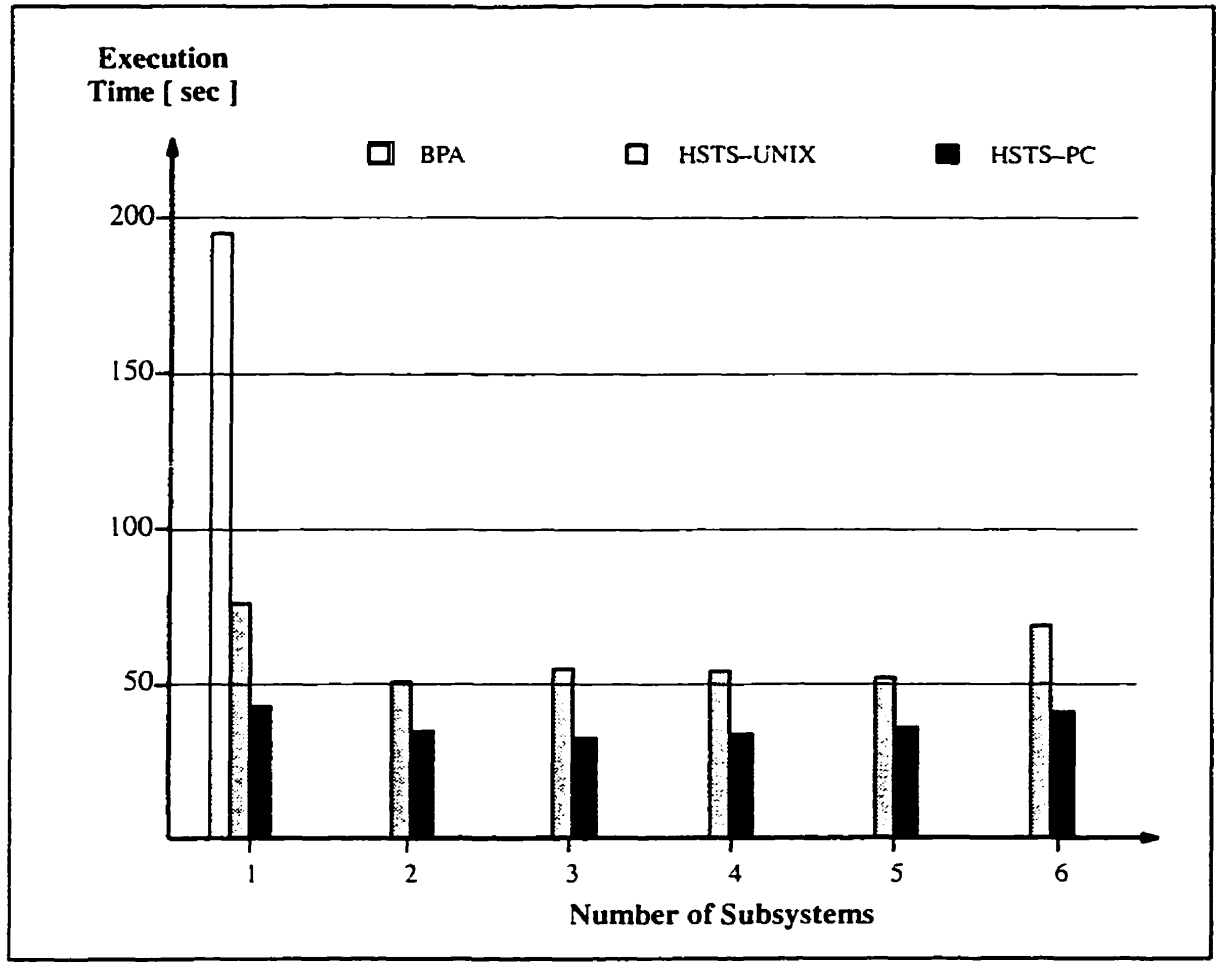


Figure 3.6. Total Execution Time of a 6–second Simulation Run for BPA and HSTS Transient Stability Programs Solving 505–bus Test System

The execution times for BPA and HSTS cases were measured. The CPU time comparison for a 6–second simulation run with a time–step Δt = 5ms on the Sparc2 UNIX workstation and the 200MHz Pentium PC are presented in Tables 2–4 and Figure 3.6. The HSTS program takes approximately 25% of the time required by the BPA program on the same Sparc2 workstation machine. Further reduction of approximately 40% is achieved by running the HSTS program on a 200MHz Pentium PC. It can also be observed that the system splitting itself produces computational savings because the LDU inverse is applied to smaller subsystem matrices and results in less fillins that improves the overall system matrix sparsity. Those gains, however, can be overwhelmed by the amount of extra work required to process interface buses which generally increase in number as the number of subsystems grows.

Chapter 4

HSTS Program Implementations

In this Chapter, parallel implementations of the High Speed Transient Stability (HSTS) program are discussed.

4.1 Parallel Processing for Power System Problems

The following definition of parallel computer was given by Ian T. Foster [10] and is used here because it is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks and workstations and multiple–processor workstations:

> *A **Parallel Computer** is a set of processors that are able to work cooperatively to solve a computational problem.*

The need for faster computers is driven by the demands of both computation–intensive and data–intensive applications and the problem of transient stability solution of large power system networks falls in these categories. The performance of the fastest computers in the market is growing exponentially with a factor of about 10 in every five years. The performance of a computer depends directly on the time required to perform a basic operation and the number of these basic operations that can be performed concurrently. The time to perform a basic operation is ultimately limited by processor 'clock cycle' time which is decreasing rapidly but for recent computers it is already approaching physical limits (such as due to the speed of light). To circumvent these limitations new strategies are being developed by the Very Large Scale Integration (VLSI) complexity theory to utilize internal concurrency in a chip.

These new strategies, however, are expensive because in order to decrease the processing time T by a certain factor, the total area A of a chip must be increased by the square of that factor [10] ie. the product AT^2 remains approximately constant. This AT^2 result means that

not only is it difficult to build individual components that operate faster, it may not even be desirable to do so. For example, if we have an area n^2A of silicon to use in a computer, we can either built n^2 components, each of size A performing an operation in time T, or built a single component of size n^2A performing the same operation in time T/n. The system of n^2 components is potentially n times faster than the single component.

A variety of techniques used to overcome the performance limitations on a single computer include: ***Pipelining*** (different stages of several instructions execute concurrently) and ***Multiple Function Units*** (several multipliers, adders, etc., are invoked by a single instruction stream).

Another important trend in computing is the enormous increase in the capability of networks that connect computers. By the end of the 1990s, bandwidth in excess of 10 Gbits per second is expected to be commonplace. This trend makes it feasible to develop applications that use physically distributed resources as if they were part of the same computer.

Although ***Distributed Computing*** differs from ***Parallel Computing*** the basic task of developing programs that can run on many processors is common for both. Programs for multi–processing can share processor resources, data code and devices. The fundamental requirements for parallel software includes : ***concurrency***, ***scalability***, ***locality*** and ***modularity***. These properties are briefly discussed below.

Concurrency, which refers to the sharing of resources in the same time frame, becomes a fundamental requirement for algorithms and programs for multiple processors located not only inside each computer but also across a network.

Programs for parallel computing may experience substantial increase in processor count over the lifetime of the target hardware. Therefore, ***scalability***, or resilience to increasing processor count, becomes another important feature for protecting software investments.

A basic model used to represent a single machine is the ***von Neumann Computer***. This model comprises a central processing unit (CPU) connected to a storage unit (Memory) as shown in Figure 4.1.

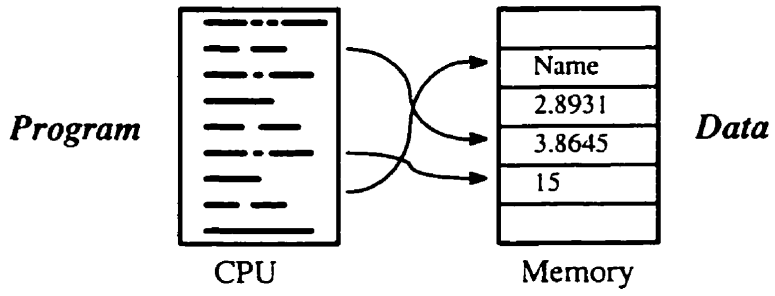


Figure 4.1. The von Neumann Computer.

The CPU executes a stored ***program*** that specifies a sequence of read and write operations on various type of ***data*** stored in the memory. This simple model has proved remarkably robust for many years.

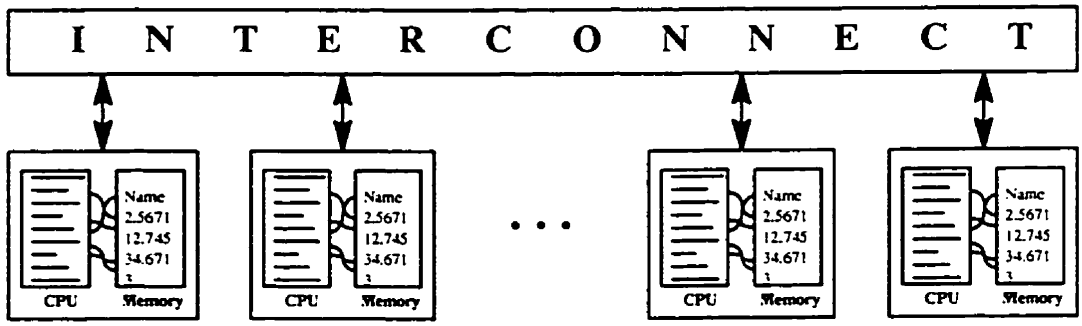


Figure 4.2. Idealized Parallel Model of Multicomputer.

A parallel machine model or ***Multicomputer*** comprises of a number of von Neumann computers, or ***nodes***, linked by an ***Interconnecting Network*** as shown in Figure 4.2. Each computer executes its own program, accesses local memory and may send and receive messages over the network. Messages are used to communicate with other computers and to read and write remote memories.

Accesses to local memory are less expensive than accesses to remote memory. That is, read and write are less costly than send and receive. This property is called ***locality*** and is the third fundamental requirement for parallel software.

While it is possible to program a single node computer in terms of sequential instructions, modular design techniques are applied for parallel programming models. Complex programs are constructed from simple components and components are structured in terms of higher–level abstractions such as data structures, iterative loops or procedures. Parallel processing introduces additional sources of complexity that deal with the problem of how to manage the execution of many processors and coordinate inter–processor interactions. In this context ***modularity*** becomes the fourth fundamental requirement for parallel software.

In a parallel programming model mechanisms are needed that allow concurrency and locality and that facilitate development of scalable and modular programs. Certain abstractions are needed that are simple to work with and that match the architectural model of the multicomputer. For this purpose two abstractions fit these requirements particularly well : the ***task*** and the ***channel.*** These abstract terms permit discussion about concurrency, locality, and communication in a machine–independent fashion and provide a basis for the modular construction of parallel programs.

Parallel computation consists of many tasks that can be executed concurrently. A task encapsulates a sequential program and local memory (virtual von Neumann machine) and a set of ***inports*** and ***outports*** define its interface to its environment. Tasks can perform basic operations like reading and writing or sending and receiving messages. Outport–inport pairs can be connected by message queues forming communication channels between tasks.

The multicomputer parallel machine models using the task/channel programming approach are widely used in parallel algorithm design, analysis, and implementation. Tasks can be mapped to physical processors in various ways. One or more tasks can be mapped to a single processor. In parallel programming model, channel indicates that computation in

one task requires data in another task (*data dependency*) in order to proceed with the task execution.

In spite of the rapid progress made in hardware technology for building a new generation of computers to date, it is still very difficult and painful to program computers for parallel processing. Few shared–memory computers such as the Balance and Alliant have resident compilers for limited set of languages (FORTRAN or C) which can convert software to parallelized execution modules with no effort on the part of the user. Since conventional programs are often not written with parallelism in mind, gains can only be slightly greater than 1.0 regardless of the number of processors used. In order to achieve significant gains by parallelization, modern software development must concern itself with issues of concurrency, scalability, locality, and modularity as discussed earlier.

Software development for parallel processing on the existing computers is currently very time consuming due to lack of parallel programing tools, debuggers, and efficient parallel compilers. Program code must be broken down into parallel tasks manually and distributed to each processor. Synchronization of all processes and all data communication is under programmer control. The code developed in such a manual way is thoroughly optimized by taking advantage of parallelism, which may not be visible to an unsophisticated compiler. Typically, a program developed for one processor is not transparent to other local memory machine because language extensions have not been standardized.

Parallel processing hardware architecture considers two general system classes :

a) Single Instruction Multiple Data (SIMD) class of machines with shared or local memory.

   SIMD class includes vector processors such as the Cray, IBM 3090/VF and also MPP, Connection Machine, etc.

b) Multiple Instruction Multiple Data (MIMD) class of machines with shared or local memory.

   MIMD class with local memory includes Distributed Processing Systems and such ma-

chines as the iPSC and NCube. MIMD class with shared memory includes the BBN Butterfly, Balance, Encore, Alliant FX–8, etc.

MIMD class includes also specialized architectures such as those designed for neural networks, transputers and the parallel vector processors which use more than one vector pipeline simultaneously. Distributed Processing Systems are an important subclass of MIMD machines.

Algorithm development includes the design and analysis of new numerical and symbolic methods to match existing or new architectures. It also involves testing algorithms to ensure accuracy and evaluation of algorithms performances. The performance of many parallel applications depends critically on the quality of the partitioning scheme used for decomposing calculation tasks across the processors of a parallel computer. Application performance is directly linked to the progress of the slowest partitioned part of the calculation and to whether all partitions are the same in terms of storage size and work load. Identifying and implementing the appropriate partitioning algorithm is crucial to ensuring high processor performance.

The parallel processing algorithm may begin to show its limitations with increasing problem complexity by requiring a great deal of execution time or not providing accurate results. The number of instructions in a program segment, or a ***grain***, can be used as a simplest measure of computational intensity. ***Grain size*** or ***granularity*** is commonly described as *fine*, *medium*, and *coarse* depending on the processing levels involved.

Latency is a time measure of the communication overhead incurred between machine subsystems. There are different types of latencies :

– ***memory latency*** – time required by a processor to access the memory,

– ***broadcast latency*** – time required for a processor to send a message to other processors,

– ***synchronization latency*** – time required for two processors to synchronize each other.

Computational granularity and communication latency are closely related and they both affect parallelization performance. By balancing granularity and latency, one can achieve better performance of a computer system. Various latencies are attributed to machine archi-

tecture, implementation technology, and communication patterns involved. The latency can in fact impose a limiting factor on the scalability of the machine size.

The complexity of an algorithm for solving complex problems on a computer is determined by the execution time and the storage space required. The ***execution time*** of a parallel program is defined as the time that elapses from when the first processor starts executing a problem solving program to when the last processor completes the execution. During execution, each processor spends a certain amount of time for computing, communicating, or idling. The ideal performance of a computer system demands a perfect match between machine capability and program characteristics. The testing of parallel algorithms must be done on actual parallel machines because complex parallel architectures and communication schemes are difficult to simulate on a sequential machines.

In more complex algorithms with variable amounts of work per tasks and unstructured communication patterns, efficient agglomeration and mapping strategies may not be obvious to the programmer. In these cases, optimization can be imbedded in the algorithm by applying the ***load–balancing*** or ***task–scheduling*** methods.

For a specific problem, such as the transient stability, the program must be designed to maximize the use of processing power of the computational hardware on which it is going to be executed. The HSTS program is designed as a general multiprocessing program that can be implemented on various computer architectures. In order to use the processing power efficiently certain program adjustments must be made to suit adequately to a given hardware. In the following Sections, two implementations of the HSTS program are described. The parallel and distributed processing implementations are different, so that the nature of program adjustment will also be different.

4.2 HSTS Implementation on Real Time Digital Simulator (RTDS)

In the following section, a parallel implementation of the HSTS program on the RTDS hardware is described.

4.2.1 The Real Time Digital Simulator (RTDS) Hardware

The RTDS is a special purpose computer [6] designed primarily to perform power system electromagnetic transient simulations. Parallel processing techniques were applied in order to achieve the necessary computation speed required for continuous real–time operation. Real–time operation, is achieved when all of the calculations required within a single time–step can be completed within the chosen time–step.

The RTDS is divided into units of hardware referred to as *racks*, with each rack housing twenty printed circuit boards. Eighteen of those boards are identical and contain two digital signal processors and associated external hardware. These are the Tandem Processor Cards (TPC). One board within each rack is the Workstation Interface Card (WIC) and is used to communicate with a host computer workstation over an Ethernet based local area network. The final board contained within a rack is the Inter–rack Communication Card (IRC) and is used to communicate with other racks comprising the RTDS. Figure 4.3 illustrates the RTDS hardware architecture.

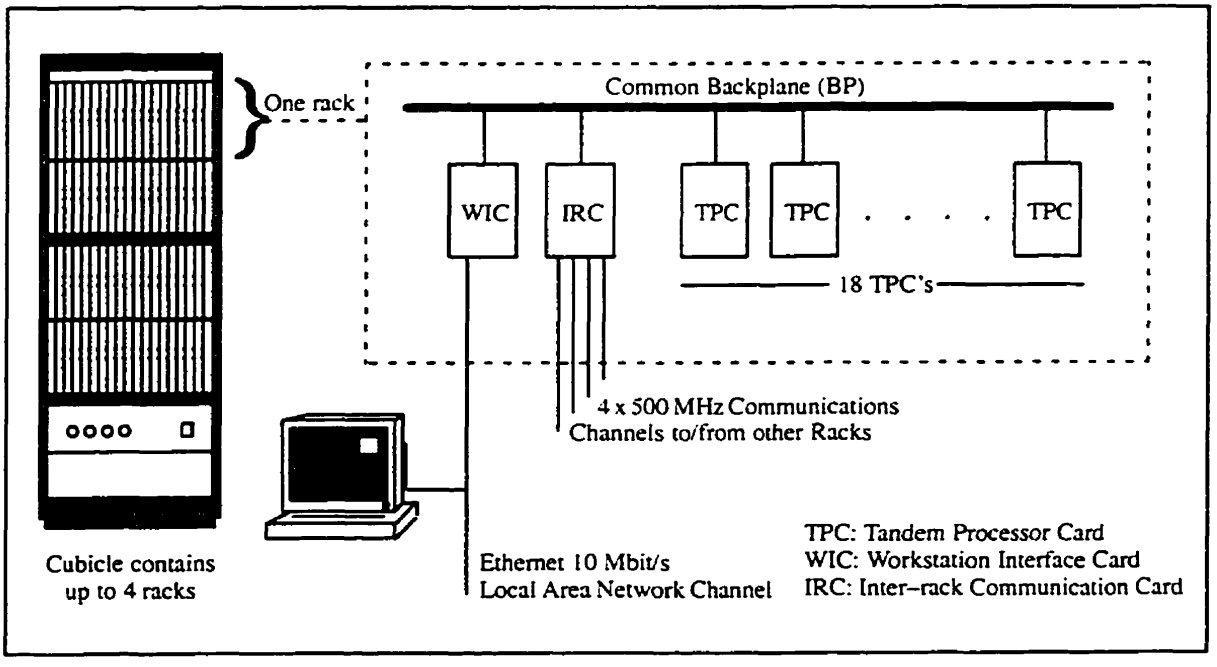


Figure 4.3. RTDS Hardware Architecture.

The TPC may be viewed as a pair of processors (two DSPs) with various banks of internal and external memory as shown in Figure 4.4 The RTDS hardware uses the NEC μPD77240 processor which has two internal banks of memory (RAM0 and RAM1) each containing 512 words.

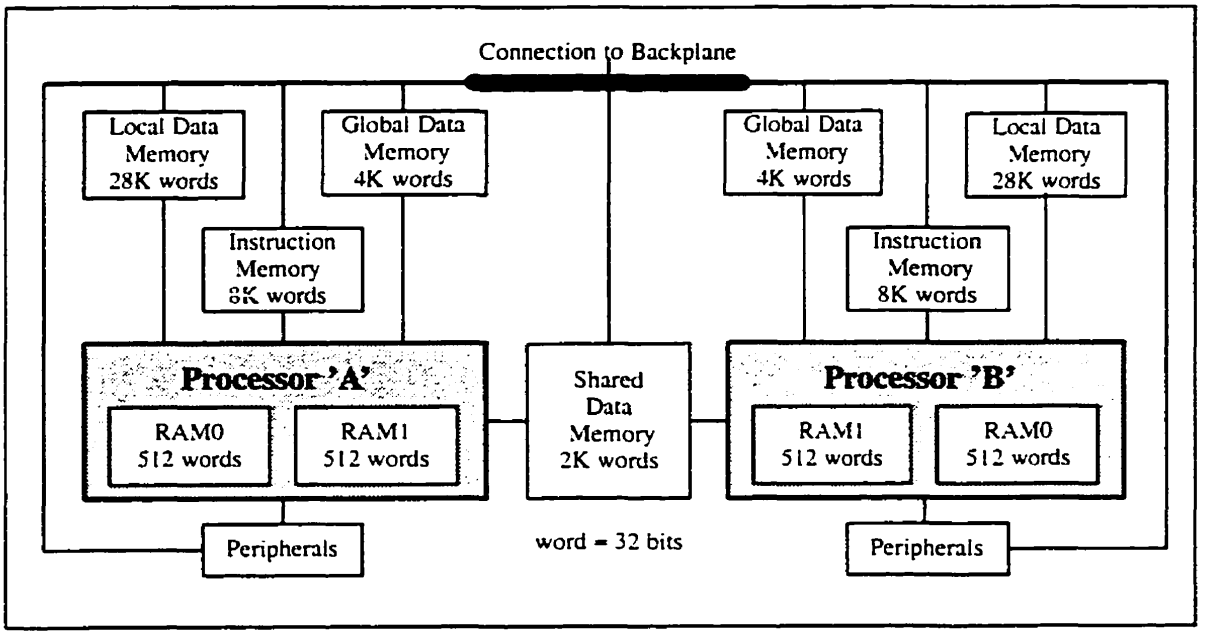


Figure 4.4. TPC Memory Banks.

Neither internal RAM bank may be directly accessed from the Backplane. Processor internal memory banks RAM0 and RAM1 are initially loaded from external memory and are subsequently addressed internally by the processor on each board.

Numerous cubicles, each containing up to four RTDS racks may be interconnected to form a large power system simulator. In theory, the number of racks comprising a RTDS is unlimited, although there is a hardware limitation in that a single rack may be directly interconnected to at most four other racks. By replacing one processor card with a second inter–rack communications card, however, the restriction may be relaxed so that one rack may be interconnected to at most eight others.

A significant effort was put into the development of various levels of software in order to ensure that the RTDS was user friendly. For transient stability simulation, the power sys-

tem model is entered from data files in a standard PSS/E like format. A Graphical User Interface similar to the PSCAD/EMTDC program can be developed for commercial version of the HSTS program.

A compiler interprets the power system model and generates data files. Combined data files and the executable code form download files for the processors comprising the RTDS.

A combined operator's console and data acquisition system is available via software running on the host computer workstation. While a simulation is running, the user is able to interact with the RTDS, and also data generated by the RTDS may be captured and plotted on the host workstation. Since the operator's console and data acquisition system are integrated, it is possible to capture data showing the response to a user initiated disturbance.

4.2.2 HSTS Implementation on RTDS

The biggest obstacle in obtaining real–time transient stability solution appears to be the repetitive solution of a large size network matrix equation $\mathbf{Y} \cdot \mathbf{V} = \mathbf{I}$. The HSTS algorithm was developed to deal with this problem in the multi–processor environment. System size reduction by Bus Tearing method and sparsity utilization by W–matrix were adapted to overcome the computationally–intensive solution problems. Partitioning scheme was also developed to balance the workload of the RTDS processors.

The W–matrix method applied on the RTDS may reach two types of limitations :

A. **Time limitation** : the total time required for the network solution and communication between processors should be completed in the real–time within every time–step of 5–10 msec.

B. **Memory limitation** : the problem size assigned to each DSP for parallel processing must fit within the size of memory blocks associated with the processors of the RTDS hardware.

The time in limitation A is the maximum of the execution times of the 36 processors within one RTDS rack required for a single–step computation. This time depends on :

– total problem size (number of system buses, and number dynamic models involved)

– density of admittance matrix and type of system models required for simulation run

– error margin and the maximum number of iterations allowed in the iteration loop

– size of a problem assigned to each DSP (number of buses and dynamic models in a partition)

Balancing processor workload is essential in this issue and equal–size partitioning normally will not suffice because of the uneven density of system admittance matrices and dynamic equations applied at various system buses. One method for balancing processor workload was proposed in section 2.5.3

The network solution algorithm based on the W–matrix Method and associated library of phasor domain models were originally programmed in "C" computer language as a stand alone program for use on a single processor workstation computers. Compiling "C" programs for parallel computers is done internally and requires no effort for pre–programming and code preparation. Although the NEC assembly code must be generated for the specific application on the existing hardware of RTDS, the reference software in "C" will be universal for any other possible application on commercially available parallel processing computers including future versions of the RTDS.

In order to run the High Speed Transient Stability program on the RTDS parallel computer, the "C" version is translated into a NEC code (assembly language) required by the μPD77240 digital signal processors (DSP) used in the hardware. This can be done either using the C–NEC compiler or by manual translation of the C–coded program to a NEC–assembly language program. Since the compiler produced very inefficient code, some of the most critical functions had to be translated manually to form an extended HSTS NEC–Library as shown in Figure 4.5.

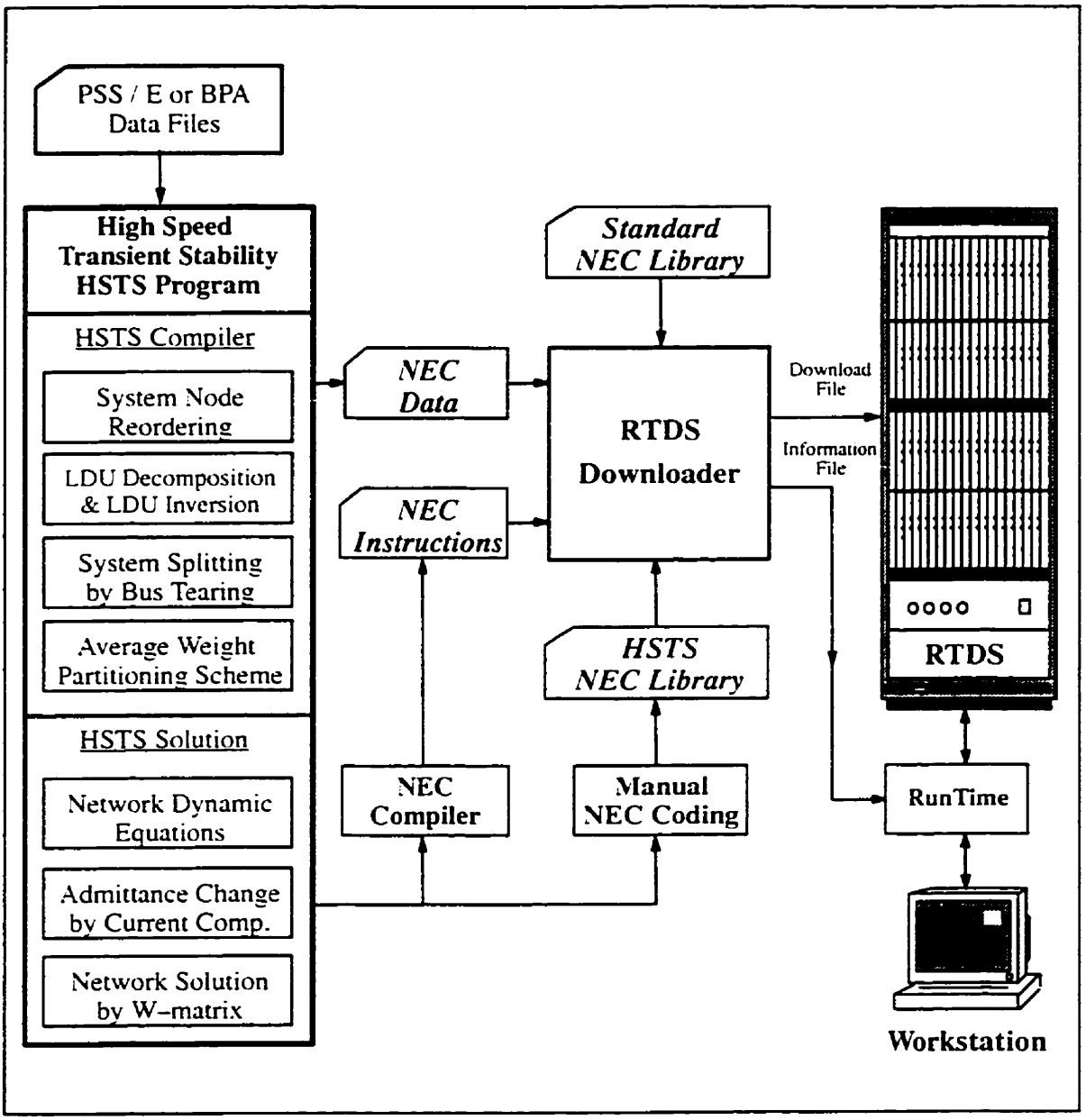


Figure 4.5. Implementation of the HSTS Program on RTDS.

It is also necessary to have a down–loading compiler program installed on a host computer for loading the NEC code and system data into the RTDS. The main assignment for the down–loader is to read and interpret all system specifications and to group appropriate data and instruction words into hexadecimal files for each individual digital signal processor involved in a simulation run on RTDS.

Memory Considerations :

For limitation B, the memory blocks of each individual DSP in the RTDS shown in Figure 3.4 are considered to be : 28K of External Data Memory (EXM), 1K of Internal Data Memory(RAM0 & RAM1), and 4K of Global Data Memory (BP) for backplane communications (within which 4x256 words is reserved for inter–rack communications).

The network solution by W–matrix method requires the following complex computations:

$\mathbf{I'} = \mathbf{D}^{-1} \cdot \mathbf{W} \cdot \mathbf{I}$ where $\mathbf{W} = \mathbf{L}^{-1}$

$\mathbf{V} = \mathbf{W}^{T} \cdot \mathbf{I'}$ where $\mathbf{W}^{T} = \mathbf{U}^{-1}$

To perform the two–step network solution computations both complex arrays **I** and **I'**, which are of the full system size, must be transferred to each DSP via Backplane Memory. The solution vector **V** is also complex and of the full system size but it doesn't have to be broadcast because dynamic equations and bus voltages are solved on the same processors.

System splitting by Bus Tearing method applied in the proposed solution method can reduce problem size to a smaller subsytem size. The length of arrays **I** and **I'** that need to be transferred are also reduced to subsystem size.

Since Global Data Memory size is 4K, only a maximum of 2K complex values can be broadcast and therefore the maximum number of system buses for processing on one RTDS rack can not exceed 2,000 buses. This allows an average 55–bus partition for each of DSP on the 36 processor rack.

The network solution requires storage of arrays and matrices in the Local Data Memory (EXM) and transfer data through Global Data Memory (BP) as shown in Figure 4.4. To evaluate memory requirements we consider, for example, the number of system buses N = 500, partition size M = 55, and the density of the W–matrix d = 10%. The amounts of memory space (not including the data for dynamic model) is calculated as follows :

Global Data Memory (Backplane BP) :

Transfer 0 – current vector **I** : 2xN = 1,000 ≤ 4K

Transfer 1 – working vector **I'** : 2xN= 1,000 ≤ 4K

Transfer 3 – voltage vector **V** : 2xN = 1,000 ≤ 4K

Local Data Memory (External EXM) :

current vector **I** :	2xN	= 1,000
working vector **I'** :	2xN	= 1,000
voltage vector **V** :	2xN	= 1,000
triangular **W** matrix :		
sparse admittances	2x(d*0.5*MxN)	= 2,750
row index array :	M	= 55
column index array :	d*0.5*MxN	= 1,375
triangular $\mathbf{W}^T$ matrix :		
sparse admittances	2x(d*0.5*MxN)	= 2,750
row index array :	M	= 55
column index array :	d*0.5*MxN	= 1,375
diagonal $\mathbf{D}^{-1}$ matrix :	2xM	= 110

The total memory requirement for storing a 55–bus partition on a single DSP is approximately **9K** which is less than the limit of **28K** for the Local Data Memory, and leaves about **19K** for other data such as fault, machines, loads, exciters and governors of all buses assigned to a single DSP.

Another memory limitation comes from the size of the Instruction Memory which for the RTDS processors is equal to 8K. The size of instruction data is given by the size of the hexadecimal NEC translation of the RTDS program and is the same for each processor. If the code exceeds 8K, the processor may have to be "specialized" to receive only the code

for specific type of computation like network solution, generator equations, non–linear load equation, and so on.

In order to perform transient stability solutions on the RTDS hardware by the W–matrix method for large systems, it is necessary to utilize many racks. As far as the memory is concerned, subsystems of 2,000 buses could be performed on one rack with an average 55–bus partition assigned to each DSP. However, due to limitations of the RTDS communication architecture, which limits the list of broadcasts to 4K, this number must be reduced to around 500 buses per rack.

To allow processing of large systems on the RTDS hardware the HSTS Algorithm described in Chapter 3 is considered. The seven–step procedure is reviewed here to address the RTDS specific problems associated with this implementation.

All data for RTDS processors is partitioned and pre–calculated in Step 1. Current injections to system buses are initialized and are updated by dynamic models in Step 2 and 3 for voltage calculations in the network solution algorithm.

According to this algorithm the computation of Tearing Bus Voltages $\mathbf{V}_t$ in Steps 3 & 4 is not parallelized and is solved locally by each processor. The calculations of subsystem current injections $\mathbf{I}_i$ in Step 5 and bus voltages $\mathbf{V}_i$ in Steps 6 & 7 are structured for parallel processing as shown below.

Computation of Tearing Bus Voltages (Steps 3 & 4 of the HSTS Algorithm)

Suppose that the Tearing Bus subsystem consists of N_t nodes and each subsystem N_i nodes respectively. The matrix $\mathbf{Y'}_{ti} = \mathbf{Y}^{T}_{ti} \cdot \mathbf{Y}_{ii}^{-1}$ required for computation of the current injections to the interface subsystem $\mathbf{I}_{ti}$ is pre–calculated according to the formula (2.55) and is stored in sparse form on each processor memory for local computations. The computation of current injections $\mathbf{I'}_t$ and voltages $\mathbf{V}_t$ repeated on each processor may cost less time than the communicating between processors required otherwise.

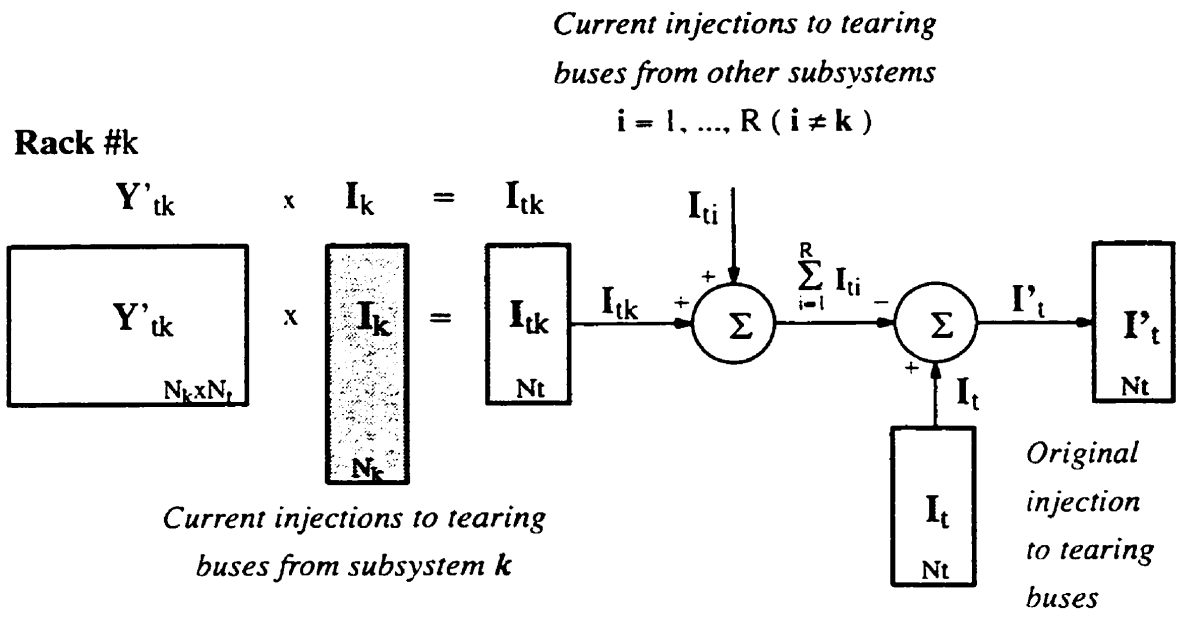


Figure 4.6. Computation of Currents Injections to Tearing Buses.

The admittance matrix $\mathbf{Y'}_{tt}$ for the interface subsystem and its inverse are pre–calculated in the algorithm Step 1. Although bus tearing voltage calculation problem is of a small size, the sparsity of the interface admittance matrix $\mathbf{Y'}_{tt}$ can still be utilized and the LDU decomposition and LDU inverse performed in the same manner as for any other subsystem admittance matrix. W matrix $\mathbf{W}_t = [\mathbf{L'}_{tt}]^{-1}$ is also formed and the subsystem network equation is solved by :

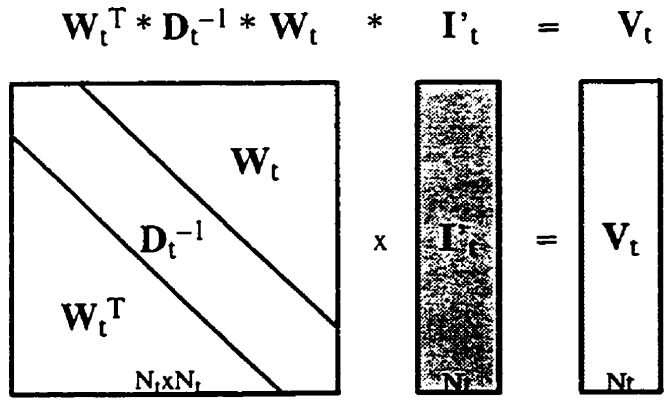


Figure 4.7. Computation of the Tearing Bus Voltages.

Parallel Computation of Subsystem Current Injections (Step 5 of the Algorithm)

Each rack computes its own injection current array $\mathbf{I'}_i$ required for subsystem bus voltage calculations on each processor. The partitioning scheme, however, assigns a certain number of busses to be solved on one processor so that the update of current injections can also be performed only for those buses and the results exchanged among all processors on the same RTDS rack.

First the current injections $\mathbf{I}_i$ to subsystem buses are updated by the solution of differential equations. The original subsystem current injections $\mathbf{I}_i$ are then modified by the amount of $\mathbf{I'}_{it}$ which accounts for the effect of all bus tearing voltages $\mathbf{V}_t$ on this subsystem. Partitioning is also applied to matrix $\mathbf{Y}^T_{ti}$ so that the matrix–vector multiplication $\mathbf{Y}^T_{ti} \cdot \mathbf{V}_t$ is performed only for a few rows on each processor as illustrated in Figure 4.8. Partitions of the border interface matrix $\mathbf{Y}^T_{ti}$ are stored as sparse and the vector multiplication here is of a type: sparse row times full column.

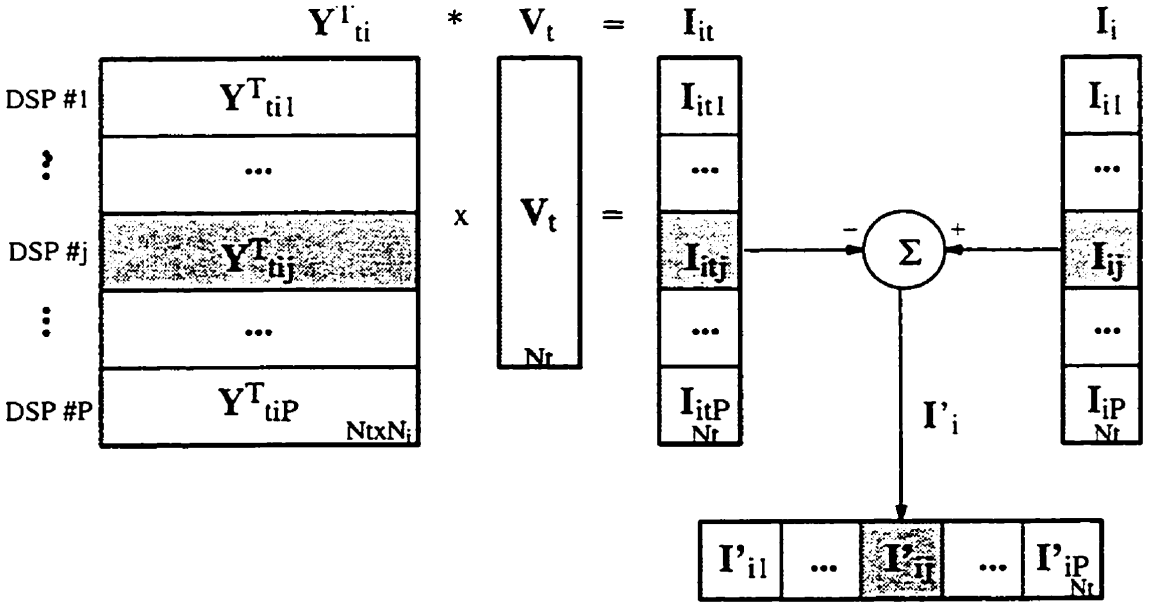


Figure 4.8. Parallel Computation of Subsystem Injection Currents.

Modified injection current array $\mathbf{I'}_i$ is put together and broadcasted via Backplane Memory BP to each processor on the same rack for parallel computation of subsystem bus voltages $\mathbf{V}_i$.

Parallel Computation of Subsystem Bus Voltages (Step 6 & 7 of the Algorithm)

For the reasons explained in section 2.4, only one W–matrix is used for each subsystem admittance matrix on one RTDS rack. Partitioning scheme divides W–matrices $\mathbf{W}_i$ and $\mathbf{W}^T_i$ into P layers so that a certain number of row–column multiplications involved in the network solution equation is computed on each processor as illustrated in Figure 4.9.

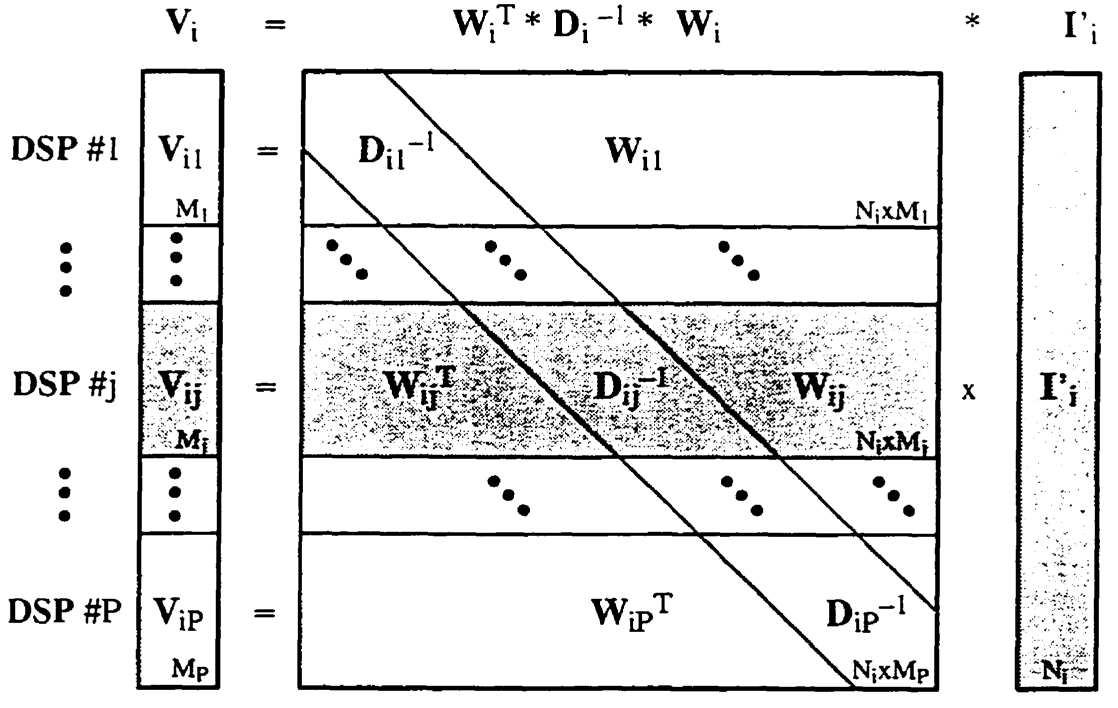


Figure 4.9. Parallel Solution of Subsystem Equations on One RTDS Rack.

For N_i nodes in subsystem i, M_j bus voltages are computed on processor j by solving one partition of subsystem matrix equation :

$$\mathbf{V}_{ij} = \mathbf{W}^T_{ij}\mathbf{D}_{ij}^{-1}\mathbf{W}_{ij} * \mathbf{I'}_i$$

This involves the following two–step W–matrix calculation :

$$\mathbf{I''}_{ij} = \mathbf{D}_{ij}^{-1} \cdot \mathbf{W}_{ij} \cdot \mathbf{I'}_i$$

$$\mathbf{V}_{ij} = \mathbf{W}_{ij}^T \cdot \mathbf{I}_i''$$

The unit–triangular sub matrices $\mathbf{W}_{ij}$ and $\mathbf{W}_{ij}^{T}$ are stored in the sparse form and the vectors $\mathbf{I'}_i$ and $\mathbf{I''}_i$ as full vector. The vector multiplication here is of a type : sparse row times full column.

The intermediate products of the two–step matrix multiplication $\mathbf{I''}_{ij}$ are exchanged among processors via the Global Data Memory (BP).

This part is probably the most computationally intensive part of the network solution algorithm and it is also repeated many times within the Time and Iteration Loops of the HSTS program. Coding should thus be done as efficiently as possible. For the NEC instruction data the vector multiplication for this part is coded manually and included in the HSTS NEC–Library of basic functions.

Communication : Inter–processor and Inter–rack Data Transfers

A complete solution for each time or iteration step is achieved in a few stages by making use of back–plane (BP) or inter–rack (IRC) communications. Four transfers are required to complete one network solution : three inter–processor transfers on the same rack and one inter–rack transfer.

First bus voltages from the last time–step solution (which are known locally by each processor) are used for solving dynamic equations of machines and loads. Current injections to system buses are updated and if admittance change took place then current compensation is also added to these current injections. Each DSP computes current injections $\mathbf{I'}_{ij}$ to those buses which are assigned to be processed as one partition group. These updated group of current injections are put together in BP to form a full subsystem current injection array which is passed back to each DSP in **Transfer 0** as shown in Figure 4.10.

By applying the subsystem–to–interface admittance matrix $\mathbf{Y'}_{ti}$, the modifications to interface current injections $\mathbf{I}_{ti}$ due to each subsystem are computed and put together in the top 256 locations of BP which is used for the IRC communication. Blocks of current modifications from each rack are received in inter–rack **Transfer 1** and they are put at BP locations

800, 900, A00, and B00 respectively for 4 other connected racks according to a communication map.

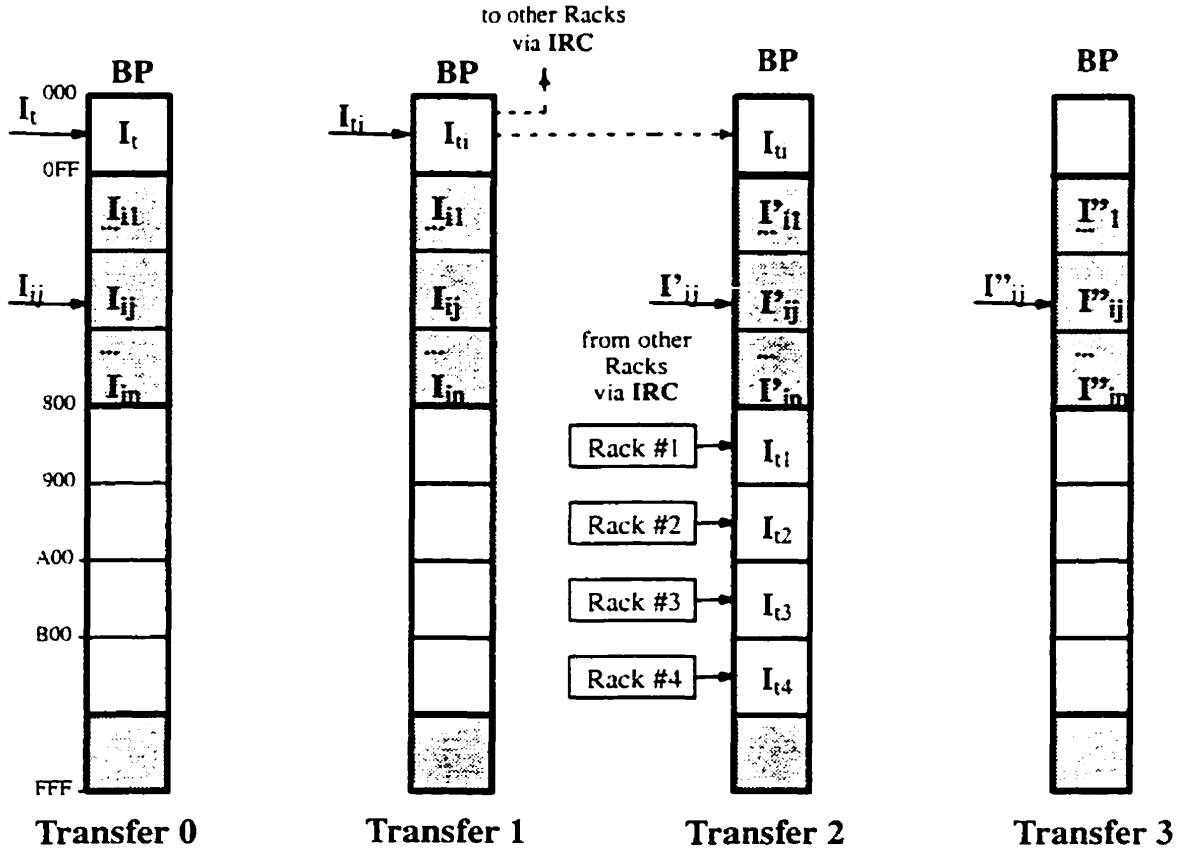


Figure 4.10. Transfers of Variables via Backplane Memory

In the next stage the modified interface current injections $\mathbf{I'}_t$ are computed and using the LDU inverse of the interface admittance matrix $\mathbf{Z}_{tt} = \mathbf{L}_t^{-1} \cdot \mathbf{D}_t^{-1} \cdot \mathbf{U}_t^{-1}$ tearing bus voltages $\mathbf{V}_t$ are computed on each DSP separately so that no transfer of data is required.

The tearing bus voltages $\mathbf{V}_t$ and the interface admittance matrix $\mathbf{Y}_{ti}$ are used to compute modifications to the subsystem current injections $\mathbf{I}_{it}$ using the admittance matrix $\mathbf{Y}_{ti}$. The modified subsystem current injection array $\mathbf{I'}_i$ is put together in BP from injections $\mathbf{I'}_{ij}$ updated on each DSP and exchanged among processors in **Transfer 2.**

A full modified current injection array $\mathbf{I'}_i$ is needed for the first part of subsystem bus voltage computation $\mathbf{I''}_i = \mathbf{D}_i^{-1} * \mathbf{W}_i * \mathbf{I'}_i$. One more time partition part of vector $\mathbf{I''}_{ij}$ is put together in BP in **Transfer 3** and the last part of the voltage computation $\mathbf{V}_i = \mathbf{U}_i^{-1} * \mathbf{I''}_i$ is now completed.

New bus voltages are used in the next step solution of system dynamic equations as described above. Since differential equations associated with a group of buses will be processed on the same DSP, the bus voltages can be stored locally only and no additional transfer between processors to exchange these voltages is required. The program returns to the top of the Iteration Loop, or if the iterations are finished, to the top of Time Loop to proceed with the next time-step computation.

Additional broadcasts may be required for the selected variables that are observed and need be uploaded to the host computer for plotting, monitoring or filing.

4.2.3 RTDS Test Results

The network solution part of the HSTS program including the W-matrix network solution method and the Current Compensation method for handling the admittance change was tested on one RTDS rack consisting of 36 processors. A 505-bus system was solved using various number of processors. The execution time for one computational cycle with one iteration was measured by observing the processor transfer request flags on the Dolch Logic Analyzer. The execution times as a function of number of processors used on a single RTDS rack are presented in Table 5 and also plotted in Figure 4.11.

The RTDS test results indicate that the most gains are achieved when using 5 to 20 processors. For more than 30 processors very small gains in execution time are observed. A full rack consisting 36 processors is therefore not an optimal number of processors to be used for parallel solution of a 500-bus system.

This results can be explained in terms of granularity or grain size for this 505–bus problem. Since one system admittance matrix was used (one rack RTDS applied), the lower triangular sparse factor matrix L after node re–ordering consists of long (505 elements), very dense (almost full) rows. These rows represents the smallest grain for processing on one processor causing that coarse grain size in this case. It would be probably better to split the test system into two or more subsystems, to reduce the grain size, and solve each subsystem on separate racks even with less than a full rack of processors.

Table 5 : HSTS Speed as a Function of RTDS Size

Number of Processors	Execution Time (one iteration)
5	41.8 msec
10	28.1 msec
15	23.6 msec
20	20.8 msec
25	19.5 msec
30	18.6 msec
36	17.7 msec

One alternative task assignment scheme for one RTDS rack may be to split the system such that one subsystem can be assigned to a TPC cards, and partition each subsystem for solving by 2–3 processor on one card (2 NEC or 3 SHARC processors). Such a scheme should improve the use of RTDS processing power but some program restructuring and transfer re–scheduling would have to be applied.

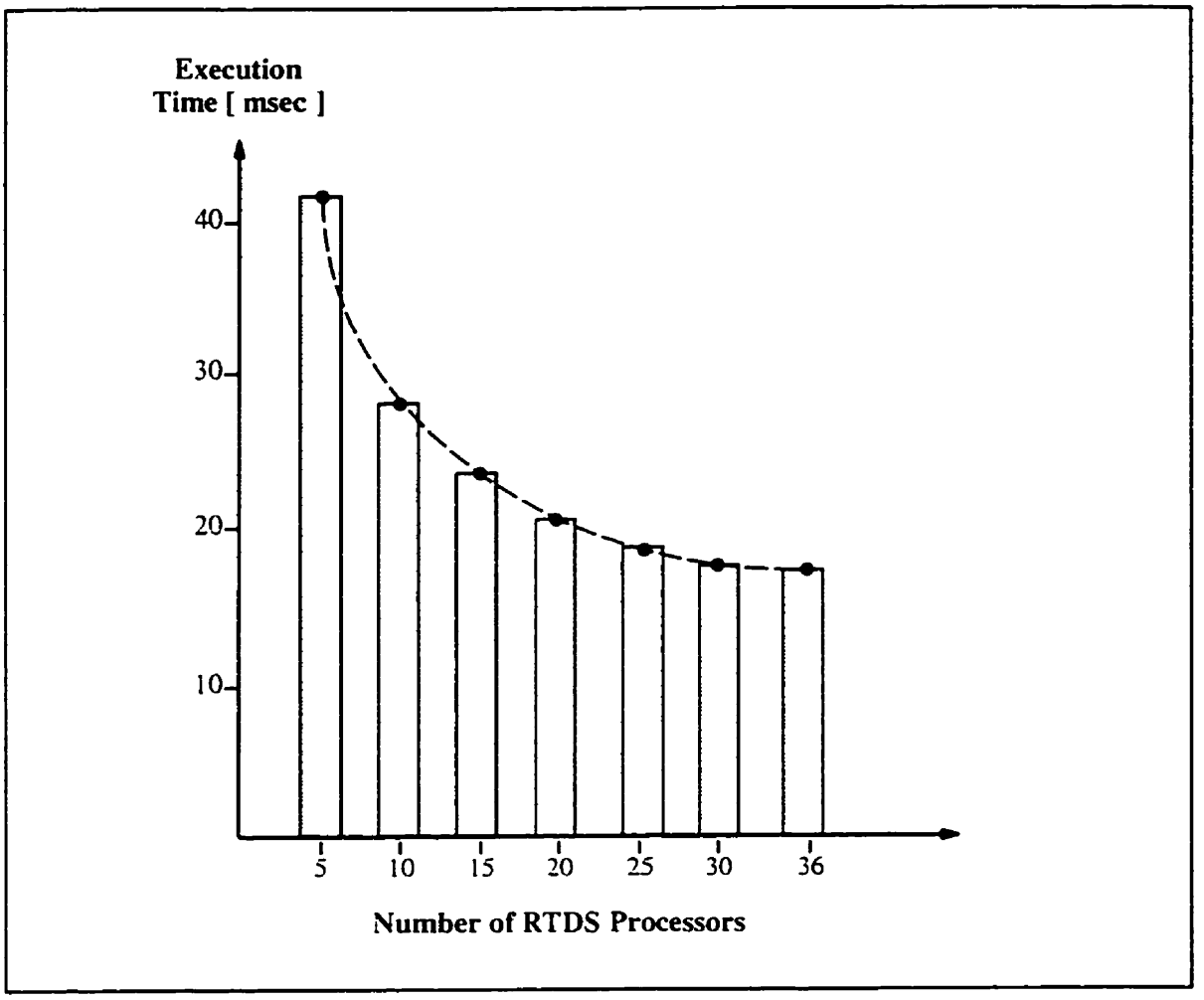


Figure 4.11. One–step Execution Time on Various RTDS Size

4.3 HSTS Implementation on Distributed Processing Systems (DPS)

Distributed Processing System is a type of multiple processor system which involves networks of computers that may not be close geographically. It is the most general form of parallel processing because it involves many different type of processors which may execute different programs, asynchronous communication channels with wide range of speeds, and architectures which are unique for each network.

Distributed Processing implies that processing will occur on more than one processor in order for a solution to be completed. Various topologies for Distributed Processing Systems can be designed. These topologies can be either static or dynamic. ***Static networks*** are

formed with point–to–point direct connections which will not change during program execution. ***Dynamic networks*** are implemented with switched channels, which are dynamically configured to match the communication demand in parallel processing programs.

In this project static networks of commodity computers connected through an Ethernet are considered. The configuration is a master–slaves scenario, with one computer freely selected as a central controller distributing and collecting data from the other nodes in the cluster. A mechanism that coordinates the scheduling of interdependent operations of a parallel application is required to run a program concurrently on separate processors. A general model of a Distributed Processing Systems (DPS) or Multicomputer is shown in Figure 4.12.

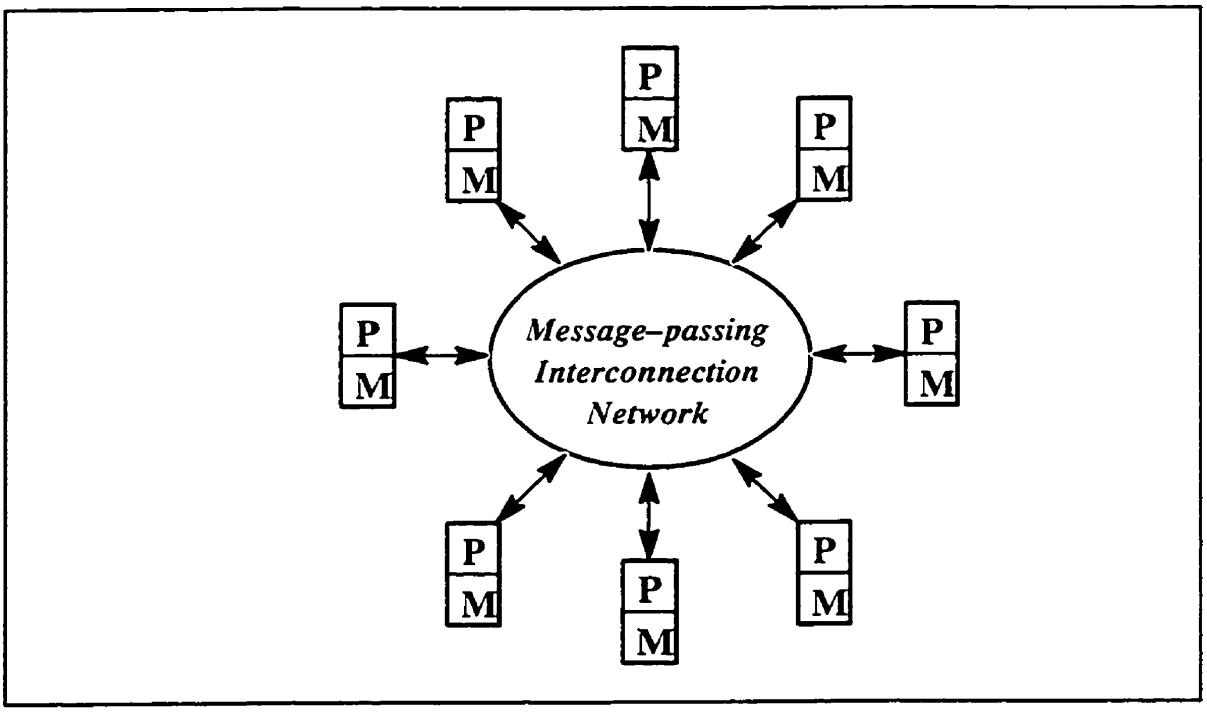


Figure 4.12. Model of a Message–Passing Multicomputer.

The DPS model consists of multiple computers interconnected by a message–passing network. Each computer consists of a processor (P), local memory (M), and disks or I/O peripherals. The message–passing interconnection network provides point–to–point connections among the computers which can have various configurations : mesh, ring, torus, or hypercube. All local memories **M** are private and are accessible only by local processors. The

Message Passing Interface Network, however, enables processors to communicate data for exchanging through the network.

Computational algorithms are traditionally executed sequentially on single processor computers. Unlike conventional sequential programs, the computations performed by Distributed Processing Systems do not yield a linear sequence of events. The inter–relationship between the events performed in distributed systems requires distributed synchronization. In the HSTS program, this is accomplished by synchronous alignment of the send() and blocking receive() instruction pairs between distributed and the central processors.

Message–passing programming is applied to develop programs for applications on Distributed Processing Systems. In parallel programming, there are many different languages and programing tools, each suitable for different classes of problem. Example systems are: Compositional C++ (CC++), FORTRAN M (FM), High Performance FORTRAN (HPF), and the Message Passing Interface (MPI). Implementation of the HSTS program on Distributed Precessing Systems is based on the **Message Passing Interface** (**MPI**) library of functions and macros that can be used in C, FORTRAN, and C++ programs [18]. The MPI was developed in 1993–1994 and is one of the first standards for programming parallel processors. MPI is a complex system which comprises at present 129 functions of numerous parameters and variants.

4.3.1 Message Passing Interface (MPI)

MPI provides functions essential for communication between processes. This communication is based on the concept of ***communicator*** which is a collection of processes that can send messages to each other. The actual message–passing in programs is carried out by 'Send' and 'Receive' functions which consists envelops (general data about receiver and sender) and the data itself.

Collective communication can be made between two processors (point–to–point) or among more then two processors. When a single process sends the same date to every process

the communication is called ***broadcast***. There are also available various communication modes such as : *standard*, *buffered*, *synchronous*, and *ready* which can be used for specific purposes.

One programming environment for MPI development is provided by the **MPICH** for Windows NT software. This implementation, called **MPICH/NT**, allows processes to communicate with each other either via shared memory or via the network depending on where the receiving process is located. For this research work a public domain and freely available MPICH/NT implementation developed at the Mississippi State University is used. An alternative UNIX implementation is the MPI/LAM developed at the Ohio Supercomputing Centre.

MPICH operates on both Intel architecture and DEC Alpha platforms and is supporting a range of multiprocessing system configurations. An **mpirun** program is provided for precess startup. Processes can be run on a default set of nodes and the process placement can be controlled by use of configuration files. The system devices are integrated into MPICH in an optimal way to bring high performance for the messaging system.

With MPICH/NT, a dedicated cluster of computers on an existing network can act as one parallel computer solving one compute–intensive problem. The Microsoft Visual Studio environment and Digital Visual FORTRAN are supported which offers extensive capabilities to support debugging for parallel programming.

In MPI programming, the processes involved in the execution of a parallel program are identified by a sequence of non–negative integers called ***ranks***. When the program is compiled and run with more than one processors the process is as follows :

1. A copy of the same executable program is placed on each processor,
2. Each processor receives its own data required to perform all assigned tasks,
3. Processes execute different instructions according to processor ranks.

The above is based on the Single Instruction Multiple Data (SIMD) paradigm in which the effect of different programs running on different processors (MIMD) is obtained by taking branches within a single program on the basis of process rank. A process can find out its own rank (MyRank) by calling the function :

```
MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

The path of execution for communication functions is selected on the basis of this rank.

A multicomputer is specified as a simple list of machine names in a file for which MPI applications must be synchronized so that all processes locate each other before user code is entered. A simple SIMD application can be specified on the ***mpirun*** command line while more complex configuration is described in a separate file, called an ***application scheme***.

Six basic MPI functions that are most frequently used for parallel programming are :

MPI_Init	– to initiate MPI computation
MPI_Finalize	– to terminate MPI computation
MPI_Comm_size	– to determine number of processors in MPI process group
MPI_Comm_rank	– to determine my processor identifier
MPI_Send	– to send a message
MPI_Recv	– to receive a message

All but the first two functions take a ***communicator*** as an argument. A communicator identifies the process group and context with respect to which the operation is to be performed. For the basic program the only communicator needed is MPI_COMM_WORLD. It is predefined in MPI and consists of all the processes involved in a computation.

The actual message–passing is carried on by the functions MPI_Send and MPI_Recv. In order for the ***message*** to be successfully communicated, the system must append some information to the data that the application program wishes to transmit. This additional information forms the ***envelope*** of the message which consists of the receiver rank, the sender rank, the tag, and the communicator.

4.3.2 HSTS Implementation on DPS

As emphasized in the previous sections, because the High Speed Transient Stability (HSTS) program was developed with parallel implementation in mind, the implementation on Distributed Processing Systems (DPS) or Multicomputer requires a minimal adaptation. A fully scalable multiprocessing version of the HSTS C–language program is applied in this implementation. The partitioning scheme, which was important for parallel processors on each RTDS rack, is included in the program but not used unless multiprocessor computers are available in the network. Communication, which was based on the Virtual Distributed Shared Memory data exchange, is now replaced with the MPI–based cross–network message passing routines.

Since effective parallel processing compilers do not yet exist, the HSTS program is generically structured to allow application on most common parallel or distributing processing systems using a single processor compiler. This program is modularized and can be automatically reconfigured to specific computer hardware architectures.

In the HSTS program the computation for solving the problem and the data operated on by this computation are decomposed into small tasks. For effective decomposition of a transient stability problem a power system is split into a desired number of subsystems one for each computer (or multicomputer) in the network.

All basic tasks are designed to perform the 7–step HSTS algorithm described in Chapter 3. A ***task assignment scheme***, which is a method of allocating problem tasks to processors, is included in the initialization part of the HSTS program. Proper communication required to coordinate task execution is also designed in the general HSTS program structure as shown in Figure 4.13.

The EXM[k] ; k = 1,2, ..., K in Figure 4.13 represent the partitioned data blocks which are sent to network computers at the beginning of a simulation run.

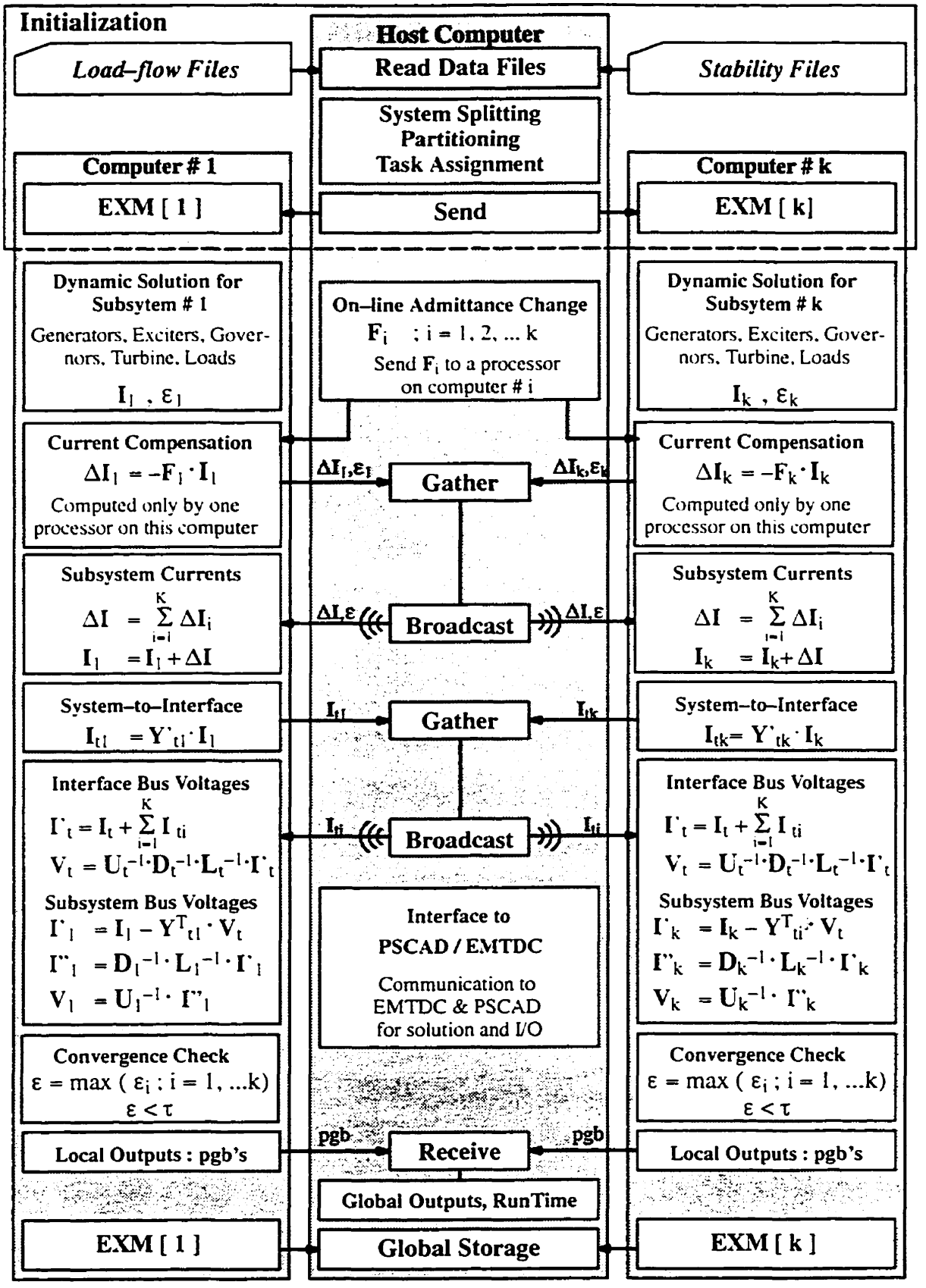


Figure 4.13. HSTS Program Structure for Multiprocessing Implementations

The **Manager / Worker** task assignment scheme is applied for implementing the HSTS program on Distributed Processing Systems as shown in Figure 4.14. In this scheme a central manager task is given responsibility for problem decomposition and task allocation. This manager task is assigned to the Root Processor rank 0. Each worker process, which is processor of rank greater than 0, repeatedly executes a problem task assigned by the manager. This task can be either to perform part of the HSTS solution or it can be a task totally different in nature such as the interface to the EMTDC/PSCAD program also shown in Figure 4.14.

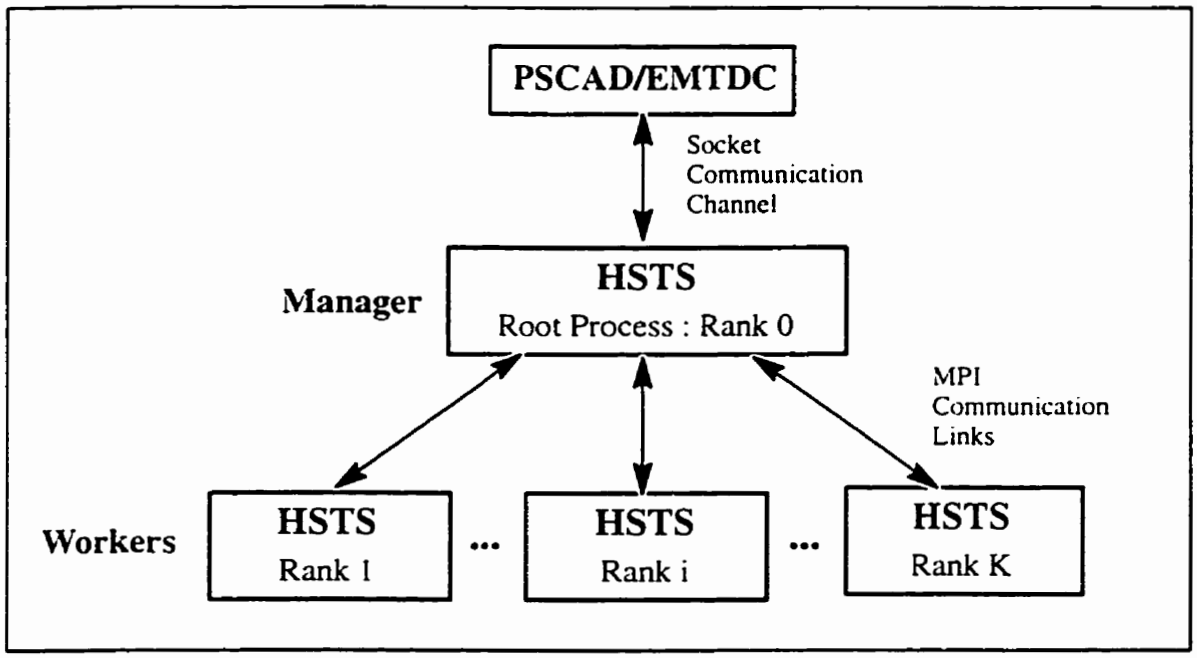


Figure 4.14. Manager/Worker Task–scheduling Scheme for HSTS Implementation on DP

The main task for a worker processor is to perform a solution of one partition of a subsystem problem. If K worker processors are located on K single–processor computers the simplest method for task assignment is to split the power system into K subsystem and assign a subsystem i to a processor which rank is equal to i (MyRank = i).

After finishing their tasks workers report to the manager by sending messages consisting of essential results required by other workers to continue their task computations. Manager receives all the messages and sends back collective messages to all workers. The manager also collects the final solution results, once every simulation time–step, and either stores

them in output files or displays selected quantities using a GUI interface such as the RunTime of the PSCAD program.

In the DPS implementation, communication must be done through the network and thus the communication latency will typically be larger than in the RTDS implementation unless fast network software and hardware are used. When one subsystem is solved by one computer, the requirement for data exchange between processors is similar to the inter–rack communication (IRC) in the RTDS implementations : one communication per iteration for the subsystem current injections to interface buses is required for the local computation of tearing bus voltages. Similar communication is also required during events such as switchings or system faults when subsystem current compensations must be exchanged. The size of these communication messages is small because it is proportional to a small number of interface buses or busses affected by admittance change in the first and second cases respectively.

Message–passing programming models are by default ***nondeterministic*** which means that the arrival of messages sent from processors A and B to a third process C, is not defined. It is the programer's responsibility to ensure that the messages reach their destinations in deterministic order when this is required. MPI provides a mechanism to create communication channels for point–to–point communication that allow construction of deterministic models. However, for the two Gather–Broadcast type of communication shown in Figure 4.13 the order of receiving messages is not critical as long as the message tags contain the information of where the received message came from. The manager can construct the collective messages for workers, based on the task assignment map which is also used at the beginning of a simulation run to distribute the partitioned data to all processors.

4.3.3 DPS Test Results

The HSTS multiprocessing solution method has been tested on a 100 MHz Ethernet LAN which connects UNIX, Windows '3.1, '95, and NT–based machines. Transient stability solution for a 505–bus test system was performed on selected homogeneous cluster of 7 Windows NT computers. First, the whole system was solved on one machine, and then the system was split into k = 2. .... 6 subsystems and solved on k+1 machines with one computer acting as a manager.

The main purpose of those tests was to demonstrate that the proposed multiprocessing solution method could produce significant speedups in the computational time. There is always certain amount of communication time associated with data exchanges required to perform such a solution. This amount of time depends on the type of network and the specialized hardware applied for fast communication. This aspects were not studied in this research work because they require expensive computer network upgrades.

In order to validate the method, the total computation and total communication times were observed separately during a 6–second simulation run on each machine in a cluster selected for the HSTS solution. The test results are presented in Table 6 and corresponding graphs are presented in Figures 4.15 and 4.16.

It can be observed that the computation time is reduced from almost 40 seconds for one machine to the range of simulation time for clusters of 6 and 7 machines. It can also be observed that the workload for workers (machines 1–6) is approximately balanced.

Table 6 : HSTS Execution Time on DPS (with MPI Communication)

HSTS Cases	Process Rank	Execution Time		
		Computation	Communication	Total*
1 subsystem / 1 machine	1	39.7 sec	0.1 sec	39.8 sec
2 subsystem / 3 machines	0	6.5 sec	94.4 sec	100.9 sec
	1	12.6 sec	88.3 sec	100.9 sec
	2	14.0 sec	86.9 sec	100.9 sec
3 subsystem / 4 machines	0	6.3 sec	142.1 sec	148.4 sec
	1	8.3 sec	140.1 sec	148.4 sec
	2	7.8 sec	140.6 sec	148.4 sec
	3	7.9 sec	140.5 sec	148.4 sec
4 subsystem / 5 machines	0	6.00 sec	387.6 sec	393.6 sec
	1	7.26 sec	386.5 sec	393.7 sec
	2	5.47 sec	388.3 sec	393.8 sec
	3	6.47 sec	387.2 sec	393.7 sec
	4	5.34 sec	389.9 sec	394.4 sec
5 subsystem / 6 machines	0	6.25 sec	602.5 sec	608.7 sec
	1	5.95 sec	602.9 sec	608.8 sec
	2	4.80 sec	604.0 sec	608.8 sec
	3	5.22 sec	604.1 sec	609.4 sec
	4	4.72 sec	604.1 sec	608.8 sec
	5	5.40 sec	603.4 sec	608.8 sec
6 subsystem / 7 machines	0	5.82 sec	529.5 sec	535.3 sec
	1	6.01 sec	529.4 sec	535.4 sec
	2	4.57 sec	530.8 sec	535.3 sec
	3	5.70 sec	529.6 sec	535.3 sec
	4	4.22 sec	531.2 sec	535.4 sec
	5	5.33 sec	529.9 sec	535.3 sec
	6	4.61 sec	530.9 sec	535.5 sec

Simulation cases : 505–bus test system, 6–second run, time–step $\Delta t = 5$ msec
* Excluding the initialization time which averages approximately 3.5 sec.

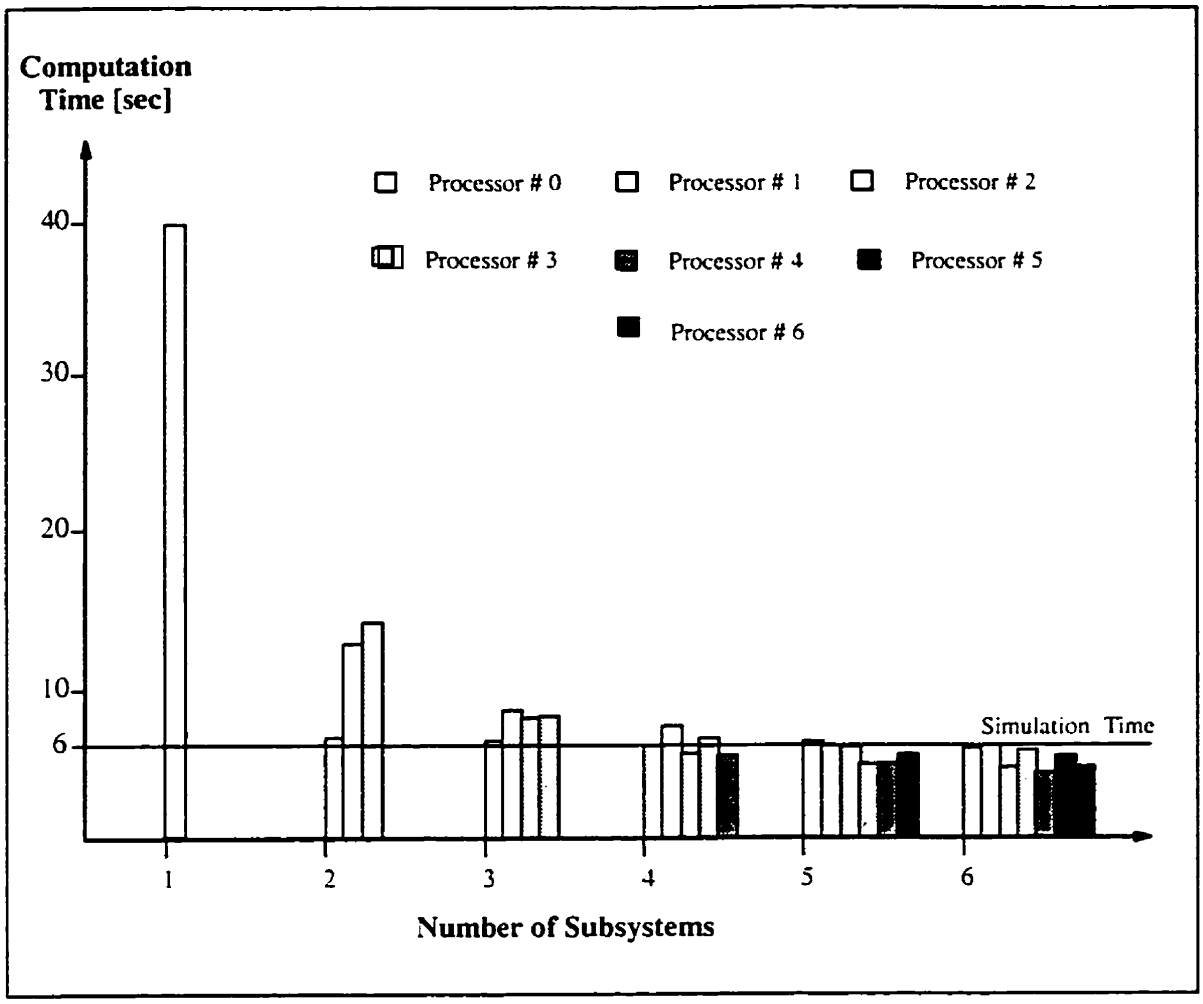


Figure 4.15. Processor's Total Computation Time on Various DPS Cluster Sizes.

In Figure 4.15, for cases with more than one processor, the first column representing the computation time of the root processor rank 0 is different than columns of the other processors. This is because in those cases manager processor does not participate directly in the solution process and thus its load is different and normally not balanced with the worker processors.

It can also be observed that, as the number of subsystems grows, problem granularity makes it more difficult to balance the processor load.

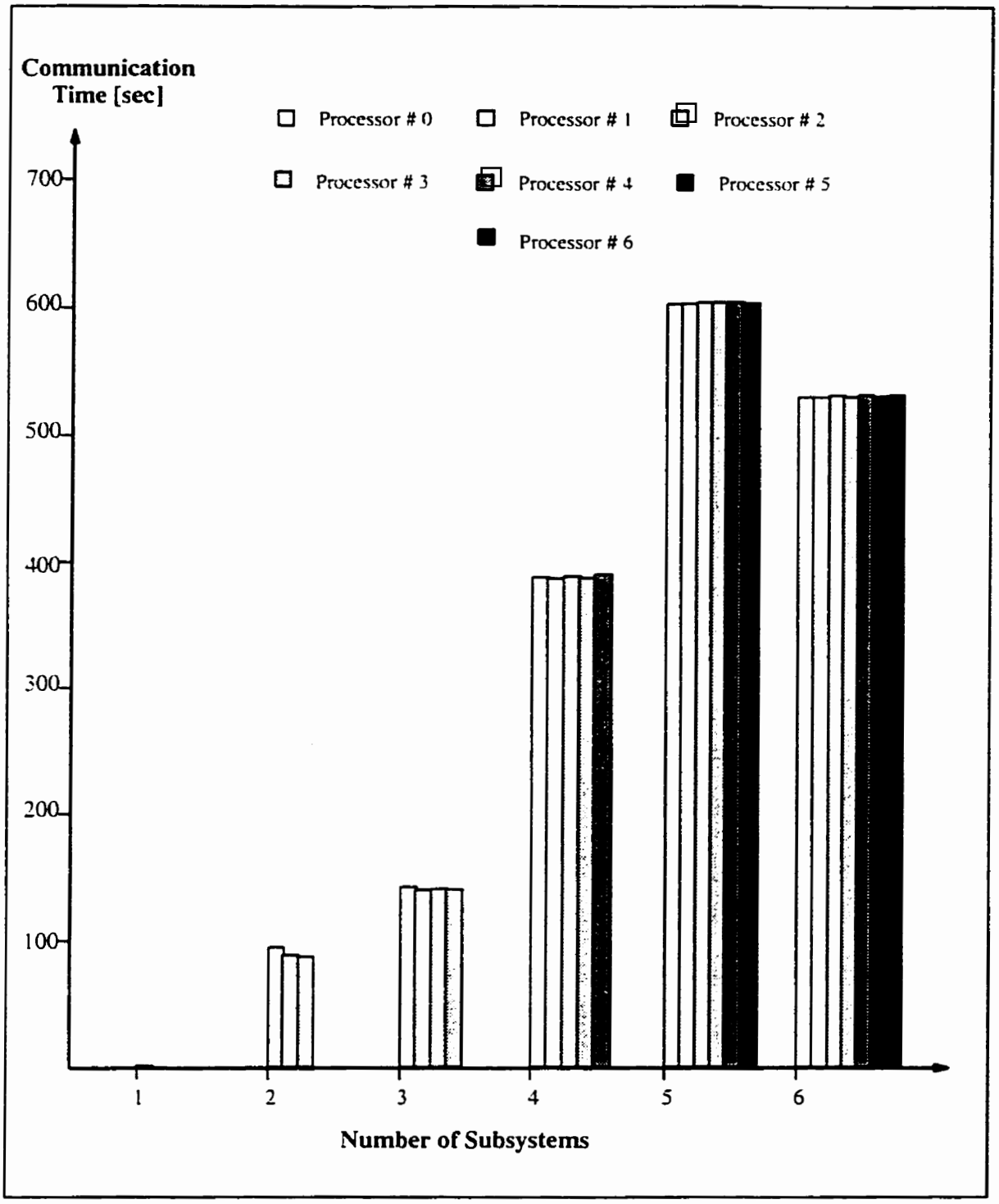


Figure 4.16. Processor's Total Communication Time on Various DPS Cluster Sizes.

Chapter 5

Conclusions and Recommendations

The main objective of this research work was to achieve a high speed stability solution which could be applied on parallel or distributed processing hardware platforms such as the RTDS and Distributed Processing System of Local Area Networks.

Various methods were considered and tried out on different computer hardware. Two iterative network solution algorithms based on the Gauss–Seidel and the Bergeron methods were developed and examined for convergence and computational speed. Although both solution methods readily fit into the parallel architecture, they generally suffer from computational inefficiency caused by the inherent slowness of convergence of the iterative network solution process for large system sizes. Therefore, a direct solution approach has been chosen in this research work for developing an algorithm for high speed stability solution. The achievements are summarized and conclusions and recommendations are presented in this Chapter.

5.1 Major Contributions

The following is a summary of the major contributions which have been accomplished in this research work :

A. Multiprocessing algorithm for high speed transient stability solution

In order to achieve high speed transient stability solution for large systems, a new multiprocessing algorithm has been developed in this thesis. This algorithm combines several techniques useful for parallel processing applications which include the following :

- LDU–decomposition and LDU–inverse for processing sparse matrices
- W–matrix method for solving network equations

- Re–ordering scheme to minimize number of fill–ins in the W–matrices
- Bus Tearing method for splitting large network into smaller subsystems
- Current Compensation method for handling the changes of system admittances
- Partitioning scheme for solving one subsystem on many processors operating in parallel.

A very important feature of this new algorithm is the **scalability** achieved in two levels of parallelization. Firstly, the large stability problem is decomposed into smaller subsystem problems using the System Splitting by Bus Tearing method. Each subsystem can be solved relatively independently on one multicomputer with minimal communication requirements because only a few connecting nodes are needed to tie the subsystem solutions together.

The second level of parallelization is achieved by applying the load–balancing partitioning scheme to solve subsystem problems by processors of a multicomputer. Since this parallel processing requires more intensive data exchange between processors, the communication should be done via shared memory such as found on a RTDS rack.

This two–level parallelization scheme allows a very flexible method of adjusting the solution method to various computing network architectures including parallel and distributed processing network configurations.

B. High Speed Transient Stability (HSTS) multiprocessing program

A stand alone version of the High Speed Transient Stability (HSTS) program has been written in 'C' computer language for applications on single processor computers (UNIX and PC) and for establishing a "template source code" for implementations on parallel and distributed processing hardware. Power system models developed in other projects at the Manitoba HVDC Research Centre are included in the dynamic part of the HSTS Program and parallelized in order to perform a complete solution of the transient stability problem. Those models include Classical and Detailed Synchronous Machines, Exciters, Power System Stabilizers, Governors, Non–linear Loads, Multi–terminal DC Links and various Faults.

Proper inter–process communication routines are developed for the RTDS and Distributed Processing System implementations. This communication must provide both the cross–network and inter–processor data exchange which is accomplished by applying the Message passing Interface (MPI) communication software.

C. Parallel processing implementation of the HSTS program

A basic version of the HSTS program (network solution and system admittance change) has been implemented on the RTDS for evaluation of the proposed network solution method in a parallel processing environment. The load–balancing partitioning scheme is the main mechanism that allows the parallel solution in this implementation.

The execution time for solving the 505–bus test system measured for various number of processors applied in a single RTDS rack, has shown that significant gains can be achieved when up to 20 processors are applied. Granularity of the problem causes the gains to saturate and the execution time to level off around 17 msec for one complete network solution computational cycle.

Alternative methods for the HSTS program implementations on future versions of the RTDS hardware (racks of cards consisting 3 SHARC processors) considers the assignment of one subsystem to 3–6 processors so that more than one subsystem can be solved on one RTDS rack. This should better utilize the RTDS processing power for solving the stability problem.

D. Distributed processing implementation of the HSTS program

A distributed processing implementation of the HSTS program on a Local Area Network has been completed to examine program flexibility to adjust to various network configuration. The proposed method considers various number of computers and various number of processors on each computer that can be involved in a collective solution process.

It was demonstrated that solving the problem on a network of distributed processors can effectively reduce computational time but at the expense of increasing communication time.

It is believed that the emerging fast network technologies can soon eliminate the high communication latency and allow high–speed or even real–time transient stability solution of large scale systems on a scalable cluster of commodity computers.

5.2 General Conclusions

The proposed multiprocessing algorithm implemented in the HSTS program deals with the fundamental requirement for parallel software very effectively. The ***concurrency*** requirement is accomplished by converting the general matrix solution into parallel, independent tasks that can be executed concurrently on many processors. For the ***scalability*** requirement, the two–level partitioning and task assignment schemes form a powerful mechanism for effective problem decomposition appropriate for various computer architectures and available network resources. The requirement of ***locality*** is one of the prime concerns in the proposed method. The data dependencies and communication requirements are minimized by assigning to one processor a group of system buses along with dynamic devices connected to those buses. ***Modularity*** of the HSTS program is maintained at various program levels. At the bottom level, there are primitive functions for basic operations such as complex vector and matrix multiplications. At higher levels are the basic program modules which are used to construct program functions performing dynamic and network solutions, admittance change, current compensation and other program blocks.

The main goal in developing a parallel algorithm is to maximize its parallelism and to minimize the data dependencies between the parallel parts. The trade–off here is that more parallelism involves more processors, which may require more data sharing and thus more communication time. The algorithm applied in the HSTS program handles those problems very effectively and, although more extensive tests are still needed, the basic tests performed for this research work confirm that. The number of inter–processor communications per computational cycle, as well as the message lengths, are minimized due to a special combination of techniques applied in the solution method.

5.3 Future Recommendations for Speed Improvements

After a more detailed analysis of the HSTS execution process, some additional savings in computation and communication times can be achieved by further optimization of the program parallelization. Solving of the interface subsystem, for example, can be assigned to the host computer (manager) and the dynamic solution for the interface buses can be performed in parallel to similar computations for subsystem busses performed by other computers (workers). The interface bus voltages can be computed by the manager and returned to the remote processors for local updates of the current injections and voltage computations. In this case, the overhead time due to the repetitive computations of the interface subsystem on each processor, can be reduced.

The main problem for distributed processing remains the cross–network communication. For solving the high communication latency problem, an upgrade of hardware is necessary. High Performance Computing and Fast Network technologies must be utilized to achieve a better performance on a Local Area Network. Recently, many research groups report on achieving a high computational speed using the fast network techniques.

One software tool designed to synthesize groups of computers into a high–performance environment has been developed by Professor Andrew Chien at the University of Illinois and is called the High Performance Virtual Machine (HPVM). Using a High Performance Virtual Machine (HPVM) software, a group of off–the–shelf computers can be synthesized to deliver a peak performance of between 100 and 200 billion floating–point operations per second. A high–performance message communication layer, the Illinois Fast Messages (FM), can send messages between processors over high–speed networks delivering bandwidth of just under 80 megabytes per second and a latency under 11 microseconds using the Myrinet interconnect. A variety of of Application Programming Interfaces (API's) has been built on the top of FM and includes the MPI used in the HSTS program.

The HVPM and Myrinet interconnect are viewed by the author as one of the most promising platform for the HSTS implementation which can eliminate the large communication overhead observed in the regular LAN. A much more expensive alternative would be to apply HSTS on High Performance Computers such as the 2600 MultiComputer Series delivered by the Computer Signal Processing & Imaging (CSPI). This system consists of 6 cards, each incorporating four 200 MHz PowerPC processing elements interconnected by an 8–port crossbar switch. CSPI uses Myrinet high speed (gigabytes per second) packet communication and packed routing technology to implement a switched network solution. It also supports the MPI for multiprocessor control and inter–processor communication.

References

[1] H.W. Dommel, "Digital Computer Solution of Electromagnetic Transients in Single and Multiphase Networks", IEEE Transactions on Power Apparatus and Systems, vol. PAS–88, No. 4, April 1969.

[2] J.S. Chai, A. Bose, "Bottlenecks in Parallel Algorithms for Power System Stability Analysis", IEEE Transactions on Power Systems, Vol. 8, No. 1, February 1993.

[3] H. Taoka, I. Iyoda, H. Noguchi, N. Sato, T. Nakazawa, "Real–Time Digital Simulator for Power System Analysis on a Hypercube Computer." Transactions on Power Systems, Vol. 7, No. 1, February 1992.

[4] Y. Sekine, K. Takahashi, T. Sakaguch, "Real–Time Simulation of Power System Dynamics." Transactions on Power Systems, Vol. 7, No. 1, February 1992.

[5] K. Werlen, H. Glavitsch, R. Bacher, "From Bergeron's Method to Node–oriented Loadflow Solution." , Proceedings of the 11–th Power Systems Computation Conference, Volume II, Avignon France, August 1993.

[6] R.P. Wierckx, W.J. Giesbrecht, R. Kuffel, X. Wang, G.B. Mazur, M.A. Weeks, A.M. Gole, "Validation of a Fully Digital Real–time Electromagnetic Transients Simulator for HVDC Systems & Control Studies", Proceedings of Athens Power Tech Conference, Volume II, Athens Greece, September 1993.

[7] P. M. Anderson, A. A. Fouad, "Power System Control and Stability" , The Iowa State University Press, 1977, pp. 38–48

[8] J. Q. Wu, A. Bose, J. A. Huang, A. Valette, F. Lafrance, "Parallel Implementation of Power System Transient Stability Analysis", IEEE/PES 1994 Summer Meeting, San Francisko, CA.

[9] D. J. Tylavsky A. Bose, "Parallel Processing in Power System Computation", An IEEE Committee Report by Task Force of the Computer and Analytical Methods Subcommit-

tee of the Power System Engineering Committee, Transactions on Power Systems, Vol.7, No. 2, May 1992.

[10] Ian T. Foter, "Designing and Building Parallel Programs', Addison–Wesley Publishing Company, 1995.

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical Recipes in C", Cambridge University Press, 1992.

[12] R. T. Byerly, E. W. Kimbark, "Stability of Large Electric Power Systems", IEEE Press, 1974.

[13] P. Kundur, "Power System Stability and Control", McGraw–Hill, Inc. 1994

[14] O. Alsac, B. Stott, W. F. Tinney, "Sparsity–oriented Compensation Methods for Modified Network Solutions", IEEE PES 1982 Summer Meeting, San Francisco, California, 1982.

[15] M. La Scala, G. Sblandorio, R. Sbrizzai, " Parallel–in–time Implementation of Transient Stability Simulation on a Transputer Network", IEEE/PES 1993 Summer Meeting, Vancouver, Canada, 1993.

[16] W. T. Kwasnicki, D. A. Woodford, X. Wang, A. M. Gole, "Bergeron Based Network Solution Algorithm for High Speed Dynamic Stability", 12–th Power System Computation Conference, Dresden, Germany , 1996.

[17] "Feasibility Assesment of Transient Stability Solution in Real Time", Canadian Electrical Association Report 347 T 868, 1994.

[18] P. S. Pachaco, "A User's Guide to MPI", University of San Francisko, 1995

[19] "LAM Overview", Ohio Supercomputer Center, http: www.osc.edu/Lam/lam/overviev.html

[20] K. Hwang, "Advanced Computer Architecture : Parallelism, Scalability, Programmability", McGraw–Hill, Inc., 1993.

[21]. CRIEPI Report – "Integrated analysis Software for Bulk Power System Stability", Central Research Institute of Electric Power Industry, 1–6–1, Otemachi, Chiyoda–ku, Tokyo, Japan.

[22]. B.M Zhang and Shousun Chan, "Advanced Power Network Analysis." Print–Hall of Tsinghua University, September 1993.

[23]. G. Kron, "Diakoptics – A Piecewise Solution of Large Scale Systems.", Elect. J., Vol. 158, 1957 to Vol. 162, 1959.

[24]. H. H. Happ, "Piecewise Methods and Applications to Power System.", John Wiley & Sons, Inc., 1980

[25]. F. F. Wu, "Solution of Large Scale Networks by Tearing", IEEE Trans. on Circuits and Systems, Vol. CAS–23, 1976, pp. 706–713.

[26]. A. Sangiovani – Vincentlli, et al., "A new Tearing Approach – Node Tearing Nodal Analysis", Proc. of IEEE Symp. Circuit and Systems, 1976, pp. 143–147

[27]. B. M. Zhang, et al., "Unified Piecewise Solution of Power System Networks Combining Both Branch Cutting and Node Tearing", Int. J. of Electrical Power and Energy Systems, Vol. 11, No. 4, 1989, pp. 283–288

[28] J. Sherman and W. J. Morrison, "Adjustment of an Inverse Matrix Corresponding to Changes in the Elements of a Given Column or a Given Row of the Original Matrix", (abstract) Ann. Math. Statistics, Vol. 20, 1949, p. 621

[29] J. Sherman and W. J. Morrison, "Adjustment of an Inverse Matrix Corresponding to Changes in One Element of a Given Matrix", (abstract) Ann. Math. Statistics, Vol. 21, 1950, pp. 124–127.

[30] J. Q. Wu, A. Bose, " A New Successsive Relaxation Scheme for the W–matrix Solution Method on Shared Memory Parallel Computers", IEEE Power Industry Computer Application Conference, Salt Lake City, Utah, 1995.

[31] M. K. Enns, W. F. Tinney, F. L. Alvarado, "Sparse Matrix Inverse Factors", IEEE/PES 1988 Summer Meeting, Portland, Oregon, USA, 1988.

[32] W. F. Tinney, J. W. Walker, "Direct Solution of Sparse Network Equations by Optimally Ordered Trangular Factorization", Power Industry Computer Applications Conference, 1967.

[33] M. La Scala, A. Bose, D.J. Tylavsky and J.S. Chai, "A Higly Parallel Method for Transient Stability Analysis", IEEE Power Industry Computer Application Conference, Seatle, 1989.

[34] T. Oyama, "Heterogeneous Parallel Processing for Power System Analysis", First International Conference on Digital Power System Simulators – ICDS'95, College Station, USA, 1995.

[35] P. K. Sinha, "Distributed Operating Systems. Concepts and Design", IEEE Press, New York, 1997.

[36] M. Bruggencate, S. Chalesani, "Parrallel Implementations of the Power System Transient Stability Problem on Clusters of Workststions", Super Computing '95 Conference, 1995.

[37] M.L. Crow, M. Ilic, "The Parallel Implementation of the Waveform Relaxation Method for Transient Stability Solutions", IEEE/PES Winter Meeting, Atlanta, Gorgia, 1990

[38] A. Padilha, A. Morelato, "A W–Matrix Methodology for Solving Network Equations on Multiprocessor Computers", IEEE/PES Summer Meeting, San Diego, California, 1991.

[39] H.W. Dommel, N. Sato, "Fast Transient Stability Solutions", IEEE Winter Meeting, New York, N.Y., 1972.

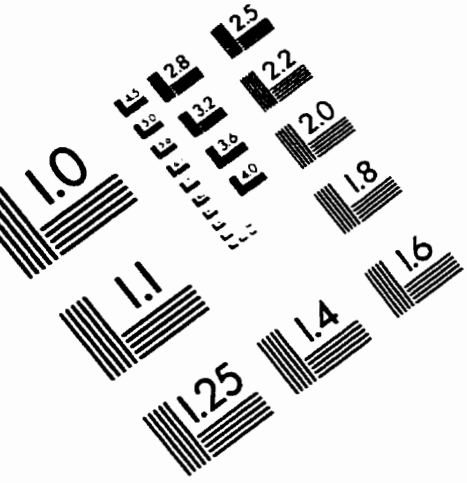
[40] G. Aloisio, M.A. Bochicchio, M. La Scala, R. Sbrizzai, "A Distributed Computing Approach for Real–Time Transient Stability Analysis", IEEE/PES Summer Meeting, Denver, Colorado, 1996.

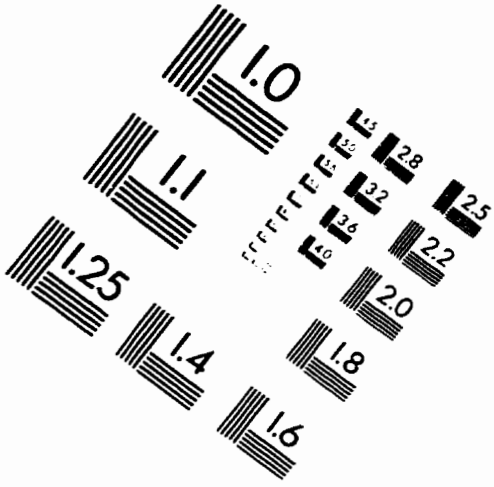
[41] J. Chai, N. Zhu, A. Bose, D. Tylavsky, "Parallel Newton Type Method for Power System Stability Analysis Using Local and Shared Memory Multiprocessors", IEEE Trans. on Power Systems, Vol. 6, No. 4, 1991

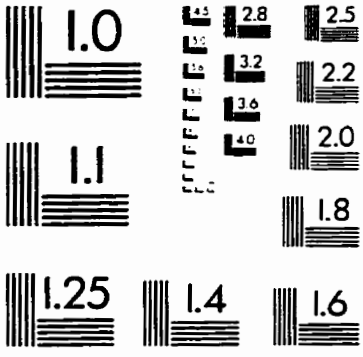
[42] K. Lau, D. Tylavsky, A. Bose, "Coarse Grain Scheduling in Parallel Triangular Factorization and Solution of Power System Metrices", IEEE Trans. on Power Systems, Vol. 6, No. 2, 1991

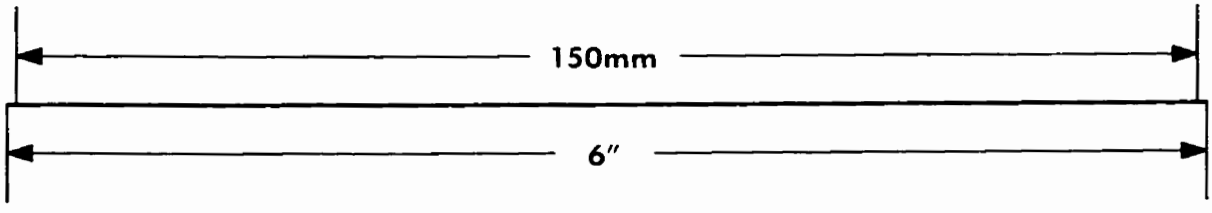
[43] M. Enns, W. Tinnay, F. Alvarado, "Spara metrix Inversion factors", IEEE Trans. on Power Systems, Vol. 5, No. 2, 1990

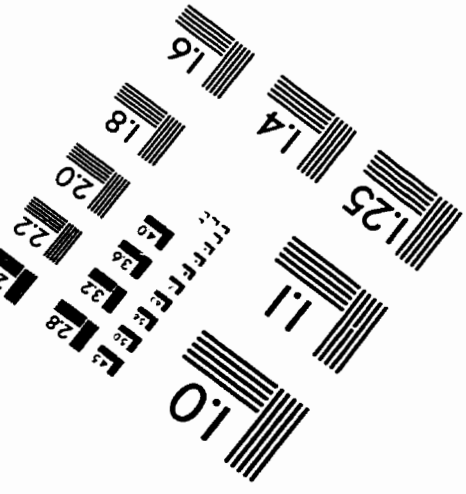
IMAGE EVALUATION
TEST TARGET (QA-3)

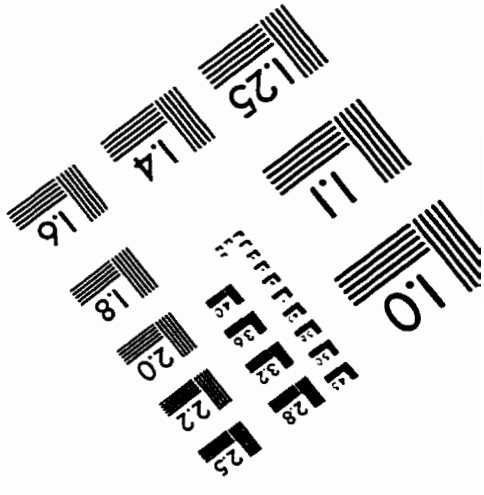












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989