1976

A special purpose multiprocessor for the simulation of dynamic systems.

Hatcher, William Lloyd

Northwestern University

https://hdl.handle.net/10945/17950

A SPECIAL PURPOSE MULTIPROCESSOR FOR THE SIMULATION OF DYNAMIC SYSTEMS

William Lloyd Hatcher

NORTHWESTERN UNIVERSITY

A SPECIAL PURPOSE MULTIPROCESSOR FOR THE SIMULATION OF DYNAMIC SYSTEMS

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical Engineering

by

WILLIAM LLOYD HATCHER III

Evanston, Illinois

June 1977 [6]

Thesis
H3176

To Jean,

who is more intelligent by far

than I.

TABLE OF CONTENTS

TABLE OF FIGURES AND TABLES

TABLE OF GRAPHS

ABSTRACT

Dynamic systems have grown to enormous size and complexity. The ability to simulate the systems has greatly helped in the design and operation of these systems. Inspite of advancements in model simplification techniques and refinements in solution procedures, many of todays systems are prohibitively expensive to simulate with the required accuracy. It was felt that the equations describing the system could be solved in parallel with savings in computer time. To this end the solution process was studied to find large scale (macro) parallelism. Macroparallelism was found both in the equations of the model and in the solution processes. Once this parallelism was found, requirements for and gains achievable by multiprocessors which use this inherent parallelism were developed.

An efficient set of model equations is obtained by converting the differential equations to algebraic equations by the implicit multi-step integration fomulas. The entire set of equations is then solved by an iterative algorithm. The Chaotic Relaxation and Newton-SOR Algorithms exhibit a high degree of parallelism which can be increased by ordering the equations to the near block diagonal form. This ordering is possible because of the sparsity present in models of large systems. The Gauss-Seidel and true Newton Algorithms are not obviously executable in parallel, but by ordering the equations to a bordered block diagonal form parallelism is exposed.

The requirements for sharing data are dictated by the form of the equations, while the control requirements depend on the algorithms. The near block diagonal form algorithms exchange solution data through shared memory, and it is the contention over this shared memory which limits the

gains achievable by parallel execution. Simple high order multiprocessors can efficiently execute the Chaotic Relaxation and Newton-SOR Algorithms. The bordered block diagonal form algorithms require that the processors solving the diagonal blocks alternate active solution periods with the processor solving the border block (cut-set processor). The cut-set processor exchanges all required information while the other processors are idle. The gains achievable are limited by the time required for the cut-set processor to solve the equations. Simple high order multiprocessors are developed to efficiently execute the Gauss-Seidel and Newton Algorithms, but the gains achievable are dependent on the extent to which the equations decompose to the bordered block diagonal forms.

The algorithms were programmed to determime the control and data sharing requirements. From these requirements the delays from parallel solution were estimated. The convergence rates of the algorithms were not altered by parallel solution, so that a rough ranking of the parallel solution techniques was made.

ACKNOWLEDGMENT

The author wishes to express his sincere appreciation to Dr. F. M. Brasch, Jr., of the Electrical Engineering Department of Northwestern University, for his direction and help during all phases of the author's Ph.D. program, particularly for his stimulating discussions and helpful suggestions during this dissertation.

The author also wishes to thank Dr. J. E. Van Ness, of the Electrical Engineering Department of Northwestern University, for his help and advice on Power Systems Models.

The author is indebted to Mrs. Babette Goldhammer for her skill and speed in typing the manuscript.

Finally the author is grateful to the U. S. Navy who sponsored the authors education through the Junior Officer Scientific Advancement Program (Burke Scholarship).

CHAPTER I

INTRODUCTION

Dynamic simulation is the prediction of the way in which a system will change with time. The system can be as simple as a resistor/capacitor filter or as complex as a modern oil refinery. For the RC filter, dynamic simulation would predict the voltage levels within the filter for a future time, based on the known starting voltages and input voltages. For an oil refinery, dynamic simulation might be used to predict what change in the quantity of a specific output would result from a change in the inputs, again based on the initial conditions and the other inputs.

The equations describing the system, called the model, are developed from the mathematical relationships within the system. Dynamic simulation is the solution of these equations which predicts the response of the actual system under the same conditions.

The development of the computer has enhanced the ability to solve the equations, allowing the system which can be simulated to grow in size and complexity. Without the computer to solve the equations, it would be nearly impossible to predict the response of large systems except by building and testing the actual system. The large complex systems of today can be designed and built, because the computer allows the actual system response to be predicted from the mathematical models of the system.

As the size and complexity of the models has grown, the cost of simulation has increased. The cost is directly related to the time required to solve the equations defining the system and the size of the equations. There

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

are uses of dynamic simulation where the time required to solve the problem is critical for reasons other than cost. An example is the use of dynamic simulation to provide control signals for the system. In this case, the changes in the system must be predicted in time to allow the appropriate controls to be instituted. If the solution cannot be found in real time, it cannot be used to prevent undesirable changes in system variables.

This thesis investigates the possibility of using a multiprocessor to reduce the time required to solve the dynamic simulation problem. The multiprocessor is basically many independent computers which can efficiently exchange control and solution information. The investigation begins by developing the parallelism of the algorithms used to solve the dynamic simulation problem and the parallelism inherent in the structure of the equations in the model. It then develops the computational requirements of the parallel algorithms and finally proposes and evaluates multiprocessor structures to execute the algorithms.

The goal of the use of multiprocessor structures is to achieve a solution in $1/n^{th}$ the time required for a single processor, where n is the number of computing elements in the multiprocessor. The restrictions on the size of n depend on the method of solution and the structure of the multiprocessor. It is shown that because of the small amount of information exchange needed for solution of the dynamic simulation problem, a large number of processors can be used efficiently.

I.A. Dynamic Simulation Problem

Simulation techniques were developed before the computer and involve

more than the analysis of a dynamic system. The first technique to be developed was iconic simulation, the use of scale models to study a system. The most recent development is discrete event simulation using random number generators to model probabilistic events. This thesis studies a third type - dynamic simulation - which is among the oldest types of simulation. [1] The more recent discrete event simulation is used within the study of dynamic simulation.

Dynamic simulation consists of establishing the mathematical relationships between elements of the system, then solving the equations to predict the changes which will occur in the system. This thesis only studies methods of solving the equations defining the system. The development of the equations which define large systems usually is not complicated. Once the mathematical relationships for the elementary parts of the system are known, their equations can easily be combined to develop the model describing the system.

Normally, the equations defining elementary systems can be solved analytically. But when complex systems are modelled, analytical solutions are available only for simplified linear representations. The first method developed for solving these equations was the use of numerical methods to approximate the solution. Originally, it took teams of mathematicians long periods of time to solve the models of what would be considered a simple system today. In 1922, the differential analyzer was invented. It allowed mechanical devices which integrate differential equations to be built. Scon thereafter the electronic analog computer replaced the mechanical version. The electronic analog computer allowed the solution of much larger and more complex systems models. The analog computer lacked the accuracy required

for solution of models over a long period of time. Further, it may require a difficult and time consuming process to program the model on the analog computer. When the first digital computer was built, one of the initial applications was dynamic simulation. As the speed of the digital computer improved, hybrid computers, a combination of digital and analog computers, were designed and built. The hybrid computer combined the speed of solution possible on analog computers, with the accuracy available in digital computers, to simulate complex models over long periods of time. As the speed of the digital computer improves it is becoming capable of solving the high frequency component equations previously only economically solved by analog computer.

Although the digital computer has many advantages over the analog computer, it is expected that analog computation will remain. Just as the digital computer has been improved so has the analog computer. The analog computer started out a higher level of sophistication so further improvements were more difficult and little research has been done of the analog computer. Digital simulation has been studied to find parallel solution methods, a feature analog computation has always used.

Dynamic simulation has become an extremely helpful tool in all phases of system studies, development, operation, and experimentation. The system designer can use dynamic simulation to insure the satisfactory completion of the system. Simulation proves the feasibility of a proposed system. The specifications for the parts of the system are developed by simulating the operation of the system. Simulation studies help develop operating procedures for the system by testing its response to numerous possible conditions called contingency planning. Through the use of dynamic simulation, a system can

be designed and developed with considerable confidence.

Dynamic simulation is also useful for the operation of a system. For some systems the only means of providing control information is through simulation. For control purposes the speed with which the dynamic simulation problem can be solved is critical. The response of the system must be predicted in time for control signals to be instituted which prevent unwanted changes in the output.

An example of this use of dynamic simulation for control is the newest gun-fire control system used by the Navy. This has typically been an exclusive analog application. However, the speed and maneuverability of modern aircraft and the need to compute the solution for multiple targets has resulted in the use of a digital computer.

Scientists also use dynamic simulation. It provides a means for scientists to perform experiments which are too costly to develop on the real system, or which might result in grave damage to the system if actually performed. Simulation can also provide information on parts of the system where measurements cannot be made.

For all these and other uses, dynamic simulation provides more information at less cost than other possible approaches. Without the ability to solve the dynamic simulation problem many modern achievements would not be possible.

The electric power industry is one of the largest users of dynamic simulation. Because of the high fixed costs and high reliability required in this industry, it has led the search for improvements in modeling. The power industry must be positive that the changes it makes in any system will not have any adverse effects on the system's operation. This is

accomplished by simulating the effect of the proposed changes.

A model for a power system consists of the interconnection of smaller models representing the generating stations and loads, and the network interconnecting these parts. The model of a generating station may contain from two to thirty differential equations depending on the degree of complexity required for the simulation. The equations describe the actions of the various parts of the station as a result of changes in the power requirements of the network at the location of the station. As the complexity of the model is increased (usually to allow a longer period of time to be accurately simulated),more parts are included in the model. For example, at the lowest level, a model for a generating system might be only a simple representation of the generator. If more complexity is desired, the generator model would be expanded and the voltage regulator added. Then the turbine and governor, and finally the boiler would be included to give a more complex model.

The vastly different uses of electric power result in a wide variety in the models which represent these loads. The simplest representation is by a constant power requirement at each node. Other simple representations are a constant current requirement or constant impedance. When the power requirements of load change with time, more complex models are needed. Then differential equations, similar to the generator station models, are required. An example of such a load is a large induction motor.

The network supplying the electrical power to the loads from the generators is represented by algebraic equations. Points where a generator or a load is connected to the network or where several power lines of the network are connected is known as a bus. Each bus requires a complex algebraic

equation to represent its effect on the time scale of interest. Normally a bus has only a few power lines and/or a generator or load connected to it, and only the variables for the connected parts occur in the equation for the bus. Even though there may be hundreds of busses in the model, the equation for each bus seldom depends on more than ten variables, and often fewer. The busses are all interconnected but one bus never connects to all of the other busses.

The models of the different parts of the power system are easily combined to form the complete model. The generator station equations are made to depend on the voltage of the bus which connects the generator to the remainder of the system. Similarly, the nonconstant load models depend on the voltage of their connecting bus. If a bus is to be added, then the algebraic equation is included and its variable added to all bus equations which have power lines connecting the existing busses to the new bus. The addition of a power line requires adding new coefficients to the busses it connects.

When a model must represent the power system of a large utility company, which may cover a multi-state region, the number of equations in the model becomes quite large. It is not difficult to combine the individual parts to form these large models. Normally, as the parts are more distant from the object to be studied, the models are simplified. Since the generator and load models only depend on the bus to which they are connected, and any one bus only connects to a few other busses, the equations of the model exhibit a property known as sparsity. Sparsity is the occurrence of only a small number of the possible variables in each equation and occurs frequently in large sets of equations. For power systems, sparsity results

from the small number of busses connected to any one bus.

For the power system the variables represent current, voltage, power, and control signals. Nonlinearities occur throughout the system primarily as a result of the representation of limits and saturation. A total model may consist of thousands of nonlinear differential and algebraic equations.

Any large dynamic simulation problem exhibits many of the properties described for the power systems. The mathematical relationships are defined by large sets of algebraic and differential (or difference) equations. Quite often these equations are simplified to the linear form to ease solution, but the nonlinearities sometimes have to be included in the model to obtain accurate results. Sparsity also exists in most large dynamic simulation problems.

This thesis assumes the following form of the differential and algebraic equations for the dynamic simulation problem:

$$\begin{aligned} \underline{\dot{x}} &= \underline{F}(\underline{x}(t),\underline{y}(t),\underline{u}(t),t) \\ \underline{0} &= \underline{G}(\underline{x}(t),\underline{y}(t),\underline{u}(t),t) \end{aligned} \qquad \text{I.A.1}$$

Where $\underline{x}$ is the state vector, $\underline{y}$ the algebraic vector, $\underline{u}$ is the vector of inputs, and t is time.

This research uses models of electric power systems wherever examples are required. These models are typical of all dynamic simulation problems, and the techniques proposed by this thesis can be used to reduce the time required to solve the equations of any dynamic simulation problem.

I.B Computer Solution of the Dynamic Simulation Problem

One of the first uses and still one of the largest uses of the computer is the simulation of dynamic systems. Certainly without the computer, simulation would not have progressed to its current widespread use. However, the systems analyst has been able to increase the size and complexity of the models faster than the computer has progressed. The time required by currently available computers restricts the size of the model to be simulated and the availability of the solution. Because of the ease of developing large models and the degree of interconnection of power systems, there is a need to solve larger dynamic simulation problems at a faster rate.

Many people have tried to reduce the time required for solution of the dynamic simulation problem. Some, Gear [4], Davison [5], Dommel and Sato [6], and Wu [7] have approached the problem by developing new algorithms for approximating the solution. Others,Davison [8], Chidambara [9], Undrill and Turner [10], Anderson [11], and Van Ness [12] have developed methods of reducing the size of the model without losing vital information. Neither group has achieved sufficient success to relieve the modeler's concern over the costs of performing dynamic simulation. This leaves a vast demand for the ability to simulate larger systems at more reasonable speeds.

As early as 1959, Gauss [14] realized that for a computer to solve a problem most efficiently, the structure of the computer must be based on the requirements of the problem. Lehman [15] and Rosenfeld [16] suggested that a problem must be examined thoroughly to discover the inherent parallelism which can be used effectively, rather than to try to find parallelism at the instruction level. Korn [17] first suggested the dynamic simulation problems suitability for parallel processing while Lehman [15] & Rosenfeld [16]

have examined the solution of systems of algebraic equations. None of them developed the ideas far enough to demonstrate the actual gains or the resulting computer structure, nor did they investigate the problems which might occur. Much more recently, Wu [7] has again suggested the multiprocessor structure for the solution of power system problems. Wu's efforts, however, are mainly in structuring the "diakoptics" type algorithms, which, unfortunately are applicable only to linear systems.

There have also been efforts to develop parallel numerical methods to solve differential equations. Miranker and Liniger [18] structured predictor-corrector type integration methods so that the prediction and correction equations could be executed on different processors. Nievergelt [19] proposed using a large parallel processor to integrate a single differential equation by using different processors for different time intervals. These methods found a much lower level of parallelism than examined by this thesis.

This thesis will examine the dynamic simulation problem to determine the major parallel paths available during execution which may be exploited to increase the speed of execution. Computing structures will be proposed which best suit the inherent parallelism. Execution speeds for these structures will be estimated. The underlying hypothesis is that the computing structure which most resembles the structure of the dynamic simulation problem will provide the most efficient execution of the model.

I.C Proposed Computer Development

Predominant computer technology dictates that problems must be solved in a serial fashion. Yet Gonzalez and Ramamoorthy [23], Baer and Russell [22], and others have shown that many problems can be solved in

parallel and have developed a graph theoretic methods of demonstrating the parallelism inherent in a problem or computer program. The numerical methods used to solve the dynamic simulation problem are readily analyzed by these techniques and may be shown to exhibit a very high degree of parallelism.

Not only has parallelism been found in computer programs but also advanced parallel computer designs have been formed and built. ILLIAC IV [24, 25, and 26] a parallel processor, was built and many other designs proposed (some were built). ILLIAC IV, in the terminology developed by Flynn [27], is a single instruction multiple data (SIMD) computing structure. That is, all processors execute the same instruction, but on different parcels of data. Another category proposed by Flynn is a multiple instruction multiple data (MIMD) computing structure. C.mmp [28] is an MIMD computer developed at Carnegie Mellon University specifically to study artificial intelligence problems. For equivalent processor features, the MIMD structure is more expensive but has greater capabilities than the SIMD structure. This thesis attempts to extract the successes and avoid the failures of these studies, in order to reduce the solution time of the dynamic simulation problem.

The set of equations representing a dynamic system consists of a large number of simultaneous nonlinear differential (or difference) and algebraic equations. Each equation defines a specific variable. The set of equations exhibit a high degree of sparseness and in general includes nonlinear terms scattered throughout.

To reduce the time required to solve the large sets of equations of the dynamic simulation problem, methods are developed to solve many of the equations concurrently. Since each equation requires different operations to be

performed on different data and there is no correlation between either the operations to be performed or the data to be used among the equations solving the equations concurrently requires independent control of the processing elements both for the operations to be performed and for the data to be used. In Flynn's terminology, the computing structure has to be of the MIMD form. Loosely linked processors each capable of executing its own independent computer program are required. The other computing structures fail in some respect to be able to solve completely concurrently the equations of the dynamic simulation problem.

The major problem in all parallel computation is the sharing of information between the parallel processes. This sharing can delay the solution process in two ways. A processor may have to idle until the information it requires is computed by another processor. Or a processor may be delayed by the physical restriction that only one processor can use a single resource of the multiprocessor. Most often this single resource is shared memory and the delay is known as memory contention.

This thesis proposes methods to minimize the delays normally resulting from multiprocessor solution by reducing the information which must be shared. The primary method used to reduce information sharing is to provide each processor with a private copy of all information available at the start of the solution process. This information is stored in a local memory which only one processor can access,to insure no delays will be encountered in accessing this data. (Rosenfeld [16] mentioned the use of local memories, but did not analyze their effects.)

The other method of reducing the amount of information which must be shared depends on the properties of the equations of the dynamic simulation

problem. By properly grouping the equations into blocks, the amount of information which must be shared between blocks can be minimized.

Computing structures are proposed and analyzed to determine the delays which will result from exchanging data. From this analysis it is possible to estimate the point where, with the addition of more processors, the resulting increase in the delays from sharing data will negate the gains in execution speed.

This analysis requires actually programming the numerical methods used to solve the dynamic simulation problem. From the programs, accurate estimates of the amoung of shared information and the time available to exchange the information can be obtained. Time is measured in memory cycles to make the analysis independent of the actual processor used. The delays are based on the ratio of the number of memory accesses requiring shared data to the total number of memory references. The actual decrease in solution time is based on the number of memory references in the parallel path plus the delays encountered,compared to the number of memory references a single processor would require. The numerical methods are presented in Chapter Two and analyzed in Chapter Three.

Chapter Four proposes multiprocessor structures which can exploit the parallelism found in the dynamic simulation problem. The structures are analyzed to determine the delays encountered and the time required for solving the dynamic simulation problem.

Chapter Five discusses further methods of reducing the sharing of data and decreasing the time required for solution of the dynamic simulation problem.

CHAPTER II

NUMERICAL SOLUTION METHODS

The dynamic system model under consideration consists of a large set of simultaneous equations of two types (see equation I.A.1). The first type is a first order differential equation, which may be nonlinear. With the differential equations, initial conditions must be specified. The second type is an algebraic equation, which also may be nonlinear. Because of the nonlinearities, direct solution is normally not possible, and iterative methods are required to find an approximate solution.

There are three methods of solving a set of algebraic and differential equations. Either the algebraic equations can be eliminated and the resulting equations solved, or the equations can be solved separately and the solutions matched for consistency, or the difference equations which approximate the solution of the differential equations may be combined with the algebraic equations and the entire set can be treated as algebraic equations [6].

Elimination of the algebraic equations is a long and difficult process. The resulting differential equations are nonsparse and require many more operations to solve than the original equations. When the equations are solved separately, the sparsity remains and the solution process is efficient unless the separate solutions are inconsistent. If this happens, the time required to find this solution has been wasted, and a new attempt must be made to find a consistent solution. Experience indicates that inconsistent solutions frequently occur in practice.

Recently Dommel and Sato [6] showed that the implicit integration schemes have many advantages. These schemes require an iteration process to find the next estimate; this allows the difference equations to be solved

[illegible]

concurrently with the algebraic equations. The concurrent iteration process insures that the solution found is consistent. The sparsity of the equations is maintained to include the block parallelism. Finally the implicit integration schemes allow the step size to be increased beyond that possible for the other methods. Since the solution of the entire set of equations in this format is no more difficult than would be required to solve the algebraic equations, the solution of the dynamic simulation problem by expressing the differential equations as algebraic equations through the use of the implicit integration schemes has become the accepted solution method.

II.A Numerical Integration of Differential Equations

Numerical integration methods are a key factor in the solution process of the dynamic simulation problem. Numerical methods were first used before mechanical integration machines were developed. Now, even though the actual solution algorithms are those for algebraic equations, the numerical integration formulas are needed to express the differential equations as algebraic equations. In this section the integration methods are presented, then in the next section the methods of solving the resulting set of algebraic equations are presented.

The simplest method of approximating the solution of a differential equation is Euler's Method. This method begins with an initial condition and estimates a new solution point a small time step away along the direction of the initial derivative. At the new estimate, this method evaluates the new derivative and again moves a small time step in the direction of the derivative for the next estimate. By repeating this procedure a series of points is calculated which approximates the solution to the differential

equation. The series is calculated successively from the following formula:

$$x(t+h) = x(t) + h*F(x(t), y(t), u(t), t) \qquad \text{II.A.1}$$

where h is the time step. The accuracy of Euler's Method over many time steps, depends proportionately (linearly) on the size of the time step. For almost all uses this high degree of error is unacceptable.

To increase the accuracy of the approximation the amount of information on which the approximation is based must be increased. Methods with improved accuracy either use more points as a basis for the new estimate, or use more calculations of the derivative, or both. When the methods provide direct solution of the new estimate they are said to be explicit methods. For increased accuracy, the new estimate is included in the formulas and the solution is obtained by iteration. These methods are known as implicit.

The Runge-Kutta methods use several evaluations of the derivative over a single step. These methods are in general the most accurate of the explicit methods and are simple to apply. They have a disadvantage in that they require that the step size be much smaller than the time constant of the highest frequency component of the solution. Quite often these very high frequencies are of negligible magnitude and could be reasonably ignored. However, the Runge-Kutta Methods require the accurate approximation of the high frequency component to be accurate. For this reason the Runge-Kutta Methods will not be considered in this thesis. (For a complete discussion of Runge-Kutta methods see [4], [29], [36].)

The multi-step methods use the estimate of the solution for several time steps as a basis to predict the next estimate. The explicit multi-step

[illegible]

methods are less accurate than the explicit Runge-Kutta schemes. With the multi-step methods the next estimate is easily added as one of the points of the basis. The addition of this point greatly increases the accuracy of the solution. An added benefit is the ability to increase the step size beyond the time constant of the highest frequency component of the solution without losing accuracy. An explanation of the differences between the explicit and implicit methods is given in [30] as:

> ... an intuitive example can be given by using the first order, $K = 1$, integration method and the differential equation $X' = -\lambda X \; \lambda \geq 0$.
>
> In the explicit case, the formula becomes:
>
> $$X_n = X_{n-1} + hX'_{n-1} = (1-h\lambda)X_{n-1}$$
>
> Since the exact solution $X(t) = Ce^{-\lambda t}$, $\lambda > 0$ is a positive decreasing function we know that $0 < X_n < X_{n-1}$ or $0 < 1-\lambda h < 1$. Therefore $h < 1/\lambda$.
>
> In the implicit formulation of the same example the integration formula becomes:
>
> $$X_n = X_{n-1} + hX'_n = \frac{X_{n-1}}{1+h\lambda}$$
>
> Since X is a positive decreasing function $1+h\lambda > 1$ which is true for all $h > 0$.
>
> As one can see in this example, in the explicit case the step size h is restricted by the size of λ while no such obvious restriction is indicated in the implicit use.

The simplest multi-step implicit method is the trapezoid method. This method does not provide an analytic estimate of the new solution. It uses the average of the present and new derivative to step to the new

point. Since the new derivative cannot be found without knowledge of the new point the estimate of the new point is obtained by iteration. The formula for the trapezoid method is:

$$x^{i+1}(t+h) = x(t) + {}^{h}/{2}\,[F(x(t),\ y(t),\ u(t),\ t) + F(x^{i}(t+h),\ y^{i}(t+h),\ u(t+h), t+h)] \qquad \text{II.A.2}$$

which is an algebraic equation and can be solved by the same techniques used to solve the algebraic equations in the model. These methods are presented in the next section.

An easy,accurate method of starting the solution procedure is to use the current position as the first estimate of the new point. With this estimate, the first iteration reduces to Euler's Method. When the trapezoid method is solved for many steps the error in the estimate of the solution will be on the order of the square of the step size.

Higher order implicit methods also exhibit the ability to use larger step sizes. To some extent, the accuracy of the solution also increases with the order of the method,thus allowing larger step sizes for a desired solution accuracy. Associated with the use of additional points for predicting the new estimate in the higher order methods is an increase in the computation required for predicting this estimate. Both the accuracy required for the solution and the time available to find that solution must be used to establish the step size and order of the method to be used. The results of this thesis are based on the trapezoid method, but no assumptions are used which would prevent the application of higher order methods. (More detailed discussion of the multi-step methods can be found in [4],

[29], [31].)

All integration schemes discussed herein remain valid when a set of differential equations is to be solved. The only requirement is that all equations are integrated over the same time step together. With the implicit integration methods, the equation for each variable must be iterated until all variables have converged. Apparent convergence of one variable is not sufficient, and in fact, that variable may change as the other equations are iterated further. Normally, as the number of equations grows, so will the number of iterations required for convergence.

II.B Numerical Solution of Algebraic Equations

By use of the implicit integration formulas the equations of the dynamic simulation problem have all been converted to algebraic equations. Some of the equations resulting from differential equations and many of the original algebraic equations are nonlinear. This normally prevents the direct solution of the equations, requiring that the solution be found by iteration. This thesis considers two types of iteration methods, the linear methods and the Newton methods. The linear iterative methods use the fixed point form of the algebraic equations to calculate the new estimate, and have the property that the present error may be bounded by a constant factor times the previous estimate's error. The Newton methods require the evaluation of the function and its derivative, to form a set of linear equations whose solution is the new estimate. The solution of these equations cause the error of the estimate to be reduced at a quadratic rate. In general all iterative methods can be expressed as:

$$\underline{y}^{k+1} = \underline{\emptyset}(y^k)$$

where if $\underline{y}^* = \underline{\emptyset}(y^*)$ then $\underline{0} = \underline{G}(\underline{y}^*)$ II.B.1

To simplify the discussion, the remainder of this section will discuss the general algebraic equations $\underline{0} = \underline{G}(\underline{y})$ or more generally $\underline{v} = G(y)$.

To develop the linear iterative methods, assume also that the equations are linear, $\underline{v} = G\underline{y}$. G is then expressed as the matrix sum G = D-L-U where D is the elements of the diagonal of G and L and U the elements of the lower and upper triangular parts of G respectively. The general iterative formula is now

$$D\underline{y}^{k+1} = (L + U)\,\underline{y}^k + \underline{v} \qquad \text{II.B.2}$$

The three linear methods to be presented can be expressed in terms of the decomposition of the matrix differ only in the location in the algorithm where the new estimate, $\underline{y}^{k+1}$, is used in the calculations of the other new estimates. The Jacobi method solves for all of the new estimates based on the last value and then updates the variables to the new estimate. (The calculations are all based on the same point for each variable.) The Jacobi algorithm appears as:

$$\underline{y}^{k+1} = D^{-1}(L + U)\underline{y}^k + D^{-1}\underline{v}$$

or

$$\underline{y}^{k+1} = B\,\underline{y}^k + D^{-1}\underline{v} \qquad \text{II.B.3}$$

The Gauss-Seidel method uses the new estimates of the variables in the calculations as soon as the new estimate is found. This frees the storage required to save both values of the variables. The Gauss-Seidel method appears as:

$$\underline{y}^{k+1} = (D - L)^{-1} U\underline{y}^{k} + (D - L)^{-1}\underline{v}$$

or

$$\underline{y}^{k+1} = C\,\underline{y}^{k} + (D - L)^{-1}\underline{v} \qquad \text{II.B.4}$$

The third method, Chaotic Relaxation, was proposed by Rosenfeld [16], and studied by Chazan and Miranker [31], for the solution of algebraic equations by a multiprocessor. Like the Gauss-Seidel method, it updates the values of the variables as soon as the new estimate is calculated. Because of the use of a multiprocessor for the calculations, the order with which the variables is updated is not constant. The decomposition is different for each iteration. This prevents expressing Chaotic Relaxation in the terms of equations II.B.3 and 4.

The development of the linear iterative methods for linear equations does not restrict their application to only the linear equations. For non-linear equations, each equation is solved for its diagonal variable (i^{th} variable in the i^{th} equation).

$$y_i^{k+1} = \hat{\emptyset}_i(y_j^k) \quad \text{for } j \neq i \qquad \text{II.B.5}$$

The rate of convergence of the linear iterative methods can be improved by the use of an acceleration factor, α. The acceleration factor

modifies the update either by reducing or increasing the magnitude of the correction applied to the variable. The use of an acceleration factor maybe thought of as altering the eigenvalues of the iteration matrix in order to increase the rate of convergence. With acceleration the linear iterative methods appear as:

$$\underline{y}^{k+1} = \alpha\underline{\emptyset}(\underline{y}^{k}) + (1-\alpha)\underline{y}^{k} \qquad 0 < \alpha < 2 \qquad \text{II.B.6}$$

For $\alpha < 1$ this is known as under relaxation, and for $\alpha > 1$, over relaxation.

The other iterative methods considered by this thesis are the Newton Methods. The Newton algorithms achieve a quadratic convergence rate by projecting along the slope of the equations at the current estimate to the axis. The intersection gives the new estimate. The slope of the equation is computed from the Jacobian, $J = \partial\underline{G}(\underline{y}) / \partial\underline{y}$. The current value of the equations is also required, $\underline{G}(\underline{y})$. The new estimate is:

$$\underline{y}^{k+1} = \underline{y}^{k} - J^{-1}\, \underline{G}(\underline{y}^{k}) = \underline{\emptyset}(\underline{y}^{k}) \qquad \text{II.B.7}$$

The evaluation of the equations, $\underline{G}(\underline{y})$, requires approximately the same amount of time as one complete linear iteration. By interspersing the evaluation of the Jacobian with the evaluation of the functions, the Jacobian can usually be calculated with only a few additional operations. However, the solution of the resulting equations, $J\underline{\delta} = \underline{G}(\underline{y})$, requires on the order of n^3 operations. The increase in the rate of convergence usually more than compensates for the increase in the number of operations.

A combination of the Newton and linear methods can be used to reduce the total number of operations required for solution of the Newton update, $\underline{\delta}$. The equations and the Jacobian are evaluated as above. The Jacobian is partitioned and the diagonal blocks solved. The off diagonal values are then compensated for by iterating the diagonal solution. If iterated until convergence, then a true Newton update is found. With a set number of iterations of the Newton update, the convergence rate is less than quadratic. This method, known as Newton-SOR, appears as:

$$\underline{\delta}^{m+1} = J_D^{-1} [\underline{G}(\underline{y}^k) + J_H \underline{\delta}^m] \qquad \text{II.B.8}$$

$$\underline{y}^{k+1} = \underline{y}^k + \underline{\delta}^* \qquad \text{II.B.9}$$

Where J_D is the diagonal blocks of the Jacobian and J_H the off diagonal blocks. $\underline{\delta}^*$ is the unique solution of II.B.8.

The Newton-SOR method saves operations because the solution of the diagonal blocks requires $\Sigma\ s{*}n_i^3$ operations rather than the n^3 operations. (s is the number of shared variables required by that block, n_i the number of variables in the block, and n the total number of variables). When the equations are sparse, the number of operations required to solve for the Newton update can be reduced.

The solution time of the linear methods is reduced by accelerating the convergence rate. For the Newton methods the rate of convergence is much higher and further improvement is not needed. However, several methods of reducing the number of operations required to solve for the Newton update have been developed. The first method is LU Factorization. After LU

Factorization was developed, it was found that further savings are possible by properly ordering the equations. Another improvement was obtained by sparsity programming.

LU Factorization is an efficient method of solving a set of linear equations. The matrix of coefficients is decomposed by elementary transformations into a lower triangular and upper triangular matrix. The decomposition results in two trivial sets of equations which can be solved by substitution. If $J\underline{\delta} = \underline{G}$ then $LU = J$ so $L\underline{z} = \underline{G}$ and $U\underline{\delta} = \underline{z}$. This method of solution requires the minimum number of operations for a general set of equations. [59]

When the equations are sparse it was found that by arranging the equations in the LU Factorizations in the proper order, the number of coefficients which were filled in (changed from zero to nonzero value) could be reduced. The process of permuting the order of the equations to achieve a sparse LU Factorization is known as optimal ordering.

The most widely used optimal ordering expresses the equations by a connection graph. The equations defining a variable become nodes with branches to all other nodes (variables) which occur in the equation. An arbitrary node of lowest degree is chosen as the first node to be eliminated. The paths which would be removed by the elimination of that node are replaced by new branches between the remaining nodes, and the first variable is deleted. Again the node of minimum degree is chosen for elimination, the new branches added, and the node deleted. This process is repeated until all nodes are eliminated. The order with which the nodes are eliminated becomes the order the equations should be included in the LU Factorization. This ordering does not produce the true optimal ordering, but the

increased computation required to find a closer estimate of the true optimal ordering is not normally regained in the solution of the equations in true optimal order. [33]

When the number of zero coefficients is very large, many additional operations can be saved by sparsity programming. These techniques omit the operations whose results are known to be zero before the operations begin.

Through the use of optimal ordering and sparsity programming, the number of operations required to solve for the Newton update can be made proportional to n for large, very sparse sets of equations.

Generally, the linear iterations require a much smaller amount of computation than the Newton methods, but also have a much slower convergence rate. All three of the linear methods require approximately the same amount of computation, and the Jacobi method requires a slight increase in the amount of memory required, but not nearly as much more memory as the Newton methods require. When the methods are arranged into parallel form other advantages are displayed. To better be able to choose the algorithms which are used to solve the dynamic simulation problem, the convergence properties of the methods must be examined.

The convergence properties of the algorithms can be divided into two categories, the region of convergence, and, given convergence, the rate with which the error of the estimate is reduced. The analysis possible on the convergence properties and on the number of operations required to compute a new estimate does not conclusively support one method. Only because of the vast reductions in the number of operations required to compute the Newton update and the experience with the convergence properties of the

Newton method, has it become the accepted solution method for power systems solution. Even though the analysis can be extended for the multiprocessor solution, only experience will establish the superior method.

For the convergence of the linear iterative methods, the equations will be restricted to being irreducibly diagonally dominant. That is, the coefficient of the i th variable in the i th equation is larger in absolute value than the sum of the absolute values of the other coefficients of that equation. Because of the structure of the network equations of the power systems, diagonal dominance normally occurs. Proper selection of the time constant insures diagonal dominance for the differential equations. It is well known that the linear methods do not converge for some of the power system models. This is because the diagonal dominance has been destroyed in these models by the existence of large capacitor banks and/or other devices which cause relatively large off diagonal coefficients.

A major difficulty in comparing the rate of convergence of these solution processes is that some methods may converge while other methods diverge for the same set of equations. Diagonal dominance is a normal property of the network equations, which, if present, allows the convergence of the methods to be proven, and the rates of convergence compared.

For linear diagonally dominant sets of equations the true Newton method is equivalent to direct solution, and the Newton-SOR method can be shown to converge. In this case the direct solution of the Newton method produces the highest convergence rate, followed by the Newton-SOR method. The Gauss-Seidel method has the highest convergence rate of the linear methods studied followed by Chaotic Relaxation and then the Jacobi method. (Actually the Chaotic Relaxation method convergence rate requires all

equations to be updated equally.)

The notion of diagonal dominance can be extended to nonlinear equations, called M-functions, and similar results obtained. For nonlinear equations, the acceleration must be limited to under relaxation to prove convergence. But it is expected that by monitoring the convergence progress, over relaxation could normally be used.

For the Newton methods, the equations must exhibit certain continuity and invertability conditions in order to prove convergence. Diagonal dominance is not necessary and if singularities exist may not even be sufficient for the convergence of the Newton methods. The Newton-SOR method does require conditions comparable to block diagonal dominance for the convergence of the SOR iterations.

The remaining part of this section shows the theoretical justification for these claims. The proofs are omitted if the results are unchanged from the source. The following definitions are required to prove the desired results.

The convergence properties are established on a compact subspace $D \subset R^n$, with the following partial ordering, $\underline{x} \leq \underline{y}$ implies $x_i \leq y_i$ for $i = 1,2,\ldots, n$ and $\underline{x} < \underline{y}$ implies $\underline{x} \leq \underline{y}$ and $x_i < y_i$ for at least one $i = 1,2,\ldots, n$.

A linear mapping $A:R^n \to R^n$ is denoted by $A \varepsilon \mathcal{L}(R^n)$ and may be represented by a matrix. A nonlinear mapping A is denoted by $A:D \subset R^n \to R^n$. The mappings may also be ordered, where for $A, B:D \subset R^n \to R^n$. $A \leq B$ implies $A\underline{x} \leq B\underline{x}\ \forall \underline{x} \varepsilon D$. If $A, B \varepsilon \mathcal{L}(R^n)$ $A \leq B$ implies $a_{ij} \leq b_{ij}$ $i,j = 1,2,\ldots, n$. A mapping $F:D \subset R^n \to R^n$ is isotone (antitone) if $F\underline{x} \leq F\underline{y}$ ($F\underline{x} \geq F\underline{y}$) whenever $\underline{x} \leq \underline{y}$; $x, y \varepsilon D$, and a mapping is inverse isotone if whenever $\underline{x} \leq \underline{y}$, then $F\underline{x} \leq F\underline{y}$.

A mapping F is non-negative if $F\underline{x} \geq 0$ if $\underline{x} \geq 0$ (also implies F is isotone).

For $A \in \mathcal{L}(R^n)$ the spectral radius, $\rho(A)$, is defined as the magnitude of the largest eigenvalue of A. It is always true that $\rho(A) \leq ||A||$.

A matrix $A \in \mathcal{L}(R^n)$ is an M-matrix if A^{-1} exists and is nonsingular, and $a_{ij} \leq 0$, $i,j = 1,2,3,\ldots, n$.

For $A \in \mathcal{L}(R)$, the matrix A^+ is defined as

$$a_{ii} > 0 \quad i = 1,2,\ldots, n$$

and

$$a_{ij} = -|a_{ij}| \qquad i \neq j$$

By comparing the results from [32], [35], [37], [38], [39] and [41], it can be shown that if a matrix A ~~is irreducibly diagonally dominant, then~~ A^+ is an M-matrix and for the iteration matrix $B^+ = (I - D^{-1}A^+)$; $\rho(B) \leq \rho(B^+) < 1$.

The linear iterative methods for $G\underline{y} = \underline{v}$ can be expressed in terms of matrix sums, $G = D - L - U$. For the Jacobi method the iteration matrix is $B_j = D^{-1}(L + U)$. With Gauss-Seidel the lower triangular matrix L is included with D so that the iteration matrix becomes $B_g = (D - L)^{-1}U$. For Chaotic Relaxation the iteration matrix changes from iteration to iteration by inclusion of different rows of L $(L = L_c + L_c^*)$. The matrix is defined as $B_c = (D - L_c)^{-1}(L_c^* + U)$. By the theorems of regular splittings [37] the following comparison can be made.

$$\rho(B_g) \leq \rho(B_c) \leq \rho(B_j)$$

Also the convergence of the SOR iterations of the Newton-SOR method can be shown.

The error of the ith iterate is $e^i = y^i - y^*$, and produces an error of $e^{i+1} = |B|e^i$ on the next iteration. It can be shown that $|e^{i+1}| \leq \rho(B)\,|e^i|$. But because B changes from iteration to iteration the requirements of the proof are more complex. These comparisons are helpful in comparing the convergence rates of the methods.

Chazan and Miranker [32], developed a notation for all linear iterative methods, which is used in the theorems showing convergence. First each new estimate is counted as an iteration, j. Then $k_i(j)$ the ith component of an n+1 vector k(j), represents the age of the ith variable being used in the calculation of the new estimate of the $k_{n+1}(j)$th variable. (Age is the number of past iterations.) With this notation the linear methods are expressed as:

$$v_i = g_i\left(y_1^{\,j-k_1(j)}, \ldots, \hat{y}_i^{\,j}, y_{i+1}^{\,j-k_{i+1}(j)}, \ldots, y_n^{\,j-k_n(j)}\right)$$

and

$$y_i^{\,j+1} = \begin{cases} (1-\alpha_j)\, y_i^{\,j-k_i(j)} + \alpha\hat{y}_i^{\,j} & \text{if } i = k_{n+1}(j) \\ y_i^{\,j} & \text{if } i \neq k_{n+1}(j) \end{cases} \qquad \text{II.B.10}$$

With this notation the Gauss-Seidel method is represented by $k_i(j) = 0$ for all i, j, and $k_{n+1}(j) = (j-1) \bmod n+1$.

The Jacobi method is represented by $k_1(j) = k_2(j) = \ldots = k_n(j) = (j-1) \bmod n$, and $k_{n+1}(j) = (j-1) \bmod n+1$.

For the Chaotic Relaxation method $k_i(j)$ can take on any values, but to show convergence $k_i(j) < S$ for all i, j and $k_{n+1}(j) = i$, $i = 1,2,\ldots, n$ infinitely many times.

Theorem II.B.1 Let $A \in \mathcal{L}(R^n)$ such that A^+ is an M-matrix. Then the sequence of estimates y^i converges for II.B.10 and for α such that $0 < \alpha < 2/(1+\rho(B))$. [32]

Thus for linear sets of equations, the Jacobian J, is simply the coefficient matrix, and since if A^+ is an M-matrix A^{-1} exists so the true Newton method results in direct solution. Further since the block diagonal is a regular splitting the SOR iterations of the Newton-SOR method converge.

For linear sets of equations the convergence rates in decreasing order are Newton, Newton-SOR, Gauss-Seidel, Chaotic, and Jacobi methods.

For nonlinear equations the notion of diagonal dominance is conveyed by the following two definitions. A mapping $G: D \subset R^n \to R^n$ is off diagonally antitone if for any $x \in R^n$ the functions $\varphi_{ij}: \{t \in R^1 | x+te^j \in D\} \to R^1$; $\varphi_{ij}(t) = g_i(x+te^j)$ $i \neq j$ $i,j = 1,2,3,\ldots, n$ are antitone; and G is diagonally isotone if for any $x \in R^n$ the functions $\varphi_{ii}: \{t \in R^1 | x+te^i \in D\}$; $\varphi_{ii}(t) = g_i(x+te^i)$ $i = 1,2,\ldots,n$ are isotone. ($\{e\}$ is the orthonormal basis for R^n such that $e^i = (0,0,\ldots, 0, 1, 0,\ldots,0)^t$).

The mapping $G : D \subset R^n \to R^n$ is an M-functions if G is inverse isotone and off diagonally antitone. Further, the Jacobian of a set of M-functions is an M-matrix. With these definitions and the notation of Chazan and Miranker, convergence for nonlinear sets of equations can be proven.

Theorem II.B.2 Let $G: D \subset R^n \to R^n$ be a continuous, off diagonally antitone and strictly diagonally isotone. Suppose for some $v \in R^n$ there exists $\underline{x}^0, \underline{y}^0 \in D$ such that $x^0 < y^0$; $J = \{x \in R^n | x^0 \leq x \leq y^0\} \subset D$; $Gx^0 \leq v \leq Gy^0$ (a). Then for $\epsilon \in (0,1]$ and any sequence $\{\alpha_j\} \in [\epsilon, 1]$ the iterates $\{x^j\}$ and $\{y^j\}$ given by the solution of II.B.10 (b) starting from x^0 and y^0, respectively, are uniquely defined and satisfy

$$x^0 \leq x^j \leq x^{j+1} \leq y^{j+1} \leq y^j \leq y^0$$

$$G\,x^j \leq v \leq G\,y^j \qquad j = 1,2,\ldots \qquad \text{(c)}$$

as well as

$$\lim_{j\to\infty} x^i = x^* \leq y^* = \lim_{j\to\infty} y^j \qquad Gx^* \leq v \leq Gy^* \qquad \text{(d)}$$

Proof: The proof follows [35] and proceeds by induction. Suppose for some $j \geq 0$, $i \geq 0$

$$x^0 \leq x^j \leq y^j \leq y^0 \qquad Gx^j \leq v \leq Gy^j$$

$$x_i^{j-k_i(j-1)} \leq x_i^{j-k_i(j)} \leq y_i^{j-k_i(j)} \leq y_i^{j-k_i(j-1)} \qquad \text{(e)}$$

From (a) define the functions

$$\gamma(\nu) = g_i(x_1^{j-k_1(j)},\ldots, x_{i-1}^{j-k_{i-1}(j)}, \nu, x_{i+1}^{j-k_{i+1}(j)},\ldots)$$

$$\beta(\nu) = g_i(y_1^{j-k_1(j)},\ldots, y_{i-1}^{j-k_{i-1}(j)}, \nu, y_{i+1}^{j-k_{i+1}(j)},\ldots)$$

are defined for $\nu \in [x_i^0, y_i^0]$. From (e) and the off diagonal antitonicity of G

$$\beta(\nu) \leq \gamma(x_i^{j-k_i(j)}) = g_i(x^j) \leq v_i \leq g_i(y^j) =$$

$$\beta(y_i^{j-k_i(j)}) \leq \gamma(y_i^{j-k_i(j)}) \qquad \text{(f)}$$

By continuity and strict isotonicity of γ and β, (f) implies the existence of a unique $\hat{x}_i^j$ and $\hat{y}_i^j$ for which

$$\beta(\hat{y}_i^j) = v_i = \gamma(\hat{x}_i^j) \; ; \; x_i^j \le \hat{x}_i^j \le \hat{y}_i^j \le y_i^j$$

where the relation $\hat{x}_i^j \le \hat{y}_i^j$ is a consequence of the tonicity properties of G. Because $\alpha_j \epsilon [\epsilon, 1]$

$$y_i^j \ge y_i^{j+1} = (1-\alpha_j)y_i^j + \alpha_j \hat{y}_i^j \ge \hat{y}_i^j \ge \hat{x}_i^j$$

$$\hat{x}_i^j \ge x_i^{j+1} = (1-\alpha_j) x_i^j + \alpha_i \hat{x}_k^j \ge x_i^j$$

which shows (e) holds for i = 1,2,..., n. Hence $x^j \le x^{j+1} \le y^j$.
From this it follows that

$$g_i(y^{j+1}) \ge g_i(y_1^{j-k_1(j)}, \ldots, \hat{y}_i^j, \ldots, y_n^{j-k_n(j)}) = v_i$$

and

$$g_i(x^{j+1}) \le g_i(x_1^{j-k_1(j)}, \ldots, \hat{x}_i^j, \ldots, x_n^{j-k_n(j)}) = v_i$$

This completes the induction and proof of (c). Clearly then the limits exist.

$$\lim_{j \to \infty} x^j = x^* \le y^* = \lim_{j \to \infty} y^j$$

Now since

$$y_i^{j+1} \geq \hat{y}_i^j = -\frac{1}{\alpha_j}(y_i^j - y_i^{j+1}) + y_i^j \geq -\frac{1}{\varepsilon}(y_i^j - y_i^{j+1}) + y_i^j$$

$$x_i^{j+1} \leq \hat{x}_i^j = -\frac{1}{\alpha_j}(x_i^{j+1} - x_i^j) + x_i^j \leq -\frac{1}{\varepsilon}(x_i^{j+1} - x_i^j) + y_i^j$$

imply that $\lim_{j\to\infty} \hat{x}_i^j = x_i^*$ and $\lim_{j\to\infty} \hat{y}_i^j = y_i^*$.

Therefore $Gx^* = v = Gy^*$.

Theorem II.B.3 Suppose the conditions of the above theorem holds: Let $k_i^1(j)$ and $k^2(j)$ be given such that $k_i^1(j) = k_i^2(j)$ $i = 1,2,\ldots,n$ except for ℓ when $k_\ell^1(j) < k_\ell^2(j)$ (a). Then for $1 \neq \ell$, $Gx^0 < v$.

$$g_i(x_1^{j-k_1^2(j)},\ldots, x_\ell^{j-k_1^2(j)}) \leq g_i(x_1^{j-k_1^1(j)},\ldots, x_\ell^{j-k_\ell^1(j)},\ldots,x_n^{j-k_n^1(j)})$$

and if $v < gy^0$ then

$$g_i(y_1^{j-k_1^1(j)},\ldots, y_n^{j-k_i^1(j)} \leq g_i(y_1^{j-k_1^2(j)},\ldots, y_n^{j-k_n^2(j)})$$

$$\hat{x}_i^{j^2} \leq \hat{x}_i^{j^1} \leq \hat{y}_i^{j^1} \leq \hat{y}_i^{j^2} \qquad \text{(c)}$$

Proof: From the previous theorem and (a) $x_\ell^{j-k_\ell^2(j)} \leq x_\ell^{j-k_\ell^1(j)}$. The off diagonal antitonicity of G yields (b) and the inverse isotonicity yields (c). The same steps hold for $\hat{y}_i^j$.

This shows that for the linear methods discussed, the Gauss-Seidel method reduces the error in the estimates at the highest rate and the Jacobi at the lowest, with Chaotic Relaxation in the middle.

The Newton methods have different requirements for convergence than the linear methods.

<u>Theorem II.B.4</u> Let $G: D \subset R^n \rightarrow R^n$ where $x^0 \epsilon D$, $G(x^*) = 0$, G' is continuous at x^* and $G'(x^*)$ is nonsingular. Then the Newton iterations

$$x^{k+1} = x^k - [G'(x^k)]^{-1} G(x^k) \qquad (a)$$

converge to x^*. Further, if there exists a γ and ρ such that

$$||G(x) - G(x^*)|| \leq \gamma||x - x^*||^\rho$$

then the Newton method converges superlinearly and if $G''(x^*)hh \neq 0$ $\forall h \epsilon R^n, h \neq 0$, then the Newton method converges quadratically. [38]

For the Newton-SOR method the convergence of the SOR iterations as well as the Newton iterations must be proved.

<u>Theorem II.B.5</u> Let $G: D \subset R^n \rightarrow R^n$ be G-differentiable on D and if G is inverse isotone or an M-function, then for any $x \epsilon D$ at which $G'(x)$ is nonsingular, the derivative is inverse isotone or an M-function respectively. [38]

<u>Theorem II.B.6</u> If $G: D \subset R^n \rightarrow R^n$ is an M-function on D then any subfunction is also an M-function. [35] Let $G'(x) = B(x) - C(x) = D(x) - L(x) - U(x)$ be regular splittings of G where B is block diagonal and D is diagonal then

$$\hat{H}_\alpha(x) = [(1-\alpha)I(x) + \alpha D^{-1}(x)\ L(x) + U(x)]$$

$$H_\alpha(x) = [(1-\alpha)I(k) + \alpha B^{-1}(x^k)\ C(x^k)]$$

From these mappings if G (x) is an M-matrix then $\rho(H_\alpha) \leq \rho(\hat{H}_\alpha) < 1$ for $\alpha \epsilon (0, 1)$. Now let m represent the number of SOR iterations.

<u>Theorem II.B.7</u> If G'(x) is an M-matrix, then $\rho(H_\alpha^{m+1}) \leq \rho(H_\alpha^{m}) < 1$ and as $m \rightarrow \infty$ the rate of convergence of the Newton-SOR iteration approaches that of a true Newton method. [38]

Thus when all methods converge the rates of convergence is ordered increasingly as true Newton, Newton-SOR m steps, Newton-SOR 1 step, Gauss-Seidel, Chaotic, & Jacobi.

II.C <u>Methods of Ordering the Equations</u>

The last section discussed ordering the equations to reduce the calculations required to solve for the Newton update. There are other methods of ordering which can also increase the parallelism inherent in the algorithms used for solving the dynamic simulation problem. These ordering methods make use of the sparsity of the equations, to reduce the coupling between blocks of equations. The blocks are then used as the primary method of assigning equations to the processors for parallel solution. There are two forms into which the equations can be arranged which have been found pertinent for increasing the effective parallelism. The first form attempts to arrange the equations so that the diagonal blocks have the fewest interconnections. Ideally, the equations of one block would have zero coefficients for all of the equations outside the block. But as the number of blocks is increased, some equations will exist with coefficients that cannot all be ordered into a diagonal block. This equation is then included with the block which minimizes the number of nonzero coefficients outside the blocks. These nonzero off block diagonal terms occur randomly throughout the matrix. This form of equations is called the near block diagonal

form. See Figure II.C.1 for a matrix representation of the ordered equations.

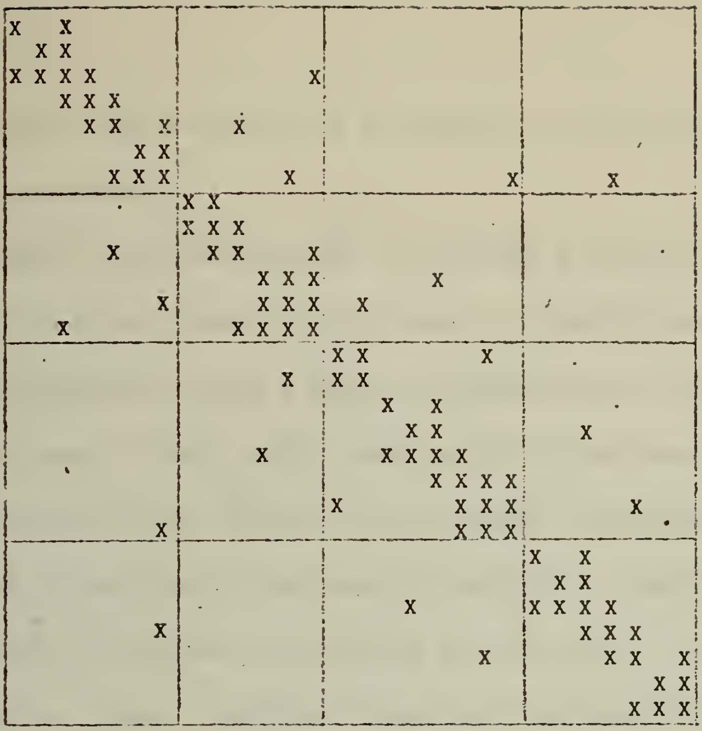
The other ordering method also attempts to arrange the equations so that the nonzero coefficients are in the diagonal blocks or the last block column. The equations which cannot be ordered into this form are included in the last block, where nonzero coefficients can occur randomly in any column.

These equations are known as the cut-set equations, since they cut the remaining equations into disconnected blocks. The equations of the diagonal blocks still have nonzero coefficients outside the diagonal block, but now these nonzero coefficients are grouped into the last block column. See figure II.C.2 for the matrix form of these equations, known as the bordered block diagonal form.

Ordering the equations to one of these forms can result in benefits beyond increasing the parallelism. The concentration of the zero coefficients greatly reduces the programming efforts normally associated with sparsity programming. Furthermore, just as optimal ordering reduced the number of operations required to solve for the Newton update, these orderings reduce the number of operations required to solve for the SOR equations to iterate in the Newton-SOR method. The near block diagonal form also reduces the number of variables which must be iterated for the Newton-SOR method.

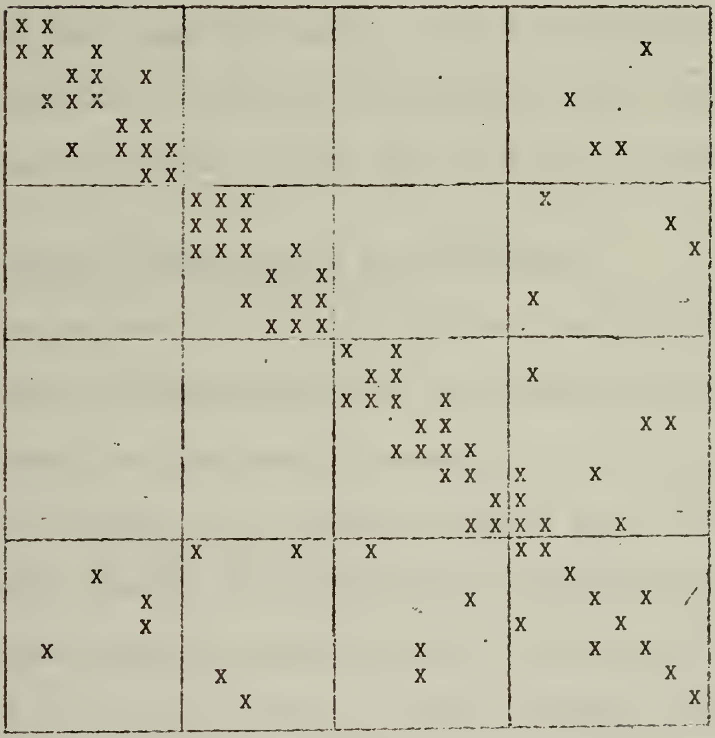
Carré [44], proposed a method of ordering the equations to the near block diagonal form. The methods used to obtain the bordered block diagonal forms can also produce a near block diagonal ordering.

There are several methods of ordering the sparse set of equations to the border block diagonal form. A true optimal solution exists but



Example of near block diagonal form

Table II.C.1



Example of bordered block diagonal form

Table II.C.2

the computation required to achieve this ordering may not be worth the effort required.

There are three methods of finding a bordered block diagonal form of the equations used in this thesis. Each is based on the graph model of the equations, with a node for each variable and a branch for each nonzero coefficient in the equation defining that variable. By finding the minimum vertex cut-set of the graph, the equations which should be ordered to the last block can be isolated. They are the equations which define the variables represented by the nodes of the cut-set. Once the cut-set is found, the other equations can easily be ordered to the disconnected blocks.

The algorithm of Ogbuobiri, et al [45], for finding the bordered block diagonal form of a set of equations appears to be the most efficient method. This algorithm and the others are presented in Chapter Five when the importance of ordering is discussed. The appendix demonstrates the performance of these methods on a test set of equations.

II.D Parallel Execution of the Algorithms

The purpose of this thesis is to examine multiprocessor computing structures for the execution of the dynamic simulation problem. The general theory for the parallel execution is to assign blocks of equations to each processor. The processors would then iterate their equations concurrently, sharing the required data between the processors. This section will prove that the algorithms can be iterated concurrently, and still converge to the same solution as serial iteration. The iterations are the only part of the solution which requires a large amount of time. The

initialization and other program set up tasks will not be considered for parallel execution.

A general purpose computer is assumed to be available for the compilation and initialization and of the programs. The general purpose computer would also perform the formation of the network matrices, and accomplish the plotting and other output functions. The use of the general purpose computer for these functions does not mean that the functions cannot be performed in parallel,only it means that the necessary modifications for parallel execution have not been studied. The amount of computation required for these operations does not appear to be sufficient to warrant parallel execution.

The most practical method of integrating differential equations is to convert the equations to algebraic form and iterate the differential equations with the algebraic equations. The conversion by use of the trapezoid rule of integration is included in the initialization and is not part of the parallel algorithms. The major iteration is within one time step, but the computation required at convergence before starting the next time step will be considered and included in the parallel algorithms.

The amount of parallelism differs with each algorithm and with the structure of the problem. First the general linear iterative algorithms parallel form is shown, followed by the parallel form of the Newton methods. The exact specifications of the parallel algorithms is left until Chapter Three where programs to execute these algorithms are developed and analyzed.

The linear iterative algorithms require the evaluation of a function defining each variable. The algorithms differ only in the sequence in which the variables are updated. The Jacobi scheme updates the variables only after a new value is calculated for all of the variables.

<u>Theorem II.D.1</u>: Let $B:R^n \to R^n$ be the Jacobi iterative functions and y^0 the initial estimate. Then if the new estimate is found by:

$$y_i^{k+j} = b_i(\underline{y}^k) + \underline{v} \qquad \text{II.B.3}$$

The sequence $\underline{y}^k$ found by updating the Jacobi variables by a serial process and $\hat{\underline{y}}^k$ found by updating the Jacobi variables by a parallel process, are identical for all k.

The proof is obvious because of the Jacobi method's application of the correction at the end of each iteration.

Because the parallel Jacobi algorithm uses the same data for all processors, the parallel solution must be synchronized. Convergence is determined by all processors through the use of control points in the parallel segments of the code. Control points are used to insure the required synchronization between processors, and to modify the execution sequence. Information from all processors is required before the individual processor is allowed to pass through the control point.

For the Jacobi algorithm all of the processors must be executing the same parallel segment of code. The code performs three distinct operations; computing a new estimate of the variables; applying this estimate to the variables; and advancing the time step. Control points are inserted in the code between each of these operations. The control points force the processors to be idle until all processors reach the point. Chapter Four discusses the implementation of control points, and their placement in the parallel algorithms.

The Chaotic Relaxation algorithm was designed for parallel execution. It evaluates the same functions as the Jacobi algorithm, but the new values of the variables are used as soon as they are calculated. For this reason, the Chaotic Relaxation scheme does not produce the same sequence of estimates as the Jacobi algorithm. It was shown [33] that whenever the Jacobi algorithm converges, the Chaotic Relaxation algorithm also converges to the same point. When over or under relaxation is used, the optimal acceleration coefficient will be different than the coefficient used for the Jacobi scheme. Normally the coefficient will be closer to unity for the Chaotic Relaxation.

The Chaotic Relaxation algorithm allows the elimination of the control points between the calculation of the new estimate and its application. All of the algorithms require the control point before the advance of the time step.

Although the Gauss-Seidel Algorithm is very similar to the Jacobi and the Chaotic schemes, the updating procedure for the Gauss-Seidel algorithm makes it difficult to execute in parallel. A straightforward implementation requires the ability to skew the solution process. An example best demonstrates this problem. Suppose a three variable set of linear algebraic equations was to be solved by the Gauss-Seidel process. The functions to be evaluated would be identical to the functions associated with the Jacobi or Chaotic schemes. In matrix form the iteration equations are

$$\begin{bmatrix} y_1^{i+1} \\ y_2^{i+1} \\ y_3^{i+1} \end{bmatrix} = \begin{bmatrix} 0 & a_{12} & a_{13} \\ a_{21} & 0 & a_{23} \\ a_{31} & a_{32} & 0 \end{bmatrix} * \begin{bmatrix} y_1^i \\ y_2^i \\ y_3^i \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \qquad \text{II.D.1}$$

Because of the time of the updating,the actual equations are evaluated as:

$$y_j^{i+1} = \sum_{k=1}^{j-1} a_{jk}*y_k^{i+1} + \sum_{k=j+1}^{n} a_{jk}*y_k^{i} + v_j \qquad \text{II.D.2}$$

In order to illustrate the skewing required the evaluation process is displayed in Table II.D.1. Time is displayed across the table and the blank areas indicate idle periods for processors. The delays resulting from, and the control required for, the skewed execution of the Gauss-Seidel scheme limit the gains achievable by parallel execution.

The Gauss-Seidel Algorithm is executable in a parallel manner much more efficiently when the equations are in the bordered block diagonal form. This form allows the updating to proceed in a sequential order and still use multiple processors. The use of the bordered block diagonal form exposes parallelism at the block level. A processor can thus be assigned to update the variables of a block. To illustrate the parallel algorithm assume the equations are linear. The matrix of coefficients, A, can be partitioned into the following form:

$$A = \begin{bmatrix} A_{11} & 0 & \cdots & A_{1n} \\ 0 & A_{22} & 0 & A_{2n} \\ \vdots & & \ddots & \vdots \\ A_{n1} & A_{n2} & & A_{nn} \end{bmatrix} \qquad \text{II.D.3}$$

Now partition $\underline{y}$ into the same blocks, and let the subscripts represent

Equation 1	v_1 +	$a_{12}y_2^k$	+	$a_{13}y_3^k$ =	y_1^{k+1}		v_1 +	$a_{12}y_2^{k+1}$
Equation 2	$=y_2^k$		v_2 +	$a_{23}y_3^k$		$a_{21}y_1^{k+1}$	$=y_2^{k+1}$	
Equation 3		+ $a_{32}y_2^k$	$=y_3^k$		v_3 +	$a_{31}y_1^{k+1}$	+	$a_{32}y_2^{k+1}$
	1	2	3	4	5	6	7	8

Time periods

Skewing required for Gauss-Seidel Algorithm

Table II.D.1

the blocks rather than the variables. The Gauss-Seidel algorithm could then be performed in the following three steps:

(a) $z_i = A_{ni} \cdot y_i^k(t)$; $i = 1,2,\dots, n$ II.D.4

(b) $y_n^{k+1}(t) = \omega \cdot \sum_i^n z_i - (\omega - 1) \cdot y_n^k(t)$ II.D.5

(c) $y_i^{k+1}(t) = \omega \cdot \{A_{ii} \cdot y_i^k(t) + A_{in} \cdot y_n^{k+1}(t)\} - (\omega - 1) \cdot y_1^k(t)$

$i = 1,\dots, n-1$ II.D.6

The necessity of storing and retrieving the portions of the cut-set update in steps a and b only slightly increases the number of operations that must be performed.

Theorem II.D.2: Steps (a) and (c) can be executed in parallel for all blocks, and still maintain the sequential updating of the variables.

Proof: For induction assume the k^{th} iteration has been achieved. Calculation of the z's requires only data available before any processor begins computation. (This is equivalent to the evaluation of the Jacobi functions.) Therefore the z's can be calculated in parallel. Now the second step, (b), must be performed. Evaluation of step three, (c), requires data from the final block, found in step two, and from the diagonal block. Because of the bordered block diagonal form, no processor requires data from other than the cut-set variables. Therefore step (c) can be executed in parallel. Now by updating the variables within a block sequentially, the order of updating becomes first the cut-set variables, in

order: then the diagonal block variables, in order. The fact that all of the diagonal blocks are updated in parallel does not matter since none of the new values are required outside the block until the cut-set variables must be updated.

Basically these algorithms have been modified to allow parallel execution rather than developing new parallel algorithms. This insures the convergence properties remain intact, with only the Chaotic Relaxation algorithms convergence rate depending on the number of processors. The Gauss-Seidel algorithm still convergences at the highest rate and the Jacobi at the lowest, for linear iterative methods. However, the execution time must also include the delays which result from controlling the parallel execution and from the sharing of data.

Parallel execution of the Newton algorithms is much more difficult. The increased computation required for these algorithms both helps and hinders parallel execution. Parts of the algorithm are very easily executed in parallel, such as the evaluation of the functions and solving for the elements of the Jacobian. The largest part of the computation, the solution of the Newton update, is very difficult to perform directly in parallel.

Evaluation of the functions, $\underline{G}$, and solution of the elements of the Jacobian, J, require only the previous values of the variables. This is equivalent to the computations required for a Jacobi iteration, where all of the data required for the operations is available before the evaluation begins. Therefore the following theorem is stated without proof.

Theorem II.D.3: The evaluation of the functions and the evaluation of

the elements of the Jacobian may be executed in parallel with the same results as serial execution.

The remainder of the Newton algorithm involves the solution of the equation for the Newton update, $J\delta = \underline{G}$. The linear iterative algorithms could be used to find $\underline{\delta}$ in parallel, but this should be avoided because of the time required for the linear algorithms to converge. Another prospect is to invert the Jacobian. Pease [54], proposed two parallel algorithms for this inversion. The schemes required the sharing of all the elements of the Jacobian and a high level of control. The methods are infeasible for a large Jacobian or a high order multiprocessor. There are two methods which simplify this solution considerably. The Newton-SOR algorithm overcomes the problem by avoiding the direct solution of the entire set of equations. The other method uses the bordered block diagonal form of the Jacobian to allow parallel direct solution.

The Newton-SOR algorithm solves the block equations explicitly then uses these results to iterate the shared variables to convergence, yielding a true Newton update. The iterations use Chaotic Relaxation, so that the Newton-SOR algorithm can be executed in parallel. By ordering the equations in the near block diagonal form the number of variables which are shared and thus the number of variables which must be iterated, is minimized. As the number of shared variables increases, the time required by the Newton-SOR algorithm increases cubicly. For a small number of shared variables, the ability to solve the blocks directly increases the region of convergence and reduces the execution time compared with simply a linear iterative method to solve the Newton update equations. The parallel Newton-SOR algorithm has basically the same sparsity and iteration control

requirements as the Chaotic Relaxation algorithm.

When the equations are in the bordered block diagonal form, the Newton update equations can be solved directly by block LU Factorization. As with the bordered block diagonal Gauss-Seidel algorithm, there are three steps required for solution. Because of the bordered block diagonal form the equations are already nearly in factored form. The initial equations are:

$$\begin{vmatrix} J_{11} & 0 & \cdots & J_{1n} \\ 0 & J_{22} & \cdots & J_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ J_{n1} & J_{n2} & \cdots & J_{nn} \end{vmatrix} * \begin{vmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_n \end{vmatrix} = \begin{vmatrix} G_1(\underline{y}^k) \\ G_2(y^k) \\ \vdots \\ G_n(\underline{y}^k) \end{vmatrix} \qquad \text{II.D.7}$$

The first step of the solution process is performed by all processors other than the cut-set processor. These processors factor the diagonal blocks into lower and upper triangular parts:

$$\text{(a)} \quad J_{ii} = \begin{bmatrix} L_{ii} & U_{ii} \end{bmatrix} \qquad i = 1,2,\ldots,n-1 \qquad \text{II.D.8}$$

and use the factored form to eliminate the last block row. This elimination results in the formation of a modification to the equations for the cut-set update. To the cut-set block must be added $J_{ni}\ U_{ii}^{-1}\ L_{ii}^{-1}\ \ J_{in}$ and to the cut-set functions must be added $J_{ni}\ U_{ii}^{-1}\ L_{ii}^{-1}\ G_i(y^k)$. In the second step the cut-set processor adds in the modifications by:

$$\text{(b)} \quad Z = J_{nn} + \sum_{i=1}^{n-1} J_{ni} U_{ii}^{-1} L_{ii}^{-1} J_{in}$$

$$Q = G_n(\underline{y}^k) + \sum_{i=1}^{n-1} J_{ni} U_{ii}^{-1} L_{ii}^{-1} G_i(\underline{y}^k) \qquad \text{II.D.9}$$

resulting in the equations appearing as:

$$\begin{vmatrix} U_{11} & 0 & \cdots & L_{11}^{-1} J_{1n} \\ 0 & U_{22} & \cdots & L_{22}^{-1} J_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & & Z \end{vmatrix} \begin{vmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_n \end{vmatrix} = \begin{vmatrix} L_{11}^{-1} G_1(\underline{y}^k) \\ L_{22}^{-1} G_2(\underline{y}^k) \\ \vdots \\ Q \end{vmatrix} \qquad \text{II.D.10}$$

The cut-set processor continues the second step by solving $Z\underline{\delta}_n = Q$ for the cut-set updates. After the conclusion of the solution process, the non-cut-set processors can solve for their variable's updates in the third step. In (c) the cut-set variable's updates are used in the back substitution calculations which yield the remaining updates.

$$\text{(c)} \quad U_{ii}\delta_i + L_{ii}^{-1}J_{in}\delta_n = L_{ii}^{-1}G_i(y^k) \qquad i = 1,2,\ldots, n-1 \qquad \text{II.D.11}$$

Block LU Factorization when performed in parallel results in the processors being idle for periods of time. During step (a) the cut-set processor is idle and the others are active. Then in (b) the cut-set processor is active and the others idle. Finally the others are active again and the cut-set processor is idle. The ability to solve in parallel is shown by:

<u>Theorem II.D.4</u>: The blocks of the Jacobian can be decomposed in parallel and the back substitution can be performed in parallel for the Newton iterative algorithm in bordered block diagonal form.

<u>Proof</u>: Since the Jacobian is in the bordered block diagonal form, decomposition of the diagonal blocks requires only the data within the blocks. The last row can also be eliminated in parallel, but this requires the summation of the elements of the cut-set block. Back substitution within the blocks requires only the previously found values of the cut-set variables, to solve for the variables within each block. Therefore, parallel solution of these two parts is possible.

Again the Newton methods were not altered to arrange the algorithms into parallel form. This preserves the convergence characteristics of these algorithms, and the increase in the speed of solution will depend only on the delays resulting from the sharing of data between processors and the control of the execution. The true Newton methods provides the highest rate of convergence, but the method requires the bordered block form for parallel execution. The Newton-SOR algorithm has a much faster execution rate if the equations are ordered to the near block diagonal form, but neither the convergence properties nor parallel execution depends on achieving a block form.

II.E Sparsity Programming

Sparsity programming has been discussed in several sections. It can be defined as a programming technique to be used to avoid computations whose result is known to be zero before the computation begins. Sparsity must be considered in all programs which require matrix like operations on a large number of variables. The solution time of the dynamic simulation problem can be reduced through use of sparsity programming.

Sparsity programming began as methods to try to fit a large sparse matrix into a small core memory. Later it was realized that a large amount of computation could also be saved. The savings in memory and execution speed vary with the number of nonzero elements. Sparsity techniques usually require at least three memory locations to store a coefficient, whereas nonsparse methods would require only one. For sparsity techniques to save execution time, an even higher ratio of zero elements is required.

There are three basic sparsity programming techniques. [33]. The first is to use an index array to point to nonzero elements. Another is to use linked lists. The third is to use linear code. The first two methods are primarily to save momory, but the linear code increases the storage required. Linear code greatly reduces the execution time and uses secondary storage efficiently. Linked lists are used when the exact number and location of the nonzero elements is unknown prior to execution.

Linked list techniques are used to allow the addition and deletion of nonzero elements. Typically there are two arrays. One is the list of the beginning of each row, the other the list of the column numbers and the values, followed by the memory location of the next element of the

row. By changing the memory location of the next nonzero value, an addition or deletion to the list can be made. Quite often the programmer needs not only the next element of a row but also the next element of the column. With the addition of an array pointing to the start of each column and two addresses after each value, the matrix could be stepped through by rows or columns. The additional address would point to the next element of the column.

Linear programs are created by a special compiler, which unwinds the "DO LOOPS" normally used in matrix operations to a linear string of instructions. With the linear code, the coefficients are treated as single variables rather than as matrix elements. Unwinding the "DO LOOPS" requires a great amount of code, usually much more than is saved by the other sparsity programming methods. The advantage is that only a small amount of this code is required in memory at one time. The reduction of core required results from moving sections of the code in and out of main memory as the sections are needed. Since most modern computers are capable of moving these sections of code without interrupting the execution stream of the computer, an overall savings in execution time is achieved.

The last method to be discussed is similar to the linked list, except that the location of the next nonzero element is always the next memory location. By always using the next memory location, the address of the next nonzero element can be omitted. This saves approximately one third of the linked list memory requirements. However, it requires knowledge of all the nonzero elements before the computation begins. Additions or deletions to the list require a complete relisting of the

nonzero elements. This is still less work than is required for the linear code method.

More general programming techniques which might be included under sparsity programming are the ordering methods discussed in section II.C. By ordering the coefficients to block diagonal or bordered block diagonal form, several smaller more dense matrices may be stored rather than the large sparse matrix. This allows the omission of a large percentage of the zeros without the other expenses associated with sparsity programming.

Quite often a combination of these techniques is used. For the programs of this thesis, the near block diagonal and the list methods will be combined. The diagonal blocks will remain as matrices, but the off diagonal elements will appear as single variables. It is expected that with the bordered block diagonal form, the submatrices will provide a sufficient reduction in the zeroes so that other sparsity methods will not be required.

For the multiprocessing environment, sparsity programming maps directly into the reduction in sharing of data. It would be extremely wasteful to access a shared variable only to multiply it by a zero coefficient. The sparsity which exists in the dynamic simulation problem, allows the equations to be ordered into the block forms and reduces the sharing of data between the processors executing these blocks to a small value,so that parallel processing is an efficient solution process.

CHAPTER III

ANALYSIS OF PARALLEL ALGORITHMS

In section II.D. it was shown that the algorithms required for the dynamic simulation problem may be stated in a form suitable for parallel execution. It is still questionable whether it is practicable to execute these algorithms in their parallel form. This chapter analyzes the algorithms in detail to determine the requirements for and advantages of parallel execution of the dynamic simulation problem.

In this chapter, the algorithms for the simulation of dynamic systems will be shown to generate far greater savings in execution time than costs from complexity. The major costs to be examined are the requirements for sharing data and the control required for parallel execution. The advantage is an increase in execution speed.

Parallel execution of algorithms requires the exchange of data between processors. The dynamic simulation problem can be arranged to a form with a high degree of parallelism because only a small amount of data must be shared. Costly additional hardware is required for each piece of data which must be shared concurrently. If only one path is provided for sharing data, then quite often a processor must sit idle waiting its turn to use this path. With the dynamic simulation problem one path is all that is required for a high order multiprocessor.

Traditionally, multiprocessors have taken advantage of the ability of multiple banks of memory to provide data faster than the processor could use it. In this way several processors can share the same memory space. Delays for sharing memory result only when two or more processors require a memory

location in the same bank. Sometimes when more than one processor requires the same memory location, it is because the processors are executing the same instructions, and the memory location contained the next instruction. This contention over the instructions is easily solved by providing multiple copies of the frequently executed segments of code.

Although both processors and memories have increased in execution speeds, the cost of the increase in memory speed has been disproportionate. For most commercially available processors it is possible to buy memory faster than required. However, the cost of this memory is many times more expensive than memory which operates at the speed required by the processor. Since multiple copies of the programs are required, two levels of memory can be used to avoid the use of expensive high speed memories (faster than the cycle time of the processor). The local level of memory contains the information (program and data) required only by its associated processor. The second level provides for the sharing of data required by one of the other processors. By using shared memory for only the data which must be shared, a large number of processors can retrieve the needed data with little delay.

This simple look at the multiprocessor structure is required for the development of the algorithms. Chapter IV presents the details of and comparison of different structures.

The programs to be presented in this chapter are actually the parallel segments of the algorithms. Except where indicated each segment is executed by a different processor. Unlike the algorithms of Chapter II, these programs show the entire requirements for each processor, rather than just

the procedure within a single time step. The programs include the control points showing the number of times the processors must be synchronized. The actual methods of achieving synchronization will be left until Chapter IV. Synchronization forces data to be shared within sections of the code rather than throughout all of the operations. The parallel segments of code establish the rates with which data must be shared.

In this chapter the advancement of the time step upon convergence of the equations is included in the analysis.

The increase in execution speed of the parallel algorithms is based on the ratio of the longest parallel instruction stream compared to the single processor instruction stream. The delays which result from the parallel execution must be added to the execution time for the longest stream. The speed advantage will never be n-fold for n processors because of the delays resulting from sharing data and achieving synchronization. The algorithms are designed to achieve the highest possible increase in execution speed. The delays depend on the multiprocessor configuration and control structure, and as such are presented in Chapter IV.

The input of data and the output of results is not discussed. Even though this is an extremely important part of the solution process, it requires almost no execution time. There are many direct memory access devices which perform these functions without interrupting the execution of the processor. Another possibility is to assign I/O to the controller, which can retrieve the required outputs as they are exchanged between processors. Since a General Purpose Computer is considered available to compile the multiprocessor program, it could also handle the I/O conversion and plotting requirements.

With dynamic simulation, the sparsity of the problem being solved affects the rate at which data must be shared and the number of operations the programs must execute. When a specific problem is required to complete the analysis of data rates and execution time,one block of the Commonwealth Edison high voltage distribution system is used. By using an actual system, some assurance is gained that the results are "typical" of all systems. The analysis uses typical blocks from the decomposition algorithms of section II.C. A detailed description of the system and all the blocks of the decompositions are in Appendix A. No attempt is made to use any parallelism below the block level.

All of the data derived in this chapter are based on the number of operations each algorithm requires to advance the time step and the number required to perform one iteration within the time step. Since it is difficult to compare the convergence rates of the different algorithms the overall increase in execution speed possible with the different algorithms is based on the experimental results that the Newton iteration converges from five to seven times faster than the linear iterations. [58]

The analysis in this chapter is used in Chapter IV to propose actual computing structures and to predict the performance of each structure. Analysis of the algorithms shows only two classes of multiprocessor structures are required.

III.A. The Parallel Jacobi Algorithm

The updating process of the Jacobi Algorithm requires the evaluation of a function for each variable. After all functions are evaluated, the correction (solution of the function) is added to each variable. All

functions include those functions assigned to different processors. This is the first point in the Jacobi algorithm where the controller must exert its power to insure all processors are synchronized in the instruction stream. After evaluation of all the functions the correction to the variables must be applied and tested for convergence. If convergence is achieved by all processors for all the variables, then the controller causes the advancement to the next time step. If convergence is not achieved then the processors repeat the function evaluations based on the new values of the variables. Again the processors must be synchronized to insure this evaluation uses the new values of the variables. The three control points required for the proper execution of the Jacobi Algorithm divide the algorithm into three distinct solution steps; the evaluation of the functions, the correction of the variables, and the advance to the new time step.

The other difficulty of the parallel algorithms is sharing data. The Jacobi algorithm shares data in all three steps. First the evaluation of the functions requires the value of variables from outside the block of variables being updated by the processor. Next when the variables are being corrected, the block variables which are required by other processors, must be stored in the shared memory. (They are also stored in local memory since they are required at high frequency by the local processor.) And finally after the variables have converged, the time step is advanced and the new predicted value of the variables required by other processors will be stored in shared memory. The sharing of data is restricted by the control points. The control points also require information from the other processors.

The instructions for the parallel segments of the algorithm appears to be just a smaller problem using the Jacobi algorithm. The only visible difference is the control requirements. Basically for the parallel algorithm the number of operations required is one-n^{th} the total number, where n is the number of processors. The reduction in sharing by grouping the variables into the near block diagonal form prevents the exact division into equal parts. The fastest execution is achieved when the differences in block size are minimal. Except for the delays resulting from sharing variables and the delays from synchronizing processors the parallel Jacobi algorithm is n times as fast as the single processor algorithm.

Further analysis of the parallel algorithm requires a detailed list of the instructions necessary for execution of the algorithm. The instructions required for one of the parallel segments is given in Appendix B. Only the instructions for one generator are shown in the block because the differences between generator models is small. More generators would require duplication of that part of the instructions. This is not the most complicated model of a generator nor is it the simplest. It is the simplest possible model still representing all the different parts of a generating station. Since each block normally contains more than one generator, the number of operations required for the generator variables is multiplied by the number of generators. The network equations are regular so they can be executed by matrix operations. Sparsity is considered more for reducing execution time than the reduction of storage, but some storage is saved. The actual location of the use of shared variables requires knowledge of the block of variables. One typical

block from the Commonwealth Edison system is shown in Table III.A.1. The actual values of the coefficients are only useful in predicting the rate of convergence of this model compared to another model. They provide no information on the convergence of this algorithm compared to another algorithm. The symmetry of the network equations allows the delineation of the block variables which are required by other processors. Any equation which requires an external variable for its evaluation, defines a local variable which will be required by another processor.

From the instruction list and the table of sparsity the actual number of operations required for execution can be developed. Table III.A.2 summarizes the operations in general and for this specific block. The variables for the general case are:

NGEN	is the number of generators of the block.
NROW	is the number of network variables.
NNZRO	is the number of nonzero coefficients.
NVS	is the number of variables required from outside the block (number of variables shared).
NLS	is the number of local variables required outside the block.

All of these variables are easily found for any specific block of a model. More general variables are:

ITS	is the number of iterations for convergence within the time step.
INS	is the number of memory accesses for machine instructions.

	50	115	120	255	45	135	265	60	65	305	335	310	475	385	75	420	435	235	240	295	290	370	205	225	300	80	130	380	260	145	190
50	X			X																								X			
115		X									X																	X			
120			X	X		X																								X	X
225	X		X	X	X																X					X			X		
45				X	X	X																			X						
135			X		X	X		X																X							
265							X	X													X										
60						X	X	X	X																						
65								X	X					X	X			X													
305										X	X														X						
335		X								X	X	X			X																
310											X	X											X								
475													X	X																	
385									X				X	X	X																
75									X		X			X	X																
420																X				X											
435																	X			X											
235									X									X	X	X		X									
240																		X	X	X											
295																X	X	X	X	X	X										
290				X			X													X	X									X	
370																		X				X					X				

Typical near block diagonal block

Table III.A.1

Summary of Required Operations for the Jacobi Algorithm

For the Generator Equations

	Memory Accesses	Divisions	Multiplications	Additions
function evaluation	141	0	47	36
variable updating	143	0	39	39
advance time step	144	0	40	28

For Network Equations

function evaluation	18*NROW+ 12*NGEN+ 10*NNZRO	4*NGEN+ 2*NROW	8*NGROW+ 8*NGEN+ 4*NNZRO	7*NROW+ 8*NGEN+ 5*NNZRO
variable updating	18*NROW	0	4*NROW	5*NROW

For the Designated Block

Generator function evaluation	423	0	141	106
Network function evaluation	952	56	408	438
Generator variable updating	429	0	117	117
Network variable updating	396	0	88	110
Advance time step	432	0	120	84

Table III.A.2

From Table III.A.2 expressions for the usage rate of shared variables can be developed. Without considering the control points, the access rate of shared memory would be

$$\frac{ITS*[\ 2*NVS + 2*NLS\ ] + 2*NLGVR}{ITS*[\ 296*NGEN + 36*NROW + 10*NNZRO + INS\ ] + 144*NGEN + INS} \qquad \text{III.A.1}$$

This is the ratio of the shared variable accesses compared to the total number of memory accesses. The Jacobi algorithm requires synchronization at different points of the instruction stream. This means III.A.1 is incorrect and the ratios must be compared within the separate steps of the solution process. The first step is the evaluation of the functions. ~~The requirement for shared variables comes from the~~ function requiring external variables. The ratio of shared variables during this period is:

$$\frac{2*NVS}{153*NGEN + 20*NROW + 10*NNZRO + INS} \qquad \text{III.A.2}$$

The next step in the solution process is the variable updating period. Here the values obtained in the previous period are used to correct the variables. The access to shared variables comes from the updating of local variables required by external processors. The ratio of shared accesses during this period is

$$\frac{2*NLS}{143*NGEN + 20*NROW + INS} \qquad \text{III.A.3}$$

The final step of the instruction stream is the advance to the next time interval. This section of code is only entered once every ITS iterations. Delays in this section of code have much less effect on the overall increase in execution speed. The only shared variables in this section are the generator voltages that are required by external processors for function evaluation. The ratio of shared access during this final step of the iteration process is:

$$\frac{2*LGVR}{144*NGEN + INS} \qquad \text{III.A.4}$$

These ratios indicate the very low rate with which data must be shared between processors. The rate depends on the sequence in which the instructions are executed. If the acceleration coefficient were applied during the function evaluation, the ratios would be significantly different. The ratio for the function evaluation is

$$\frac{2*NVS}{263*NGEN + 32*NROW + 10*NNZRO + INS} \qquad \text{III.A.5}$$

and the ratio for the variable update is

$$\frac{2*NLS}{33*NGEN + 8*NROW + INS} \qquad \text{III.A.6}$$

This coding method is inefficient because the rates required for sharing

data between processors vary to such an extreme. The multiprocessors would have to be able to supply the data at the higher rate even though this rate is required only for a short period. In the coding of the algorithms the rate of sharing data will always be spread as evenly as possible throughout the program.

To estimate the actual rates required the values of the variables for the Commonwealth Edison System are as follows:

NGEN	=	3
NROW	=	22
NNZRO	=	52
NVS	=	12
NLS	=	9
LGVR	=	1

When the instructions are executed on the CDC6400, the number of machine instructions for each section of the program is as follows:

INS FCN EVAL	=	154*NGEN + 80*NROW + 8*NNZRO
INS VAR UPDT	=	124*NGEN + 8*NROW
INS TME ADV	=	130*NGEN

The most crucial section of the program for sharing data is the variable updating procedure. For the Commonwealth Edison System during this section approximately 1.3% of the accesses would be for data within the shared memory. The function evaluation requires the lowest access rate or 0.08% of the accesses to shared memory. However, if the code was not properly ordered and the acceleration coefficient were applied during function evaluation, the shared memory access rate would be 3.6%, or almost three times the rate of the proper code. This demonstrates the importance of careful coding of the parallel algorithms.

III.B. The Chaotic Relaxation Algorithm

The Chaotic Relaxation Algorithm was designed to be a parallel algorithm. It evaluates the same functions as the Jacobi Algorithm to compute an update for each variable. But, rather than waiting until all updates are computed before they are applied, the Chaotic Scheme corrects the values of the variables as soon as a new value is predicted. This means that the values used to compute the correction may use the estimated values from different iterations, In fact, it is an advantage that the Chaotic Algorithm uses the latest possible value without regard to which iteration produced that estimate. The correction procedure repeats until all of the variables converge.

The Chaotic Algorithm depends on the number of processors. If the number of processors is equal to the number of variables to be updated, then the Chaotic and Jacobi Algorithms are identical. At the other limit, if there is only one processor, the Chaotic Algorithm is identical to the Gauss-Seidel algorithm. For the number of equations in the power system model, it is infeasible to have a processor for each equation, and the execution rate is not increased with just one processor, so the equations are grouped into blocks with each block assigned to a processor. To minimize the sharing of variables between processors, the equations are ordered to the near block diagonal form. With the Chaotic Scheme's independence from iteration numbers, as soon as a processor completes updating the variables of the block assigned to it, it can immediately restart the updating procedure. There is no requirement to synchronize the processors. The deletion of the synchronization required in the Jacobi Algorithm results in the increase of the efficiency of the processors.

This approach is different from that of the designers of the algorithm. They intended the algorithm for a general purpose multiprocessor with totally shared memory. In the original form each processor would update all of the variables, as if it were the only processor. Actually, several processors would be updating the variables, each remaining approximately equidistantly separated in the list of variables. If the blocks of variables required equal computational time, these two methods of applying the Chaotic Relaxation Algorithm would be equivalent. With different computation times for each block, the convergence rate depends on the least frequently updated variables.

The instructions for the Jacobi and Chaotic schemes are identical, with only the order changed. However, the application of the update, as soon as it is computed, slightly reduces the total computation. The real savings in computation time comes from the elimination of the synchronization. The synchronization at the advance of the time step remains because of the requirement of convergence of all variables. Because of the updating during the evaluation, the access rate of shared memory is not restricted to the short interval of variable update as in the Jacobi. This results in a much lower rate of use of shared variables. The rearrangement of the instructions into the Chaotic Relaxation Algorithm is shown in the appendix. The same block variables (Table III.A.1) are used when sparsity is used for the prediction of access rates. A summary of the operations required for the execution of the algorithm is given in Table III.B.1.

Using the variables presented in the previous section the requirements for the access to shared memory for the instructions within a time step

can be expressed by the following ratio:

$$\frac{2*NVS + 2*NLS}{290*NGEN + 36*NROW + 10*NNZRO + INS} \qquad \text{III.B.1}$$

After convergence the advance of the time step is identical to the Jacobi Algorithm and results in the following ratio of access rates:

$$\frac{2*LGVR}{144*NGEN + INS} \qquad \text{III.A.4}$$

The combination of the two sections of code from the Jacobi algorithm results in reducing the shared memory access rate to 0.8% for the Commonwealth Edison System. This is lower than the rate required for the time step advance, but so seldom is this code executed that the smaller value should be used.

From the shared memory usage rates established in these two sections, and from the convergence rates shown in the previous chapter, the conclusion is drawn that the Chaotic Scheme is preferable to the Jacobi method for multiprocessors. Since the requirements of these two algorithms on the multiprocessor structure are identical only the Chaotic Relaxation Algorithm is studied further.

The execution speed of the Chaotic Algorithm is important for comparison with the other algorithms. Comparison is much more difficult with the other algorithms, than with the Jacobi algorithms because of the different requirements the algorithms make on the multiprocessor, the differ-

Operations Required for Chaotic Relaxation

	INS Accesses	DATA Accesses	Divisions	Multi.	Additions
Generator Variable Updt	310*NGEN	278*NGEN	0	85*NGEN	75*NGEN
Network Variable Updt	55*NROW+ 8*NNZRO 25*NGEN	34*GROW+ 12*NGEN+ 10*NNZRO	4*NGEN+ 2*NROW	16*NROW+ 8*NGEN+ 4*NNZRO	16*NROW+ 8*NGEN+ 4*NNZRO
Advance Time Step	130*NGEN	144*NGEN	0	48*NGEN	34*NGEN

Table III.B.1

ences in the regions of convergence. By combining rate of convergence and execution speed, general comparisons are possible. However, only experience with a wide variety of problems will be conclusive.

The measure of execution speed is based on the number of memory accesses required for one iteration of the algorithm. This value can easily be obtained from the information contained in Table III.B.1. The Chaotic Algorithm requires the following number of memory references:

$$625*NGEN + 89*NROW + 18*NNZRO \qquad \text{III.B.2}$$

For the Commonwealth Edison System block of Table III.A.1. approximately 5800 memory references are required for one iteration.

III.C The Bordered Block Diagonal Gauss-Seidel Algorithm

The Gauss Seidel Algorithm is a linear algorithm like the Chaotic and Jacobi Algorithms. It uses the same functions to compute the updates for each variable. The algorithm differs from the Jacobi in that the updates are applied as soon as computed. Unlike Chaotic Relaxation, the Gauss-Seidel Algorithm updates the variables in strict sequential order. The immediate updating of the variables is not difficult with parallel processing. However, the strict sequential order with which the update is computed requires a high degree of control to implement. The difficulty of sequential updating can be avoided by ordering the variables to the bordered block diagonal form. With this form there is parallelism even in the sequential updating. After the cut-set variables are updated, the other blocks can be updated in parallel without destroying the sequential

order. This results from the lack of dependence on the variables in a block on the variables of another block other than the cut-set. By simply updating the variables of a block sequentially, the entire list of variables is updated sequentially. This divides the algorithm into three parts: the updating of the cut-set variables, the updating of the other variables, and the advance of the time step.

An unexpected advantage of the bordered block diagonal form is the method by which this form shares data. The only requirement for the sharing of data is between the cut-set processor and another processor. Only two processors need share a piece of data. Of course the cut-set processor must share data with each other processor. Each processor must provide different information to the cut-set processor, but it provides exactly the same information to each other processor. By providing a separate shared memory between each processor and the cut-set processor, only two processors will have to share the same memory. To the cut-set processor, part of this shared memory could appear as a single shared memory. The cut-set processor could store its variables for all processors with a single instruction per variable value. With only two processors sharing a memory, the contention over that memory can be made inconsequential.

The lack of parallelism in the correction of the cut-set variables can be overcome by several methods. The simplest is for each processor to compute its portion of the update. The last block row depends only on the block variables, and could be solved by the block processor by sharing the portion of the update rather than the variables required to compute that portion. The cut-set processor would have to sum these partial

corrections with the correction for its block. This still leaves the other processors idle for the final steps of the cut-set variable correction, and the cut-set processor idle while the other variables are being corrected. This implementation provides some parallelism with only a small degree of control. The control synchronizes the processors after each stage of the program to insure completion of that section of the program. The processors require synchronization after the separate portions of the correction to the cut-set variables, and after complete correction to the cut-set variables. Synchronization is always required for the advancement of the time step.

By increasing the control, the idle time of the processors can be lowered. First, assume the representation of numbers by the processors allows a symbolic representation of a number which would not be encountered in normal computation, say minus infinity. To start the algorithm, fill shared memory with this number. Then, allow the processors to begin to compute their portion of the cut-set correction. After a correction is computed it replaces the minus infinity stored in that location. Thus the cut-set processor can also begin to compute the correction. If when accessing shared memory for a portion of the update, a minus infinity is encountered,the cut-set processor would wait for the other processor to compute the correction. The added testing occurs during time the processor would be idle. After the cut-set processor successfully reads a portion of the update, it refills that location with minus infinity for the next iteration. Likewise, after the other processors have computed their portions of the update, they can begin to correct their variables. The cut-set processor will update its variables and store them in shared

memory, replacing the minus infinity stored there. If another processor finds a cut-set variable equal to minus infinity it will know that the cut-set processor has not yet corrected that variable. The processor then waits until the new value for that variable is stored. The last time a processor uses a variable, it replaces that variable with minus infinity. The major idle periods have been removed, and the parallelism has been extended at slight cost. The cut-set processor is still idle for a large part of each iteration but it does not delay the other processors as greatly with this form. If the other processors had generator and/or load models included in their variables, but the cut-set did not, then the other processors would seldom have to wait for cut-set variables. The generator model solution time would more than cover the cut-set variables updating.

Another possibility, if the simulation includes extensive generator and load modeling, is to have the cut-set processor update the cut-set variables with no help, while the other processors update the generator and load equations. (This prevents the cut-set variables from including these variables, but this does not restrict the cut-set choice.) The other processors would then share the value of their variables rather than the portion of the cut-set update. With the same use of minus infinity, the correct value of an iteration would be insured. To make this scheduling feasible, the number of computations for the generator equations must cover the computation of the cut-set update. This requires, on the average, at least three generator equations for every cut-set variable. Again the cut-set processor would be idle for part of each iteration, but this is the fastest method per iteration when the generator equations cover the cut-set equations.

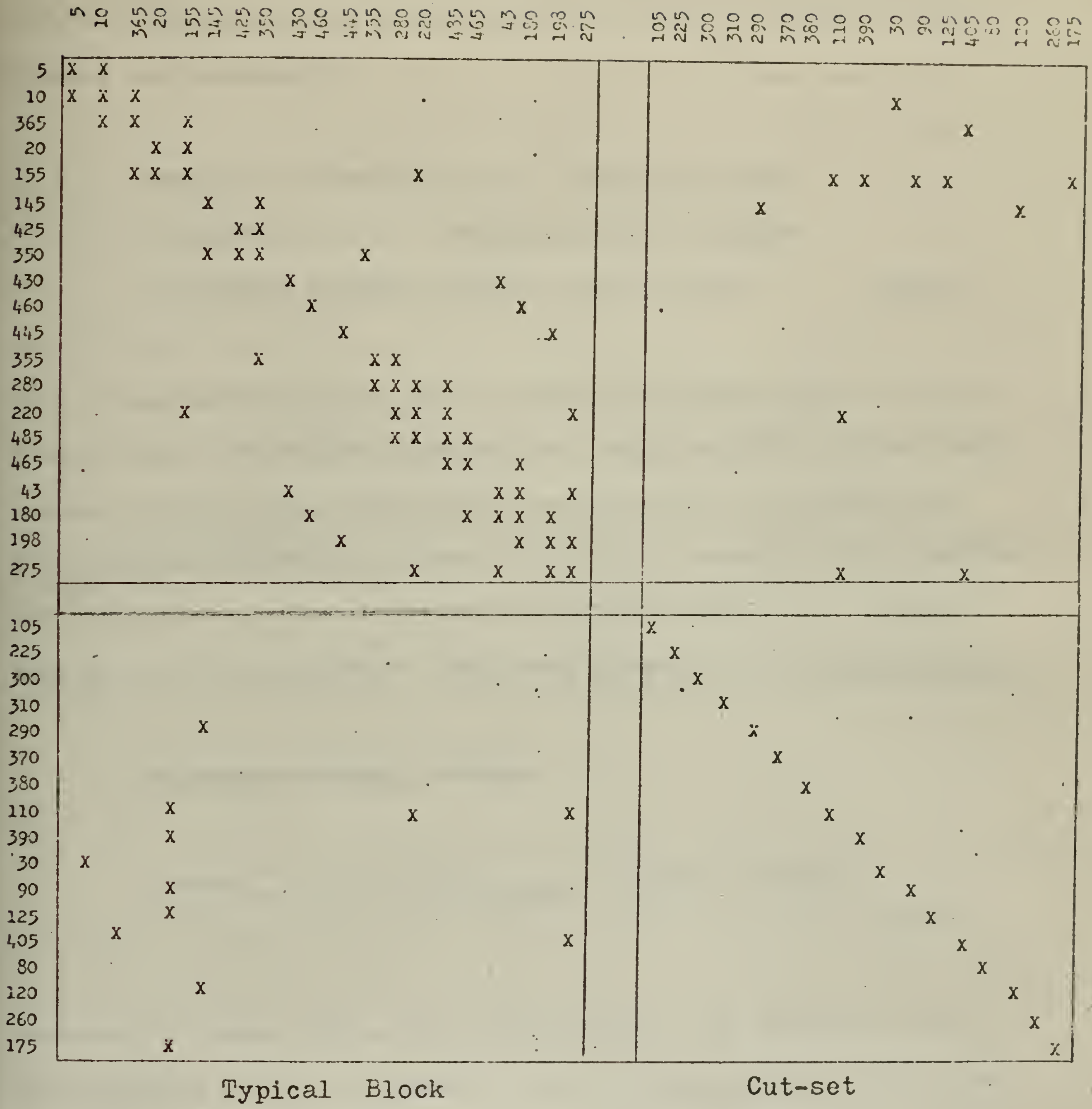
The same basic instruction sequence is used for each implementation. Appendix B contains a listing of this code. For the first implementation, synchronization would occur after the first block of network equations, and again after the second block. For the other implementations these synchronizations do not occur. For the third implementation the cut-set processor executes all the instructions required to correct the cut-set variables. This eliminates the loop of code which computes the partial updates. Further the loop of code for correcting the cut-set variables is identical to the code required by the other processors for their network equations. The number of operations for each of these sections of code is given in Table III.C.1. From this table the summaries for all the implementations of this algorithm can be derived.

Since the sharing of data is not a restrictive problem with the bordered block diagonal form, the critical information is the speed of execution. The Gauss-Seidel Algorithm is known to converge faster than the Chaotic Relaxation Algorithm. Comparison of the execution speeds of the two algorithms is difficult because of the differences in the requirements made on the multiprocessor structure by each algorithm. A typical bordered block diagonal decomposition of the Commonwealth Edison High Voltage Distribution System is given in Table III.C.2. As with the near block diagonal decomposition of Table III.A.2, this decomposition is for a five processor structure. (For the near block diagonal form there are five diagonal blocks; for the bordered block diagonal there are four diagonal blocks plus the cut-set.)

One method of measuring speed of execution is by counting the number of operations required for one iteration of the update of the variables.

	Inst. Memory Access	Data Memory Access	Divisions	Multiplications	Additions
Generator Variable update	310*NGEN	278*NGEN		85*NGEN	75*NGEN
Partial cut-set update	25*PROW + 8*NNZRO	5*PROW+ 10*NNZRO		4*NNZRO	1*PROW+ 4*NNZRO
Cut-set update with Partials	56*NROW+ 16*NPAR+ 8*NNZRO	34*NROW+ 6*NPAR+ 10*NNZRO		16*NROW+ 4*NNZRO	16*NROW+ 2*NPAR+ 4*NNZRO
Cut-set update with no partials	49*NROW+ 8*NNZRO	34*NROW+ 10*NNZRO	2*NROW	16*NROW+ 4*NNZRO	16*NROW+ 4*NNZRO
Block Network Variable update	49*NROW+ 25*NGEN+ 8*NNZRO	34*NROW+ 12*NGEN+ 10*NNZRO	2*NROW	16*NROW+ 8*NGEN+ 4*NNZRO	16*NROW+ 8*NGEN+ 4*NNZRO
Advance time Step	130*NGEN	144*NGEN		48*NGEN	34*NGEN

Table III.C.1



Bordered Block Diagonal Form of Commonwealth Edison System

Figure III.C.2

From Table III.C.2 this is an easy task. For the first implementation with the code synchronized at the network equations, the number of memory accesses per iteration is

$$\begin{aligned}&625*NGEN + 30*NROW(Partials) + 18*NNZRO(Partials)\\&+ 90*NROW(cut\text{-}set) + 18*NNZRO(cut\text{-}set) + 22*NPARS\\&+ 83*NROW + 18*NNZRO + \text{Synchronization delays}\end{aligned} \qquad \text{III.C.1}$$

This is significantly larger than the number of accesses required for the Chaotic Scheme. The other implementations reduce the total required memory accesses, however, the delays become less predictable. The third implementation requires the fewest memory accesses per iteration, but it requires the computation time for the generator variables to cover the computation time of the cut-set variables. The number of memory accesses required is

$$625*NGEN + 83*NROW + 18*NNZRO$$

or

$$83*NROW_{cut\text{-}set} + 18*NNZRO_{cut\text{-}set} + 83*NROW + 18*NNZRO \qquad \text{III.C.2}$$

whichever is greater. This third implementation, with the cut-set processor updating the cut-set variables while the other processors update the generator variables, requires approximately the same number of memory references for the program parts as does the Chaotic Algorithm. One difference is that the generator and load models for the Gauss-Seidel Algorithm are

divided among one fewer processor than they are for the Chaotic Algorithm. For the Commonwealth Edison System the different algorithms compare as follows. The Chaotic Algorithm requires approximately 5800 memory references and the first implementation of the Gauss Seidel Algorithm requires approximately 7000 memory accesses, but the third implementation requires only 5200 memory references. For the third implementation the four generator models cover the updating of the cut-set variables. Since a completely different multi-processor structure is required for these algorithms, this small speed advantage is not significant. Both algorithms are carried on into the next chapter for the design of multi-processing structures.

III.D. The Newton-SOR Algorithm

The Newton algorithms are very different than the linear algorithms discussed in the previous sections. The Newton Algorithms develop a set of equations, the solution of which yields the new estimate of the variables of the dynamic simulation problem. The development of the equations and their solution requires many times the computation effort of the linear algorithms. However, this additional computation results in quadratic rather than linear convergence, and normally produces convergence when the linear methods fail. The additional computation divides the Newton Algorithms into four steps.

The first step of the Newton methods is the solution of the function defining the variables. This function is similar to the functions used to predict the new estimates of the linear methods. The second step is the formation of the Jacobian. The Jacobian consists of the partial derivatives of these functions with respect to each variable, and is the coefficient

of the set of equations. Quite often the solution of the function is found at the same time the Jacobian is formed. Both of these sections can require the value of any variable; sharing may be required. The Jacobian and the solution of the functions define a linear set of equations. The third step of the algorithm is the solution of these equations to provide the new estimate of the variables. Since the equations are linear, they can be solved directly. It is the direct solution of these equations which requires the largest amount of computation, and is the most difficult section to arrange in parallel form. For the Newton-SOR Algorithm a solution to the Jacobian equations is found by direct solution of the diagonal blocks and iteration of the off-diagonal elements. Since the Jacobian exhibits the same sparsity pattern as the functions of section one, the sparsity from the Chaotic Algirthm carries over to this section. The difficulty of the Newton-SOR Algorithm is the iteration of the off-diagonal elements, each of which requires variables from outside the block diagonal. By using the near block diagonal form of the equations, the number of off-diagonal elements is minimized. A processor can be assigned to each block for parallel execution, by sharing the values of the variables of the off-diagonal elements. This requires a high rate of sharing data between processors. The fourth step of the algorithm is the advancement to the next time interval. It is almost identical to this step in the previous algorithms.

As with the other algorithms the variables are of two types, state variables and algebraic variables. The functions of the first part differ for the two variables types. For the state variables, the functions represent the change in the derivative for this time step from the previous iteration to the present iteration. The solution has been found when the

derivative does not change from one iteration to the next. Calculation of the derivative is identical to the linear algorithm method. The code for this section, given in Appendix B, appears different because of the use of matrix representation of all of the variables, and the calculation of the Jacobian interspersed with the function evaluation. The functions for the algebraic variables are different from the linear algorithms. The functions are in the original form given, rather than the fixed point problem. The value of the function defines the error of the variables from a functional value of zero. For the power utilities simulations, the algebraic variables are further divided into current and voltage. The functions for the current variables define the difference between the current of the generator or load and the current of the node connecting to that device. The voltage functions define the imbalance between the nodes. In the previous algorithms the current equations are combined into the nodal equations for the loads. The voltage equations are similar to the equations of the previous algorithms.

The evaluation of the functions often includes calculation of the elements of the Jacobian. This can be used to reduce the total computation by interspersing the solutions. The number of iterations required for convergence, may not increase significantly if the Jacobian is not recomputed at every iteration. Usually the Jacobian is recomputed and resolved only if the solution requires more than a few iterations, or if the step size or some other major variable is changed. (Such as the change due to the occurrence of a fault.) Because of the large amount of computation required to solve the Jacobian equations, the added iterations may actually result in a reduction in computation time. After the Jacobian equations are solved once, the next solution can be ob-

tained in only a few operations.

For the Newton-SOR Algorithm the solution of the Jacobian equations is not completely direct. Symbolically the inverse of each block of the Jacobian multiplies its block row and function. This process is totally parallel, requiring no information from another processor. If the Jacobian is not going to be recalculated after every iteration then the inverse must be saved for multiplication by the next functions. Computationally this procedure is inefficient. A matrix can be inverted and multiplied times another matrix in the same number of steps as it can be inverted. Computationally the block diagonal is decomposed into an upper and lower diagonal matrix. The new values for the off diagonal elements and the functions are found by solving the lower triangular equations then the upper triangular equations. The triangular form allows direct solution. The solution of these equations tends to fill in the off-diagonal columns. Columns of the Jacobian which were all zero remain such, but columns with one or more nonzero elements tend to be completely full after solution. This is a result of the fill in of the matrix inverse. The fill in does not increase the number of variables which must be shared between processors but it does require every variable to depend on the external variables. The iterations to correct for the off-diagonal elements require only the external variables. These external variables depend in return on the iterated values of other external variables, including some of the local variables. But only a small number of the local variables are required by other processors. Only these variables must be iterated to convergence. The variables not required by other processors are simply corrected after the convergence of all the shared variables.

To show the fill in and near block diagonal form,the block of the Commonwealth Edison System of Table III.A.1 is repeated, with one of the sets of generator variables. In this version, Table III.D.1, the fill in due to the LU Decomposition is denoted by an F, the original elements by X, and the fill in due to inversion by I. The off-diagonal elements need all the values for the corrections. This shows the optimal ordering of the generator and network variables.

In the actual computation it is easier to find the error in the previous estimate than the actual new estimate. For this reason the last part of the solution of the Jacobian equations is the addition of the correction to the old estimate of the variables. The use of the error allows convergence testing before solving the Jacobian equations.

The Newton-SOR Algorithm requires the sharing of data in three of the steps. Data must be shared for the evaluation of the functions and for the solution of the elements of the Jacobian. The other period requiring the sharing of data is during the third step of the solution process. The iteration of the corrections due to the off-diagonal elements requires the values of external variables. The later sharing rate is the critical rate. Control of the algorithm is required to insure that the evaluation of the functions and elements of the Jacobian use the new estimates of the variables. Convergence must be tested over all processors in two places. First, for the convergence of the variables, after the evaluation of the functions, and again for the convergence of the corrections due to the off-diagonal elements. All of the control points are shown in the program code.

The sparseness of the functions is still used for their evaluation. Since the generator and other models of the block diagonal are themselves

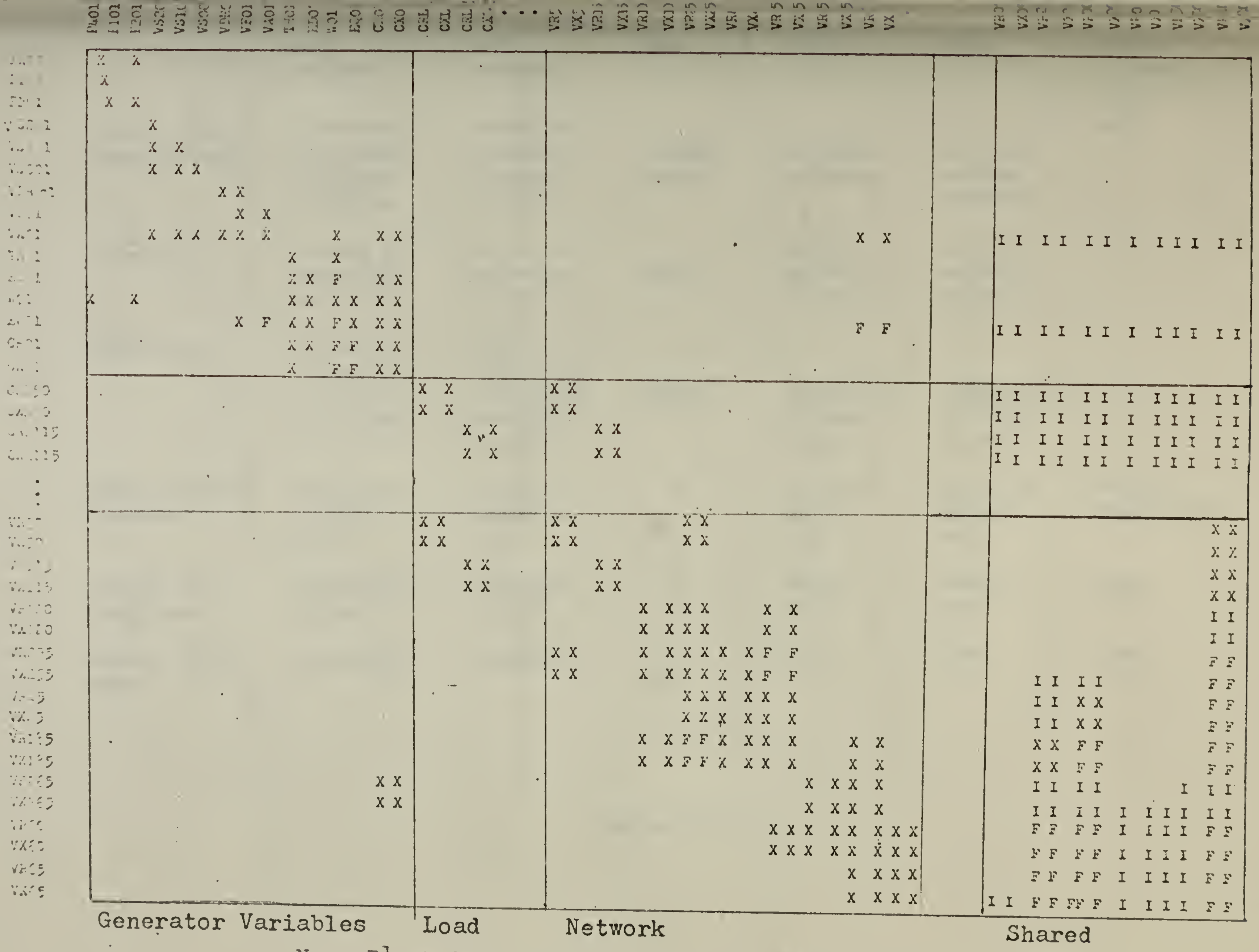
block diagonal, it is possible to reduce the computation of the decomposition by sparsity techniques. The entire block diagonal matrix is stored, but arrays indicate the last element of each column and row to reduce the computation. The off-diagonal values are compressed to successive columns.

To determine the rates required for sharing data and the increase in execution speed, the summary of Table III.D.2 is provided. The variables used in that table and in the equations to follow are defined below:

NVBLS	Number of variables in the block
LARU	Length of the average row beyond the diagonal
LACL	Length of the average column below the diagonal
NGEN	Number of generator models in block
NLOAD	Number of load models in block
NROW	Number of network nodes in block
NNZRO	Number of nonzero Y matrix entries in block
NEXV	Number of nonlocal variables used in model
NITS	Number of iterations used for convergence of inner loop
NLVR	Number of the local variables required by other processors
NSTV	Number of state variables in the block

From the summary of table III.D.2 the extremely high rate of sharing new values of the corrections during the iterations is shown. The ratio of shared memory accesses to all memory accesses is presented in III.D.1.

$$\frac{(\mathrm{NEXV} + 1)}{(54 + 9*\mathrm{NEXV})} \qquad \text{III.D.1}$$



Near Block Diagonal Form of Commonwealth Edison System

Figure III.D.1

	INST. Accesses	DATA Accesses	Div.	Multi.	Add.	Shared Accesses
Function and Jacobian Eval.	490*NGEN+ 71*NLOAD+ 20*NROW+ 21*NNZRO	348*NGEN+ 36*NLOAD+ 14*NROW+ 19*NNZRO	5*NGEN +NROW	139*NGEN +4*NNZRO	115*NGEN +3*NROW +4*NNZRO	NEXV
LU Decomp.	NVBLS*[14+ LACL*(28+ LARU*6)]	NVBLS*[14+ LACL*(8+ LARU*6)]	NVBLS	NVBLS*[LACL*(1+ 2*LARU)]	NVBLS*[LACL*(1+ 2*LARU)]	0
Back Subs.	6*NEXV+9+ (NVBLS-1)* [19+NEXV*(16+3*LARU) +3*LARU]	4*NEXV+6+ (NVBLS-1)* [10+NEXV*(4+6*LARU)+ 6*LARU]	NVBLS	(NVBLS-1)* [2+LARU+ NEXV]	(NVBLS-1)* (1+ NEXV *LARU)	0
ITERATE Shared Variables	NITS*[26+ 3*NEXV]* NLVR	NITS*[28+ 6*NEXV]* NLVR	0	NTIS*[3+ NEXV]* NLVR	NITS*[3+ NEXV]* NLVR	NITS*[1+ NEXV]*NLVR
Correct All Variables	NVBLS*[20+ 2*NEXV]	NVBLS*[8+ 6*NEXV]	0	NVBLS* NEXV	NVBLS* NEXV	NEXV
Advance Time Step	19+4*NSTV	10+6*NSTV	0	NSTV	NSTV	NLVR

Table III.D.2

This is a favorable ratio only for a small number of external variables. The ratio can be improved by sacrificing convergence speed. By using a Jacobi updating scheme rather than the Chaotic replacement, the external variables would only have to be accessed once a loop. This would lower the ratio to the value in III.D.2.

$$\frac{(NEXV + 1)}{(54 + 9*NEXV)*NLVR} \qquad \text{III.D.2}$$

For the Commonwealth Edison System of Table III.D.2, the Chaotic Replacement would require 7.5% of the memory accesses to be to shared memory. For the Jacobi replacement only 1% would be required. This would allow more processors to share the same shared memory. The execution time would not be changed, but the convergence rate would be slower.

For the Newton-SOR Algorithm the iterations due to the off-diagonal elements is almost the only time data that must be shared between processors. Shared data is required for the function and Jacobian evaluation and for the advance of the time step, but the rate during these periods is many times slower than that required during the iterations.

Also from the Table III.D.2, an estimate of the time required for one Newton iteration can be found. The number of iterations of the SOR loop is an unpredictable parameter which varies with problem and even with exact decomposition. The time in terms of memory accesses is shown in III.D.3.

$$\begin{aligned} TIME = {} & 838*NGEN + 107*NLOAD + 34*NROW + 40*NNZRO + \\ & NVBLS*[28 + LACL*(36 + LARU*12)] + 10*NEXV + \\ & 15 + (NVBLS - 1)*[29 + NEXV*(20 + LARU*9) + LARU*9] + \\ & NITS*[NLVR*(54 + NEXV*9)] + NVBLS*(28 + NEXV*8) \end{aligned} \qquad \text{III.D.3}$$

After the direct solution time is shown in the next section, a reasonable upper bound on the number of inner iterations required for convergence can be developed.

III.E The Bordered Block Diagonal Newton Method Algorithm

The bordered block diagonal Newton method follows the same four steps used by the Newton-SOR method. But as the bordered block diagonal Gauss-Seidel Algorithm differs from the Chaotic Algorithm, so the bordered block diagonal Newton Algorithm differs from the Newton-SOR Algorithm. The differences appear in the third step of the solution process, where the Jacobian equations are solved. By arranging the variables in the bordered block diagonal form, these equations can be solved directly, rather than iterated to convergence. The cost of the direct solution is increased computational requirements, idle time for processors, and more stringent sparsity requirements. As with the bordered block diagonal Gauss-Seidel Algorithm the unexpected benefit is lowered memory contention.

The bordered block diagonal Newton Method requires decoupling the variables into blocks and a cut-set. The network variables provide all of the interconnections and form the entire set of variables which must be examined for decomposition. Since all other processors are idle while the cut-set processor is solving for the new cut-set variables maximum efficiency is obtained by minimizing the number of cut-set variables. Thus even the load models connected to nodes of the network are not included in the cut-set variables. These loads are included in the other diagonal blocks.

The solution of the functions defining the error of the variables and

the elements of the Jacobian can require the values of the cut-set variables as well as the local variables. But none of the other processors' variables are required. The equations are identical to those required in these sections in the Newton-SOR Algorithm. The fourth section, advancement of the time step, is also identical to the Newton-SOR Algorithm.

The third section of the program, the solution of the Jacobian equations, is the only point where the programs differ. Since the Jacobian has the same sparsity as the functions, the Jacobian is in the bordered block diagonal form. To solve a set of equations the last chapter showed that the matrix of coefficients should be decomposed into the product of an upper and lower triangular matrix. The bordered block diagonal Jacobian is already very near this form because of the sparsity pattern. The diagonal blocks can be decomposed in parallel so they are each of this form, leaving only the block row of the cut-set variables. When the other processors eliminate these rows for the complete LU Decomposition, they modify the diagonal block of the cut-set variables. The cut-set processor must add in the modifications before this block can be decomposed. After decomposition is complete, the process of back substitution can begin. First, the back substitution of the cut-set variables is accomplished. Then the other processors can perform the back substitution for their variables in parallel. The completion of the back substitution yields the correction to each variable, which can be applied completely in parallel. After correction the process repeats.

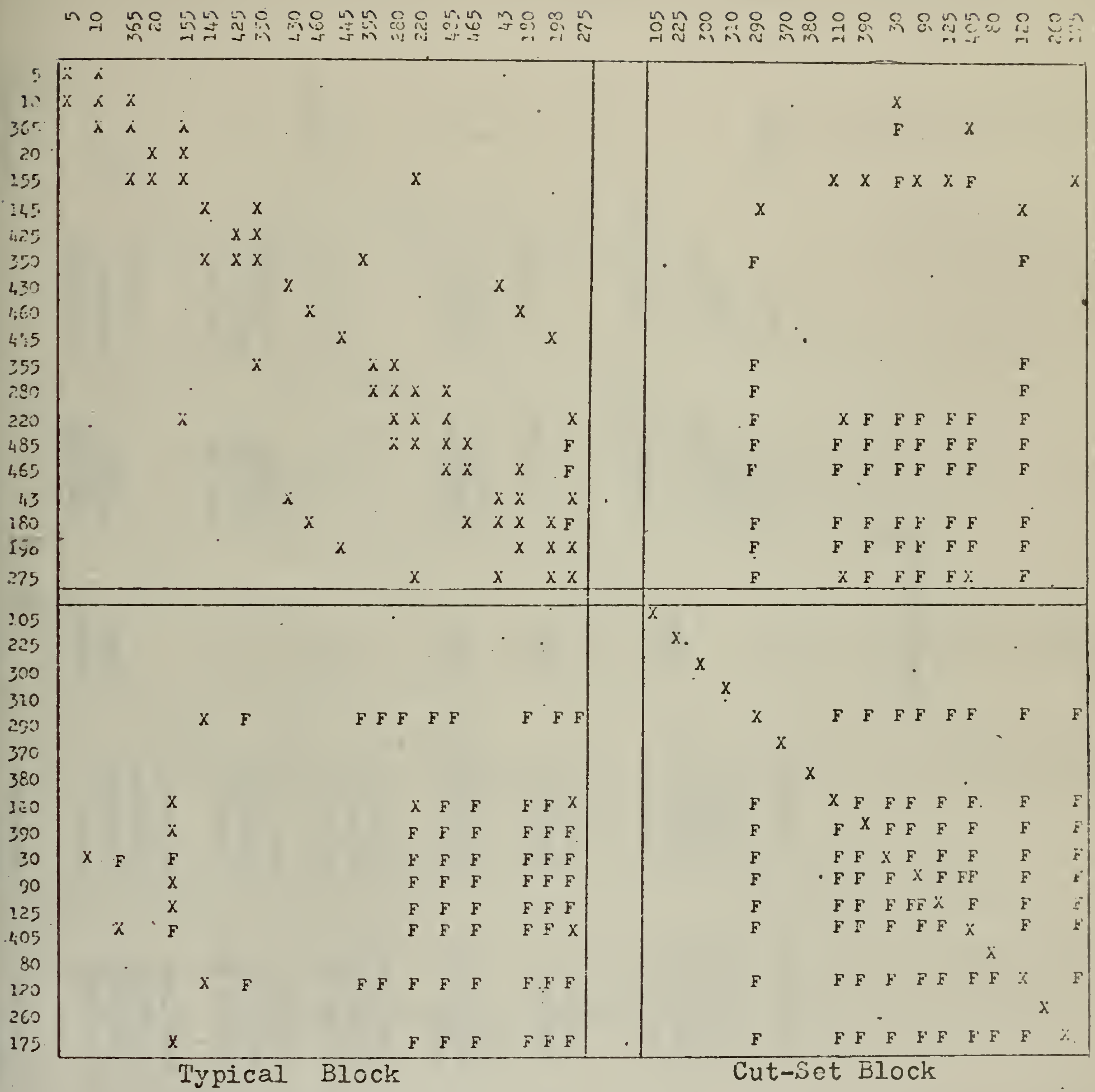
The bordered block diagonal Newton Algorithm requires every processor to be able to share data with the cut-set processor, and the cut-set processor to provide the cut-set variables to every processor. It requires the cut-set

processor to be idle while the other processors are decomposing the Jacobian and performing the back substitution for computation of the correction. These other processors are idle while the cut-set processor performs the same computations on the cut-set. The idle periods provide all of the synchronization required by this algorithm. The only other control function required is the determination of convergence across all processors. This could be included in the duties of the cut-set processor with little difficulty.

The instructions required to execute this algorithm are given in Appendix B. The instructions for the cut-set processor are set off from the other instructions in their proper sequence. The bordered block diagonal form of the variables was given in Table III.C.2, however it is repeated here in Table III.E.1, so that the fill-in which results from the LU Decomposition can be designated. The summary of the operations is given in Table III.E.2.

As with the bordered block diagonal Gauss-Seidel Algorithm the sharing of data is not difficult with the bordered block diagonal Newton Algorithm. Since only two processors are required to share any one piece of information, the data rates need never be as high as for the other algorithms. This leaves the time required for execution as the major point of analysis. This time is given in terms of memory accesses in III.E.1.

$$\begin{aligned}&838*NGEN + 107*NLOAD + 34*NROW + 40*NNZRO +\\ &NVBLS*[30 + LACL*(38 + 8*LARU)] + NP*(5*NCSV^2 + 3*NCSV)\\ &+ 4*NCSV^3/3 + 44*NCSV^2 + 79*NCSV - 19 + (NVBLS - 1)*\\ &[46 + 7*LARU + 7*NCSV] \qquad \text{III.E.1}\end{aligned}$$



Bordered Block Diagonal Form of
Commonwealth Edison System
Figure III.E.1

	INSTR. Accesses	DATA Accesses	DIV	MULTI	ADD	SHARED Accesses
Function and Jacobian eval.	490*NGEN+ 71*NLOAD+ 20*NROW+ 29*NNZRO	348*NGEN 36*NLOAD+ 14*NROW+ 19*NNZRO	5*NGEN +NROW	139*NGEN *4NNZRO	115*NGEN +3*NROW +4*NNZRO	NEXV
LU Block Decomposition	NVBLS*[15+ LACL*(30+ LARU*3)]	NVBLS*[15+ LACL*(8+ LARU*5)]		NVBLS*[LACL*(1+ LARU)]	NVBLS*[LACL*(1 +2*NARU)]	0
Cut-Set Sum- mation of ex- ternal mods	NCSV*[10+ NCSV*(16+ 2*NP)+2*NP]	NCSV*[4+ NCSV*(5+ 3*NP)+ NP	0	0	NP*NCSV^2	NP*NCSV^2
Decomposition of Cut-Set	NCSV^3+17* NCSV^2/2+21* NCSV/2	NCSV^3/3 + 19*NCSV^2/2+ 67*NCSV/2	NCSV	NCSV^3/3+ NCSV^2	NCSV^3/3 +NCSV^2	0
Back Subs. of Cut-Set	3*NCSV^2+ 21*NCSV- 21	2*NCSV^2+ 16*NCSV- 11	NCSV	-1/3NCSV	-5/6NCSV	
Back Subs of Blocks	3*NCSV+5+ (NVBLS-1)* [32+3*LARU +3*NCSV]	4*NCSV+8+ (NVBLS-1)* [14+4*LARU 4*NCSV]	NVBLS	(NVBLS-1)* [2*LARU* NEXV]	(NVBLS-1)* (1+NEXV* LARU]	
Advance Time Step	10+6*NSTV	19+4*NSTV	0	NSTV	NSTV	NSTV

Table III.E.2

Both the bordered block diagonal Newton and the Newton-SOR Algorithms converge at the same rate, thus to compare their solution rates only the time required for one Newton iteration need be compared. Using the Commonwealth Edison System the ratio of memory accesses yields the execution rates of III.E.2.

$$\frac{\text{NSOR}}{\text{BBDN}} = \frac{810*\text{NITS} + 46600 + \text{Jacobian eval}}{50000 + \text{Jacobian eval}} \qquad \text{III.E.2}$$

This suggests that 4 SOR iterations can be allowed before the Newton Algorithm will be faster. But the summation of the modifications to the cutset block can be accomplished during the modification process by using the software flag as in the bordered block diagonal Gauss-Seidel Algorithm. The ratio then reduces to the value of III.E.3.

$$\frac{\text{NSOR}}{\text{BBDN}} = \frac{810*\text{NITS} + 46600}{37000} \qquad \text{III.E.3}$$

This shows that the true Newton Algorithm should probably be used if the equations can be arranged into the bordered block diagonal form. The vast differences required in the multiprocessor structures which can execute these algorithms weakens the apparent superiority of the true Newton method.

III.F Comparison of the Algorithms

This chapter presented four algorithms which are suitable for parallel execution of the dynamic simulation problem. In the next chapter multiprocessor structures are presented that are capable of executing these

algorithms. In this section the algorithms are compared to show the similarities that exist among them. It is shown that only two different types of multiprocessing structures are required.

The most obvious similarity is that the Chaotic Relaxation and the Newton-SOR algorithms use the near block diagonal form of the equations, while the Gauss-Seidel and true Newton methods require the bordered block diagonal form. The form of the equations determines the type of data sharing that is required for the solution.

The near block diagonal form algorithms share a small portion of the variables among all processors. Each processor uses one or more of the shared variables during the calculation of the new estimate of the local variables. However, since only data is shared, no processor will try to obtain access to shared memory on the cycle immediately after receiving the value of a shared variable. The Chaotic and Newton-SOR algorithms require access to shared data at different rates. The memory contention resulting from the use of shared memory is also different for these two algorithms. The Chaotic Relaxation algorithm requires access to the shared variables at a much lower rate than the Newton-SOR algorithm. The Newton-SOR algorithm requires one additional control point, but any processor which can efficiently execute the Newton-SOR algorithms can efficiently execute the Chaotic Relaxation algorithm.

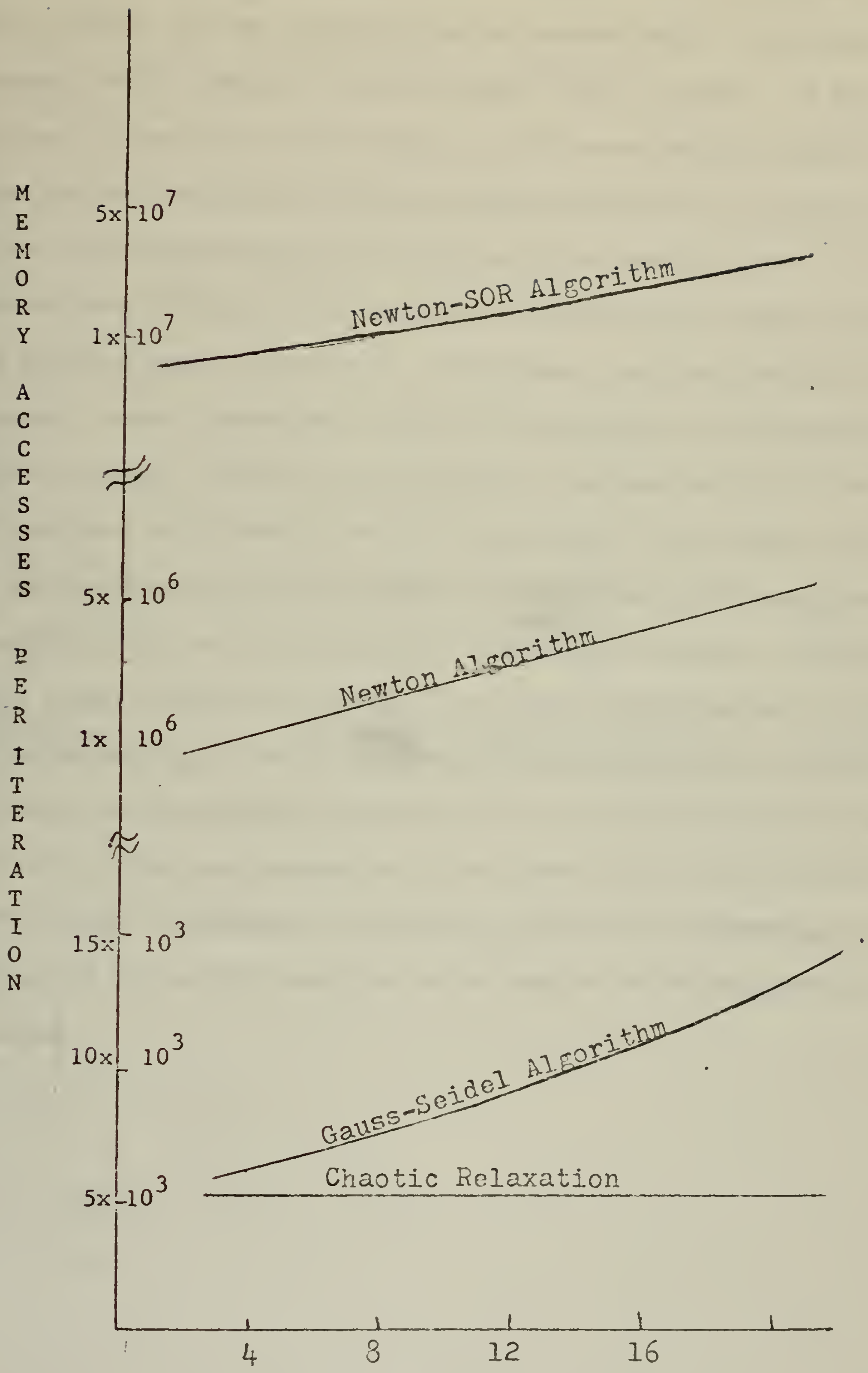
The bordered block diagonal algorithms share solution information in an entirely different manner. With these schemes only the cut-set processor requires information from the diagonal processors, and the cut-set processor must supply information to the other processors. Both the Gauss-Seidel and the true Newton algorithms require the cut-set processor

and the other processors to alternate execution periods. Since the processors alternate execution periods, memory contention does not cause delays. The control requirements for the two algorithms are identical.

Prediction of the delays due to parallel execution requires information about the multiprocessor structures. The convergence rates of the algorithms were compared in the last chapter. This chapter has presented the number of operations that are required to compute a new estimate. As was the case when comparing the convergence rates of the algorithms, there is difficulty in comparing the number of operations required to compute the new estimate. The delays encountered in parallel solution differ with the form of the equations. Table III.F.1,developed from the equations of this chapter, shows the number of operations the algorithms require for equations with 5% nonzero elements within the blocks. Stagg [59], suggested that 5 to 7 linear iterations are required for every Newton iteration allowing a rough comparison of the algorithms'solution time.

The number of operations do not include the delays which result from parallel execution, so the resulting comparison cannot be final. It does suggest that the linear methods should converge in less time than the Newton algorithms. This is contrary to the experience gained with the single processor implementation, where the Newton method solves the model in less time. The use of the linear iterative methods raises doubts about convergence since these methods are known not to converge for some sets of equations. When the delays are estimated in the next chapter, the comparison will be refined.

In spite of the analysis available, questions remained on the proper utilization of the SOR iterations with the Newton-SOR method. Solution



Number of Shared Variables

Comparison of Execution Rates

Figure III.F.1

time and sharing rate are dependent on the number of SOR iterations required. Further the SOR iterations can be stopped before convergence is reached, and the Newton iterations should still converge. To test the effect of variation in the number of SOR iterations the load flow problem for the Commonwealth Edison system of Appendix A was ordered to the near block diagonal form and solved by the Newton-SOR method. The SOR iterations were repeated until convergence, repeated seven times (approximately half the number required for convergence), and only computed once. The highest overall convergence rate was obtained with the convergence of the SOR iterations. Because of the computer time required, the slower algorithms were only iterated ten and fifteen times respectively. The seven SOR iteration algorithm converged at approximately the same rate as the converged SOR iteration algorithm. The single iteration algorithm was much slower converging requiring ten Newton iterations for every five of the other two algorithms when convergence was approached. The first few iterations were approximately the same. It is expected that more detailed study of the convergence properties of the Newton algorithms would find that the highest convergence rate would be achieved by increasing the accuracy required of the SOR iterations as the Newton iterations approached convergence.

CHAPTER IV

PROPOSED MULTIPROCESSOR STRUCTURES

Before the parallel algorithms were developed in Chapter Three, the rudiments of a computing structure were discussed. The computing structure consists of two levels of memory, multiple processors and a controller. One level of memory, the local level, is private to each processor. The other level of memory, shared memory, is connected to the processors by a common bus. The information required to be exchanged between processors is stored in this shared memory. The other processors can then read the information from shared memory. The difficulty is that only one processor per memory cycle can access a single shared memory. The other processors must wait until the shared memory is free to exchange information. In addition to exchanging data, the processors must pass information required to control the execution sequence. This information is much simpler, consisting of a signal from each processor designating which part of the execution sequence the processor is performing and the status of the iterations. A controller would assimilate these signals and provide other signals to each processor to alter the sequence of instructions. The most common signal would indicate convergence of an iteration, and would instruct the processors to advance the time step.

This simple description indicates all of the major components required for every computing structure. First, each of the components is explained. Then the components are assembled into different structures. The ability of each structure to execute the dynamic simulation problem is predicted. The structures are only capable of efficiently executing one of the two classes of algorithms, either the algorithms using the near block diagonal

ordering of the equations of the simulation, or the algorithms requiring the bordered block diagonal ordering.

After each structure is developed, the execution speed is predicted. The structures are modeled to estimate the delays which will result from contention over shared memory. The contention models used are derived from the multiprocessor model developed by Skinner and Asher [61]. The model yields a multipicative factor, called the stretching factor, which shows the amount by which the execution time is increased due to memory contention. This factor does not include the effects of the variation of the convergence rate between the parallel algorithms.

IV.A Multiprocessor Components

There are three parts to the computing structures considered here. The first part is the computing elements. The computing element consists of the processor and local memory, and any other devices required for an independent computer. The second component of the structure provides for the exchange of data. For the near block diagonal algorithms it consists of a shared memory, a common bus, and an arbiter. For the bordered block diagonal algorithms, the cut-set processor handles the sharing of data. The final component is the controller. It may be as simple as a few external logic devices, or as complicated as another processing element. In fact, the control functions may be assigned to one of the processing elements.

In this discussion, as few specifications as possible are made on the processing element. It is assumed to be a commercially available minicomputer, although micros or full GP computers might as easily fill the position. A few features will enhance the execution of the dynamic simulation problem.

The major requirement is the efficient execution of arithmetic operations on floating point numbers. The sparsity programming of the dynamic simulation problem can more easily be performed for a processor with some indexing capabilities. The actual instruction set and execution speeds do affect the final structure, but the analysis of this thesis is based on a high level language implementation of the algorithms. The local memory associated with each processor is of any form efficient for that processor. The only requirement is that it is of sufficient size to contain the entire parallel portion of the program and data required by that processor.

The ability of the structure to exchange information between processors determines the success or failure of the structure. If the information cannot be efficiently exchanged, the use of parallel processing is not efficient. For the near block diagonal form algorithms, the simplest device which is used for exchanging data is the single shared memory. All processors can gain access to this memory by a common bus. Only one processor may gain access to the bus (and shared memory) during one shared memory cycle. When more than one processor requests the bus, an arbiter grants access to the bus to only one processor. The bus, arbiter, and the shared memory appear as one unit. If a processor gains access to the bus, then the requested information is provided at the end of the memory cycle. The other processors must wait until the next memory cycle and repeat the request. Because of the structure of the algorithms, after a processor gains access to shared memory, that processor waits several memory cycles before there is another request for shared memory. To insure against unreasonable delays, the number of processors should be limited so that, on the average, no processor will make two requests for shared memory before every processor requesting

shared memory has had at least one opportunity for access.

The arbiter is a simple hardware device which insures only one processor is granted access to the bus during a shared memory cycle. Because of the number of memory cycles between a single processor's request for shared memory, any scheme used by the arbiter to grant access should result in a "fair" order of selection. A true random, or round robin system would be preferred. But even sequential polling, giving priority to the closest processor on every poll, would not significantly alter the delays. A true random arbiter was used for the models.

The cut-set processor accomplishes the sharing of data for the bordered block diagonal algorithms. This could be achieved by a common bus connecting the cut-set processor to the other processors. The cut-set processor needs access to the local memory of these other processors only while they are idle. This eliminates the need for separate shared memory and an arbiter.

If the portion of the local memory of the diagonal processors where the cut-set variables are stored appears as a single memory to the cut-set processor, the efficiency of the bordered block diagonal algorithms will be increased. In this case, the cut-set processor will be able to store a copy of each of its variables for every processor with only one store instruction per variable.

The final part of the computing structure is the controller. The controller insures that the execution process proceeds in the prescribed manner by all processors. Action is required by the controller at the control points within the algorithms. The controller achieves synchronization of the processors at these points of the programs, and signals changes in the program, such as moving to the time step advance. The controller is not

expected to be able to provide all of the control required for major program changes, such as beginning execution of a new simulation.

There are two possible devices to act as the controller. One, a totally external controller, uses logic hardware to provide control signals based on the signals provided by each processor. The other controller, one of the processing elements, would examine shared memory for the status of all of the other processors and, based on this information, provide signals to alter the execution sequence of the other processors. However, the use of a processor as a controller implies the control functions will require many memory cycles to implement since the processor would have to perform operations to determine the control required.

The simplest control scheme is achieved by each processor setting hardware flags with the information representing that processor's current status. External logic would then use the information from these flags to determine the need for altering the execution sequence. If a change is needed the external logic either could interrupt the processors, with the interrupt providing the information required to alter the instruction sequence, or set flags which, upon reading by the processor, would indicate the new instructions to execute. By using external logic for control there is less delay than would result from the software approach of exchanging semaphores through shared memory.

With the bordered block diagonal algorithms, the cut-set processor can perform the control functions without the delays which normally result from using a processor to control a general purpose multiprocessor, since the controlling functions occur when the cut-set processor is otherwise idle.

With all of the parts of the computing structures defined, the next sections show how they might be combined to execute the algorithms required for the solution of the dynamic simulation problem. The structures are divided into two groups: the structures for the near block diagonal algorithms, and the structures for the bordered block diagonal algorithms. First the model which predicts the actual execution times is described.

IV.B The Multiprocessor Model

The structures proposed must be evaluated to predict the delays that will result from sharing data, and the accompanying increase in execution time. The multiprocessor model developed by Skinner and Asher [63], is modified slightly to provide this prediction.

The Skinner and Asher model use the theory of Markov Chains to predict the stretching effect on execution time delays due to memory contention. A one step transition matrix is developed. The elements of the matrix are the probabilities of the processors being delayed due to the other processor's actions. Skinner and Asher assumed each processor attempted to access shared memory by Bernoulli trials with probability p. When more than one processor attempts access, access goes to one processor with probability Π. The processors failing to receive access to shared memory repeat the request on the next memory cycle. Skinner and Asher assumed that a processor could repeat the request even if it was granted access the previous cycle. When the probabilities for each processor are the same, the model can be simplified to represent the major states of the multiprocessor. For example, a major state might represent three processors delayed waiting for shared memory. This is essential for models the size required for this thesis.

The model of Skinner and Asher was modified primarily to reflect the fact that a processor, would not request shared memory immediately after receiving it. The modification involved splitting the <u>no processor delayed</u> state into a state of <u>no processors requesting shared memory</u> and a state of <u>one processor requesting shared memory</u>. With the state split,the probabilities can be modified to reflect the restrictions on the processors'repetition of requests to shared memory, and the actual usage rates of shared memory can be predicted. If the probabilities were not modified this model would provide the same values as Skinner and Asher's.

By solving the Markov Chain model (finding the steady state probabilities) the expected value of the occurrence of the different states can be predicted. These probabilities provide the information needed to predict the additional time that will be required to execute the problem because of memory contention. This information is expressed as the stretching factor and is computed as the inverse of the probability that a processor will not be delayed. The ratio of the multiprocessor total execution time to that of the single processor is the stretching factor divided by the number of processors.

IV.C <u>Structures for the Near Block Diagonal Algorithms</u>

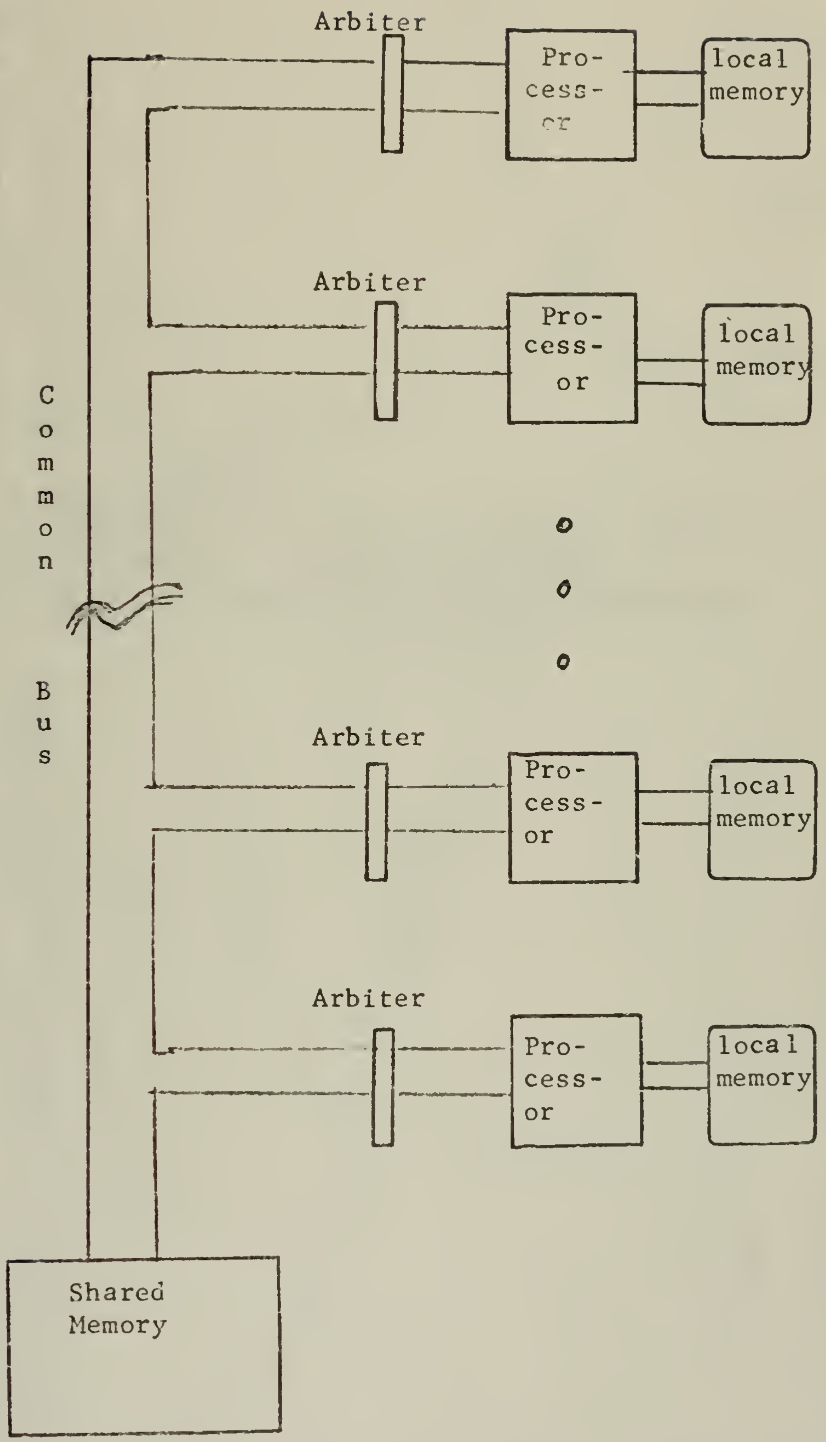
The Chaotic Relaxation and Newton-SOR Algorithms, are enhanced by ordering the equations of the simulation to a form with diagonal blocks and as few interconnections as possible. This near block diagonal form minimizes the sharing of data between processors, and improves the convergence rate of the Newton-SOR Algorithm. The near block diagonal form is

not essential for these algorithms. Likewise, the structures of this section are designed for near block diagonal equation solution, but they will execute the algorithms for nonsparse equations. The cost of non-sparse problems would be smaller problem size and increased delays due to memory contention.

The Chaotic Relaxation Algorithm requires the ability for all processors to share data. Analysis in Chapter Three showed each processor will try to access shared data not more than two percent of the memory cycles. Control is required only to insure all processors advance the time step when all variables have converged. The Newton-SOR Algorithm has the above requirements, with seven and one half percent of the memory cycles going to shared memory during the iterations, and in addition,requires control to insure convergence of all processors during the secondary iteration.

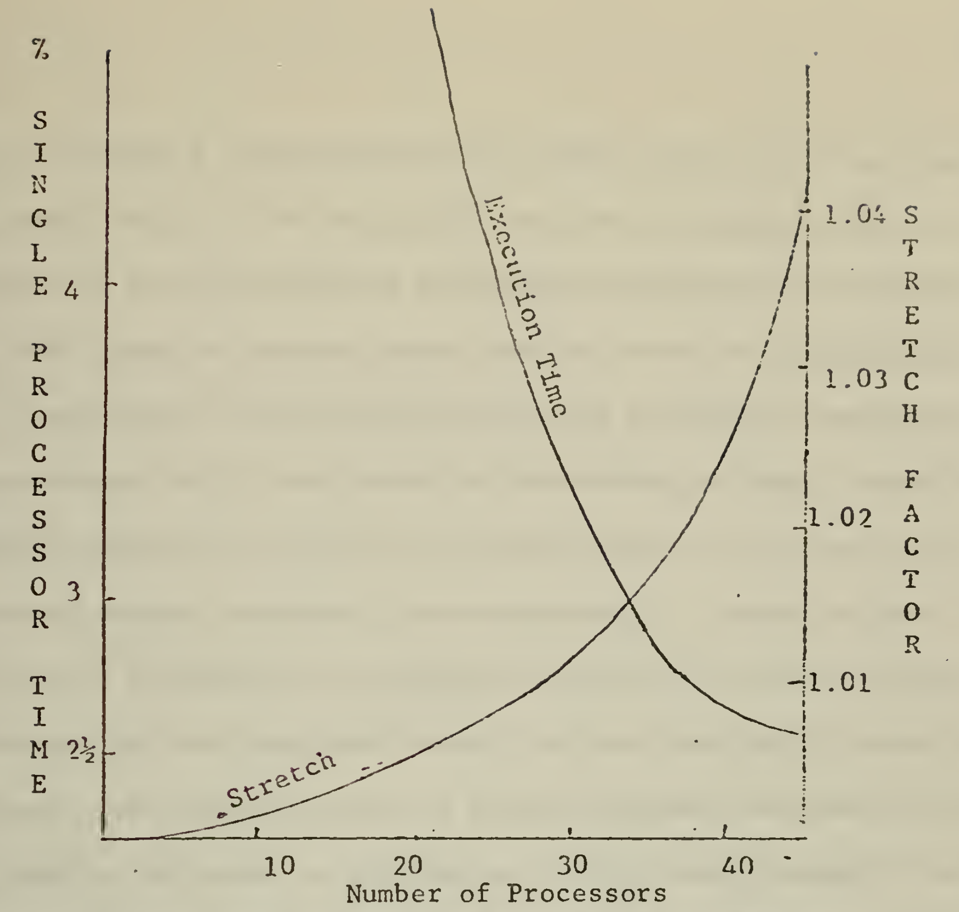
The simplest multiprocessor structure is the use of a common bus to connect a number of processing elements. Figure IV.C.1 shows one such structure. The common bus consists of address, data and, control lines. An arbiter would control access to the bus. The controller would consist of logic devices connected to each processor. This is the model assumed during the development of the Chaotic and Newton-SOR Algorithms. The arbiter grants access based on the next processor desiring access in the loop. Graph IV.C.1 shows the stretching effect as more processors are added to the structure for the Chaotic algorithm. It also shows the expected decrease in execution time. This is expressed as the percent of the time a single processor would require. The corresponding quantities are repeated for the Newton-SOR Algorithm in Graph IV.C.2.

From the first graph it is apparent that the delays from memory



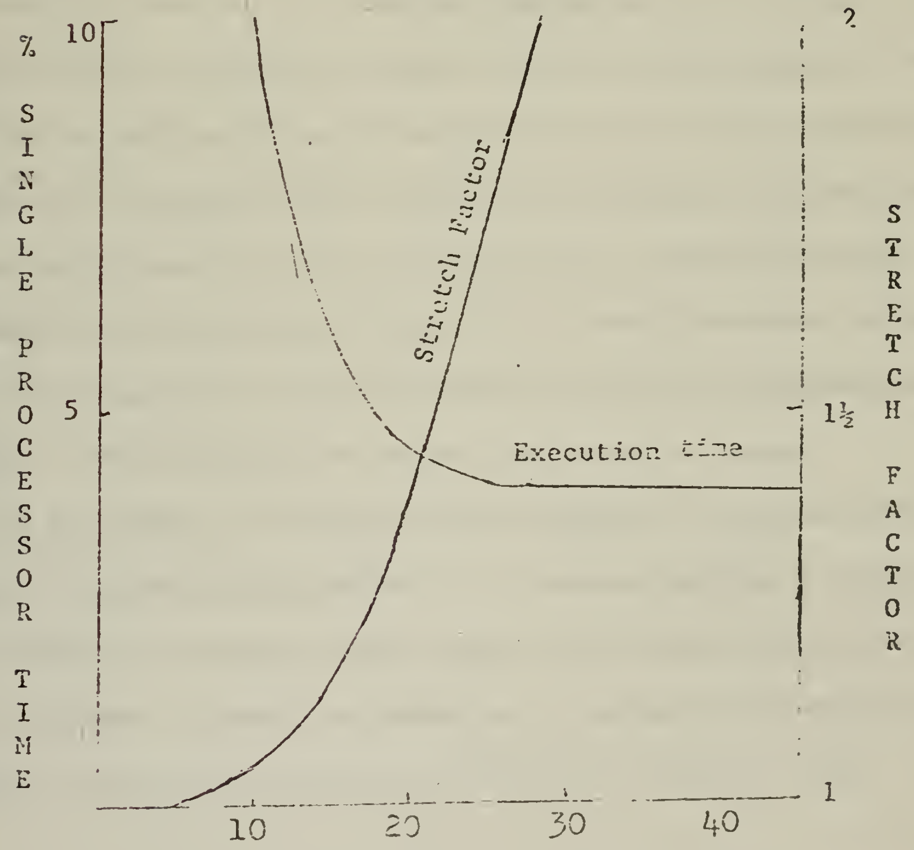
Near block diagonal form Multiprocessor (Single Shared Memory)

Figure IV.C.1



Number of Processors
Chaotic Relaxation with single shared memory

Graph IV.C.1



Number of Processors
Newton-SOR with single shared memory

Graph IV.C.2

contention do not have a significant effect until almost all of the memory cycles are used. After all of the cycles are used, a further increase in execution speed is not attainable by increasing the number of processors. Saturation (100% usage) of shared memory does not occur as expected when the number of processors is the reciprocal of the individual processors shared memory access rate. This number of processors and their expected problem solution time are called the idealized number of processors and idealized multiprocessor execution time respectively. Because of the stretching effect, the number of processors required to saturate shared memory is larger than the idealized number, and even with this larger number of processors the execution time is longer than the idealized execution time. For example, with Chaotic Relaxation, each processor could be expected to use shared memory for two percent of its total memory cycles. Fifty processors would be used in the idealized multiprocessor structure, achieving solution in 2% of the time that a single processor would require. However, because of the stretching effect of memory contention, fifty processors use shared memory for only 95% of the total possible access to shared memory, and solution would require 2.17% of the time that a single processor would require. Addition of approximately twenty five more processors would only reduce the time required for solution down to 2.1% of the single processor solution time. The idealized two percent can never be reached.

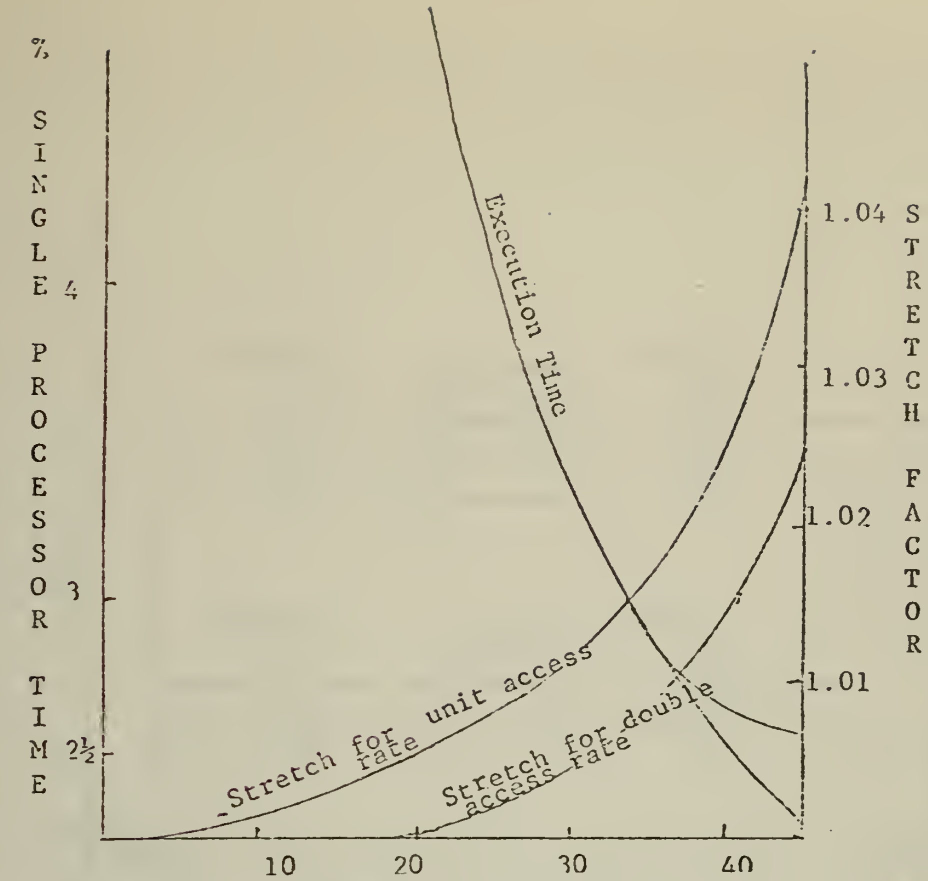
To chose the number of processors which should be included in the multiprocessor is a difficult problem. It requires setting a value on the cost of an additional processor and a value on the time saved by using an additional processor. Clearly the addition of processors beyond the number which saturates shared memory gains no decrease in execution time. Below

this number, every additional processor results in a smaller reduction in execution time. The graph of the solution time versus the number of processors becomes almost level when the number of processors reaches between two-thirds and four-fifths of the idealized number. For general simulation studies it is expected that this range includes the appropriate number of processors to include in the multiprocessor. It is difficult to foresee using more than this number of processors.

The easiest method to improve the performance of this structure is to increase the rate with which the shared memory can provide data. If the shared memory had an access rate twice that required by the processors, or if shared memory had multiple access ports then the data could be supplied to two processors every memory cycle. Graphs IV.C.3 and 4 present the results for this structure with even multiples of the shared memory access rate.

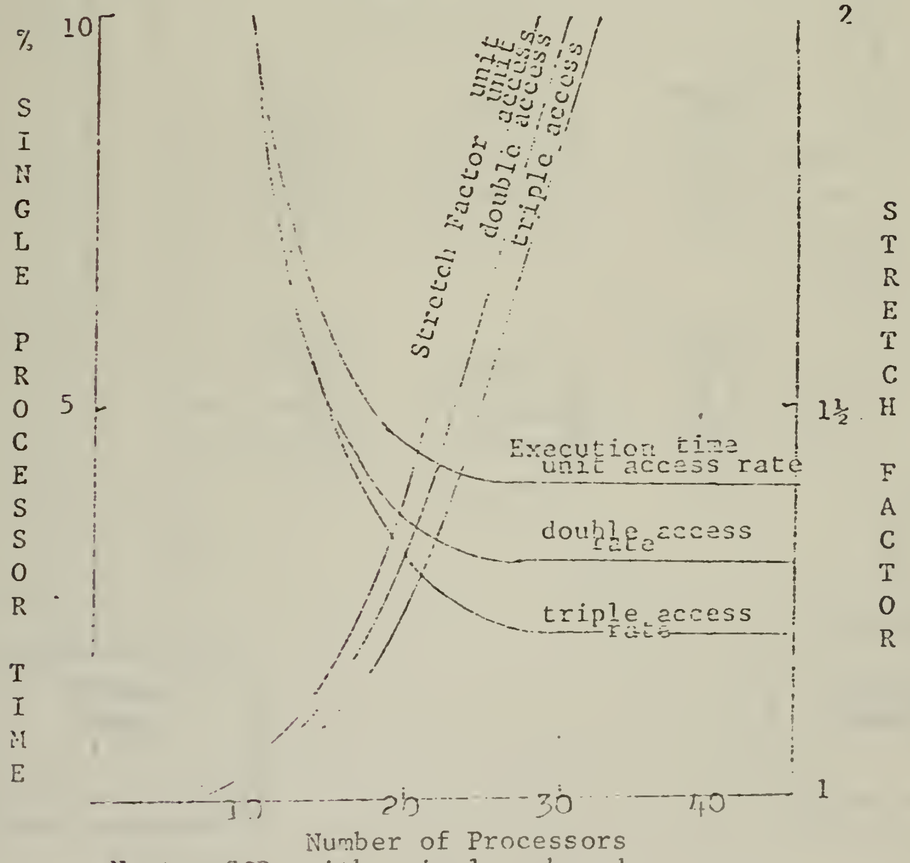
Similar results are obtained with a multiprocessor having multiple shared memories. Figure IV.C.2 shows a two memory version. Each processor is connected to two common buses. Each bus has its own arbiter and shared memory. With two shared memories each processor would access one of the shared memories roughly half as often. (Equal portions of the shared data is stored in each memory.) Each of the buses in the multiple shared memory version would be equivalent to the single bus of the first structure. Graphs IV.C.5 and 6 show the results for the multiple shared memories. (As the structures become more complicated the models of the structures must be simplified for solution.)

The multiple memories can also be further improved by increasing their access rate.



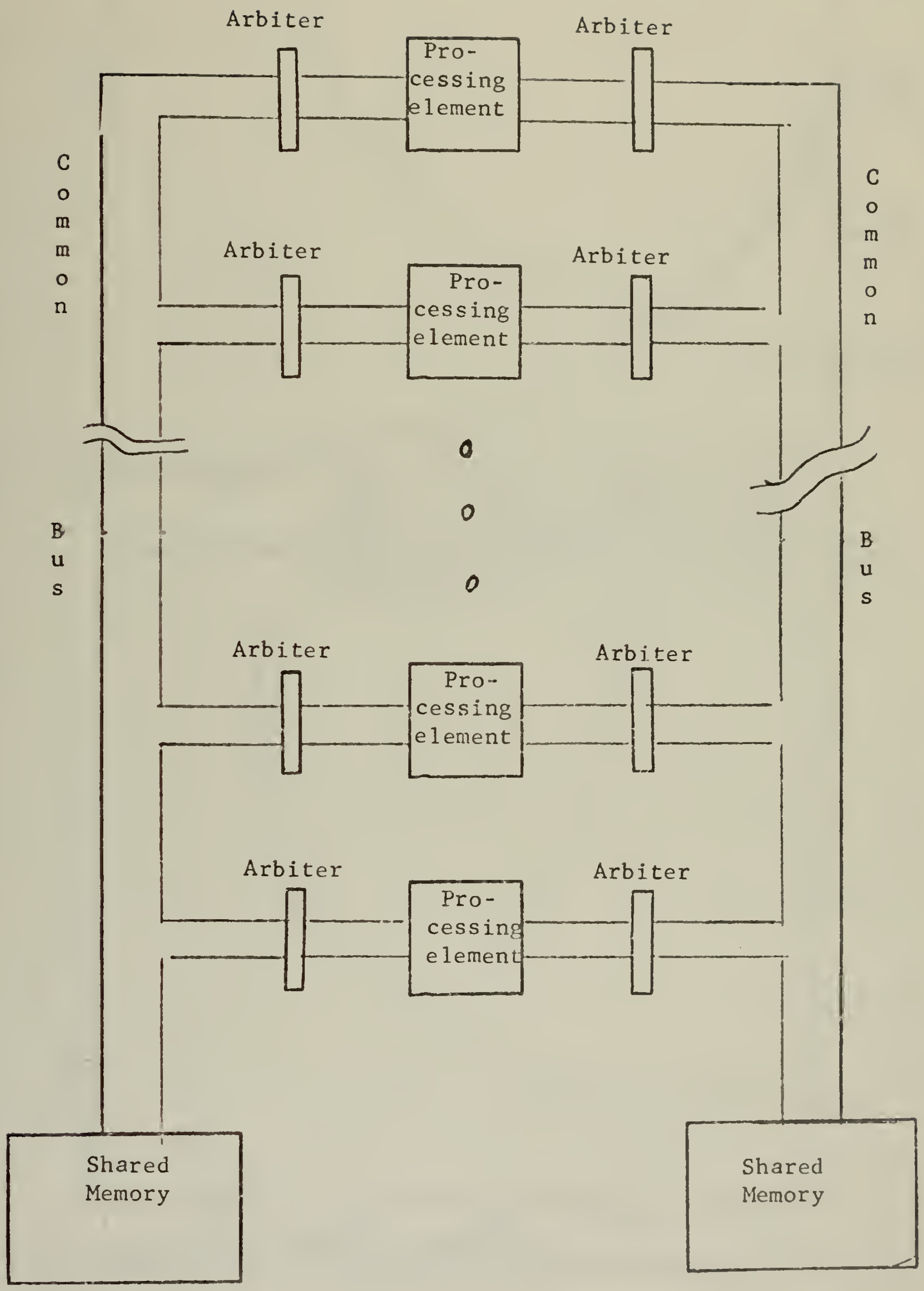
Chaotic Relaxation with single shared memory

Graph IV.C.3



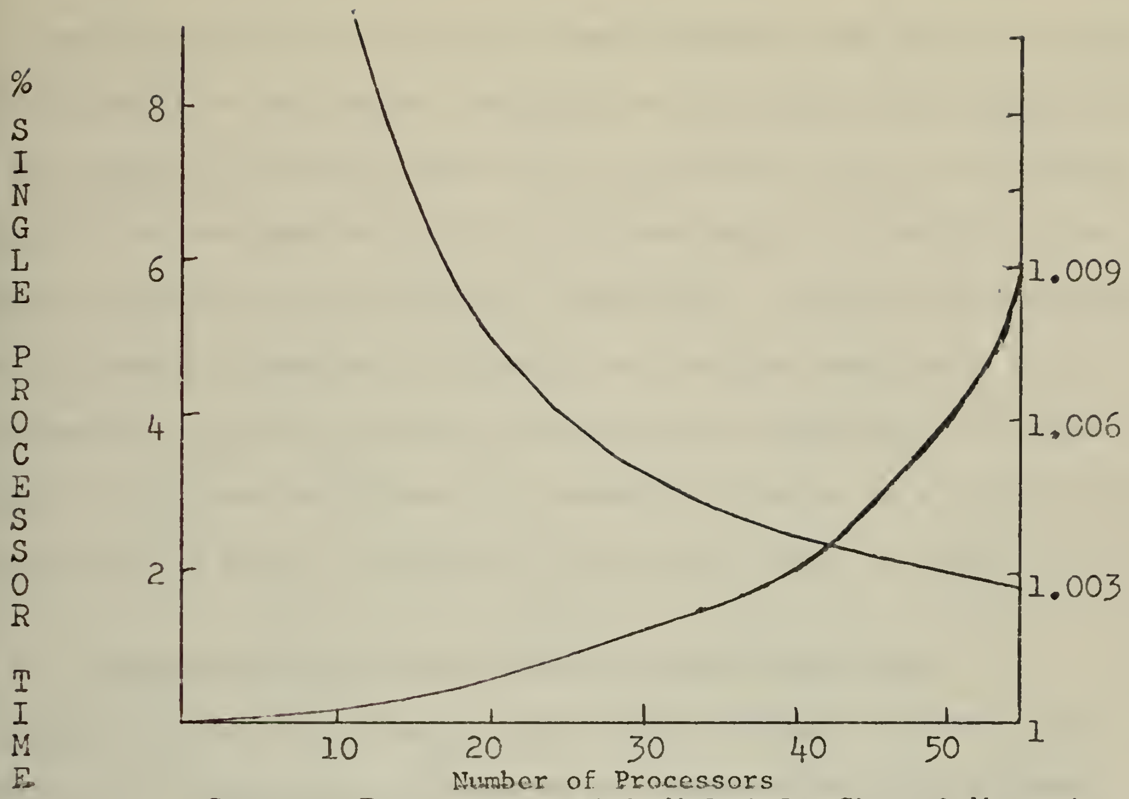
Newton-SOR with single shared memory

Graph IV.C.4

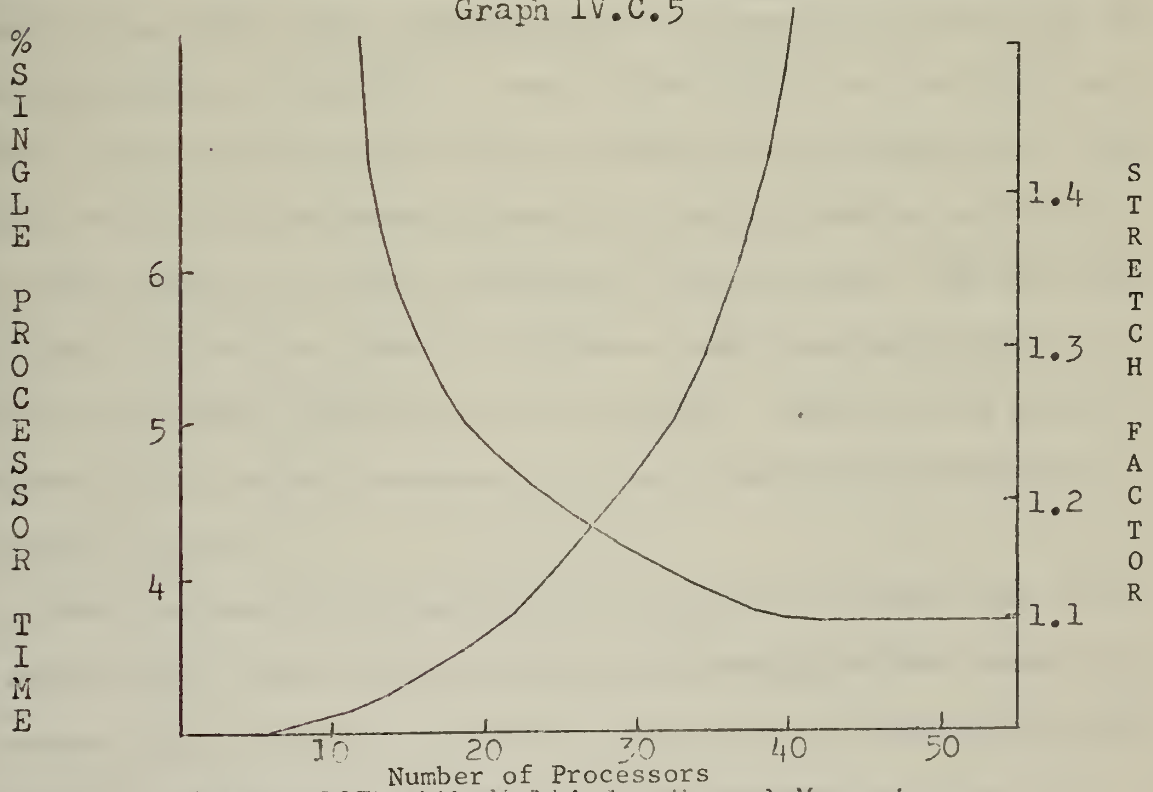


Near block diagonal form Multiprocessor (Two Shared Memories)

Figure IV.C.2



Chaotic Relaxation with Multiple Shared Memories
Graph IV.C.5



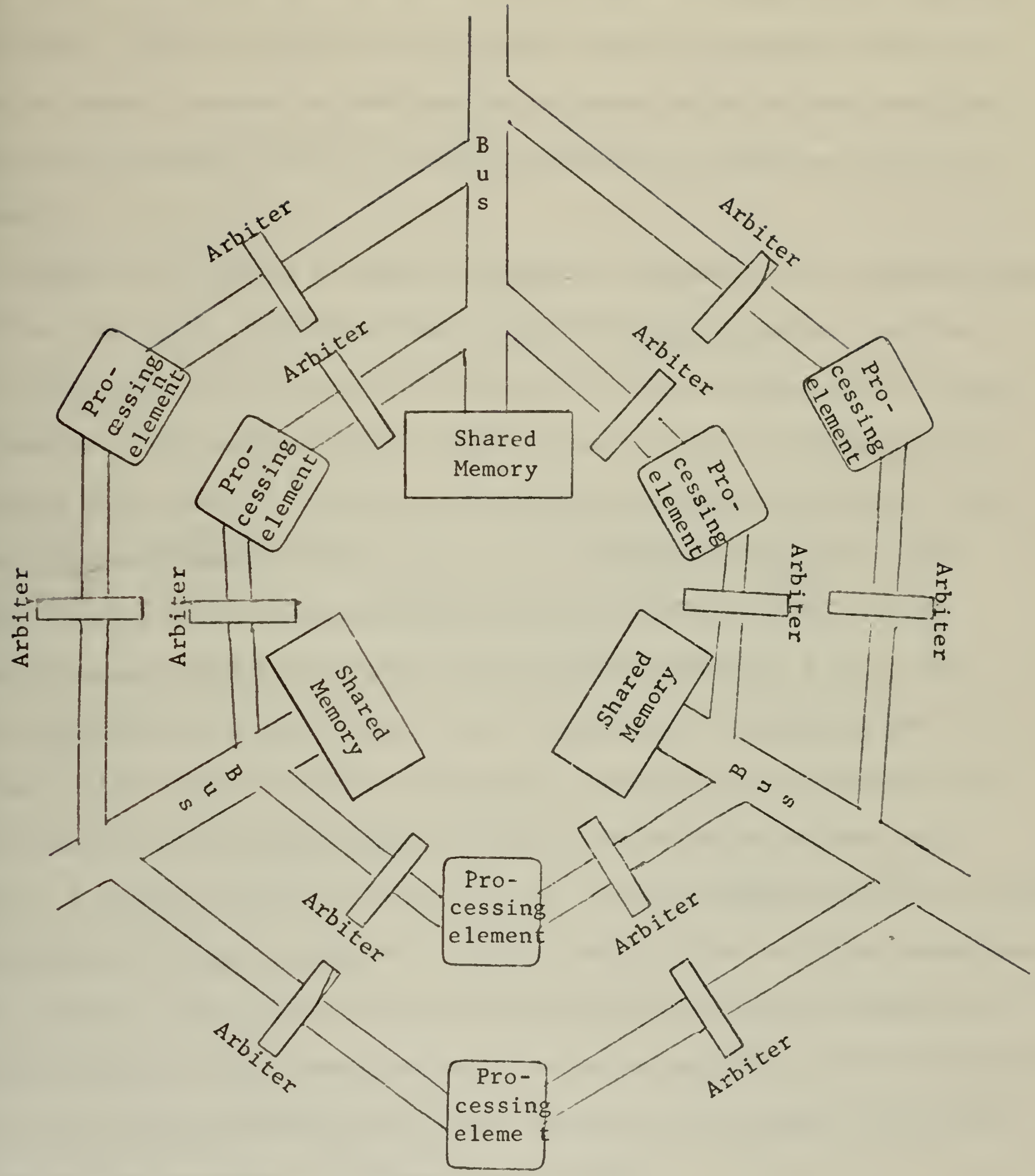
Newton-SOR with Multiple Shared Memories
Graph IV.C.6

Another structure uses multiple shared memories but does not connect all processors to each memory. Depending on the problem the decomposition of the dynamic simulation problem may be structured so all of the shared variables are not required by all of the processors. A savings in cost could be achieved by minimizing the connections. Modeling such a structure for the number of processors typically used requires many states to be represented. With this structure the individual probabilities for each group of processors would have to be specified, leading to a prohibitively large model. A typical structure of this type is shown in Figure IV.C.3.

IV.D Structures for the Bordered Block Diagonal Algorithms

The bordered block diagonal algorithms use the parallelism of the equations of the simulation to execute the Gauss-Seidel and the Newton method in parallel. This decomposition of the equations requires that the cut-set processor be capable of communicating with all of the other processors. The other processors need not communicate with each other. This places the cut-set processing element in a central location, with all of the other processing elements connected to this processor. The controller must still receive data from all processors, and disseminate signals to alter the execution sequences of all processors. Because of the central location of the cut-set processor, it is very capable of performing the necessary control.

The structures of the bordered block diagonal algorithms need to be divided further. Both the Gauss-Seidel and Newton Algorithms can be implemented with the cut-set processor's instructions overlapping the other processor's instructions, or with the cut-set executing only when the



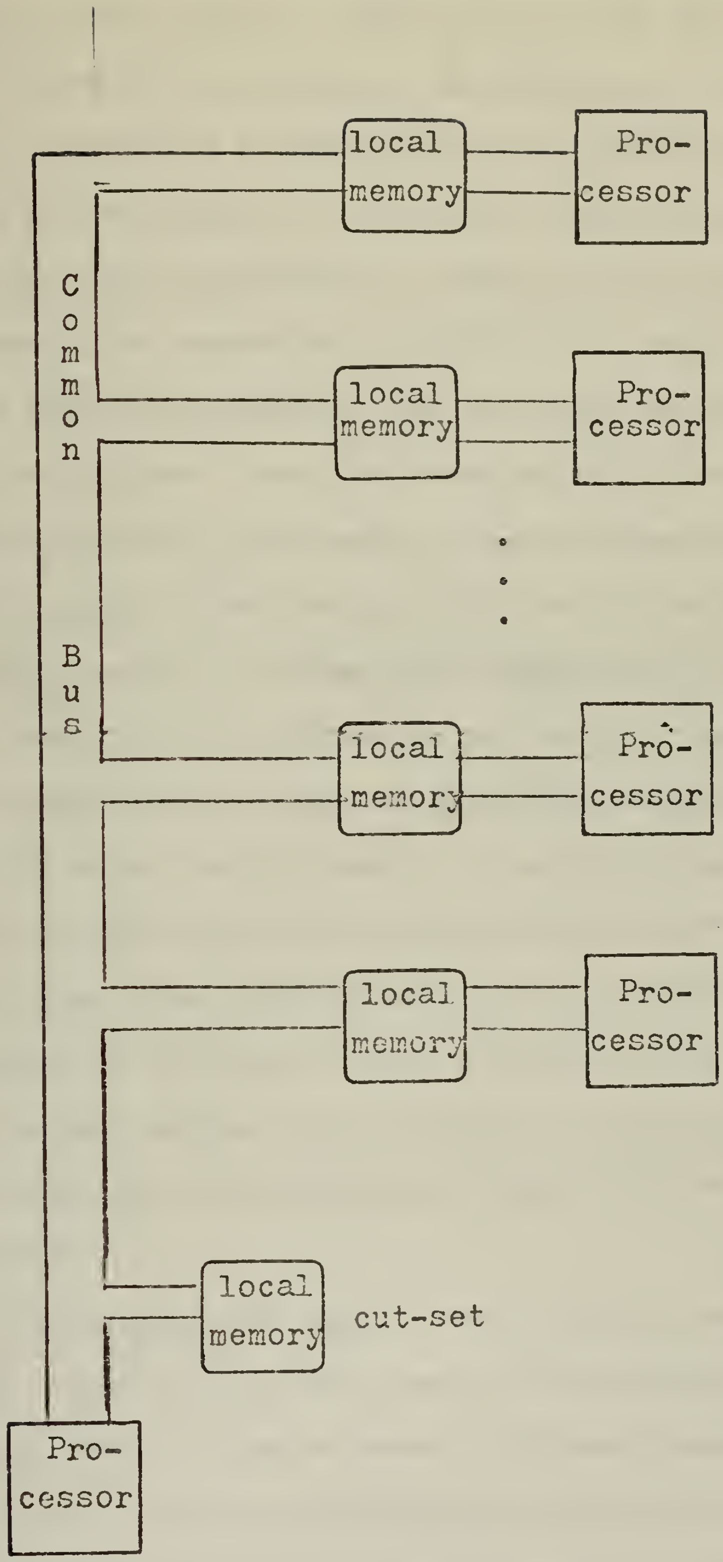
Near block diagoanl form Multiprocessor (Partially connected Shared Memeory)

Figure IV.C.3

other processors are idle and vice-versa. With no overlap of instructions, the multiprocessor structure is simplified because of the removal of memory contention. Thus no arbiter or even shared memory is required. There is cost in terms of execution speed and in the usage of the cut-set processor. The cut-set processor could be used for convergence testing and step size control.

Figure IV.D.1 shows a suggested computing structure for an implementation with no overlapping of instructions. This structure is simpler than the first multiprocessor of Section C, because there is no possibility of memory contention. The shared memory is actually part of each of the processing elements local memory. The cut-set processor can address locations of the other processors memory over the bus. It is suggested that part of the local memories have the same address, so that when the cut-set processor stores the values of its variables, each processor receives a copy. The cut-set processor can also address each local memory individually for the values of the local variables it requires. When the cut-set processor is idle, the bus is off so the other processors do not interfere with each other. A simple external controller, or the cut-set processor itself,is required to signal the cut-set processor to begin execution after the other processors have gone idle. The cut-set processor can signal the other processors to resume execution as its last instruction before going idle or into its control mode. The cut-set processor could also detect convergence and deposit information to signal the processors to advance the time step.

In Chapter Three a possible increase in execution rate was shown by having the cut-set processor and the other processors execute instructions concurrently. This complicates the structure of Figure IV.D.1 greatly.

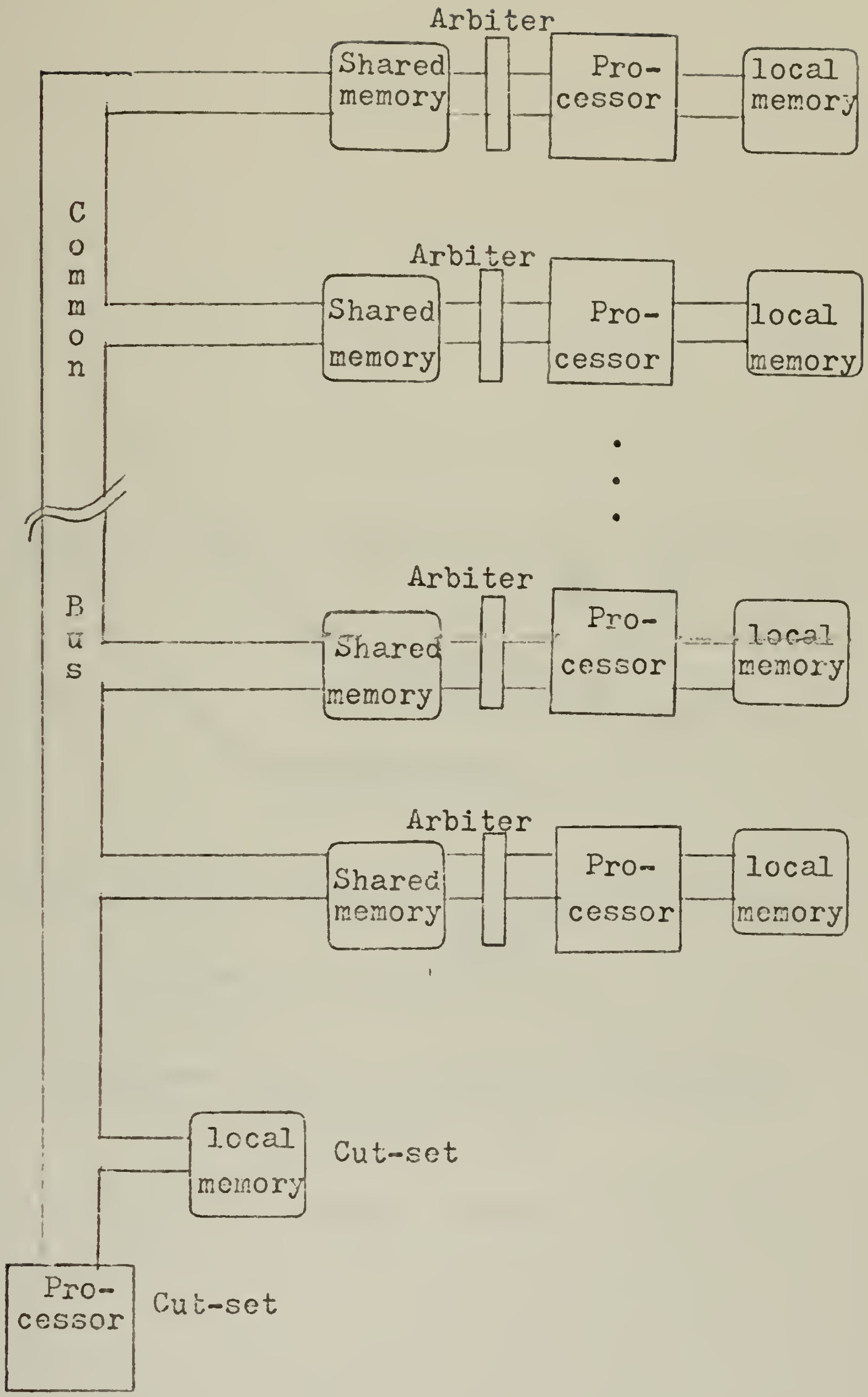


Bordered Block Diagonal Form Multiprocessor
Nonoverlapping Instructions
Figure IV.D.1

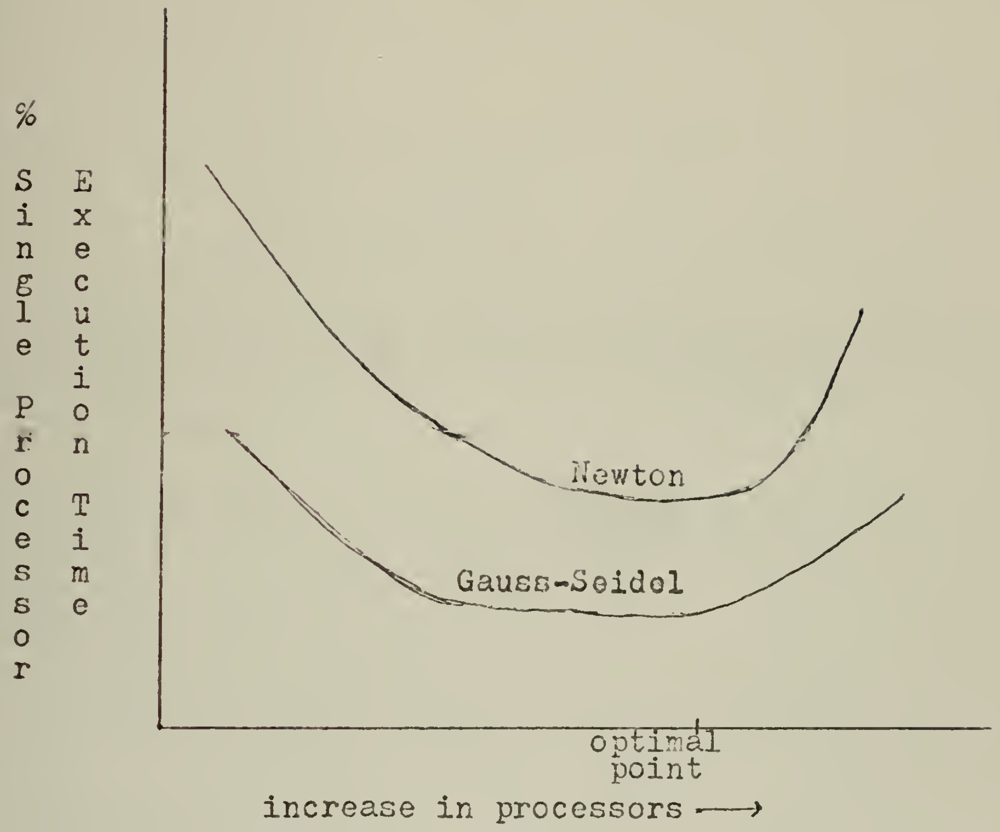
Now there must be separate sections of shared memory, and arbiters to control access to these memories. Figure IV.D.2 shows the suggested additions to be made to the structure. One advantage is that now the cut-set processor is capable of providing all of the control functions.

Even with the overlapping of instructions the delays which result from memory contention are inconsequential. However, there is another problem dependent feature which extends the execution as the number of processors grows. When a problem is decomposed into more parts the number of variables in the cut-set grows. Since the execution of the cut-set equations demands time proportional to the number of cut-set equations, an increase in their number greatly affects the execution rate of the entire problem. Each separate problem will decompose into blocks which will call for a processor for each block. An increase in the number of processors should be viewed as an increase in the size of the problem to be executed rather than a method to reduce the time required to execute a single problem. Graph IV.D.1 shows how the execution time decreases as more processors are added. The execution rate of the bordered block diagonal algorithms depends on the number of cut-set variables. As their number increases the largest portion of the processors are idle for longer periods. The increased idle time of the processors prevent the solution time from decreasing further.

One feature which if added would slightly increase the execution rate, but would be of more benefit by removing the need for the many arbiters, is dual port shared memory. The local processors could be assigned the first of the two possible memory cycles and the cut-set processor the second. Synchronizing the processors out of phase would be



Bordered Block Diagonal Form Multiprocessor
Overlapping Instructions
Figure IV.D.2



Approximate Effect of Number of Processors on Execution Time

Graph IV.D.1

much simpler than the control required to insure single processor access requirements. The added cost comes from the increased memory speeds.

CHAPTER V

DECOMPOSITION EFFECTS

This thesis has studied the dynamic simulation problem to determine the parallelism inherent in the problem and the requirements of a computing structure to exploit this parallelism. In the process of this study, the problem of decomposing the equations into blocks with the minimum number of interconnections has surfaced as a strategic factor for parallel execution. Although each system of equations only has to be decomposed the first time it is to be simulated, the methods of achieving this decomposition are haphazard at best. The algorithms which do exist are too inefficient to apply to sets of equations of the size described in this thesis. In addition, the actual decomposition has different effects on each of the algorithms. The least affected algorithm is Chaotic Relaxation and the most affected is the true Newton. The convergence rates of the Chaotic Relaxation and Newton-SOR schemes are affected by the actual decomposition, but only the number of operations required to compute the iteration is affected for the bordered block diagonal form Gauss-Seidel and Newton Algorithms. In fact, the same number of iterations are required for the parallel bordered block diagonal form algorithms as for the serial Gauss-Seidel and Newton Algorithms.

In this chapter the difficulties of the decompositions are studied and the effects are described.

V.A <u>The Problems of Decomposing Equations</u>

Kron [44], suggested solving large sets of equations by "tearing" the problem into smaller parts. For the power system problem, Kron

proposed tearing the network equations into small parts by eliminating some connecting lines. The solution of the individual parts was then iterated to balance the values on either side of the torn line. The modifications used to produce the convergence of the iterations, required that the equations of the smaller parts be linear. The requirement of linearity is considered too restrictive for for the dynamic simulation problem. Even with Kron's method, the choice of which lines to tear depended mainly on the intuition of the system analyst.

The notion of cutting the sets of equations comes from the use of optimal ordering for LU Factorization. By properly ordering the equations, the number of nonzero coefficients formed by the LU Factorization can be minimized. This minimizes the number of operations required to solve the equations. Several methods have been developed to properly order the equations, but only by trying all possible orderings is a true optimal ordering achieved. Associated with the optimal ordering is the construction of computer programs to perform operations only on nonzero coefficients. The sparsity programming techniques require a large amount of computation to calculate the next nonzero coefficient. The data structure problems can be simplified by concentrating the nonzero coefficients into blocks. The blocks can be programmed without the use of sparsity programming. The use of these blocks reduces the execution time of the algorithms compared to the nonblock methods with sparsity programming.

The methods used to find block orderings are related to the methods used for optimal ordering. To find the decompositions which minimize the nonzero coefficients outside the blocks would require essentially trying all possible orderings. Methods are presented to find approximation to

this block decompositions.

Carré [46] proposed a method of ordering the equations to the near block diagonal form. This method minimizes the value as well as the number of off block diagonal nonzero coefficients. The minimization of the value of these coefficients reduces the number of iterations that the Newton-SOR method requires. This scheme begins by forming a graph model of the equations. The branches are then listed in the order of the absolute value of the coefficient of that branch. The branches are added in decreasing order of the value of their coefficients until a tree is formed. (Non-tree branches are skipped when necessary.) The disconnected parts of the tree correspond to the different blocks of equations. If a block turns out to be larger than permissible that block can be subdivided by applying the same procedure to it. After choosing the variables the equations within each block should be put in optimal order for the LU decomposition.

The bordered block diagonal form also orders the equations so that the nonzero terms are in diagonal blocks. When the number of blocks is increased, the equations with coefficients of noncut-set variables which cannot be ordered into a diagonal block become a member of the cut-set. The cut-set equations form the last block. Two methods of finding this ordering are presented. Each is based on a graph model of the equations.

To find the smallest set of equations, the cut-set, whose deletion will divide the remaining equations into non-interacting sets, is equivalent to the min-cut max-flow Graph Theoretic Problem. The graph theoretic procedures can be used to find the cut-set. To do this, the highest degree vertex of the graph model of the equations is labeled the source. The

next highest mode not directly connected to the source is labeled the terminal. The minimum vertex cut-set is then found between the source and the terminal by an algorithm such as the one presented by Frank and Frisch [49]. If the number of vertices in this cut-set is equal to the degree of the terminal vertice then the source vertex becomes a member of the last block. If the minimum cut-set is smaller,then the cut-set is put in the last block. In either case the nodes of the last block are eliminated and the graph reduced again. If only the source vertex was added to the cut-set then the process is repeated until the graph is disconnected.

For graphs representing power systems networks, finding a useful decomposition may be difficult. The power system has been designed to insure continued operation in spite of failures in the system. Deletion of the cut-set is equivalent to those buses failing. The cut-set is therefore much larger for the power system network, than for a typical random graph for the same size and sparseness. When the algorithm presented by Frank and Frisch [49] is applied to a power system graph, the cut-set is typically the nodes adjacent to the terminal node. This produces the diagonal block of only the terminal node. A block size of one is not efficient for parallel execution, so the algorithm must be forced to find larger blocks. This increases the computation and reduces the optimality of the solution. Further the algorithm does not provide for finding more than two blocks at a time except for symmetric graphs. The minimum cut-set for dividing a graph into two subgraphs may not be contained in the cut-set for dividing the graph into three subgraphs. There is speculation in the literature that optimal ordering and block decomposition is an NP Complete problem in graph theory. [72]

A much more direct solution process has been proposed by Ogbuobiri, et al [45] for power systems. This method overcomes some of the problems of the graph theoretic procedures by attacking the problem in a different manner. The method groups the most highly connected nodes, allowing the cut-set to be chosen as the least connected nodes. The method does not help to decompose a large block into smaller ones. It can also be used to find the near block diagonal form by not extracting the cut-set variables.

A great deal of computation can be saved in the decomposition problem by reducing the graph model. The reduction process combines all nodes of degree two or less with their adjacent node. Further, all parallel paths and self loops are deleted. The resulting reduced graph can be shown to be either 1) a graph of five or more nodes all of degree three or greater, 2) a complete graph of four nodes, or 3) two nodes connected by a single branch. The latter two of these reduced graphs can be put into bordered block form easily. The complete graph of four nodes requires three of the nodes to form the cut-set. Depending on how the reduction proceeded there will be one, two, or three other blocks for the diagonal. A graph of two nodes with one branch comes either from parallel paths or one path of nodes of degree two. For parallel paths the cut-set is the two remaining nodes and each path is another block. For one path a center node is the cut-set with either side the other blocks. Only the reduced graph of five or more nodes needs further study to find the cut-set. [50]

The power system networks do provide some benefits. The graph can be related to geographical locations which can aid in locating possible

cut-sets. Experience with the system will lead the analyst to knowledge of the weakest links in the network. These weak links are natural choices for the cut-set. For power systems, reliability studies may show which buses are the weakest links in the network.

For the transient stability problem, there is one feature which can be used to always identify a cut-set. By reducing the network equations, the resulting non-sparse block of equations acts as a cut-set for the generator and load models. These models only connect through the network. This method is used in [29] to reduce the execution time for a single processor by simplifying the sparsity programming task.

V.B <u>Effects of Decomposition on the Near Block Diagonal Algorithms</u>

The difficulty of finding the optimal ordering for both the near block diagonal form and the bordered block diagonal form was presented in the last section. In this section the effects of the ordering are presented to allow the analyst to choose the degree of decomposition allowable. For the Chaotic Relaxation and Newton-SOR schemes, the decomposition aids in reducing the amount of sharing of data required and by improving the convergence rates. However, these two algorithms will achieve the correct solution without the decomposition.

For the Chaotic Relaxation the exact decomposition does not affect the number of operations required to perform one iteration. It does vary the individual processor's rate of requesting shared memory. This rate is directly related to the number of nonzero off block diagonal coefficients. But even for dense equations the highest rate of requesting shared memory is trivially small.

The remaining effect is the change in the convergence rate of the

Chaotic Relaxation with different decompositions. Proper decomposition can improve the convergence rate considerably. With improper decomposition the updating of the variables could become a Jacobi scheme, where the values for the variables used to compute the update are an iteration or more old. With block diagonal decomposition the updating sequence could be a Gauss-Seidel scheme. Without the block diagonal decomposition the equations are not updated exactly in sequential order. This reduces the convergence rate to a value less than that of the Gauss-Seidel method. The convergence rate is reduced further when an old iteration value is used to compute an update. This can only occur from a shared variable. The possibility of using an old iteration is reduced primarily by insuring that each processor requires approximately the same number of operations to compute an iteration. (Equally sized blocks is the simplest measure of this time.) Also minimizing the number of external variables required by a processor reduces the possibility of using an old iteration value.

For the Newton-SOR Algorithm the rate of using shared memory is independent of the decomposition (except for totally disjoint equations). This results from only the nonzero off block diagonal entries being used during the iteration for the Newton update. The actual number of operations required for each iteration and the convergence rate of the Newton update does depend on the decomposition. The number of Newton iterations is not affected by the decomposition.

To minimize the number of operations, the number of columns outside the block diagonal part of the Jacobian which contain nonzero values must be minimized. The actual number of nonzero entries in each of these

columns is not significant. Thus when the decomposition is found, the number of external variables required by each block is the value which must be reduced.

The convergence rate of the Newton update is maximum when the processors all require the same number of external variables. However, the convergence rate is not as significant as the increase in the number of operations caused by the addition of more external variables. (The number of operations grows as n^3.) If given the prerogative the off block diagonal entries should be of the smallest absolute magnitude, to provide the highest possible convergence rate.

To summarize,the decomposition for the Chaotic Relaxation Algorithm should strive for equal sized blocks even if additional off block diagonal entries are required. For the Newton-SOR Algorithm the number of external variables must be minimized.

V.C. Effects of Decomposition on the Bordered Block Diagonal Algorithms

The decomposition of the network equations provides the only high level parallelism possible in the true Newton and Gauss-Seidel Algorithms. If the equations could not be decomposed, there is no straightforward method of using a multiprocessor to reduce the time required for the solution of the dynamic simulation problem by these algorithms. Experience indicates it is reasonable to assume that the equations can be decomposed to the bordered block diagonal form. In this section the properties of the cut-set are related to the execution rate.

Since the parallelism is found in the equations to be solved and not in the algorithms used to solve them, the convergence rates of the algorithms

are not altered. All of the solution steps required by the algorithms are performed in the same manner as they would be for serial execution. This results in the solution being found in the same number of iterations, regardless of the number of processors used to calculate the iterations.

With the bordered block diagonal form algorithms, the number of operations per iteration is the sum of the operations for the parallel blocks and the cut-set block. By increasing the number of blocks, more processors can be used to reduce the time required to compute the diagonal blocks portion of the iteration. But the solution time of the cut-set is not reduced, further the size of the cut-set is typically increased by decomposing the equations into more blocks. With more equations to solve,the cut-set processor requires a larger portion of time, so that increasing the number of blocks may actually increase the time required for solution. Since the Newton and Gauss-Seidel algorithms require such differing amounts of computation per iteration, the effects of increasing the size of the cut-set is different.

For the Gauss-Seidel Algorithm the cut-set variables must be iterated while the other processors are idle. This requires the cut-set processor to sum the portions of the update from each processor. But not all cut-set variables are affected by all processors. Thus the computation for the cut-set processor only grows as the sum of the number of cut-set variables affected by each processor. There is no difference to the cut-set processor between only one variable of a diagonal block affecting a cut-set variable, and all variables of the diagonal block affecting the cut-set variable. Since so little computation is required by the cut-set processor, for the

bordered block diagonal form of the Gauss-Seidel Algorithm, an increase in the size of the cut-set has only slight effect.

For the true Newton algorithm the cut-set processor must sum the modifications from all of the other processors and then solve the resulting equations. The solution time for these equations grows as n^3, where n is the number of cut-set variables. Since the equations of the cut-set are dense after the modifications are summed, more operations may be required to solve a smaller number of equations than the other processors require to solve a larger number of sparse equations. Thus when the number of blocks is increased, increasing the size of the cut-set, the computation time of the cut-set is dramatically increased while the time for the diagonal processors is only partially decreased. The true Newton Algorithm requires that the number of cut-set variables be minimized to yield the fastest possible solution.

For the bordered block diagonal form Gauss-Seidel Algorithm it is possible to use more than one cut-set processor, by assigning different cut-set variables to each processor to update. The cut-set variables would have to be assigned so that the variables would not depend on variables assigned to the other cut-set processors. However, seldom do the cut-set variables depend on other cut-set variables. The cut-set processors would have to compete for the available accesses to the local processor's memories for the cut-set processors the available accesses are sufficient for several processors to share without serious contention. The use of several cut-set processors would relieve the bottleneck caused by the solution of the cut-set variables, resulting in even higher execution rates than projected in this thesis.

To use multiple cut-set processors for the bordered block diagonal form Newton Algorithm requires a different philosophy in finding the cut-set. The cut-set variables must be divided into layers, and the solution process itself gains another layer where processors are idle. The cut-set is cut into nearly disconnected parts whose complete solution requires another cut-set processor. To find a decomposition which is itself decomposed into disconnected parts is not difficult. First the set of equations are cut into two near equal subsets. This first cut is the lowest level of the cut-set. The large subsets are then cut into smaller blocks. But the cut-set from each subset is not dependent on the equations of the other subset. Solution of the Jacobian equations can now proceed in five steps. First the diagonal blocks are triangularized and the last block row eliminated. The subset cut-set processor then sums the modifications to its variables, and forms the modification to the lower cut-set. This lowest cut-set is solved and the back substitution begins. The subset cut-set uses the lowest level cut-set variables to find the new value of the subset cut-set variables. Finally the diagonal blocks can use the two level of cut-set variables to complete the required back substitution for the remaining updates.

Using the decomposed form of the cut-set variables can save execution time even when only one cut-set processor is used. The decomposed form forces sparsity into the final cut-set equations, greatly reducing the number of operations required to solve for the cut-set updates.

CHAPTER VI

CONCLUSION

The dynamic simulation problem can be structured so that a high degree of macroparallelism exists in the solution techniques. The macroparallelism results not only from the algorithms used to solve the dynamic simulation problem but also from the actual equations of the models. Simple multiprocessors have been proposed which can use this macroparallelism to greatly reduce the time required to solve the dynamic simulation problem.

The model of a dynamic system has been considered to consist of a large number of nonlinear differential and algebraic equations. In order to efficiently solve these equations the differential equations are expressed as algebraic equations through the use of implicit multistep integration methods. The entire set of equations is now solved by the numerical methods applicable to nonlinear algebraic equations. Four algorithms have been studied in detail, Chaotic Relaxation, Gauss-Seidel, Newton-SOR, and true Newton. Macroparallelism is present in the Chaotic and Newton-SOR algorithms. The Gauss-Seidel and Newton methods depend on the parallelism for the actual equations for parallel solution.

The convergence properties of the parallel solution methods are basically the same as the serial convergence properties. Where convergence can be proven, the rates of convergence of the methods have the same relative ordering as they have in a serial implementation. The true Newton and Newton-SOR methods converge most rapidly, followed by then the Gauss-Seidel and Chaotic Relaxation Algorithms. The Newton-SOR method requires

two levels of iteration. The major iteration converge at the same rate as the Newton method, while the minor iterations converge at a linear rate. The Gauss-Seidel method can be shown to converge at a higher rate than the Chaotic Relaxation, but the rate is only slightly higher.

A feature of almost all large sets of equations, especially those of the dynamic simulation problem, is sparsity. Because of the sparsity present in the equations of the models, the equations can be ordered to either the near block diagonal form or the bordered block diagonal form. These forms concentrate the nonzero coefficients of the equations into diagonal blocks. The parallel solution techniques then solve the blocks of equations in parallel. Information must be exchanged between the parallel solution streams for those nonzero coefficients which cannot be arranged into a diagonal block.

The near block diagonal form of the equations reduces the required sharing of solution data between the processors for the Chaotic Relaxation and Newton-SOR algorithms. This form also decreases the number of operations and increases the convergence rate of the Newton-SOR method. But arranging the equations to this form is not an absolute requirement for parallel execution.

By arranging the equations into the bordered block diagonal form, macroparallelism can be obtained in the Gauss-Seidel and true Newton methods. For these methods the degree of parallelism depends entirely on the actual decomposition of the equations. Parallel solution methods for these two algorithms only slightly increases the number of operations that must be performed for an iteration, even though many processors are performing the operations concurrently. The convergence rates of the

algorithms is not affected by parallel solution.

The algorithms can also be ranked by the number of operations each requires to compute a new estimate of the variables. In general for the same problem the Newton-SOR algorithm requires the largest number of operations per iteration. The Newton method is next followed by the Gauss-Seidel and then the Chaotic Relaxation. There are several orders of magnitude difference in the number of operations required for the Newton methods, compared to the linear methods. It has been reported that the linear iterative methods require on the average 5 to 7 iterations to reduce the error an amount equal to one iteration of the Newton methods. To find an overall ranking,the delays due to parallel execution must be included with the convergence properties.

The two methods of ordering the equations basically determine the requirements for sharing data and thus the structure of the multiprocessor. The algorithms determine the rates at which the data must be shared and the control required for execution. The sharing of data causes delays in the execution either because the required information has not yet been computed or the device containing the information is busy servicing another processor. By minimizing the amount of information which must be shared, the delays due to sharing this data are reduced. The delays due to control of the solution process are inherent in the algorithms. Some of the control delays can be reduced, but the gains achievable are not as significant as possible for the sharing of data.

The algorithms of the near block diagonal form, exchange solution data conveniently through shared memory.These algorithms, the Chaotic and Newton-SOR, have the same control and shared data requirements for parallel execution,

and thus can be executed on the same multiprocessing structures. However since the Chaotic Relaxation algorithm requires shared data at a much lower rate than the Newton-SOR algorithm, approximately three times the number of processors can be executing this algorithm in the same structure with approximately the same delays from memory contention. The Newton-SOR algorithm requires several times as many calculations per iteration as the Chaotic method, and will converge at a higher rate. Since the number of operations and the convergence rate is heavily dependent on the actual problem, an actual ranking in terms of solution time requires problem solution experience.

The bordered block diagonal form algorithms exchange data in an entirely different manner so that contention from shared memory is not a problem. From the memories' point of view the Gauss-Seidel and true Newton algorithm appear to be executed by a single processor. This is because the cut-set processor accomplishes the exchange of data while the other processors are idle. Thus the delays in parallel execution result from processors being idle waiting for data to be computed, and the delays become longer as the number of variables in the cut-set are increased. As a result the advantages derived from parallel execution depend on the particular problem being solved. Experience has shown that usually,sparse sets of equations can be arranged into the bordered block diagonal form. By combining the information presented, estimates of the actual decreases in execution time can be predicted. By comparing the number of operations required for serial execution to the number of operations in the longest parallel stream plus any concomitant delays,the decrease in execution time can be

predicted. If the estimates of the convergence rate are included, a rough ranking of the parallel algorithms can be obtained.

For the Chaotic Relaxation algorithm a single processor implementation would execute the same instruction sequence but for more variables. However the convergence rate of the single processor (Gauss-Seidel) would be higher than that of a multiprocessor. Arranging the equations in the near block diagonal form, to reduce the sharing of data between processors, prevents the equations from being divided equally between the parallel processors. Therefore, if p is the number of processors, δ, the fraction of additional operations in the longest parallel path, and s is the stretching factor due to memory contention; then a multiprocessor could execute an iteration of the Chaotic Relaxation algorithm in $(1+\delta)s/p$ of the time a single processor would require. Sample coding indicates that on simple single shared memory multiprocessor, the Chaotic Relaxation algorithm could be solved by 40 processors in parallel in 2.7 percent of the time that a single processor would require. By increasing the complexity of the multiprocessor, even further reductions in execution time are possible.

With the Newton-SOR algorithm comparison is more difficult since there is usually no reason to use the Newton-SOR algorithm on a serial processor. The true Newton solution could be found in fewer operations. When the equations are too dense to be able to achieve the bordered block diagonal form or the linear iterative method does not converge, then the Newton-SOR method must be used. For completely dense sets of equations it does represent a possible savings in execution time over the true Newton method. Because of the higher rate required for sharing data, fewer

processors can share the same memory efficiently. A multiprocessor with 10 processors could execute the Newton-SOR algorithm in ½ the time a serial processors could solve the true Newton algorithm. (This figure is based on the sparsity and decomposition of the power system problem used in Chapter Three and discussed in Appendix A.)

The bordered block diagonal form algorithms are even more problem dependent than the Newton-SOR Algorithms. The parallelism of the Gauss-Seidel and true Newton algorithms depends on the achievable decomposition of the equations. The delays of parallel execution are based on the time required to solve the cut-set equations. The number of processors capable of executing the algorithms in parallel depends on the number of diagonal blocks into which the equations decompose. If the number of operations required to solve the cut-set equations is equal to the number of operations required for the largest diagonal block, then the solution time for the multiprocessor is 2/p of the single processor time. For the problem used for the Newton-SOR algorithm the decomposition indicates five processors could efficiently execute the problem. Other decompositions exist that could use more processors, and if larger problems were used for a basis then it is expected that many more processors could be used. For this problem the Gauss-Seidel algorithm could be solved in 1/3 the time of a serial processor and the true Newton algorithm could be solved in 2/3 of the time.

Methods have been discussed to improve the decompositions and increase the complexity of the multiprocessors to reduce the time required for parallel solution even further. With these improvements the bordered block

diagonal form multiprocessors would probably provide the fastest solution of the dynamic simulation problem. However, since the Chaotic Relaxation Algorithm is capable of being executed on higher order multiprocessors, the increased parallelism may provide the fastest solution. Only through actually solving many problems by all algorithms can a preferred method be established. The results do indicate that parallel solution of the dynamic simulation problem is feasible and the multiprocessors structures required to execute the parallel algorithms need not be highly complex.

This thesis has shown many directions for further research, from theoretic numerical methods to hardware controller implementation. The most obvious missing data is experimental evidence that these multiprocessors can easily be built. Unfortunately, the time requirements prohibited this approach, even though it is expected that low level parallel dynamic simulation could be performed on the Computer Science Research Network of Northwestern University. Another possibility is to model the proposed multiprocessor structures with a microprocessor network which would test the control requirements and delays from sharing data. Still for complete assurance of success, the multiprocessors must be built and the algorithms tested for many problems.

An unsolved problem which seems to be reappearing in the literature is the decomposition and optimal ordering procedures [72]. Efficient solution procedures would greatly benefit this class of problems.

There is still a wide gap between sets of equations which can be proven to converge under an algorithm, and actual convergence. The restrictions necessary to prove convergence limit the ability to compare the algorithms execution rates.

APPENDIX A

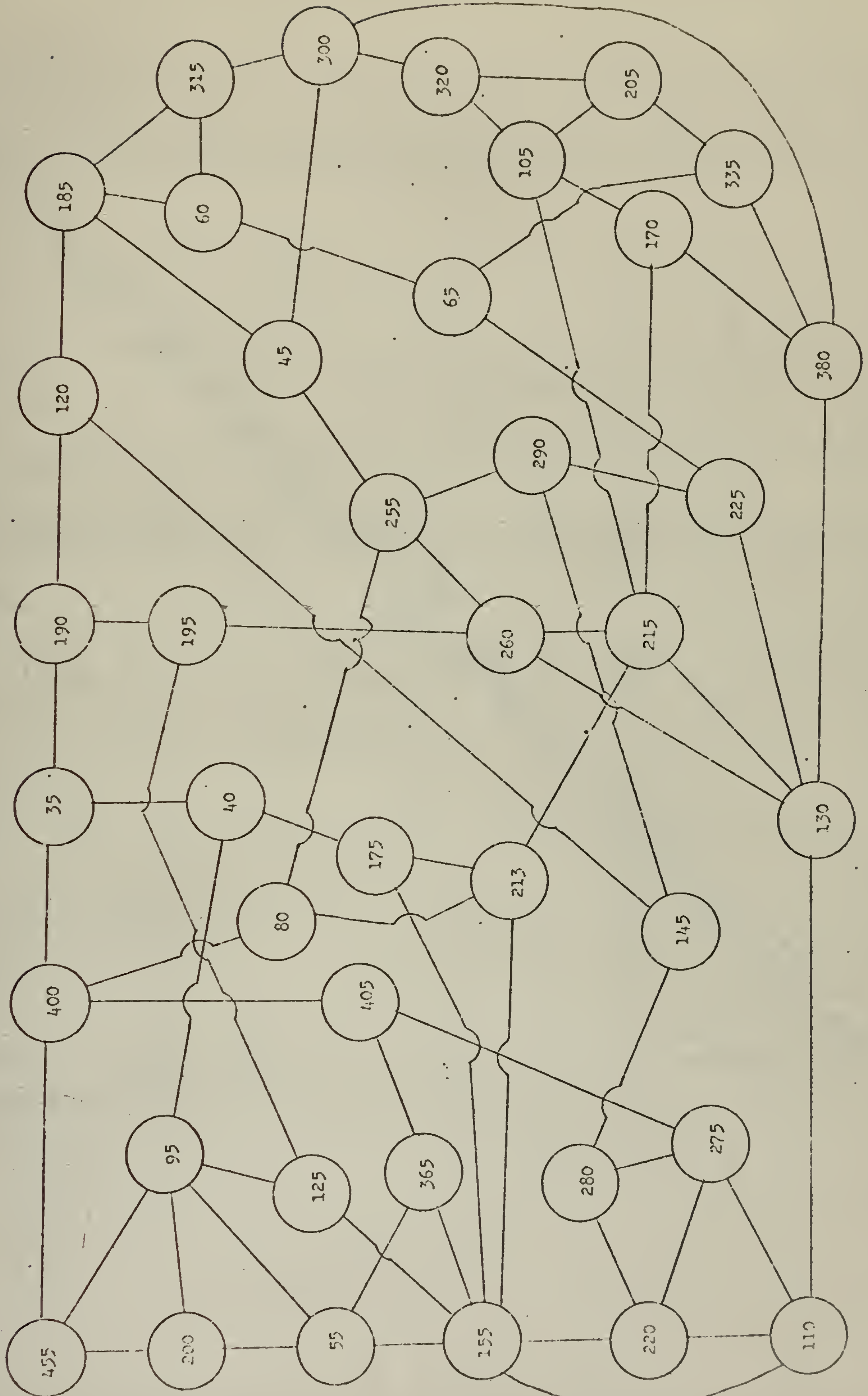
A POWER SYSTEM EXAMPLE OF A DYNAMIC SIMULATION PROBLEM

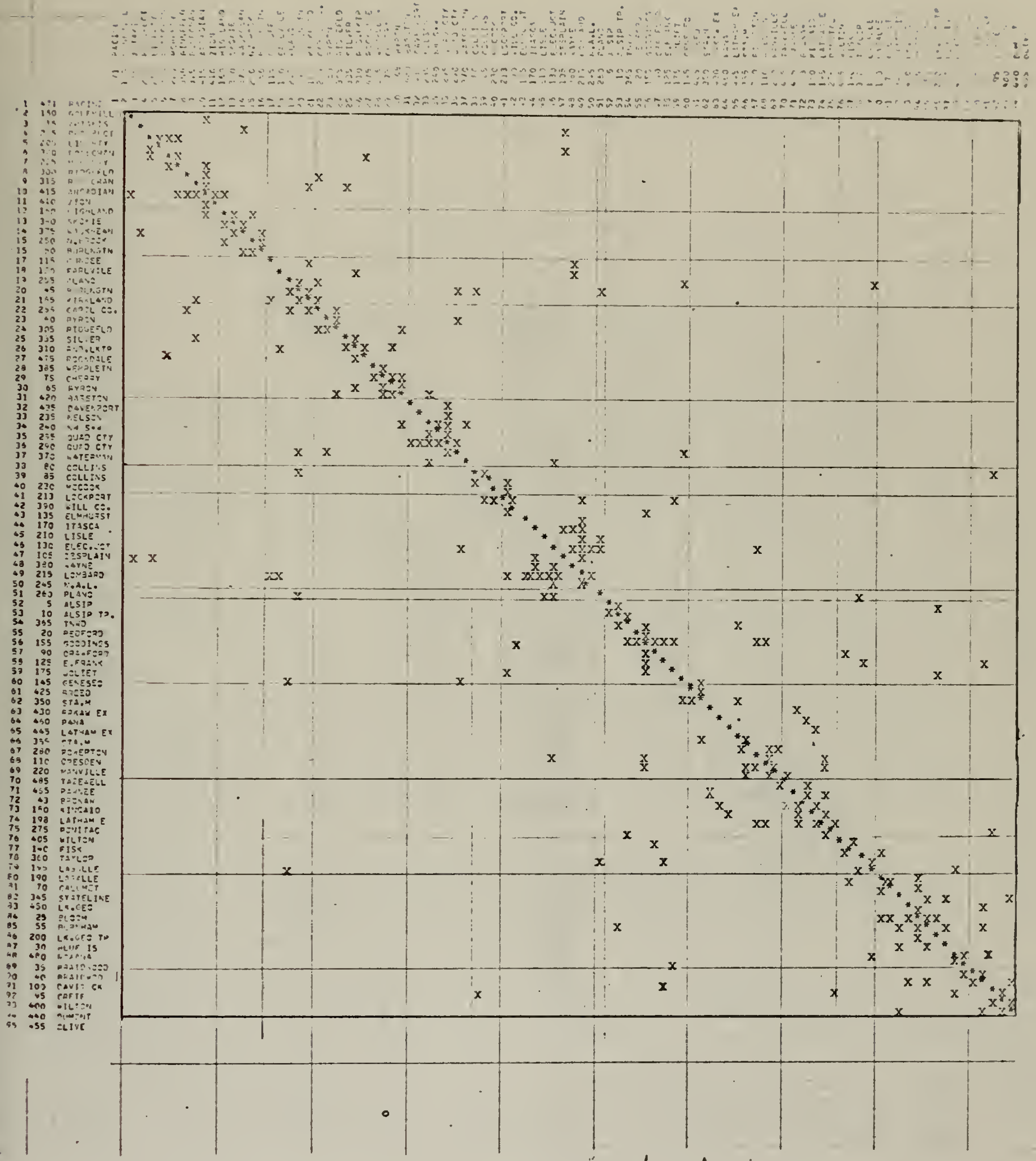
Whenever a specific problem has been required to complete the analysis of this thesis the Commonwealth Edison High Voltage Distribution System has been used. The distribution system consists of twelve generating stations, ninety-five busses, and 143 lines, seven of which are parallel to other lines. The loads are all modelled by constant loads.

The equations of the model are equivalent to those of [29], and are developed in [58]. The equations can be found in the programs of Appendix B. A list of the busses and lines connecting each bus is given in Table A.1 and A.2. After the network has been modelled by a graph, it is reduced by the procedures of Section V.A. The reduced graph is a much more manageable size with forty nodes and sixty-nine lines. The reduced graph model is presented in Figure A.3. The reduced model can be decomposed, enabling the entire network to be decomposed. A typical near block diagonal form decomposition is given in Figure A.4, and a bordered block diagonal form decomposition in Figure A.5. Neither of these decompositions are unique nor are the decompositions optimal.

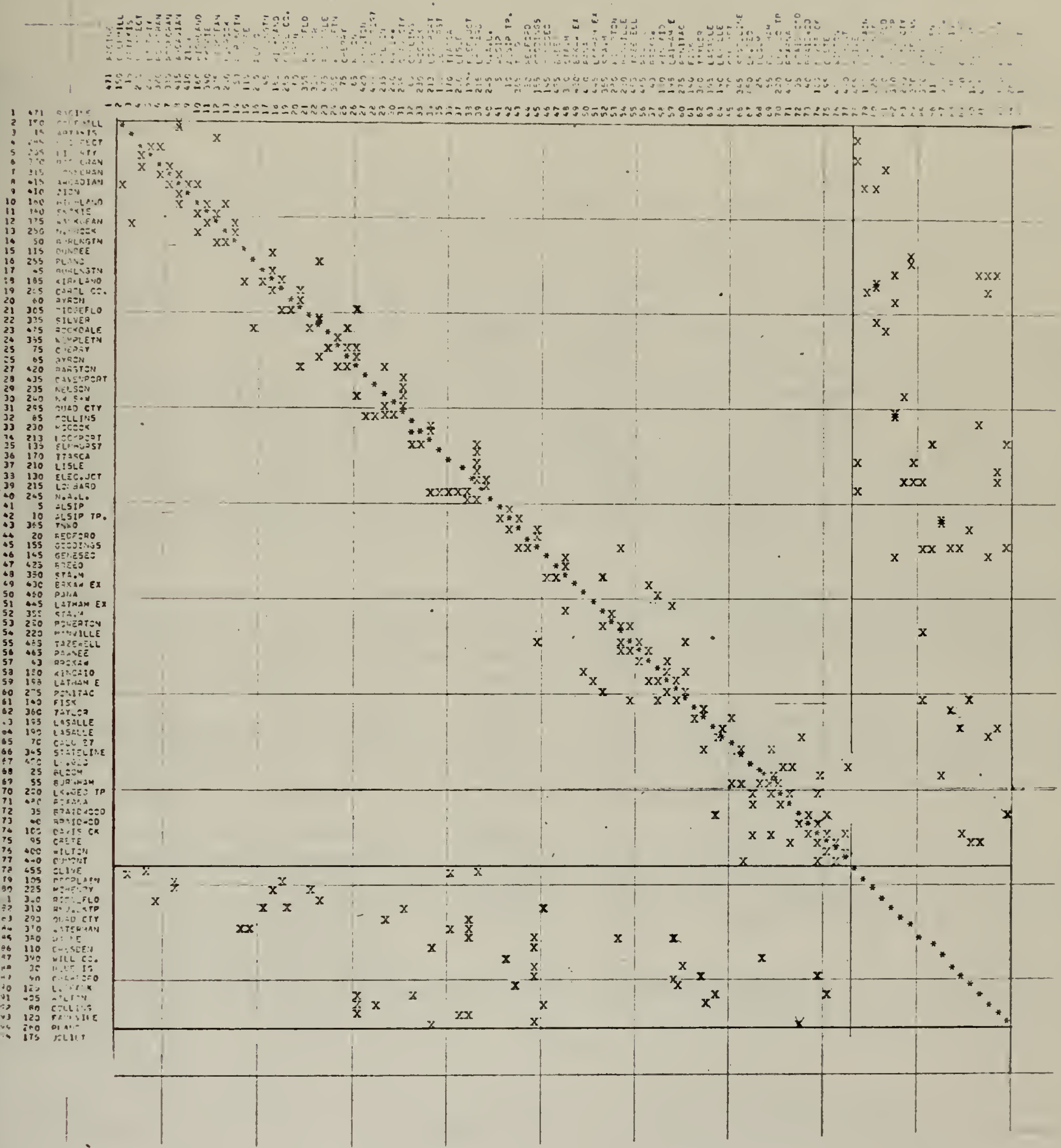
```
BUS NAME
  5 ALSIP    345          235 NELSON   345
 10 ALSIP TP.345          240 NW S+W   345
 15 APTAKIS  345          245 N.A.L.   345
 20 BEDFORD  345          250 N.BROOK  345
 25 BLOOM    345          255 PLANO    765
 30 BLUE IS  345          260 PLANO    345
 35 BRAIDWOOD765          265 CAROL CO.765
 40 BRAIDWOD 345          275 PONITAC  345
 43 BROKAW   345          280 POWERTON 345
 45 BURLNGTN 765          285 PROSPECT 345
 50 BURLNGTN 345          290 QUAD CTY 765
 55 BURNHAM  345          295 QUAD CTY 345
 60 BYRON    765          300 RIDGEFLD 765
 65 BYRON    345          305 RIDGEFLD 345
 70 CALUMET  345          310 RND.LKTP 345
 75 CHERRY   345          315 ROSECRAN 765
 80 COLLINS  765          320 ROSECRAN 345
 85 COLLINS  345          335 SILVER   345
 90 CRAWFORD 345          340 SKOKIE   345
 95 CRETE    345          345 STATELINE345
100 DAVIS CK 345          350 STA.M    765
105 DESPLAIN 345          355 STA.M    345
110 DRESDEN  345          360 TAYLOR   345
115 DUNDEE   345          365 TNWD     345
120 EARLVILE 765          370 WATERMAN 345
125 E.FRANK  345          375 WAUKGEAN 345
130 ELEC.JCT 345          380 WAYNE    345
135 ELMHURST 345          385 WEMPLETN 345
140 FISK     345          390 WILL CO. 345
145 GENESEO  765          400 WILTON   765
150 GOLFMILL 345          405 WILTON   345
155 GOODINGS 345          410 ZION     345
160 HIGHLAND 345          415 ARCADIAN 345
170 ITASCA   345          420 BARSTON  345
175 JOLIET   345          425 BREED    765
180 KINCAID  345          430 BRKAW EX 345
185 KIRKLAND 765          435 DAVENPORT345
190 LASALLE  765          440 DUMONT   765
195 LASALLE  345          445 LATHAM EX345
198 LATHAM E 345          450 LK.GEO   345
200 LK.GEO TP345          455 OLIVE    345
205 LIBERTY  345          460 PANA     345
210 LISLE    345          465 PAWNEE   345
213 LOCKPORT 345          471 RACINE   345
215 LOMBARD  345          475 ROCKDALE 345
220 MANVILLE 345          480 ROXANA   345
225 MCHENRY  765          485 TAZEWELL 345
230 MCCOOK   345
```

```
PRUS AR PRUS NAME    OBUS AR CBUS NAME     C
   5  0 ALSIP          10  0 ALSIP TP.     120  0 EARLVILE     255  0 PLANO        300  0 RIDGEFLD     305  0 RIDGEFLD
  10  0 ALSIP TP.      30  0 BLUE IS       125  0 E.FRANK      155  0 GOODINGS     300  0 RIDGEFLD     315  0 ROSECRAN
  10  0 ALSIP TP.     365  0 TNWD          125  0 E.FRANK      195  0 LASALLE      305  0 RIDGEFLD     335  0 SILVER
  15  0 ARTAKIS       205  0 LIBERTY       130  0 ELEC.JCT     215  0 LOMBARD      310  0 RND.LKTP     335  0 SILVER
  15  0 ARTAKIS       285  0 PROSPECT      130  0 ELEC.JCT     245  0 N.A.L.       315  0 ROSECRAN     320  0 ROSECRAN
  20  0 BEDFORD       155  0 GOODINGS      130  0 ELEC.JCT     250  0 PLANO        315  0 ROSECRAN     320  0 ROSECRAN
  25  0 BLOOM          55  0 BURNHAM       130  0 ELEC.JCT     370  0 WATERMAN     315  0 ROSECRAN     410  0 ZION
  25  0 BLOOM          95  0 CRETE         130  0 ELEC.JCT     380  0 WAYNE        315  0 ROSECRAN     415  1 ARCADIAN
  30  0 BLUE IS        55  0 BURNHAM       135  0 ELMHURST     215  0 LOMBARD      315  0 ROSECRAN     471  1 RACINE
  35  0 BRAIDWOOD      40  0 BRAIDWOOD     140  0 FISK         360  0 TAYLOR       350  0 STA.M        355  0 STA.M
  35  0 BRAIDWOOD     190  0 LASALLE       145  0 GENESEO      290  0 QUAD CTY     350  0 STA.M        425  1 BREED
  35  0 BRAIDWOOD     400  0 WILTON        145  0 GENESEO      350  0 STA.M        365  0 TNWD         405  0 WILTON
  40  0 BRAIDWOOD     100  0 DAVIS CK      150  0 GOLFMILL     340  0 SKOKIE       375  0 WAUKGEAN     410  0 ZION
  40  0 BRAIDWOOD     175  0 JOLIET        155  0 GOODINGS     175  0 JOLIET       385  0 KEMPLETN     475  1 ROCKDALE
  43  0 BROKAW        180  0 KINCAID       155  0 GOODINGS     220  0 MANVILLE     400  0 WILTON       405  0 WILTON
  43  0 BROKAW        275  0 PONITAC       155  0 GOODINGS     365  0 TNWO         400  0 WILTON       405  0 WILTON
  43  0 BROKAW        430  1 BRKAW EX      155  0 GOODINGS     390  0 WILL CO.     400  0 WILTON       440  1 DUMONT
  45  0 BURLNGTN       50  0 BURLNGTN      160  0 HIGHLAND     340  0 SKOKIE       440  1 DUMONT       455  1 OLIVE
  45  0 BURLNGTN       50  0 BURLNGTN      160  0 HIGHLAND     410  0 ZION         450  1 LK.GEO       455  1 OLIVE
  45  0 BURLNGTN       50  0 BURLNGTN      170  0 ITASCA       215  0 LOMBARD      450  1 LK.GEO       480  1 ROXANA
  45  0 BURLNGTN       50  0 BURLNGTN      170  0 ITASCA       380  0 WAYNE        465  1 PAWNEE       485  1 TAZEWELL
  45  0 BURLNGTN      165  0 KIRKLAND      175  0 JOLIET       213  0 LOCKPORT
  45  0 BURLNGTN      255  0 PLANO         180  0 KINCAID      198  0 LATHAM E
  45  0 BURLNGTN      300  0 RIDGEFLD      180  0 KINCAID      460  1 PANA
  50  0 BURLNGTN      380  0 WAYNE         180  0 KINCAID      465  1 PAWNEE
  55  0 BURNHAM        70  0 CALUMET       185  0 KIRKLAND     225  0 MCHENRY
  55  0 BURNHAM       200  0 LK.GEO TP     190  0 LASALLE      195  0 LASALLE
  55  0 BURNHAM       245  0 STATELINE     195  0 LASALLE      250  0 PLANO
  60  0 BYRON          65  0 BYRON         193  0 LATHAM E     275  0 PONITAC
  60  0 BYRON         185  0 KIRKLAND      195  0 LATHAM E     445  1 LATHAM EX
  60  0 BYRON         265  0 CAROL CO.     200  0 LK.GEO TP    450  1 LK.GEO
  65  0 BYRON          75  0 CHERRY        200  0 LK.GEO TP    480  1 ROXANA
  65  0 BYRON         235  0 NELSON        205  0 LIBERTY      310  0 RND.LKTP
  65  0 BYRON         385  0 KEMPLETN      205  0 LIBERTY      320  0 ROSECRAN
  70  0 CALUMET       345  0 STATELINE     210  0 LISLE        215  0 LOMBARD
  70  0 CALUMET       360  0 TAYLOR        210  0 LISLE        260  0 PLANO
  75  0 CHERRY        335  0 SILVER        213  0 LOCKPORT     215  0 LOMBARD
  75  0 CHERRY        385  0 KEMPLETN      213  0 LOCKPORT     230  0 MCCOOK
  80  0 COLLINS        85  0 COLLINS       213  0 LOCKPORT     390  0 WILL CO.
  80  0 COLLINS       255  0 PLANO         215  0 LOMBARD      245  0 N.A.L.
  80  0 COLLINS       400  0 WILTON        220  0 MANVILLE     275  0 PONITAC
  85  0 COLLINS       213  0 LOCKPORT      220  0 MANVILLE     280  0 POWERTON
  90  0 CRAWFORD      140  0 FISK          220  0 MANVILLE     485  1 TAZEWELL
  90  0 CRAWFORD      155  0 GOODINGS      225  0 MCHENRY      315  0 ROSECRAN
  95  0 CRETE         100  0 DAVIS CK      235  0 NELSON       240  0 NW S.W
  95  0 CRETE         125  0 E.FRANK       235  0 NELSON       295  0 QUAD CTY
  95  0 CRETE         200  0 LK.GEO TP     235  0 NELSON       370  0 WATERMAN
  95  0 CRETE         455  1 OLIVE         240  0 NW S.W       295  0 QUAD CTY
 105  0 DESPLAIN      150  0 GOLFMILL      250  0 N.BROOK      340  0 SKOKIE
 105  0 DESPLAIN      170  0 ITASCA        250  0 N.BROOK      375  0 WAUKGEAN
 105  0 DESPLAIN      215  0 LOMBARD       255  0 PLANO        260  0 PLANO
 105  0 DESPLAIN      285  0 PROSPECT      255  0 PLANO        260  0 PLANO
 110  0 DRESDEN       130  0 ELEC.JCT      255  0 PLANO        290  0 QUAD CTY
 110  0 DRESDEN       155  0 GOODINGS      265  0 CAROL CO.    290  0 QUAD CTY
 110  0 DRESDEN       220  0 MANVILLE      275  0 PONITAC      405  0 WILTON
 110  0 DRESDEN       275  0 PONITAC       280  0 POWERTON     355  0 STA.M
 115  0 DUNDEE        335  0 SILVER        280  0 POWERTON     485  1 TAZEWELL
 115  0 DUNDEE        380  0 WAYNE         290  0 QUAD CTY     295  0 QUAD CTY
 120  0 EARLVILE      145  0 GENESEO       295  0 QUAD CTY     420  1 BARSTON
 120  0 EARLVILE      185  0 KIRKLAND      295  0 QUAD CTY     435  1 DAVENPORT
 120  0 EARLVILE      190  0 LASALLE       300  0 RIDGEFLD     305  0 RIDGEFLD
```


185
315
300
320
205
60
105
335
170
65
45
380
120
290
225
255
190
195
260
215
35
40
130
175
213
80
145
400
405
95
125
365
280
275
455
200
55
155
220
110



1 471 RACINE
2 150
9 315
10 415
11 410 ZION
13 3-0
15 250
16 70 BURLNGTN
17 115
18 120 EARLVILE
19 255 PLANO
20 45
21 145
22 255 CAPIL CO.
23 40 BYRON
24 305 RIDGEFLD
25 335 SILVER
26 310
27 475 ROCKDALE
28 385
29 75 CHERRY
30 65 BYRON
31 420
32 435 DAVENPORT
33 235 NELSON
34 240
35 295 QUAD CTY
36 290 QUAD CTY
37 370 WATERMAN
38 80 COLLINS
39 85 COLLINS
40 230 MCCOOK
41 213 LOCKPORT
42 390 WILL CO.
43 135 ELMHURST
44 170 ITASCA
45 210 LISLE
46 130 ELEC.JCT
47 105 DESPLAIN
48 380 WAYNE
49 215 LOMBARD
50 245 N.A.L.
51 260 PLANO
52 5 ALSIP
53 10 ALSIP TP.
54 365
55 20 BEDFORD
56 155 GOODINGS
57 90 CRAWFORD
58 125 E.FRANK
59 175 JOLIET
60 145 GENESEO
61 425
62 350 STA.M
63 430 ESKAW EX
64 460 PANA
65 445 LATHAM EX
66 355 STA.M
67 280
68 110 DRESDEN
69 220 MANVILLE
70 485 TAZEWELL
71 455 PAWNEE
72 43
73 160 KINCAID
74 198 LATHAM E
75 275 PONTIAC
76 405 WILTON
77 140 FISK
78 360 TAYLOR
79 195 LASALLE
80 190 LASALLE
81 70 CALUMET
82 345 STATELINE
83 450 LK.GEO
84 25 BLOOM
85 55
86 200 LK.GEO TP
87 30 BLUE IS
88 480
89 35 BRAIDWOOD
90 40 BRAIDWOOD
91 100 DAVIS CK
92 95 CRETE
93 400 WILTON
94 440 SUMMIT
95 455 OLIVE


1 471 RACINE
2 170 [illegible]
3 15 ARTESIAN
4 [illegible] [illegible] FECT
5 [illegible] [illegible]
6 [illegible] [illegible] CRAN
7 315 [illegible]
8 415 ARCADIAN
9 410 ZION
10 160 HIGHLAND
11 140 [illegible]
12 375 WAUKEGAN
13 250 [illegible]
14 50 BURLNGTN
15 115 DUNDEE
16 255 PLANO
17 45 BURLNGTN
18 185 KIRKLAND
19 265 CARRL CO.
20 60 BYRON
21 305 [illegible]
22 335 SILVER
23 475 ROCKDALE
24 355 [illegible]PLETN
25 75 CHERRY
26 65 BYRON
27 420 BARSTON
28 435 DAVENPORT
29 235 NELSON
30 240 [illegible]
31 295 QUAD CTY
32 85 COLLINS
33 230 MCCOOK
34 213 LOCKPORT
35 135 ELMHURST
36 170 ITASCA
37 210 LISLE
38 130 ELEC.JCT
39 215 LOMBARD
40 245 N.A.L.
41 5 ALSIP
42 10 ALSIP TP.
43 365 [illegible]
44 20 BEDFORD
45 155 GIDDINGS
46 145 GENESEO
47 425 [illegible]
48 350 STA.M
49 430 [illegible] EX
50 450 PANA
51 445 LATHAM EX
52 355 STA.M
53 250 POWERTON
54 220 [illegible]VILLE
55 465 TAZEWELL
56 463 PAWNEE
57 43 [illegible]
58 150 KINCAID
59 198 LATHAM E
60 275 PONTIAC
61 140 FISK
62 360 TAYLOR
63 195 LASALLE
64 190 LASALLE
65 70 [illegible]
66 345 STATELINE
67 [illegible] [illegible]
68 25 BLOOM
69 55 BURNHAM
70 230 [illegible] TP
71 [illegible] [illegible]
72 35 BRAIDWOOD
73 40 BRAIDWOOD
74 105 DAVIS CK
75 95 CRETE
76 400 WILTON
77 440 [illegible]
78 455 CLINE
79 105 [illegible]
80 225 MCHENRY
81 [illegible] [illegible]FLO
82 310 [illegible]
83 290 QUAD CTY
84 370 WATERMAN
85 380 [illegible]
86 110 DRESDEN
87 390 WILL CO.
88 30 BLUE IS
89 50 [illegible]
90 125 [illegible]
91 405 [illegible]
92 80 COLLINS
93 120 [illegible]
94 260 PLANO
95 175 JOLIET

```
      PROGRAM JACOBI  ( INPUT,OUTPUT)
C***         THIS USES JACOBI ITERATIONS TO SOLVE THE TRANSIENT       *******
C****      STABLITY PROBLEM                                           *******
      COMPLEX C,V,Y,SUM,COX,ONE      ,P
      DIMENSION Y(100, 20),V(100),P(100),IB(10),C(100),NROW(100),IV(100,100)
     2 20) , SUM(100)     , ISV(20)
C****          DATA SHOULD BE INSERTED HERE                           *******
C****           ALL INFORMATION IS ASSUMED TO BE PRESENT AND FILLED IN
C *****                    BY THE DATA CARDS                          *******
C **          Y  IS STORED IN SPARSE FORM, THE FIRST ELEMENT OF       *******
C ***         EACH ROW IS THE INVERSE OF THE DIAGONAL ELEMENT         ******
C **         ALL OTHER NONZERO ELEMENTS ARE COMPRESSED TO THE NEXT    *******
C****         START OF PARALLEL ITERATIONS                           *******
C****          ALL PARPARATIONS MADE PRIOR TO ENTERING LOOP          ******
C****         TIME WILL BE ADVANCED BY CODE AFTER CONVERGENCE        *******
C****          CODE WILL REINTIALIZE FOR NEW TIME STEP               *******
      IBS = IB(L-1) + 1
      IBL = IB(L)
      IPNT = 0
      KNT = 0
C**          ALL PROCESSORS MUST SYNCHRONIZE AT THIS POINT IN EACH ITERATION *
      ICONV = 0
      CD01  = REAL(C(I))*SIN(TH01)  - AIMG(C(I))*COS(TH01)
      CQ01  = REAL(C(I))*COS(TH01) + AIMG(C(I))*SIN(TH01)
C******      GENERATOR STATE VARIABLES                                ******
      DTH01 = W01
      DW01 = RM01*(PM01        -(EQP01*CQ01+EDP01*CD01+XQXD01*CQ01*CD01)
     1 - D01*W01 )
      DEQ01 = TID01*(VFD1-EQ01-XDXD01*CD01 - S01 )
      DED01 = TIQ01*(XQXQ01*CQ01 - ED01 )
C******      GENERATOR STATE VARIABLES                                ******
      VD01 = ED01-RA01*CD01 + XQ01*CQ01
      VQ01 = EQ01 - RA01*CQ01 - XD01*CD01
      V(I) = CMPLX(VD01*SIN(TH01)+VQ01*COS(TH01), -VD01*COS(TH01)+VQ01*
     1 SIN(TH01) )
      V01 = SQRT(VD01*VD01 + VQ01*VQ01  ) .
```

```
***      EXCITER/REGULATOR STATE VARIABLES                                     ******
A01= TAI01*(VU01+VS01-V01-TFIK01*(VF01-VDE01) -VA01 )
~~F VA01 .LT. 0.0 .AND.DVA01 .LT. 0.0 ) DVA01 = 0.0~~
F/A01 .GT. VAMX01 .AND. DVA01 .GT. 0.0 ) DVA01 = 0.0
E01 = TFI01*(VF01-VDE01 )
~~= ABS(VF01) - ES01~~
F( TMPO .LT. 0.0 ) TMPO = 0.0
01 = TMPO*TMPO*CSI01*SGN(VF01)
~~01 = TEI01*(VA01-TKE01*VF01-SVF01)~~
***      EXCITOR/REGULATOR STATE VARIABLES                                     ******
1 = GK01*W01*TIT601
1 = TI201*(Y201-VS201)
~~1 = T3IT401*Y101 + OMTT01*VS101~~
***      SUPPLEMENTARY STATE VARIABLES                                         ******
201 = Y101
101 = T4I01*(Y101-VS101)
~~01 = Y001 - VS001*TI601~~
***      SUPPLEMENTARY STATE VARIABLES                                         ******
. = T501*Y001 + OMTT601*VS001
F VS01 .GT. VSM01) VS01 = VSM01
~~F VS01 .LT.-VSM01) VS01 =-VSM01~~
1. = P101 -TTK01*W01
~~2. = P201 + P001~~
F X201 .LT. 0.0 ) X201 = 0.0
  X201 .GT. PMX01 ) X201 = PMX01
**       TURBINE / GOVERNOR STATE VARIABLES                                    ******
~~1 =-TJI01*(CK101+X101 )~~
1 = T3I01*(X101-P201)
1 = TKI01*(OMK201*X201-P401)
*        TURBINE / GOVERNOR STATE VARIABLES                                    ******
~~4 = P401 +CK201*X201~~
*        NETWORK EQUATIONS                                                     *******
*        NROW STORES THE NUMBER OF NONZERO ELEMENTS OF EACH ROW   *******
* **       IV  THEN POINTS TO THE APPROPIATE NODE VOLTAGE FOR EACH      ******
*  NONZERO ENTRY IN Y                                                   ******
0 DO I = IBS,IBL
(I) = (0.0,0.0)
```

```
      NR = NROW(I)
      DO 115 J = 2,NR
      JJ = IV(I,J)
15    SUM(I) = SUM(I) +Y(I,J)*V(JJ)
      IF ( I .GT. LGEN ) GOTO 116
      C(I) = (SUM(I) + ONE ) /Y(I,1) + P(I) / CONJG(V(I))
16    SUM(I) = P(I)*Y(I,1)/CONJG(V(I)) - SUM(I)
10     CONTINUE

  ******        NETWORK EQUATIONS                                   *******

******          ALL PROCESSORS MUST SYNCHRONIZE AT THIS POINT IN EACH ITERATION *

  ******        UPDATE ALL VARIABLES                                *******

      DO 1000 I = IBS,IBL
      COX = W*SUM(I) + AW*V(I)
      IF ( CABS(COX) .GT. ERR) ICONV = 1
      V(I) = V(I) + COX
      IF ( I .GT. LNVR ) GOTO1000
      IS = ISV(I)
      V(IS) = V(I)
000    CONTINUE
      COR   =W*(THO1P + HH*DTHO1 ) + AW*THO1
      IF ( COR .GT.  ERR ) ICONV = 1
      THO1 = THO1 + COR
      COR = W*(WO1P + HH*DWO1) +AW*WO1
      IF ( COR .GT.  ERR ) ICONV = 1
      WO1 = WO1 + COR
      COR = W*(EQO1P + HH*DEQO1 ) + AW*EQO1
      IF ( COR .GT.  ERR ) ICONV = 1
      EQO1 = EQO1 + COR
      COR = W*(EDO1P + HH*DEDO1) + AW*EDO1
      IF ( COR .GT.  ERR ) ICONV = 1
      EDO1 = EDO1 + COR
      VDO1 = EDO1 - RAO1 * CDO1 + XQO1 * CQO1
      VQO1 = EQO1 - RAO1 * CQO1 - XDO1 * CDO1
      V(1) = CMPLX(VDO1*SIN(THO1)+VQO1*COS(THO1),VQO1*SIN(THO1)-VDO1*COS
     1 (THO1 ) )
      COR = W*(VAO1P + HH*DVAO1) + AW*VAO1
      IF ( COR .GT.  ERR ) ICONV = 1
      VAO1 = VAO1 + COR
      COR = W*(VDEO1P + HH*DVDEO1) + AW*VDEO1
      IF ( COR .GT.  ERR ) ICONV = 1
      VDEO1 = VDEO1 + COR
      COR = W*(VFO1P + HH*DVFO1 )  + AW*VFO1
      IF ( COR .GT.  ERR ) ICONV = 1
      VFO1 = VFO1 + COR
      COR = W*(VS2O1P + HH*DVS2O1) + AW*VS2O1
      IF ( COR .GT.  ERR ) ICONV = 1
      VS2O1 = VS2O1 + COR
      COR = W*(VS1O1P + HH*DVS1O1 ) + AW*VS1O1
      IF ( COR .GT.  ERR ) ICONV = 1
      VS1O1 = VS1O1 + COR
      COR = W*(VSOO1P + HH*DVSOO1 ) + AW*VSOO1
```

```
      IF ( COR .GT.  ERR ) ICONV = 1
      VS001 = VS001 + COR
      COR = W*(P101P + HH*DP101 ) +  AW*P101
      IF ( COR .GT.  ERR ) ICONV = 1
      P101 = P101 + COR
      COR = W*(P201P + HH*DP201 ) +  AW*P201
      IF ( COR .GT.  ERR ) ICONV = 1
      P201 = P201 + COR
      IF ( COR .GT.  ERR ) ICONV = 1
      COR = W*(P401P + HH*DP401 ) +  AW*P401
      P401 = P401 + COR
C******      UPDATE ALL VARIABLES                                    *******

      KNT = KNT + 1
      IF ( ICONV .NE. 0 )  GOTO 50
C******      SET CONVERGENCE FLAG TO CONTROLLER                      *********
C******      IF ALL DO NOT SIGNAL CONVERGENCE    RETURN TO 50        *********
      IF(KNT .GT. 50 ) GOTO 900

C******      ADVANCE TIME STEP                                       ******

      TMPO = TH01
      TH01 = TH01 + 2.*HH*DTH01
      TH01P = TMPO + HH*DTH01
      TMPO = W01
      W01 = W01 + 2.*HH*DW01
      W01P = TMPO + HH*DW01
      TMPO = EQ01
      EQ01 = EQ01 + 2.*HH*DEQ01
      EQ01P = TMPO + HH*DEQ01
      TMPO = ED01
      ED01 = ED01 + 2.*HH*DED01
      ED01P = TMPO + HH*DED01
      TMPO = VA01
      VA01 = VA01 + 2.*HH*DVA01
      VA01P = TMPO + HH*DVA01
      TMPO = VDE01
      VDE01 = VDE01 + 2.*HH*DVDE01
      VDE01P = TMPO + HH*DVE01
      TMPO = VF01
      VF01 = VF01 + 2.*HH*DVF01
      VF01P = TMPO + HH*DVF01
      TMPO = VS201
      VS201 = VS201 + 2.*HH*DVS201
      VS201P = TMPO + HH*DVS201
      TMPO = VS101
      VS101 = VS101 + 2.*HH*DVS101
      VS101P = TMPO + HH*DVS101
      TMPO = VS001
      VS001 = VS001 + 2.*HH*DVS001
      VS001P = TMPO + HH*DVS001
      TMPO = P101
      P101 = P101 + 2.*HH*DP101
      P101P = TMPO + HH*DP101
```

```
      TMPO = P201
      P201 = P201 + 2.*HH*DP201
      P201P = TMPO + HH*DP201
      TMPO = P401
      P401 = P401 + 2.*HH*DP401
       PM01 = P401 + CK201*(P201 +P001 )
C
C   ******      ADVANCE TIME STEP                                    ******
C
       IF(IPNT . LT. NPNT ) GOTO 2
       GOTO 1

       END
```

```
      SUBROUTINE CHAOTIC ( C,V,Y,P  )
C
C  ******       THIS USES CHAOTIC RELAXATION TO SOLVE THE TRANSIENT       ******
C  *******         STABLITY PROBLEM                                       *******
      COMPLEX C,V,Y,SUM,COX,ONE      ,P
      DIMENSION Y(100, 20),V(100),P(100),IB(10),C(100),NROW(100),IV(100,100)
     1  20) ,   ISV(20 )
C
C  *******        DATA SHOULD BE INSERTED HERE                            *******
C  ******           ALL INFORMATION IS ASSUMED TO BE PRESENT AND FILLED IN
C  ********                BY THE DATA CARDS                              *******
C
C
C  *****         Y  IS STORED IN SPARSE FORM, THE FIRST ELEMENT OF        *******
C  ******        EACH ROW IS THE INVERSE OF THE DIAGONAL ELEMENT          ******
C  *****       A LL OTHER NONZERO ELEMENTS ARE COMPRESSED TO THE NEXT     *******
C
C   ******        START OF PARALLEL ITERATIONS                          *******
C   ******         ALL PARPARATIONS MADE PRIOR TO ENTERING LOOP         ******
C   ******        TIME WILL BE ADVANCED BY CODE AFTER CONVERGENCE       *******
C   ******         CODE WILL REINTIALIZE FOR NEW TIME STEP              *******
C
      IBS = IB(L-1) + 1
      IBL = IB(L)
1     IPNT = 0
2     KNT = 0
50    ICONV = 0
        CD01 = REAL(C(I))*SIN(THO1) - AIMG(C(I))*COS(THO1)
        CQ01 = REAL(C(I))*COS(THO1) + AIMG(C(I))*SIN(THO1)
C
C   *******     GENERATOR STATE VARIABLES                               ******
C
      DTHO1 = WO1
      COR    =W*(THO1P + HH*DTHO1 ) + AW*THO1
      IF ( COR .GT.  ERR ) ICONV = 1
      THO1 = THO1 + COR
      DWO1 = RMO1*(PMO1      -(EQPO1*CQ01+EDPO1*CD01+XQXD01*CQ01*CD01)
     )  - DO1*WO1 )
      COR = W*(WO1P + HH*DWO1) +AW*WO1
      IF ( COR .GT.  ERR ) ICONV = 1
      WO1 = WO1 + COR
      DEQO1 = TIDO1*(VFO1-EQO1-XDXD01*CDO1 - SO1 )
      COR = W*(EQO1P + HH*DEQO1 ) + AW*EQO1
      IF ( COR .GT.  ERR ) ICONV = 1
       EQO1 = EQO1 + COR
      DEDO1 = TIQO1*(XQXQ01*CQO1 - EDO1 )
      COR = W*(EDO1P + HH*DEDO1) + AW*EDO1
      IF ( COR .GT.  ERR ) ICONV = 1
      EDO1 = EDO1 + COR
C
C   *******     GENERATOR STATE VARIABLES                               ******
C
```

```
SUBROUTINE  CHAOTIC   NORMAL                               CDC 6600 FTN V3.0-P336 OPT=2  07/1
              VD01 = ED01-RA01*CD01 + XQ01*CQ01
              VQ01 = EQ01 - RA01*CQ01 - XD01*CD01
              V(I) = CMPLX(VD01*SIN(TH01)+VQ01*COS(TH01),-VD01*COS(TH01)+VQ01*
             1  SIN(TH01) )
              V01 = SQRT(VD01*VD01 + VQ01*VQ01 )
        C
        C     *******      EXCITER/REGULATOR STATE VARIABLES                             ******
        C
              DVA01= TAI01*(V001+VS01-V01-TFIK01*(VF01-VDE01) -VA01 )
              IF( VA01 .LT. 0.0 .AND.DVA01 .LT. 0.0 ) DVA01 = 0.0
              IF(VA01 .GT. VAMX01 .AND. DVA01 .GT. 0.0 ) DVA01 = 0.0
              COR = W*(VA01P + HH*DVA01) + AW*VA01
              IF ( COR .GT.  ERR ) ICONV = 1
              VA01 = VA01 + COR
              DVDE01 = TFI01*(VF01-VDE01 )
              COR = W*(VDE01P + HH*DVDE01) + AW*VDE01
              IF ( COR .GT.  ERR ) ICONV = 1
              VDE01 = VDE01 + COR
              TMPO = ABS(VF01) - ES01
              IF ( TMPO .LT. 0.0 ) TMPO = 0.0
              SVF01 = TMPO*TMPO*CSI01*SGN(VF01)
              DVF01 = TFI01*(VA01-TKE01*VF01-SVF01)
              COR = W*(VF01P + HH*DVF01 )  + AW*VF01
              IF ( COR .GT.  ERR ) ICONV = 1
              VF01 = VF01 + COR
        C
        C     *******      EXCITOR/REGULATOR STATE VARIABLES                             ******
        C
              Y201 = GK01*W01*TIT601
              Y101 = T1201*(Y201-VS201)
              Y001 = T3IT401*Y101 + OMTT01*VS101
        C
        C     *******      SUPPLEMENTARY STATE VARIABLES                                 ******
        C
              DVS201 = Y101
              COR = W*(VS201P + HH*DVS201) + AW*VS201
              IF ( COR .GT.  ERR ) ICONV = 1
              VS201 = VS201 + COR
              DVS101 = T4I01*(Y101-VS101)
              COR = W*(VS101P + HH*DVS101 ) + AW*VS101
              IF ( COR .GT.  ERR ) ICONV = 1
              VS101 = VS101 + COR
              DVS001 = Y001 - VS001*TI601
              COR = W*(VS001P + HH*DVS001 ) + AW*VS001
              IF ( COR .GT.  ERR ) ICONV = 1
              VS001 = VS001 + COR
        C
        C     *******      SUPPLEMENTARY STATE VARIABLES                                 ******
        C
              VS01 = T501*Y001 + OMTT601*VS001
              IF ( VS01 .GT. VSM01) VS01 = VSM01
              IF ( VS01 .LT.-VSM01) VS01 =-VSM01
        C
              X101 = P101 -TTK01*W01
              X201 = P201 + P001
```

```
            IF ( X201 .LT. 0.0 ) X201 = 0.0
            IF ( X201 .GT. PMX01 ) X201 = PMX01
      C
      C   ******       TURBINE / GOVERNOR STATE VARIABLES                     ******
      C
            DP101 =-TU101*(CK101+X101 )
            COR = W*(P101P + HH*DP101 ) +  AW*P101
            IF ( COR .GT.  ERR ) ICONV = 1
            P101 = P101 + COR
            DP201 = T3101*(X101-P201)
            COR = W*(P201P + HH*DP201 ) +  AW*P201
            IF ( COR .GT.  ERR ) ICONV = 1
            P201 = P201 + COR
            DP401 = TK101*(CMK201*X201-P401)
            COR = W*(P401P + HH*DP401 ) +  AW*P401
            IF ( COR .GT.  ERR ) ICONV = 1
            P401 = P401 + COR
      C
      C   ******       TURBINE / GOVERNOR STATE VARIABLES                     ******
      C
            PM01 = P401 +CK201*X201
      C
      C   ******       NETWORK EQUATIONS                                       *******
      C
      C
      C  *******       NROW STORES THE NUMBER OF NONZERO ELEMENTS OF EACH ROW  *******
      C
      C  ********      IV  THEN POINTS TO THE APPROIATE NODE VOLTAGE FOR EACH  ******
      C  *****     NONZERO ENTRY IN Y                                          ******
      C
            DO 100 I = IBS,IBL
            SUM = (0.0 , 0.0 )
            NR = NROW(I)
            DO 115 J = 2,NR
            JJ = IV(I,J)
        115 SUM = SUM + Y(I,J) * V(JJ)
            IF ( I .GT. LGEN ) GOTO 116
            C(I) = ( SUM + ONE )/Y(I,1) + P(I)/CONJG(V(I))
        116 SUM = P(I)*Y(I,1)/CONJG(V(I)) - SUM
            COR = W*SUM-AW*V(I)
            IF( CABS(COR) .GT. ERR ) ICONV = 1
            V(I) = V(I) + COR
            IF ( I .GT. LNVR ) GOTO 100
            IS = ISV(I)
            V(IS) = V(I)
      100   CONTINUE
      C
      C   ******       NETWORK EQUATIONS                                       *******
      C
      C
      C
      C
            KNT = KNT + 1
            IF ( ICONV .NE. 0 )  GOTO 50
      *
```

```
     ********           SET CONVERGENCE FLAG TO CONTROLLER                             *********
     ********            IF ALL DO NOT SIGNAL CONVERGENCE       RETURN TO 50           *********
     C
     C
     C   ******          ADVANCE TIME STEP                                             ******
     C
           TMPO = THO1
           THO1 = THO1 + 2.*HH*DTHO1
           THO1P = TMPO + HH*DTHO1
           TMPO = WO1
           WO1 = WO1 + 2.*HH*DWO1
           WO1P = TMPO + HH*DWO1
           TMPO = EQO1
           EQO1 = EQO1 + 2.*HH*DEQO1
           EQO1P = TMPO + HH*DEQO1
           TMPO = EDO1
           EDO1 = EDO1 + 2.*HH*DEDO1
           EDO1P = TMPO + HH*DEDO1
           VDO1 = EDO1 - RAO1 * CDO1 + XQO1 * CQO1
           VQO1 = EQO1 - RAO1 * CQO1 - XDO1 * CDO1
           V(I) = CMPLX(VDO1*SIN(THO1)+VQO1*COS(THO1),VQO1*SIN(THO1)-VDO1*COS
          1 (THO1 ) )
           TMPO = VAO1
           VAO1 = VAO1 + 2.*HH*DVAO1
           VAO1P = TMPO + HH*DVAO1
           TMPO = VDEO1
           VDEO1 = VDEO1 + 2.*HH*DVDEO1
           VDEO1P = TMPO + HH*DVEO1    DVDEO1
           TMPO = VFO1
           VFO1 = VFO1 + 2.*HH*DVFO1
           VFO1P = TMPO + HH*DVFO1
           TMPO = VS2O1
           VS2O1 = VS2O1 + 2.*HH*DVS2O1
           VS2O1P = TMPO + HH*DVS2O1
           TMPO = VS1O1
           VS1O1 = VS1O1 + 2.*HH*DVS1O1
           VS1O1P = TMPO + HH*DVS1O1
           TMPO = VSOO1
           VSOO1 = VSOO1 + 2.*HH*DVSOO1
           VSOO1P = TMPO + HH*DVSOO1
           TMPO = P1O1
           P1O1 = P1O1 + 2.*HH*DP1O1
           P1O1P = TMPO + HH*DP1O1
           TMPO = P2O1
           P2O1 = P2O1 + 2.*HH*DP2O1
           P2O1P = TMPO + HH*DP2O1
           TMPO = P4O1
           P4O1 = P4O1 + 2.*HH*DP4O1
           P4O1P = TMPO + HH*DP4O1
           PMO1 = P4O1 + CK2O1*(P2O1 +POO1 )
     C
     C   ******          ADVANCE TIME STEP                                            ******
     *******           ALL PROCESSORS MUST SYNCHRONIZE AT THIS POINT IN EACH ITERATION *
     C
           IF(IPNT . LT. NPNT ) GOTO 2
```

```
      SUBROUTINE GUASS(P,V,C,Y,IB)
C
C
C  ******         THIS USES THE BORDERED BLOCK DIAGONAL FORM TO ALLOW      *******
C  *****          GUASS SEIDEL ITERATIONS TO BE PERFORMED IN PARALLEL      *******
C
      COMPLEX C,V,Y,SUM,COX,ONE ,TV ,P
      DIMENSION Y(100, 20),V(100),P(100),IB(10),C(100),NROW(100),IV(100,100)
     1 20),TV(20,20),NDR(100),IVP(20,20)
C
C  *******         DATA SHOULD BE INSERTED HERE                            *******
C  ******           ALL INFORMATION IS ASSUMED TO BE PRESENT AND FILLED IN
C  ********                BY THE DATA CARDS                               *******
C
C
C  *****         Y  IS STORED IN SPARSE FORM, THE FIRST ELEMENT OF         *******
C  ******        EACH ROW IS THE INVERSE OF THE DIAGONAL ELEMENT           ******
C  *****        A LL OTHER NONZERO ELEMENTS ARE COMPRESSED TO THE NEXT     *******
C
C
C   ******        START OF PARALLEL ITERATIONS                             *******
C   ******         ALL PREPARATIONS MADE PRIOR TO ENTERING LOOP            ******
C   ******        TIME WILL BE ADVANCED BY CODE AFTER CONVERGENCE          *******
C   ******         CODE WILL REINTIALIZE FOR NEW TIME STEP                 *******
C
      IBE = IB(N)
      IBN = IB(N-1) + 1
      IBS = IB(I-1) +1
      IBI = IB(I)
1     IPNT = 0
2     KNT = 0
50    ICONV = 0
        CD01 = REAL(C(I))*SIN(TH01) - AIMG(C(I))*COS(TH01)
        CQ01 = REAL(C(I))*COS(TH01) + AIMG(C(I))*SIN(TH01)
C
C   *******      GENERATOR STATE VARIABLES                                 ******
C
      DTH01 = W01
      COR   =W*(TH01P + HH*DTH01 ) + AW*TH01
      IF ( COR .GT.  ERR ) ICONV = 1
      TH01 = TH01 + COR
      DW01 = RM01*(PM01      -(EQP01*CQ01+EDP01*CD01+XQXD01*CQ01*CD01)
     1  - DD1*W01 )
      COR = W*(W01P + HH*DW01) +AW*W01
      IF ( COR .GT.  ERR ) ICONV = 1
      W01 = W01 + COR
      DEQ01 = TID01*(VF01-EQ01-XDXD01*CD01 - S01 )
      COR = W*(EQ01P + HH*DEQ01 ) + AW*EQ01
      IF ( COR .GT.  ERR ) ICONV = 1
       EQ01 = EQ01 + COR
      DED01 = TIQ01*(XQXQ01*CQ01 - ED01 )
      COR = W*(ED01P + HH*DED01) + AW*ED01
      IF ( COR .GT.  ERR ) ICONV = 1
      ED01 = ED01 + COR
C
```

```
BROUTINE  GUASS      NORMAL                        CDC 6600 FTN V3.0-P336 OPT=2  07/21/76
      C    *******    GENERATOR STATE VARIABLES                                    ******
      C
           VD01 = ED01-RA01*CD01 + XQ01*CQ01
           VQ01 = EQ01 - RA01*CQ01 - XD01*CD01
           V(I) = CMPLX(VD01*SIN(TH01)+VQ01*COS(TH01),-VD01*COS(TH01)+VQ01*
          1  SIN(TH01) )
           V01 = SQRT(VD01*VD01 + VQ01*VQ01 )
      C
      C    *******    EXCITER/REGULATOR STATE VARIABLES                            ******
      C
           DVA01= TAI01*(VQ01+VS01-V01-TFIK01*(VF01-VDE01) -VA01 )
           IF( VA01 .LT. 0.0 .AND.DVA01 .LT. 0.0 ) DVA01 = 0.0
           IF(VA01 .GT. VAMX01 .AND. DVA01 .GT. 0.0 ) DVA01 = 0.0
           COR = W*(VA01P + HH*DVA01) + AW*VA01
           IF ( COR .GT.  ERR ) ICONV = 1
           VA01 = VA01 + COR
           DVDE01 = TFI01*(VF01-VDE01 )
           COR = W*(VDE01P + HH*DVDE01) + AW*VDE01
           IF ( COR .GT.  ERR ) ICONV = 1
           VDE01 = VDE01 + COR
           TMP0 = ABS(VF01) - ES01
           IF ( TMP0 .LT. 0.0 ) TMP0 = 0.0
           SVF01 = TMP0*TMP0*CSI01*SGN(VF01)
           DVF01 = TEI01*(VA01-TKE01*VF01-SVF01)
           COR = W*(VF01P + HH*DVF01 )  + AW*VF01
           IF ( COR .GT.  ERR ) ICONV = 1
           VF01 = VF01 + COR
      C
      C    *******    EXCITOR/REGULATOR STATE VARIABLES                            ******
      C
           Y201 = GK01*W01*TIT601
           Y101 = TI201*(Y201-VS201)
           Y001 = T3IT401*Y101 + OMTT01*VS101
      C
      C    *******    SUPPLEMENTARY STATE VARIABLES                                ******
      C
           DVS201 = Y101
           COR = W*(VS201P + HH*DVS201) + AW*VS201
           IF ( COR .GT.  ERR ) ICONV = 1
           VS201 = VS201 + COR
           DVS101 = T4I01*(Y101-VS101)
           COR = W*(VS101P + HH*DVS101 ) + AW*VS101
           IF ( COR .GT. ERR ) ICONV = 1
           VS101 = VS101 + COR
           DVS001 = Y001 - VS001*TI601
           COR = W*(VS001P + HH*DVS001 ) + AW*VS001
           IF ( COR .GT.  ERR ) ICONV = 1
           VS001 = VS001 + COR
      C
      C    *******    SUPPLEMENTARY STATE VARIABLES                                ******
      C
           VS01 = T501*Y001 + OMTT601*VS001
           IF ( VS01 .GT. VSM01) VS01 = VSM01
           IF ( VS01 .LT.-VSM01) VS01 =-VSM01
           X101 = P101 -TIK01*W01
```

```
      X201 = P201 + P001
      IF ( X201 .LT. 0.0 ) X201 = 0.0
      IF ( X201 .GT. PMX01 ) X201 = PMX01
C
C   ******      TURBINE / GOVERNOR STATE VARIABLES                    ******
C
      DP101 =-THT01*(CK101+X101 )
      COR = W*(P101P + HH*DP101 ) +  AW*P101
      IF ( COR .GT.  ERR ) ICONV = 1
      P101 = P101 + COR
      DP201 = T3T01*(X101-P201)
      COR = W*(P201P + HH*DP201 ) +  AW*P201
      IF ( COR .GT.  ERR ) ICONV = 1
      P201 = P201 + COR
      DP401 = TKT01*(OMK201*X201-P401)
      COR = W*(P401P + HH*DP401 ) +  AW*P401
      IF ( COR .GT.  ERR ) ICONV = 1
      P401 = P401 + COR
C
C   ******      TURBINE / GOVERNOR STATE VARIABLES                    ******
C
      PM01 = P401 +CK201*X201
C
C   ******      NETWORK EQUATIONS                                     *******
C
C
C  *******      NROW STORES THE NUMBER OF NONZERO ELEMENTS OF EACH ROW  *******
C
C   ********     IV  ~~THEN POINTS~~ TO THE APPROIATE NODE VOLTAGE FOR EACH   ******
C  *****     NONZERO ENTRY IN Y                                        ******
C
C
*******         THE NETWORK EQUATIONS ARE SOLVED BY BLOCKS              ********
********          THE FIRST BLOCK IS THE BLOCK LAST ROW FOR THE INTERMIDATE  ****
*******           RESULTS THAT WILL BE SUMMED BY THE LAST PROCESSOR     *********
C
      DO 105  J = IBN,IBE        IF ( NROW(J) .EQ. 0 ) GOTO 105
      SUM = (0.,0.)
      NR = NROW(J)
      DO 110 L = 1,NR
      JJ = IV(J,L)
110   SUM = SUM + Y(J,L)*V(JJ)
105   TV(J,I) = SUM
C  105 CONTINUE
C*******        LAST BLOCK PARTIALLY SOLVED BY EACH PROCESSOR           ******
C
C
********          THE PROCESSOR ASSIGNED TO THE LAST BLOCK MUST SUM       ********
*******           ALL INTERMIDATE RESULTS FROM ALL PROCESSORS TO          *******
********          FIND THE ITERATION RESULTS FOR THE CUT SET VARIABLES     ********
C
******      SYNCHRONIZATION REQUIRED HERE
C
********          THE NEXT SECTION OF CODE APPLIES ONLY TO THE PROCESSOR  ********
********          ASSIGNED TO THE CUT SET VARIABLES                       *********
```

```
C
      DO 200 J = IBN , IBE
      NR = NDR(J)
      DO 205 L = 2,NR
      LL = IV(J,L)
205   SUM = SUM + Y(J,L)*V(LL)
      NR = NROW(J)
      DO 210 L = 1,NR
      LL = IVP(J-IBN,L)
      SUM = SUM + TV(J,LL)
210   TV(J,LL) = MINF
      SUM = P(I)*Y(I,1)/CONJG(V(I)) - SUM
      COX = W*SUM-AW*V(I)
      IF( CABS(COX) .GT. ERR ) ICONV = 1
200   V(J) = V(J) + COX
C
*******       END OF CODE EXCLUSIVELY FOR PROCESSSOR ASSIGNED TO CUT SET   *****
C
       DO 300 J = IBS  , IBI
      SUM = (0.,0.)
      NR = NROW(J)
      DO 305 L = 2,NR
      JJ = IV(J,L)
305   SUM = SUM + Y(J,L)*V(JJ)
      IF ( I .GT. LGEN ) GOTO 311
      C(J) = (SUM + ONE)/Y(J,1) + P(J)/CONJG(V(J))
311   SUM = P(I)*Y(I,1)/CONJG(V(I)) - SUM
      COX = W*SUM-AW*V(I)
      IF( CABS(COX) .GT. ERR ) ICONV = 1
300   V(J) = V(J) + COX
C
******      SYNCHRONIZATION REQUIRED HERE
C   ******        NETWORK EQUATIONS                                   *******
C
      KNT = KNT + 1
      IF ( ICONV .NE. 0 )  GOTO 50
C
C   ******        ADVANCE TIME STEP                                   ******
C
      TMPO = TH01
      TH01 = TH01 + 2.*HH*DTH01
      TH01P = TMPO + HH*DTH01
      TMPO = W01
      W01 = W01 + 2.*HH*DW01
      W01P = TMPO + HH*DW01
      TMPO = EQ01
      EQ01 = EQ01 + 2.*HH*DEQ01
      EQ01P = TMPO + HH*DEQ01
      TMPO = ED01
      ED01 = ED01 + 2.*HH*DED01
      ED01P = TMPO + HH*DED01
      VD01 = ED01 - RA01 * CD01 + XQ01 * CQ01
      VQ01 = EQ01 - RA01 * CQ01 - XD01 * CD01
      V(I) = CMPLX(VD01*SIN(TH01)+VQ01*COS(TH01),VQ01*SIN(TH01)-VD01*COS
     1 (TH01 ) )
```

```
          TMPO = VA01
          VA01 = VA01 + 2.*HH*DVA01
          VA01P = TMPO + HH*DVA01
          TMPO = VDE01
          VDE01 = VDE01 + 2.*HH*DVDE01
          VDE01P = TMPO + HH*DVE01
          TMPO = VF01
          VF01 = VF01 + 2.*HH*DVF01
          VF01P = TMPO + HH*DVF01
          TMPO = VS201
          VS201 = VS201 + 2.*HH*DVS201
          VS201P = TMPO + HH*DVS201
          TMPO = VS101
          VS101 = VS101 + 2.*HH*DVS101
          VS101P = TMPO + HH*DVS101
          TMPO = VS001
          VS001 = VS001 + 2.*HH*DVS001
          VS001P = TMPO + HH*DVS001
          TMPO = P101
          P101 = P101 + 2.*HH*DP101
          P101P = TMPO + HH*DP101
          TMPO = P201
          P201 = P201 + 2.*HH*DP201
          P201P = TMPO + HH*DP201
          TMPO = P401
          P401 = P401 + 2.*HH*DP401
          P401P = TMPO + HH*DP401
           PM01 = P401 + CK201*(P201 +P001 )
      C
      C
      C   ******      ADVANCE TIME STEP                                     ******
      C
          IF(IPNT . LT. NPNT ) GOTO 2
          GOTO 1
          RETURN

          END
F CHANGES MADE BY THE OPTIMIZER
S OF INVARIANT RLIST REMOVED FROM THE LOOP STARTING AT LINE   173
```

```
      *
      *        NEWTON-SOR ALGORITHM
      *
            SUBROUTINE NSOR ( X,Y,H,ICNTR)
      *
          DIMENSION X(300),Y(100,100),Z(300,300),F(300),D(100),DX(300),DP(
         2 300),NROW(100),IV(100,20),INC(100),IRS(100),IRN(100),IREV(100)
         3 ,LV(100),OTHER(2600)
      *
      ********        THIS ALGORITHM SOVES THE DYNAMIC SIMULATION PROBLEM
      ********          BY IMPLICIT INTERGRATION AND NEWTON UPATING OF THE
      ********          VARIABLES.
      *
      ********        X IS A VECTOR OF THE VARIABLES OF THE MODEL BOTH STATE
      ********            AND ALGEBRAIC.  Y IS THE BUS ADMITTANCE MATRIX ,
      ********            Z IS THE JACOBIAN OF THE EQUATIONS.  F IS THE
      ********            FUNCTIONAL ERROR OF THE VARIABLES, D THE DERIVATIVE
      ********            OF THE STATE VARIABLES, DP THE PAST DERIVATIVE.
      *
      ********        OPTIMAL ORDERING IS IMPORTANT SINCE THE JACOBIAN MUST BE
      ********          SOLVED .  THE ORDERING STATES WITH THE TURBINE/GOVENROR
      ********          THEN TO THE SUPPLEMENTAL AND REGULATOR, FINALLY TO THE
      ********          MACHINE VARIABLES.  NEXT ARE THE CURRENT EQ AFTER ALL
      ********          OF THE GENERATOR MODELS, FINALLY THE VOLTAGE LAST
      *
        1     T = 0.
        2     B = 1./(80*H)
              KNT = 0
        5     ICONV = 0
              KNT = KNT + 1
      *
      ********        TURBINE GOVERNOR VARIABLES
      *
      ********        SYNCHRONIZATION REQUIRED HERE FOR ALL PROCESSSORS
      *
              SN01 = SIN(X(10) )
              CS01 = COS(X(10) )
              CD01 = X(14)*SN01-X(15)*CS01
              CQ01 = X(14)*CS01 +X(15)*SN01
              VD01 = X(11) - RA01*CD01 + XQ01*CQ01
              VQ01 = X(13) -RA01*CQ01 -XD01*CD01
              Z(1,1) = TI5G01-B
              Z(1,3) = TIKG01
              X2G01 = X(3) + PC01
              IF( X2G01 .LT. 0.0 ) X2G01 = 0.0
              IF ( X2G01 .GT. PMAX01) X2G01 = PMAX01
              D(1) = Z(1,3)*X2G01 - TI5G01*X(1)
              F(1) = DP(1) - D(1)
              DP(1) = D(1)
              IF(ABS(F(1)) .GT. ERR ) ICONV = 1
              Z(2,2) =    -TI1G01 - B
              Z(2,12) = -TI1G01*G1K01*OMIT01
              D(2) = Z(2,12)*X(12)-TI1G01*X(2)
              F(2) = DP(2) - D(2)
              DP(2) = D(2)
```

```
      IF(ABS(F(2)).GT. ERR ) ICONV = 1
      X1G01 = X(2)-G1K01*T2IT101*X(12)
      Z(3,2) = TI3G01
      Z(3,3) = -TI3G01 - B
      Z(3,12) = TKIT01
      D(3) = TI3G01*(X1G01-X(3) )
      F(3) = DP(3) - D(3)
      DP(3) = D(3)
      IF(ABS(F(3)).GT. ERR ) ICONV = 1
      PM01 = X(1) + G2K01*X2G01
C
C*******        SUPPLEMENTAL VARIABLES
C
      Z(4,4) = -T2T601 - B
      Z(4,12) = GKT2T6
      D(4) = TI2S01*(Z(4,12)*X(12)-X(4) )
      F(4) = DP(4) - D(4)
      DP(4) = D(4)
      IF(ABS(F(4)).GT. ERR ) ICONV = 1
      Z(5,4) = -TI2401
      Z(5,5) = -TI401 - B
      Z(5,12) = TKGTT01
      D(5) = Z(5,12)*X(12) -TI2401*X(4) -TI401*X(5)
      F(5) = DP(5) - D(5)
      DP(5) = D(5)
      IF(ABS(F(5)).GT. ERR ) ICONV = 1
      Z(6,4) = -T3TT01
      Z(6,5) = OMTT401
      Z(6,6) = -TI601 - B
      D(6) = -TI601*X(6) + OMTT401*X(5) -T3TT01*X(4)
      F(6) = DP(6) - D(6)
      DP(6) = D(6)
      IF(ABS(F(6)).GT. ERR ) ICONV = 1
      VS01 = T5S01*(T3IT401*D(4)+OMTT01*X(5))+OMTT601*X(6)
      IF (VS01 .GT. VSMX01 ) VS01 = VSMX01
      IF ( VS01 .LT. VSMN01 ) VS01 = VSMN01
C
C*******        REGULATOR VARIABLES
C
      VM01 = (X(IVG01)+RT01*X(14)-XT01*X(15))**2+(X(IVG01+1)+RT01*X(15)+
     2 XT01*X(14) )**2
      VM01 = SQRT(VM01)
      Z(7,7) = TIF01 - B
      Z(7,8) = -TIF01
      D(7) = TIF01*(X(7) - X(8) )
      F(7) = DP(7) - D(7)
      DP(7) = D(7)
      IF(ABS(F(7)).GT. ERR ) ICONV = 1
      TMP = ABS(X(8)) - ESR01
      IF ( TMP .LT. 0.0 ) TMP = 0.0
      SVF01 = SIGN(TMP*TMP*CSI01,X(8) )
      PSE801 = SIGN(2.*X(8)*CSI01,X(8)    )
      IF( TMP .LT. 0.0 ) PSE801 = 0.0
      Z(8,8) = TIE01*(EK01-PSE801) - B
      Z(8,9) = TIE01
```

```
          D(8) =  TIF01*(X(9)-EK01*X(8) -SVF01)
          F(8) = DP(8) - D(8)
          DP(8) = D(8)
          IF(ABS(F(8)).GT. ERR ) ICONV = 1
          Z(9,4) = -TZS01
          Z(9,5) = TOMT01
          Z(9,6) = TZOMT
          Z(9,7) =-FKTTF01
          Z(9,8) = FKTTF01
          Z(9,9) = TIA01 - B
          Z(9,12) = TZKG01
          Z(9,14)=-TIA01*(X(IVG01)*RT01+X(IVG01+1)*XT01+RTXT01*X(14) )/ VM01
          Z(9,15) = -TIA01*(X(IVG01+1)*RT01-X(IVG01)*XT01+RTXT01*X(15))/VM01
          Z(9,IVG01) = -TIA01*(X(IVG01) +RT01*X(14)-XT01*X(15) )/VM01
          Z(9,IVG01+1) = -TIA01*(X(IVG01+1)+RT01*X(15)+XT01*X(14) )/VM01
          E01 = VJ01 + VS01-V01- FKTTF01*(X(8)-X(7))-X(9)
          IF ( X(9) .LT. VMN01 .AND. E01 .LT. 0.0 ) E01 = 0.0
          IF ( X(9) .GT. VMX01 .AND. E01 .GT. 0.0 ) E01 = 0.0
          D(9) = TIA01*E01
          F(9) = DP(9) - D(9)
          DP(9) = D(9)
          IF(ABS(F(9)).GT. ERR ) ICONV = 1
    *
    *******         GENERATOR VARIABLES
    *
          Z(10,10) = -B
          Z(10,12) = 1.
          D(10) = X(12)
          F(10) = DP(10) - D(10)
          DP(10) = D(10)
          IF(ABS(F(10)) .GT. ERR ) ICONV = 1
          Z(11,10) = -TI001*XOXO01*C001
          Z(11,11) = -TI001
          Z(11,14) = TI001*XOXO01*CS01
          Z(11,15) = XOXO01*SN01*TI001
          D(11) =-TI001*(X(11) -XOXO01*C001)
          F(11) = DP(11) - D(11)
          DP(11) = D(11)
          IF(ABS(F(11)) .GT. ERR ) ICONV = 1
          Z(12,1) = RM01
          Z(12,10) = RM01*(X(13)*C001+X(11)*C001+XOXO01*(C001*C001-C001*C00
         21))
          Z(12,11) = -RM01*C001
          Z(12,12) = -RM01
          Z(12,13) = -RM01*C001
          Z(12,14)= -RM01*((X(13) +XOXO01*C001)*CS01+ (X(11) +XOXO01*C001)*
         2 SN01 )
          Z(12,15) = -RM01*((X(13)+XOXO01*C001)*SN01-(X(11)+XOXO01*C001)*
         2 CS01 )
          D(12) = RM01*(PM01-X(13)*C001-X(11)*C001+XOXO01*C001*C001 )
          F(12) = DP(12) - D(12)
          DP(12) = D(12)
          IF(ABS(F(12)) .GT. ERR ) ICONV = 1
          Z(13,10) =-TI001*( XOXO01*C001 + PSG01)
          Z(13,13) =-TI001*(OPSG01)
```

```
          Z(13,14) = - TID01*(XDXD01*SN01 + PSG01 )
          Z(13,15) = TID01*(XDXD01*CS01 - PSGX01 )
          Z(13,8) = TID01
          Z(13,11) = -TID01*PSG01
          D(13) = TID01*(X(8) - X(13) -XDXD01*CD01 - SG01)
          F(13) = DP(13) - D(13)
          DP(13) = D(13)
          IF(ABS(F(13)) .GT. ERR ) ICONV = 1
*
********           CURRENT EQUATIONS          FOR GENERATOR
*
          Z(14,10) = -RA01*CD01 -XD01*CD01-VD01
          Z(14,11) = 1.0
          Z(14,14 ) = -RA01*SN01 + XD01*CS01
          Z(14,15) = RA01*CS01 + XD01*SN01
          Z(14,IVG01) = -SN01
          Z(14,IVG01+1) = CS01
          F(14) = VD01 - X(IVG01)*SN01 - X(IVG01+1)*CS01
          Z(15,10) = RA01*CD01 -XD01*CD01 + VD01
          Z(15,13) = 1.0
          Z(15,14) = -RA01*CS01 - XD01*SN01
          Z(15,15) = -RA01*SN01 + XD01*CS01
          Z(15,IVG01) = -CS01
          Z(15,IVG01+1) = - SN01
          F(15) = VD01 - X(IVG01)*CS01 + X(IVG01+1)*SN01
*
********           THESE EQUATIONS ARE REPEATED FOR ALLL GENERATOR MODELS
*
********           FIRST TIME THROUGH LOOP MUST START WITH ADVANCING TIME STEP
*
          IF( ICNTR . EQ. 0 ) GOTO 1000
*
********           CURRENT EQUATIONS          FOR LOADS
*

          VL01 = CABS(CMPLX(X(IVL01),X(IVL01+1)))
          VIL01 = 1.0/VL01
          SI01 = VL01 - VL01*VL01/ VOL01
          SP01 = 1.0 - (VL01*VL01)/(VOL01*VOL01)
          Z(IL01,IL01) = -X(IVL01)
          Z(IL01,IL01+1) = -X(IVL01+1)
          Z(IL01,IVL01) = -X(IL01)+COR01*X(IVL01)*(VIL01-V2IL01-PRIV01)
          Z(IL01,IVL01+1) = -X(IL01+1)+COR01*X(IVL01+1)*(VIL01-V2IL01-PRIV01
        2 )
          F(IL01) = -X(IVL01)*X(IL01)-X(IVL01+1)*X(IL01 +1)+IOR01*SI01+PCR01*SP01
        2 *SP01
          Z(IL01+1,IL01) = X(IVL01+1)
          Z(IL01+1,IL01+1) = -X(IVL01)
          Z(IL01+1,IVL01) = -X(IL01)+COX01*X(IVL01)*(VIL01-V2L01+PXV01)
          Z(IL01+1,IVL01+1) = X(IL01)+COX01*X(IVL01+1)*(VIL01-V2L01+PXV01)
          F(IL01+1) = X(IVL01+1)*X(IL01)-X(IVL01)*X(IL01+1)+COX01*SI01-PCX01
        2 *SP01
*
********           NODE VOLTAGE EQUATIONS
*
```

```
      DO 100 I = 1,IBN,2
      IVL = I + IVS
      IC = INC(I/2+.5)
      F(IVL) = X(IC)
      F(IVL+1) = X(IC+1)
      NP = NROW(I/2+.5)
      DO 110 J = 1,NP
      JJ = IV(J)
      F(IVL) = F(IVL) -Y(I,J)*X(JJ) + Y(I,J+1)*X(JJ+1)
      F(IVL+1) = F(IVL+1) - Y(I,J)*X(JJ+1) -Y(I,J+1)*X(JJ)
      IF ( ABS(F(IVL)) .GT. ERR ) ICONV = 1
      IF ( ABS(F(IVL+1) ) .GT. ERR ) ICONV = 1
      Z(IVL,JJ) = Y(I,J)
      Z(IVL,JJ+1) = Y(I,J+1)
      Z(IVL+1,JJ) = Y(I,J+1)
      Z(IVL+1,JJ+1) = Y(I,J)
110   CONTINUE
100   CONTINUE
*
      IF ( ICONV .EQ. 0 ) GOTO 1000
*
********         TEST FOR ERROR REQUIRMNETS IF MEET ADVANCE TIME STEP
********      CONVERGENCE TESTED FOR ALL PROCESSORS
*
********      START LU DECOMPOSITION OF BLOCK DIAGONAL
********         SPARSITY IS USED TO REDUCE COMPUTATION , IRS AND IRN
********         IS THE START AND END OF EACH ROW,  IEV IS START OF
********         EXTERNAL VARIABLES, INC IS END OF COLUMN (START IS DIAGONAL)
      DO 200 I = 1,N
      ISR = IRS(I)
      INR = IRN(I)
      IEV = IREV(I)
      ICN = INC(I)
      DIV = 1.0/Z(I,I)
      II = I+1
      DO 215 K = II , ICN
      Z(K,I) = Z(K,I)*DIV
      DO 220 J = II,INR
220   Z(K,J) = Z(K,J) -Z(K,I)*Z(I,J)
      DO 225 J = IEV,IEN
225   Z(K,J) = Z(K,J) - Z(K,I)*Z(I,J)
      F(K) = F(K) - Z(K,I)*F(I)
215   CONTINUE
200   CONTINUE
*
********       BACK SUBSTITUTION TO COMPLETE BLOCK INVERSION AND
********         MULTIPLY TIMES THE BLOCK ROW
*
      DIV = 1.0/Z(N,N)
      DO 305 J = IFS,IEN
305   Z(N,J) = Z(N,J)*DIV
      F(N) = F(N)*DIV
      DO 300 II = 2,N
      I = NO - II
      IP1 = I+1
```

```
 SUBROUTINE NSOR     NORMAL                    CDC 6600 FTN V3.0-P336 OPT=2  07/30/
              DIV = 1.0/Z(I,I)
              DO 310 K = IFS,IEN
              DO 315 J = IP1,INP
        315   Z(I,K) = Z(I,K) - Z(I,J)*Z(J,K)
        310   Z(I,K) = Z(I,K)*DIV
              DO 320 J = IP1,INP
        320   F(I) = F(I) - Z(I,J)*F(J)
              F(I) = F(I)*DIV
        300   CONTINUE
       *
        301   ICONV = 0
       *
       ********           SHARED VARIABLES ITERATED TO CONVERGENCE
       *
              DO 400 I = 1,NS
              IL = LV(I)
              SUM = 0.0
              DO 405 J = IES,IEN
              JJ = IV(I,J)
              SUM = SUM + Z(IL,J)*DX(JJ)
        405   CONTINUE
              COR = W*(F(IL)-SUM) - AW*DX(IL)
              DX(IL) = DX(IL) + COR
              IF ( COR .GT. ERR ) ICONV = 1
        400   CONTINUE
       *
              IF( ICONV .NE. 0 ) GOTO 301
       ********          CONVERGENCE TESTED FOR ALL PROCESSORS
       *
       *
       ********           AFTER CONVERGENCE REMAINING VARIABLES UPDATED
       *
              DO 500 I = 1,N
              IL = LV(I)
              SUM = 0.0
              JJ = IV(I,J)
              DO 505 J = IFS,IEN
        505   SUM = SUM + Z(IL,J)*DX(JJ)
        500   DX(IL ) = F(IL) - SUM
       *
       ********           ADD CORRECTION TO VARIABLES
       *
              DO 600 I = 1,N
        600   X(I) = X(I) + DX(I)
       *
       ********           TEST FOR CORRECTNESS OF VARIABLES REQUIRES FUNCTION EVALUATIO
              GO TO 5
       *
       *
       ********           ADVANCE TIME STEP
       *
       1000   IF ( T .GT. TEND ) RETURN
              IF ( KNT .LT. 4) H = 1.25*H
              IF ( KNT .GT. 6 ) H = H*.75
              DO 1010 I = 1,NSV
```

```
     SUBROUTINE  NSOR          NORMAL                                CDC 6600 FTN V3.0-P330 OPT=2  07/30/7
                        X(I) = X(I) + D(I)*H
                        DP(I) = D(I)
                 1010   CONTINUE
                        T = T + H
 335                    ICNTR = 1
                        GO TO 2
                        END
 STER ALLOCATION
 REGISTERS ASSIGNED OVER THE LOOP BEGINNING AT LINE   270
 REGISTERS ASSIGNED OVER THE LOOP BEGINNING AT LINE   293
 REGISTERS ASSIGNED OVER THE LOOP BEGINNING AT LINE   312
 MEMORY WOULD HAVE RESULTED IN BETTER OPTIMIZATION
```

```
*
*
*                          NEWTON METHOD FOR BORDERED BLOCK DIAGONAL FORM
*
            SUBROUTINE NEWT ( X,Y,H,ICNTR)
*
            DIMENSION X(300),Y(100,100),Z(300,300),F(300),D(100),DX(300),DP(
          2 300),NROW(100),IV(100,20),INC(100),IRS(100),IRN(100),IREV(100)
          3 , LV(100),OTHER(2600),FM(20,10),ZM(20,20,10)
            EQUIVALENCE  (OTHER(1),NROW(1)),(OTHER(101),IV(1,1)),(OTHER(2101),
          2 INC(1)),(OTHER(2201),IRS(1)),(OTHER(2301),IRN(1)),(OTHER(2401),
          3 IREV(1)),(OTHER(2501),LV(1) )
*
********            THIS ALGORITHM SOVES THE DYNAMIC SIMULATION PROBLEM
********              BY IMPLICIT INTERGRATION AND NEWTON UPATING OF THE
********              VARIABLES.
*
********            X IS A VECTOR OF THE VARIABLES OF THE MODEL BOTH STATE
********              AND ALGEBRAIC.  Y IS THE BUS ADMITTANCE MATRIX ,
********              Z IS THE JACOBIAN OF THE EQUATIONS.  F IS THE
********              FUNCTIONAL ERROR OF THE VARIABLES,  D THE DERIVATIVE
********              OF THE STATE VARIABLES, DP THE PAST DERIVATIVE.
*
********            OPTIMAL ORDERING IS IMPORTANT SINCE THE JACOBIAN MUST BE
********             SOLVED .  THE ORDERING STATES WITH THE TURBINE/GOVENROR
********             THEN TO THE SUPPLEMENTAL AND REGULATOR, FINALLY TO THE
********             MACHINE VARIABLES.  NEXT ARE THE CURRENT EQ AFTER ALL
********             OF THE GENERATOR MODELS, FINALLY THE ~~VOLTAGE~~ LAST
*
1           T = 0.
2           B = 1./(80*H)
            KNT = 0
5           ICONV = 0
            KNT = KNT + 1

            IF ( ICONV .EQ. 0 ) GOTO 1000
********            CONVERGENCE TESTED FOR ALL PROCESSORS
*
*
********            START LU DECOMPOSITION OF BLOCK DIAGONAL
********              SPARSITY IS USED TO REDUCE COMPUTATION  , IRS AND IRN
********              IS THE START AND END OF EACH ROW,  IEV IS START OF
********              EXTERNAL VARIABLES, INC IS END OF COLUMN (START IS DIAGONAL).
            DO 200 I = 1,N
            ISR = IRS(I)
            INR = IRN(I)
            IEV = IREV(I)
            ICN = INC(I)
            DIV = 1.0/Z(I,I)
            II = I+1
            DO 215 K = II,ICN
            Z(K,I) = Z(K,I)*DIV
            DO 220 J = II,INR
  220       Z(K,J) = Z(K,J) -Z(K,I)*Z(I,J)
            DO 225 J = IEV,IEN
  225       Z(K,J) = Z(K,J) - Z(K,I)*Z(I,J)
```

```
          F(K) = F(K) - Z(K,I)*F(I)
  215     CONTINUE
          DO 230 K = IFV,IBN
          Z(K,I) = Z(K,I)*DIV
          DO 235 J = II,INR
  235     Z(K,J) = Z(K,J) - Z(K,I)*Z(I,J)
          DO 240 J = IFV,IBN
  240     ZM(K,J,IP) = -Z(K,I)*Z(I,J)
          FM(K,IP) = -Z(K,I)*Z(I,J)
  230     CONTINUE
  200     CONTINUE
**
********           END PARALLEL PROCESSING FOR CUT-SET SOLUTION
*
*************************************************************************************************
*
*********          CUT-SET PROCESSOR INSTRUCTIONS START HERE
*
********                  ADD-UP MODIFICATIONS DUE TO CUT-SET ROW
*
          DO 250 J = IBS,IBN
          DO 255 K = IBS,IBN
          SUM = 0.
          DO 260 I = 1,NP
  260     SUM = SUM + ZM(J,K,I)
  255     Z(J,K) = Z(J,K) + SUM
          DO 265 I = 1,NP
  265     F(J) = F(J)+FM(J,I)
  250     CONTINUE
*
********                  DECOMPOSE CUT-SET BLOCK
*
          DO 300 I = IBS,IBN
          DIV = 1.0/Z(I,I)
          DO 315 K = II,IBN
          Z(K,I) = Z(K,I)*DIV
          DO 330  J = II,IBN
  330     Z(K,J) = Z(K,J) - Z(K,I)*Z(I,J)
          F(K) = F(K) - F(I)*Z(K,I)
  315      CONTINUE
  300      CONTINUE
*
********               BACK SUBSTITUTION OF LAST BLOCK VARIABLES
*
          DX(IBN) = F(IBN)/Z(IBN,IBN)
          DO 400 II = 2,IN
          I = IBNO-II
          IP1 = I+1
          DIV = 1.0/Z(I,I)
          SUM = 0.
          DO 405 J = IP1 ,IBN
  405     SUM = SUM - Z(I,J) * DX(J)
  400     DX(I) = SUM*DIV
*
*********          CUT-SET PROCESSOR INSTRUCTIONS END HERE
```

```
      *
      **************************************************************************************
      *
      ********                  THIS RETURNS TO PARALLEL PROCESSING
      *
      ********            BACK SUBSTITUTION OF VARIABLES OF OTHER PROCESSORS
      *
            SUM = 0.
            DO 505 I = IBS,IBN
      505   SUM = SUM -Z(N,I)*DX(I)
            DX(N) = SUM/Z(N,N)
            DO 500 II = 2,N
            I = NO - II
            IP1 = I + 1
            DIV = 1./Z(I,I)
            SUM = 0.
            INR = IRN(I)
            DO 510 J = IP1,INR
      510   SUM = SUM - Z(I,J)*DX(J)
            DO 515 J = IBS,IBN
      515   SUM = SUM - Z(I,J)*DX(J)
            DX(I) = SUM*DIV
      *
      ********              ADD CORRECTION TO VARIABLES
      *
      500   X(I) = X(I) + DX(I)
      *
      ********              TEST FOR CORRECTNESS OF VARIABLES REQUIRES FUNCTION EVALUATION
            GO TO 5
      *
      *
      ********              ADVANCE TIME STEP
      *
      1000  IF ( T .GT. TEND ) RETURN
            IF ( KNT .LT. 4) H = 1.25*H
            IF ( KNT .GT. 6 ) H = H*.75
            DO 1010 I = 1,NSV
            X(I) = X(I) + D(I)*H
            DP(I) = D(I)
      1010  CONTINUE
            T = T + H
            ICNTR = 1
            GO TO 2
            END
```

```
LOCATION
RS ASSIGNED OVER THE LOOP BEGINNING AT LINE     79
RS ASSIGNED OVER THE LOOP BEGINNING AT LINE    106
RS ASSIGNED OVER THE LOOP BEGINNING AT LINE    119
RS ASSIGNED OVER THE LOOP BEGINNING AT LINE    128
RS ASSIGNED OVER THE LOOP BEGINNING AT LINE    130
```

BIBLIOGRAPHY

1. Kochenburger, R.J., Computer Simulation of Dynamic Systems, Prentice Hall, Englewood Cliffs, N.J.,(1972).

2. Undrill, J.M., "Equipment and Load Modeling in Power System Dynamic Simulation", ERDA Conference, Vermont, Summer 1975.

3. Korn,G.A. and Korn T.M., Electronic Analog and Hybrid Computers (2nd Ed.), McGraw Hill, New York,(1972).

4. Gear, C.W, Numerical Initial Value Problems in Ordinary Differential Equations, Prentice Hall,(1971).

5. Davison, E.J., "An Algorithm for the Computer Simulation of Very Large Dynamic Systems", Automatica, Vol.9, pp.665-675,(1973).

6. Dommel H.W. and Sato, N., "Fast Transient Stability Solution", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-91, pp.985-991,(1972).

7. Wu,F.F, "Solution of Large-Scale Networks by Tearing", Memo No. ERL N532 (July 1975), Electronics Research Laboratory, University of California, Berkeley.

8. Davison, E.J., "A New Method for Simplifying Large Linear Dynamic Systems", IEEE Transactions on Automatic Control, Vol. AC-13, pp. 214-15, (1968).

9. Chidambara, M.R., "Two Simple Techniques for the Simplification of Large Dynamic Systems", JACC, pp.669-679,(1969).

10. Undrill J.M. and Turner, A.E, "Construction of Power System electromechanical equivalents by model analysis", IEEE Power Apparatus and Systems, Vol. PAS-90, pp.2060-2071, (1971).

11. Anderson, J.H., "Geometrical approach to reduction of dynamical system", Proc. IEE, (London) Vol.114, pp.1014-1018, (1967).

12. Van Ness, J.E., "Improving Reduced Dynamic Models of Power Systems", 1975 PICA Conference Proceedings, pp. 155-157,New Orleans.

13. Van Ness, J.E., Zimmer, Cultu,M., "Reduction of Dynamic Models of Power Systems", 1973 PICA Conference Proceedings, pp. 105-112, Minneapolis.

14. Gauss, E.J., "A comparison of Machine Organizations by their Performance at the Interactive Solution of Linear Equations" JACM,Vol. 6 pp.476-485, (1959).

15. Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors", Proc. IEEE Vol.54, pp.1889-1901, (1966).

16. Rosenfeld, Jack L., "A Case Study in Programming for Parallel Processor," Comm ACM, Vol.12, pp.645-655, (1969).

17. Korn, G.A., "Back to parallel computation: Proposal for a completely new online Simulation System using Standard Mini's for low cost multiprocessing", Simulation, pp.37-45, (1972).

18. Miranker W.L. and Linger, W.M., "Parallel Methods for Numerical Integration of Ordinary Differential Equations", Math of Computation, Vol. pp.303-370, (1967).

19. Nievergelt, J., "Parallel Methods for Integrating Ordinary Differential Equations", Comm ACM, Vol. 7, pp.731-733, (1964).

20. Schwartz, J., "Large Parallel Computers", JACM Vol.13, pp.25-32, (1966).

21. Thurber K.J. and Patton, P.C., "The Future of Parallel Processing", IEEE Transactions on Computers, Vol C-22, pp.1140-43, (1973).

22 Baer, J.L. and Russell, E.C., "Preparation and Evaluation of Computer Programs for Parallel Processing Systems", in L.C.Hobbs, (ed), Parallel Processor Systems, Technologies, and Application, pp.375-416, Spartan Books, Washington D.C., (1970).

23. Gonzalez, M.J. and Ramamoorthy, C.V., "Recognition and Representation of Parallel Processable Streaming Computer Programs", pp. 335-374. Ibid.

24. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A., "The ILLIAC IV Computer", IEEE Transactions Computers, Vol C-17, pp. 746-757, (1968).

25. Davis, R.L., The ILLIAC IV Processing Element", IEEE Transactions Computers, Vol. C-18, pp. 800-816, (1969).

26. Kuck, D.J., "ILLIAC IV Software and Application Programming" IEEE Transactions Computers, Vol. C-17, pp. 758-770, (1968).

27. Flynn, M.J., "Very High-Speed Computing Systems", Proc IEEE Vol. 54, pp.1901-1909, (1966).

28. Wulf, W.A. and Bell, C.G., "C.mmp - a multi mini processor", Proc. AFIPS 1972 Fall Jt. Computer Conf., Vol.41, Part II, pp. 765-777.

29. U.S.Department of the Interior, "Analytical Techniques for Transient Stability". Final report submitted to Bonneville Power Administration, Contract #14-03-1486N, by Federal Systems Division, IBM Corp.,(1972).

30. Gear, Charles W. "Simultaneous Numerical Solution of Differential Equations - Algebraic Equations", IEEE Circuit Theory, Vol. CT 18, pp. 89-95, (1971).

31. Gear, C.W., "The Numerical Integration of Ordinary Differential Equations", Computation of Math, Vol.21, pp.146-156, (1967).

32. Chazan, D. and Miranker, W., "Chaotic Relaxation", Linear Algebra and its applications, Vol. 2, pp.199-222, (1969).

33. Tinney, W.F. and Walker, J.W., "Direct Solution of Sparse Network Equations by Optimally Ordered Triangularization", Proc IEEE Vol. 55, pp.1801-1809, (1967).

34. Rheinboldt, W., "On M-functions and their application to nonlinear Gauss-Seidel Iterations and Network flows", J.Math Analysis and Application, Vol.32, pp.274-307, (1970).

35. Rheinboldt, W.C., Methods for Solving System of Nonlinear Equations, SIAM, Philadelphia, Pa.,(1974).

36. Henrici, P., Discrete Variable Methods in Ordinary Differential Equations, John Wiley & Sons, New York, (1962).

37. Varga, Richard S, Matrix Iterative Analysis, Prentice Hall, Englewood Cliffs, N.J., (1962).

38. Ortega, J.M. and Rheinboldt, W.C., Iterative Solution of Nonlinear Equations in Several Variables, Academic Press, New York, (1970).

39. Oretga J.M. and Rheinboldt, W.C., "Monotone Iterations for Nonlinear Equations with Application to Gauss Seidel Methods" SIAM Journal Numerical Analysis, Vol.4, pp. 171-190, (1967).

40. Porsching, T.A., "On Rates of Convergence of Jacobi and Gauss-Seidel Methods for M-functions," SIAM Journal Numerical Analysis Vol.8, pp.575-582, (1971).

41. Rheinboldt, W.C., "A Unified Convergence Theory for a Class of Iterative Processes", SIAM Journal Numerical Analysis, Vol.5., pp. 42-63, (1968).

42. Kron, G., Diakoptics: the Piecewise Solution of Large Scale Systems, MacDonald, London, (1963).

43. Happ, H.H., Diakoptics and Networks, Academic Press, New York, (1971).

44. Carre, B.A., "Solution of Load-flow Problems by Partitioning Systems into Trees", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-89, pp. 1931-1938, (1968).

45. Ogubuobiri, E.C., Tinney, W.F., Walker, J.W., "Sparsity-Directed Decomposition for Gaussan Elimination on Matrices" IEEE Transactions on Power Apparatus and Systems, Vol. PAS-89, pp. 141-150, (1970).

46. Tinney, W.F. and Meyer, W.S, "Solution of Large Sparse Systems by Ordered Triangular Factorization", IEEE Transactions on Control,Vol. AC-18, pp. 333-46, (1973).

47. Hachtel, G.D., Brayton, R.K., Gustavson, F.G., "The Sparse Tableau Approach to Network Analysis and Design" IEEE Transactions on Circuit Theory, Vol CT-18, pp. 101-113, (1971).

48. Dec, N., Theory with Application to Engineering and Computer Science, Prentice Hall, Englewood Cliffs, New Jersey, (1974).

49. Frank, Howard and Frisch, Ivan T., Communication, Transmission, and Transportation Networks, Addison Wesley Publishing Co., Reading, Mass., (1971).

50. Boesch, F.T. and Frisch I.T., "On the Smallest Disconnecting Set in a Graph",IEEE Transactions on Circuit Theory, Vol CT-15, pp.286-288, (1968).

51. Frisch, I.T., "An Algorithm for Vertex-pair Connectivity" INT J Control,Vol.6, pp. 579-593, (1967).

52. Miranker, W.L., "A Survey of Parallelism in Numerical Analysis" SIAM Review, Vol. 13, pp. 524-47, (1971).

53. Pease, M.C., "Matrix Inversion Using Parallel Processing" JACM, Vol 14, pp.757-764, (1967).

54. Tewarson, Reginald P., Sparse Matrices Academic Press, New York,(1973).

55. Reid, J.K., Large Sparse Sets of Linear Equations Proceeding of the Oxford Conference at the Institute of Math and its Application held in April 1970, Academic Press, London, (1971).

56. Rose D.J. and Willoughby, R.A., Sparse Matrices and Their Application, Plenum Press, New York, (1972).

57. Sato, N. and Tinney, W.F., "Techniques for exploiting the Sparcity of the Network Admittance Matrix", IEEE Transactions on Power Apparatus and Systems, Vol PAS-82, pp.944-950, (1963).

58. Stagg, G.W. and El-Abiad, A.H., Computer Methods in Power System Analysis, McGraw Hill, New York, (1968).

59. Westlake, Joan R., A Handbook of Numerical Matrix Inversion and Solution of Linear Equations, J.Wiley & Sons, New York, (1968).

60. Alvarado, F.L. "Computational Complexity in Power Systems", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-95, pp.1028-1037, (1976).

61. Skinner, C.E. and Asher, J.R., "Effects of Storage Contention on System Performance", IBM System J, pp. 319-333, (1969).

62. Strecker, D., "Analysis of the Instruction Rate in Certain Computer Structures", PhD Thesis 1970, Carnegie-Mellon University.

63. Blandarkar D.P. and Fuller, S.H., "Markov Chain Models for Analyzing Memory Interference in Multi Processor Computer System", Proc. First Annual Symposium on Computer Architecture, IEEE, New York, pp. 1-6, (1973).

64. Flores, Ivan, "Derivation of a Waiting-Time Factor for a Multiple-Bank Memory" JACM Vol. 11, pp. 265-282, (1964).

65. Saaty, Thomas L., Elements of Queueing Theory with Applications, McGraw Hill, New York, (1961).

66. Plummer, W.W., "A synchronous Arbiters" IEEE Transactions on Computers, Vol. C-21, pp. 37-42, (1972).

67. Anderson, G.A. and Jensen E.D., "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", ACM Computing Surveys, Vol. 7, (1975).

68. Hansen, Per Brinch, Operating System Principles, Prentice Hall, Englewood Cliffs, New Jersey, (1973).

69. Bell, C.Gordon and Newell, Allen, Computer Structures: Readings and Examples, McGraw-Hill, New York, (1971).

70. Baer, J.L., "A Survey of some Theoretical Aspects of Multiprocessing", ACM Computing Surveys, Vol. 5, pp.31-80, (1973).

71. Franta, W.R. and Houle, P.A., "Comments on Models of Multiprocessor Multi Memory Bank Computer Systems", 1974 Winter Simulation Conference, New York, pp. 87-97.

72. Rose, D.J., Tarjan, R.E., Lucker, G.S., "Algorithmic Aspects of Vertex Elimination on Graphs", SIAM J Comput, Vol.5, pp. 266-283, (1976).

VITA

NAME: William Lloyd Hatcher III

BORN: June 24, 1949

EDUCATION: B.S.S.E., U. S. Naval Academy, Annapolis, Maryland, June 1971.

M.S., Northwestern University, Evanston, Illinois, June 1975.

PUBLICATIONS:

1. F.M. Brasch, Jr. and W. L. Hatcher, "Digital Simulation and Control of a Shipboard Generating System," 1975 Power Industry Computer Application Conference, New Orleans, La., June 1975.

2. W.L. Hatcher, F.M. Brasch, Jr, and J.E. Van Ness, "A Feasibility Study for the Solution of Transient Stability Problems by Multiprossor Structures," submitted to 1977 IEEE Power Apparatus and Systems Winter Power Meeting, New York, New York, September, 1976.

14 AUG 79 25388

Thesis 166710
H3176 Hatcher
A special purpose multiprocessor for the simulation of dynamic systems.

4 FEB 77 DISPLAY

~~4 FEB 77~~ ~~DISPLAY~~

14 AUG 79 25388

Thesis
H3176 Hatcher 166710
A special purpose multiprocessor for the simulation of dynamic systems.