Techniques for power system simulation using multiple processors

Alistair James Eden Taylor

Abstract.

Techniques for Power System Simulation using Multiple Processors.

Alistair James Eden Taylor.

Submitted in 1990 for the degree of
Doctor of Philosophy.

University of Durham.

The thesis describes development work which was undertaken to improve the speed of a real–time power system simulator used for the development and testing of control schemes. The solution of large, highly sparse matrices was targeted because this is the most time–consuming part of the current simulator. Major improvements in the speed of the matrix ordering phase of the solution were achieved through the development of a new ordering strategy. This was thoroughly investigated, and is shown to provide important additional improvements compared to standard ordering methods, in reducing path length and minimising potential pipeline stalls. Alterations were made to the remainder of the solution process which provided more flexibility in scheduling calculations. This was used to dramatically ease the run–time generation of efficient code, dedicated to the solution of one matrix structure, and also to reduce memory requirements.

A survey of the available microprocessors was performed, which concluded that a special–purpose design could best implement the code generated at run-time, and a design was produced using a microprogrammable floating–point processor, which matched the code produced by the earlier work.

A method of splitting the matrix solution onto parallel processors was investigated, and two methods of producing network splits were developed and their results compared. The best results from each method were found to agree well, with a predicted three–fold speed–up for the matrix solution of the C.E.G.B. transmission system from the use of six processors. This gain will increase for the whole simulator. A parallel processing topology processor was then developed using the same partitions which could process the topology of the partitioned network and produce the necessary structures for the remainder of the solution process.

Techniques for Power System Simulation using Multiple Processors.

Alistair James Eden Taylor.

A thesis presented for the degree of

Doctor of Philosophy.

University of Durham.
School of Engineering
and Applied Science.

November, 1990

To

my parents.

Acknowledgements.

I would firstly like to thank my parents for their encouragement, advice and for providing me with opportunities that they never had. I would also like to thank my supervisor, Professor M.J.H. Sterling for guidance, and the rest of the O.C.E.P.S. Group, past and present, particularly Dr. M.R. Irving, Mr. J.O. Gann and Dr. A.P. Birch for their help. Special thanks are due to Mrs. J. Gibson for producing the 'O.C.E.P.S.' illustrations, and to Mrs. A. Shipley for all her help.

I would also like to thank the technical staff of the School of Engineering and Applied Science, particularly Mr. T. Nancarrow, Mr. I. Wallis and Mr. I. Garrett, for both their friendship and invaluable help throughout my studies in Durham.

I am grateful to Royal Ordnance plc., and Dr. D. Izod in particular, both for sponsorship during the period of this research, and for industrial experience beforehand.

Thanks to the residents of 'The Gables', particularly Yvonne, Ian, Philip and Carver. You'll know what this is for. Finally, thanks to the regulars in D.U.Y.H.A. for the complete antithesis. Walk on.

Contents.

Tables.

Figures.

Appendix D. Optimised Splits

Declaration.

The work contained in this thesis has not been
submitted elsewhere for any other degree or qualification,
and that unless otherwise referenced,
it is the author's own work.

Statement of Copyright.

Chapter 1.

Introduction.

The increasingly more complex devices and systems that are being designed, and are in use today, have required much more careful control than their simpler ancestors. The increasing use of embedded computers in such devices and systems has eased their control, by removing some of the low level, or mundane, actions from the main controller, but this controller must now, however, be able to control both the actual device, and any controller that lies between. The added complexity has made the system less amenable to direct analysis, so modelling is increasingly performed to develop and test control systems.

In large systems, there is also more economic pressure to reduce the cost of running the system than before. An electrical power system is undeniably large, and it is important that it should deliver electrical power both cheaply and reliably. These two objectives are, to a certain extent, mutually exclusive, so a compromise position must be reached. If better control can be exercised over the system, then it might be possible to reduce conservative practices while maintaining the same levels of security of supply, so again there is a need for developing and more importantly proving the algorithms which are used to control the system. Economy and reliability are also influenced by the actual plant that is installed, and how it is utilized, so a large part of the responsibility for these aims must be taken by system planners.

Simulators are extensively used to test control algorithms, and to train the operators of the system. Much effort has been expended into cockpit simulators for aircraft, both to train new crews, and provide experience in handling conditions, such as a stall, for which it is not possible to provide experience in a real aircraft without danger. The simulator should provide the most realistic simulation possible, and aircraft simulators have video displays showing the view through the plane's windows, and hydraulics to

physically move the cockpit to create the feeling of motion. A power system simulator shares many of the aims of these flight simulators, because it is highly undesirable to carry out operator training and 'what if' studies on the actual system, since this would adversely affect the supply of electricity, and might result in costly damage to the installed plant. The cost of any failure which resulted in the loss of supply would also be prohibitive.

Despite the similarities which exist between the purposes of simulating aircraft and electrical power systems, different applications do exist for such simulators due to the characteristics of the systems that are modelled. An aircraft is a small, reasonably well defined object, and while error and bad data detection are required for security, there should be little difficulty in determining the state of the aircraft. A power system, however, is spread out over a large geographical area, so communications with the main controller are made more difficult and costly, so there are fewer measurements made and fewer which are passed to the controller in approximate real–time. The remote nature of these transducers, and the long communication paths, leaves them prone to error, so several stages data validation, such as error removal and state estimation, must be applied before any of these values are used by the control package. There are, nevertheless, times when there are several interpretations which could be given to a set of results, and a reliable, accurate simulation of the system could be used to investigate some or all of the hypotheses.

1.1 Uses of a power system simulator.

A simulation of a power system would be useful for five purposes.

1.1.1 Development of control systems.

The O.C.E.P.S. group uses a simulator [114] to develop and test control algorithms before they are used on a real power system. Such a simulator should be as realistic as possible, in that it should present the same results as would come from the real system, and must run in real–time [122]. In other words, the simulator output data should match those which

would come from the actual system, both in value and time of arrival. The configuration in use by O.C.E.P.S is shown in figure 1.1.

Only a minimal user interface is required, since most of the commands would come from the control computer or the predefined schedule, with only start–up and stop commands being required from a user. The schedule is private to the simulator in the sense that it is not fed to the control package. The effects of a scheduled event appear to the controller as unexpected behaviour of the system, and it is up to the controller to identify the new system state. In general, the shedule is predetermined, but events not in the schedule will still come from the controller in real–time, and will also be generated internally within the simulator by models of protection devices on the real system. These events limit the possibility of allowing the simulator to work ahead of real–time during slack periods of operation.

1.1.2 Operator training.

Many simulators are written to train system operators to make the best use of the system [16 39 90 94 109 113 123 128 138], and to help them to recognize and cope with undesirable system events, by giving them realistic practice with a model of a system. Severe events can be simulated in the model without affecting the security of supply for the customers. The cost of failure is merely having to re–start the scenario, so a variety of solutions could be tried to see which is the most successful.

A.training simulator must have similar properties to one written for the development of control software, because it should be as realistic as possible. There must, however be more a extensive user interface, provided either as part of the simulator, or by an outside facility, e.g., it is usual for either a dummy (or back–up) control room to be used, or for the main control room to be switched over during periods of quiescent operation, and for the simulator to run on the main control computer, or its back–up. The simulator would normally be run from a master schedule, which would generate the initial events. From there on, most events would be expected to be generated from the operator or the control computer. The control computer is a vital part of the set–up, because it is part of the operator's operating environment when controlling the real system, and would be expected to analyse much of the data, and perform some operations automatically.

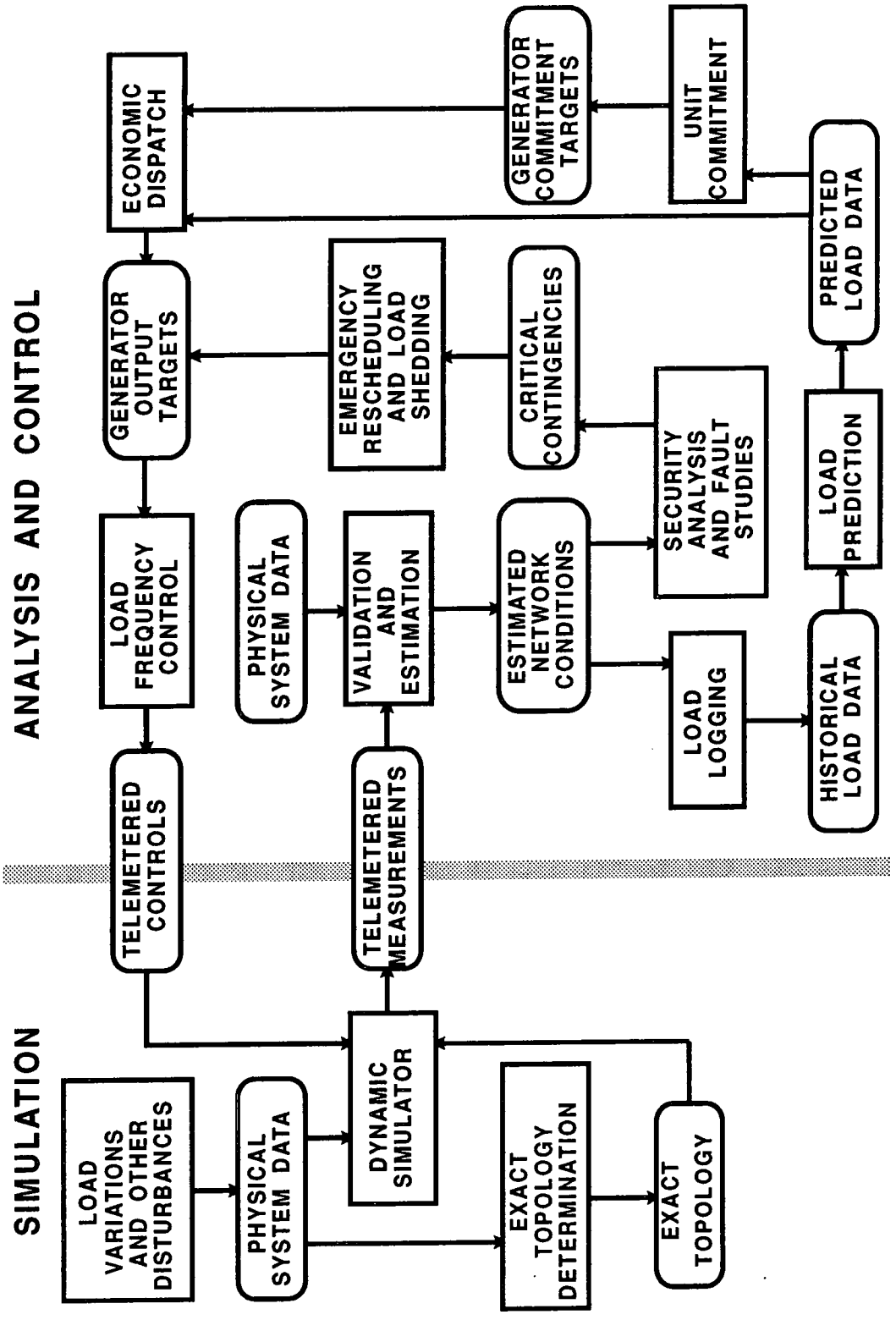


Figure 1.1 Configuration for O.C.E.P.S. simulation.

1.1.3 Teaching.

Simulators can also be used for teaching, to show the effects on the system of certain actions and events. Such simulators would normally be expected to model small systems, because the effects of a system event would usually become lost in a large system. There is less of a requirement for real–time simulation, and indeed, it might be better to trigger each time–step manually, so that all the information provided could be studied. More information than is usually generated by training simulators is required to allow students to see all aspects of the system. A graphical interface is a must, and it is useful to be able to step forwards and backwards to illustrate specific points. It would also be usual to provide facilities to select from a wide choice of predefined models, and the ability to change models simply.

1.1.4 Investigating system behaviour.

Such a simulator could also be used as an investigative tool [81] to probe the events leading to and during a severe system event. If the simulation is fast enough, it could be used during the event to assist the operators in trying to determine the most probable system state, and in trying various remedies, before they are tried on the real system. A post event investigation would certainly be instigated so that a similar problem in the future would be handled better, and this would involve the simulation of what was believed to have caused the problem, to verify that the actual system responses were consistent with this premise, and then, once the schedule had been determined, simulations of many responses to determine future operating practice.

1.1.5 Planning tool.

A simulator could also be used as a planning tool, to investigate the effects of changes to system plant or configuration. Real–time execution would not be required for most work, and so this would be most similar to the teaching simulator. Model entry would also have to be provided, in addition to the flexibility of the teaching simulator, so that new models could easily be added, and existing models changed, possibly even during a run. Later, work would transfer to one of the real–time simulators to prove the modification in a more realistic system environment.

The above cases all simulate both the network and individual items of plant. There are many simulations performed during the design of individual machines, but these do not have the large network component, and are sufficiently different from the simulation of the whole or large parts of power systems that they are not considered further in this thesis.

1.2 Methods of simulation.

Before the introduction of fast digital computers, simulation was performed using analogue components [76], with a scaled down model of the real power system. Such simulators could simulate events faster than real–time by altering the impedance values to compensate for the change in the base frequency. Simulators suffered from large physical size, and more importantly, from the slowness of reconfiguration if the conductivity of a circuit element required changing. Although electrically controlled switches were used for some connections, their large size forced the use of panel connections for less frequently operated switches. If a new item of plant were required, then one would have to be physically constructed and connected into the system. The flexibility of experimenting with new models of existing devices, or with models of completely new plant, was therefore poor.

Hybrid simulators [104] were then developed, which still used some analogue components, but performed some calculations digitally, particularly topological connection, with A/D and D/A converters interfacing the analogue and digital parts to each other.

As digital computers became faster and more capable, simulation became entirely digital. The gains in flexibility were considerable, and the physical space requirement was drastically reduced. The digital simulator must perform much more work modelling the analogue components, so a compromise must be reached between system size, model complexity and speed. Precisely where the balance is struck depends of the intended application of the simulator. For control development and operator training, real–time execution is of paramount importance, and the system size is defined by the real system, so the models have to be simpler, and less transient detail is modelled. The response time of the communications of the real system usually result in the choice of a time–step between solutions of about one second.

More detailed consideration of local controllers, and investigative work would require faster events to be modelled, which could be accomplished by relaxing the real–time constraint, or by only modelling the local part of the system in detail, and lumping the remainder of the system together, and using simpler models in it. The relevant transient behaviour would tend to be localized, so little of this information is invalidated.

There is a new, tentative move back to analogue methodology, but using digital computers. The technique is known as *homeostasis* [83], that is, the maintenance of internal equilibrium when subjected to external disturbances. Each circuit element is modelled with an arbitrarily complex model, which can only communicate with its neighbours by a limited number of variables, or system states. These variables represent the same types of property for every model in the system, so that they can all be connected together in any way desired. Candidates for these variables are currents, voltages, power flows and frequency. Each model receives inputs of these variables from its connections to the system, and attempts to alter its internal state to match these external values. The values which result from this adjustment are then passed out of the model to a global adjuster, which combines all the model outputs together, and produces new inputs for them all. A check is then made for convergance, and the process repeated until this is achieved. The simulator can then proceed to the next time step.

1.2.1 Simulator Interface to the Control Computer.

The simulated system appears to the control system just like the real system. For a network simulator, the control systems of individual items of plant are included in the plant models, so that the fast–acting control paths are handled automatically within the simulator. To aid realism, measurements and commands should be passed through an industry–standard S.C.A.D.A. (Supervisory Control And Data Acquisition) computer, sited between the simulator and control computers, or if the programs reside in the same computer, a software simulation of this process should be provided. It is important that the control package should only see what it would see if it were connected to the real system, so the controller should not be able to access the simulator's private data to determine, for example, the correct system topology. Such access would also provide an unrealistically short time delay between an event in the simulator and its arrival at the control package.

The simulator should not only model the system, but should also mimic the noise, missing data and measurement errors in the signals sent back to the controller by the actual system. The simulator continues to model the system as correctly as possible, and only then corrupts the data which is transmitted to the S.C.A.D.A.. Corruption could also be introduced in the other communication path, but this would be more troublesome, since the simulated system would not be that which was expected. This would, however, be a good test for control algorithms, but undesirable for operator training. A simulator used for teaching or investigative work should not have any errors introduced.

1.3 Power system control.

A modern power system has two main levels of control. The lower level acts at the plant level, with the aim of keeping some local measurements within certain limits. A generator, for example, usually has a target voltage and frequency for its output busbar, and local control adjusts the working fluid flow through the turbine and winding excitation to maintain these. Other control loops are used to provide back–up and to control other parameters, such as boiler firing. These individual loops are combined into a global control strategy for the generator, which is what the higher level of control sees as the characteristics of the generator. The targets, or set points, are produced by the higher level of control, which is regional or possibly national. This controller tries to optimize the economics of the system, while maintaining security of supply, while the lower controller tries to satisfy the targets which it receives.

The lower level of control can be thought of as being part of the individual items of plant, so that the higher level of control *sees* the combined characteristic. It is also capable of acting quickly, whereas the typical response time for the higher level of control is between ten seconds and a minute. Measurements are only received at the control centre and passed to the control computer and the operators every ten seconds, or so, and a decision must then be made, and the instructions sent out before any action can be taken.

The elements in the higher level of control of a modern, electrical power system are shown in figure 1.2, connected to a simulator instead of a real power system. The system measurements, such as frequency, voltages, current flows, set points and transformer ratios are fed into a validation and estimation routine to remove or reduce the effect

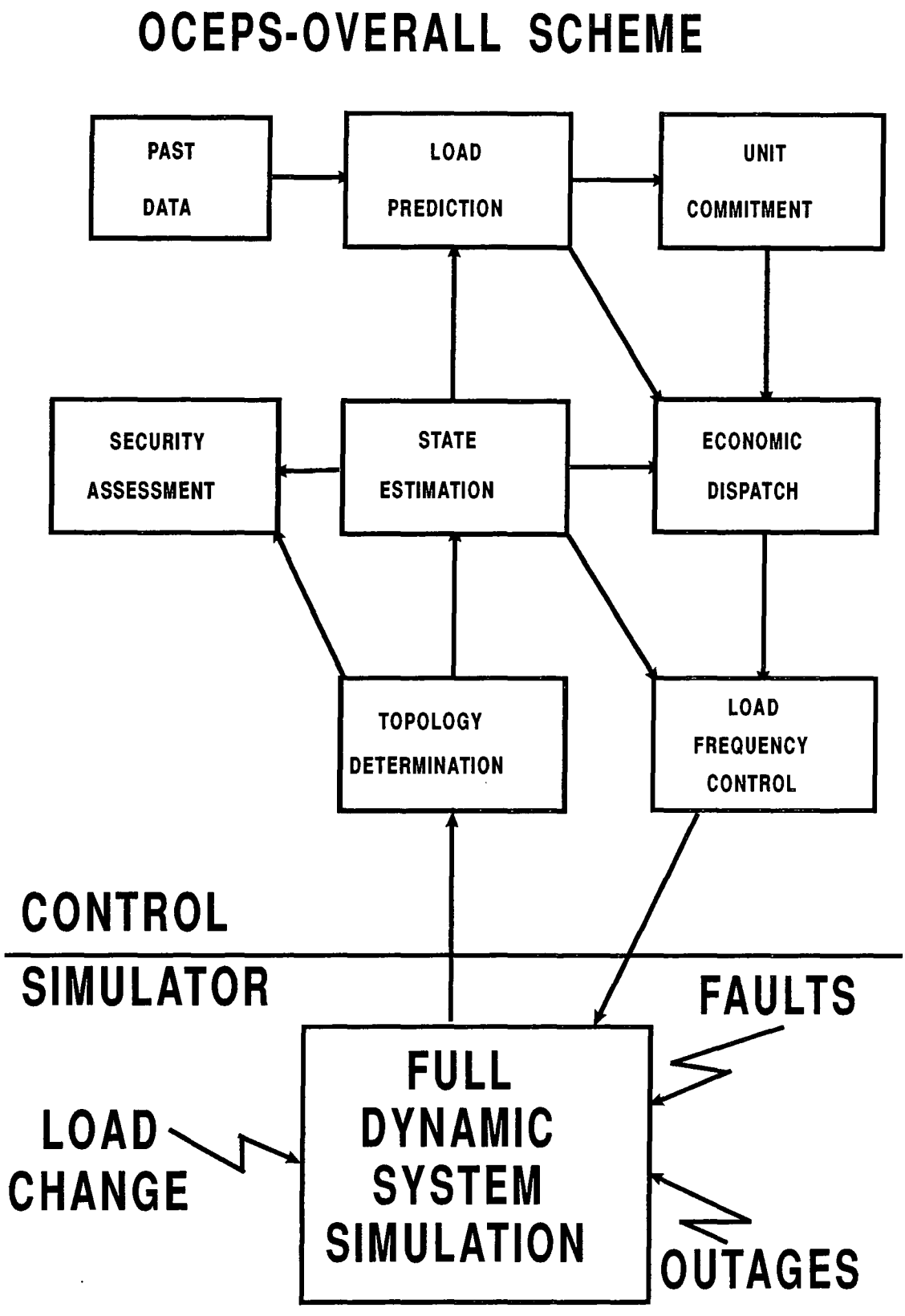


Figure 1.2 Overall O.C.E.P.S. control scheme.

of measurement errors, and to try to estimate any unavailable measurements, whether temporary or as a result of metering shortages. An estimate is then made of the current state of the system, and these values are logged and used for future processing. In the short term, they are used to adjust generator set points, which control the controllers which are local to each generator, the compensators and transformer taps, and in severe situations, commands are sent to circuit breakers to disconnect parts of the system to reduce the electrical load if the generation which is available cannot meet the required demand.

An alternating current power system is unlike most other supply systems, such as water and gas, in that it is not possible to store what is supplied at times when there is a surplus, and use it at times when there is a shortage. There are pump storage schemes, for example Foyers near Loch Ness, which store energy by pumping water uphill when there is a surplus of power, and release the energy by generating as the water is allowed to flow back, but these are costly and the exception to the norm. If the instantaneous energy generation does not precisely match the current demand, both from users and losses, then the mismatch must be converted to or from another form of energy to maintain an overall energy balance. In a power system, there is a considerable kinetic energy reserve stored in rotating mass, in the shafts of motors and generators, and any power mismatch causes their rotational speed to alter, thereby either absorbing or releasing energy and altering the frequency of the supply. Frequency is therefore a good indicator of the balance between load and generation, and is the basis for most controllers (Load Frequency Control [17]). There are also statutory requirements which limit voltage levels and frequency deviations, over several time–scales.

Looking further ahead from the current instant in time, the current and historical state of the system is used to predict the future load on the system, on a time–scale ranging from minutes to a day ahead, and this prediction is used to allocate the load between available generators, and control when and which generators participate in the system. Most generators require a run–up time before they are able to generate into the system, so they must be given advance notice of when they will be required. For a thermally fired plant (coal, oil), the boiler must be heated, and steam generated and allowed to pass through the turbine to slowly heat this to operating temperature, allowing differential thermal expansion to be kept to safe levels. Only then can the generator be used. A similar (but faster) process is required on cessation of generation. This clearly costs

money without any financial return, so for economic reasons should be minimized. It also means that generators must be kept running in reserve, so that if there is an unforeseen load increase, or a plant failure, additional generation can be brought quickly on–line, to prevent frequency excursions and load loss.

Unfortunately, the demand for power throughout the day is not constant, so some generators must be run–up and shut–down during every 24 hour period. There is also a requirement to have generation in reserve, able to pick up load with different amounts of notice, from seconds, to minutes, to tens of minutes. The faster pick–ups are referred to as spinning reserve, i.e., the plant is activated and synchronized with the system, but not generating at its full capacity. Gas turbines, with their fast start–ups, can also be used to pick up load over a time–scale of minutes. This is important, because many thermal stations can generate above their normal rating for a short time, by releasing additional steam from the boilers. This capability is limited by the boiler rating, and those of the turbine and generator, but can be used until more generation, such as gas turbines, becomes available.

The provision of adequate generation is performed by the load prediction and unit commitment. These results are passed into an economic dispatcher, which allocates actual generation targets based on system restraints, legal requirements, predicted loads and economic factors such as the cost of generating at each plant. These targets are then modified by the results of the emergency, or short–time rescheduller, and passed to the load frequency controller, which monitors frequency and adjusts generation targets on a short time scale. These values are then transmitted to the system.

1.4 Description of a power system.

An electrical power system consists of generation, transmission, distribution and loads. Electrical generation is frequently remote from the point of useage of electrical energy, which is an economic decision since it is cheaper to transfer electrical energy than to transport the fuel from where it is available, and additionally a large supply of cooling water is required if thermal power plant is used. There are also political and geographic reasons for the remote siting of generation, such as areas of high rainfall and damable rivers and lakes for hydro–generation, and remoteness from large and influential centres of population for nuclear plant.

The users of electrical power tend to be in cities and towns, so the power must be transmitted from the generation sites to the load points. It is more efficient to transmit electricity at high voltage and low current, since heating loss is the product of wire resistance and the square of the current flow. For this reason, most power transmission is done at voltages between 100kV and 1MV. These voltages require significant levels of insulation, and so must be reduced for distribution to the customers. Distribution networks feed from the transmission network, and usually use voltages between 10kV and 100kV, with voltage being progressively stepped down as the customer is neared. Distribution to the commercial (office, shop) and domestic user is at voltages of about 240V, while industrial customers may recieve higher voltages if they are bulk users, and some industrial sites may even have their own generation, which they use themselves, and feed any surplus into the National Grid. This is restricted to heavy users of electrical power, most notably aluminium smelters.

The United Kingdom has a tightly meshed National Grid [41] for transmission. Most generation is thermal, with coal and oil fueled boilers predominating, and is sited along major rivers, since these provide cooling water for the thermodynamic cycle of the working fluid, and provided easy local transport of fuel. Most coal is found in South Wales, and in the Nottinghamshire–Yorkshire coalfields. This area has many power stations because it has several suitable rivers, is relatively close to industrial Yorkshire and Lancashire, and the terrain south to London presents no difficulties for power transmission. Newer oil fired stations are more likely to be situated near the coast and an oil terminal. Nuclear stations, which contribute around 8% of british electrical power are usually coastal, since this halves the land at risk from an accident, and again provides cooling water. Despite coal and large rivers, the North East is poorly supplied, with only coal–fired Blyth and nuclear Hartlepool of any note. The distribution of major generation sites is shown in figure 1.3, which is taken from C.E.G.B. recruitment literature.

Problems on the transmission grid could cause large areas of the country to have a deficit of generation compared to load, so some load would have to be shed. The tight meshing attempts to reduce this eventuality by providing alternative, possibly less efficient, routes between any two substations [41]. The meshing is tighter and more global than is to be found in other countries where private electricity suppliers have each developed their own networks, and then linked these together with their neighbours to

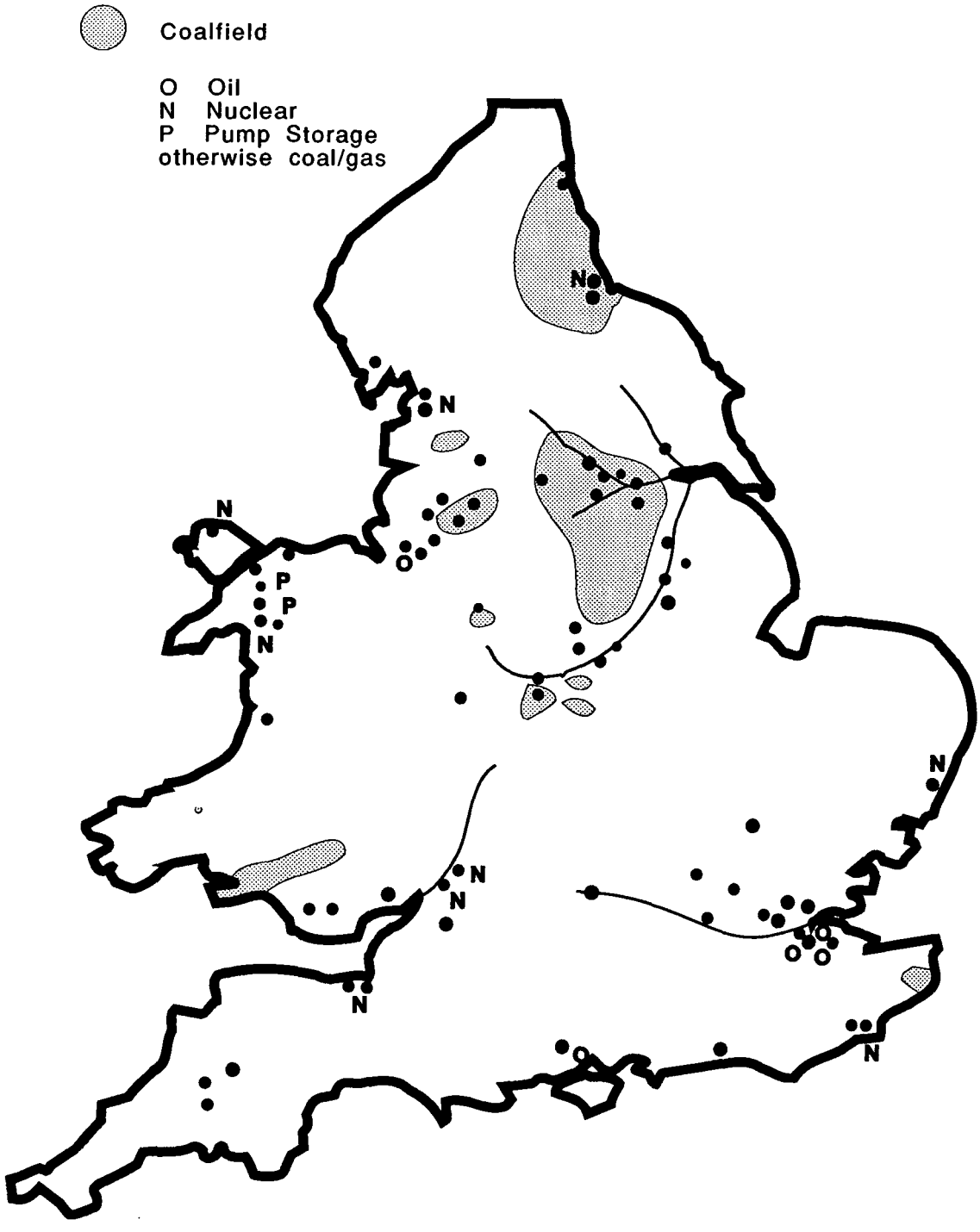


Figure 1.3 Generation sites in England and Wales.

provide power flow between them. This power flow can be an economic benefit, and can also improve reliability, or reduce the cost of providing reserve generation. Such networks split relatively easily for parallel processing, because there are generally few interconnections between the areas. It is easier to monitor and control the inter–area power flows if there are only a few inter–area connections. The United Kingdom system, in contrast, has few weak links where an area is connected to the rest of the grid by only a few lines, and these are usually lines with large power flows, which may cause numerical or stability problems with some algorithms if the links are broken in the model, to enable parts of the power system to be processed separately.

Power is generated and transmitted as three sinusoidal alternating voltages, out of phase by 120°, which makes efficient use of cables and provides constant power in a balanced load, which is desirable for motors. The domestic and most commercial load is single phase, which is obtained from one of the three phases and a local neutral point, which will usually be at approximately earth potential. If the load is balanced, then it is possible to view the system as a single phase system for some types of analysis, and this is done for large–scale simulation. A balanced load is desireable from an economic viewpoint, since the more balanced are the loads on each of the three phases, the less current will flow in the neutral line. If the neutral current can be made negligible, then an earth path can be used in transmission and distribution, and one of the cables dispensed with.

The power system in the United Kingdom has:

212	Transmission substations.
800	Electrical nodes in the transmission system in normal operation.
1600	Transmission lines.
7725 km	Total line length in the transmission system.
78	Power stations.
52 GW	Installed capacity.

There is some transmission of electrical power in the distribution system, but this is generally disregarded for transmission studies, because it is small compared to the power carried by the bulk transmission system. The individual connections from the transmission network to the distribution system can therefore be viewed as individual loads on the transmission network, with no connectivity of their own. Of course, where

an important distribution link does occur, it can be modelled. The type of load seen by the transmission system at each connection will vary with the mix of load connected to the distribution system below the connection, and how the distribution system controls its voltage and power flow levels. Loads are usually classified as constant voltage, power or current [114]. It will be seen from the size of the transmission system that it is quite sufficiently large without modelling any of the distribution system beneath it in detail.

The brief description of the number of nodes hides one of the main difficulties in the modelling and analysis of power systems, which is that the system can change configuration, sometimes quite drastically, during the study. Not only can the circuit components (lines, transformers, loads and generators) become inactive, but the points to which they connect can merge together or break apart. This is achieved by grouping busbars together in substations, and by providing switchable connections between busbars, and between busbars and other circuit elements. Each circuit element is only connected to one busbar at each of its ends. Most substations usually have all their switches closed, and so are fully connected, and are seen as one electrical node. All circuit elements connected to that substation would connect to one node when energised. Some substations are commonly run split into two or more nodes. These nodes may merge at some time during the simulation, so the equations must be reformulated to represent the disappearance of a node, and the increase in connections to another node. The converse operation must also be handled, where a new node, possibly with connections, appears in a substation.

Transmission lines are used to pass the electrical energy between nodes. Together with transformers, they are the primary connections between nodes in different substations. Transformers are used to convert between voltage levels, to adjust the phase angle, and also for voltage regulation, which is achieved by changing transformer taps, which alters the ratio of voltages across the transformer. This is can either be done automatically by sensing the voltage on a local busbar (i.e., it is handled by the simulator), or by the global controller if more complex adjustments are required.

There are other circuit elements in the transmission system that do not connect nodes together. A variety of compensators are used around the network to alter the power factor. These are added because a power factor of near unity reduces the transmission

losses, and if the power factor is allowed to become too low, instability can occur. These are generally fairly simple to model.

1.5 Characteristics of power system simulation.

1.5.1 Modelled elements.

The elements which a power system simulator must model, fall into four categories:

1) Some plant items do not need to be modelled numerically, and are only used to determine connectivity. These elements are all internal to sub-stations, connected between busbars and either busbars or inter–node connections. Busbars and circuit breakers are used only as connections, and as such, do not have equations representing them. System values do not vary with position in these items, so the voltage is assumed equal at every point on a busbar. In many cases, all the elements that the simulator actually knows about do not need to be represented in the heart of the simulator. There are usually many switching elements in any direct path between busbars; one or more circuit breakers and almost certainly more than one isolator. The circuit breakers are used to stop current flow, and must be capable of breaking fault currents above the normal connection rating. Isolators cannot break flow, but once open, would prevent flow even if the breaker were subsequently closed. Normal practice would be to open the breaker, and then the isolators if the connection were to remain broken for a significant time. Re–closure would be performed by the circuit breaker. These elements could be explicitly handled by a routine placed between the core of the simulator and the event generator and communications interface. This would convert the status of each into a combined status which the simulator core would process. This allows some parallelism, and also enables more sophisticated processing to be done on these plant items, without slowing the simulator core, as is shown in figure 1.4.

2) Network components, such as transmission lines and transformers have models which are algebraic. The models are strictly non–linear, because

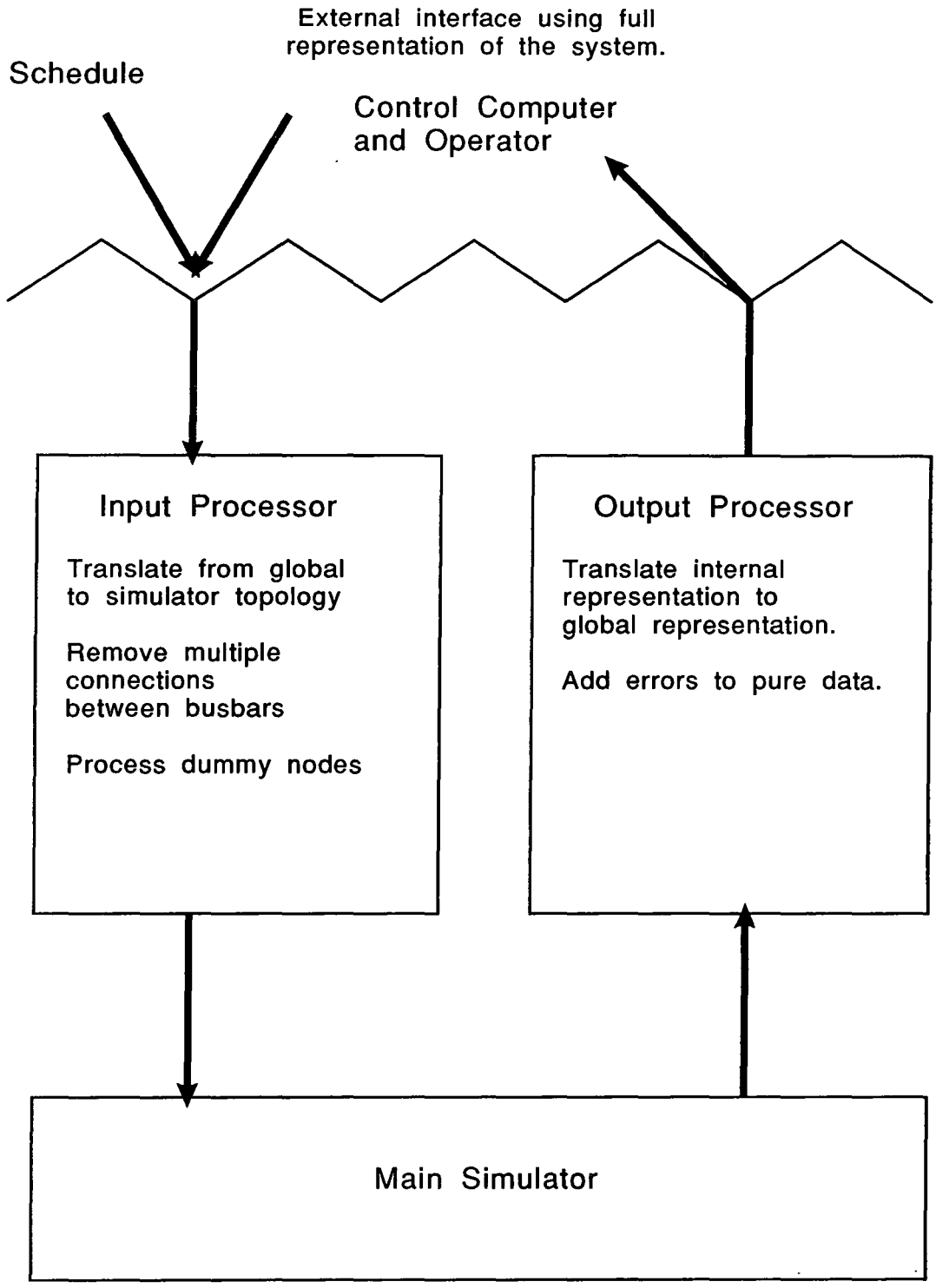


Figure 1.4 Parallelism in simulator communications.

the admittances depend on system frequency, which in turn depends on the power flows round the system. The non–linearity is not large, and some solution methods ignore it, using the system set frequency for all admittance calculations. These elements determine the possible inter–node connectivity of the system, and are energised by the state of the circuit breakers at either end.

3) Other components such as loads and compensators also require algebraic models, but are connected to only one electrical node, and therefore cannot affect the connectivity of the network. Loads are usually modelled as shunt impedances, but this simplicity hides the load model itself, which adjusts the parameters of this impedance to match the desired load characteristic. This may be one of the standard cases of: constant voltage, constant current and constant power, or some combination of these.

4) The last category consists of generation. Generators are represented by models consisting of ordinary differential equations, which must either be solved separately from the algebraic equations, or integrated before being combined with the algebraics. This simulator assumes that they do not alter the connectivity of the network, so that a generator model can only connect to one busbar.

A thorough treatment of the models in use by the O.C.E.P.S. group can be found in Rafian, Irving and Sterling [114]. A companion simulation language project to this research, removed the need to investigate the precise models of individual components, and actually made this undesirable, because the simulator should not be restricted to a particular set of models. The only restriction placed on some of the models is their connectivity. The connectivity options allowed would permit simulation for operator training and control package development, but specifically, mutual inductance between transmission lines cannot be modelled, and some forms of automatic control requiring remote automatic sensors would require special handling.

1.5.2 Solution of the models.

The models for each individual plant item define the contribution, which is made to the system, by the item, due to the current state of the system. Each model produces its own contribution, and these individual contributions must be harmonised to produce a solution which satisfies all the models. The results from the mopdels are represented by both differential and algebraic equations, and the combination of these two types of equations is one of the main characteristics of power system simulation. The problem is complicated both by the large scale, and by the real–time reconfiguration of the network, requiring that the algebraic equations must be able to be re–formed in real–time, with the resulting adjustments of the differential equations. The identifiers used for the numerical solution must therefore be capable of being re–generated in real–time. The solution methods which are applicable are described in more detail in chapter 2.

1.6 Digital Computer Hardware.

Modern simulators invariably make use of digital computers, but the ranges of performance and intended uses of these computers is large, so some are more suited to power system simulation than others. The specific demands which a power system simulator makes on computer hardware are the subject of the remainder of this thesis, particularly chapters 2—7, but briefly, a large number of relatively unstructured floating point operations are required in a given, short, time period, if the simulator is to model the system in real–time. The number of these calculations requires fast computers, and the lack of any regular pattern in the memory accesses, that access the data for them, renders many of the recent developments in very fast vector processors and parallel computer arrays inapplicable to solutions of network problems [19].

The cost differential between the best current mini–computers, which are capable of sustaining high rates of calculation, and the best microprocessors makes basing a simulator on microprocessor hardware very attractive, if the performance is sufficient for real–time simulation. Although many of these microprocessors can sustain high rates of floating–point calculations, their performance does not tend to be as well balanced as that of mini–computers, so although they may perform some tasks much faster than mini–computers, they may be much slower for others. Most problems can be formulated in such a way that memory accesses are either infrequent, very localised, or follow a

regular and predictable pattern, all of which are easy to incorporate into a general purpose design. This is not the case for network solutions, and little support is provided because this is a minority problem.

One method of increasing the solution speed for a problem is to use a faster computer, based on a faster processor and memory system. Another method is to divide the task between several or many processors, so that each has less work to do, and so can finish its allotted calculations faster. This attractive idea works well for some problems, such as the calculation of the Mandelbrot Set [107] and fluid flow problems [29], but is difficult to apply for other problems, one of which is network simulation. In order for a problem to split easily between processors, each processor must either be largely autonomous, so that it can calculate independently of all the other processors, or there must be considerable natural regularity in the problem, so that inter–processor communications can be optimised and effectively hidden.

Research into the application of parallel processors has started from the two extremes of parallel processing; the use of a few, powerful processors, and the use af many, possibly less powerful processors. The latter are represented by processor grids such as systolic arrays, the ICL DAP and Transputer arrays, while the former are either several similar processors, or the addition of specialist computers to a more general purpose host.

The unstructured network connectivity of a power system removes any usable structure from the inter–processor communications if many processors are used, and the double precision floating–point representation required for numerical stability have dictated that arrays of less powerful processors are not well suited to power system calculations, so most work has used a smaller number of more powerful processors.

The use of multiple processors or computers for some problems is not new, but the methodology has changed with the introduction of relatively cheap processors which are capable of participating in the solution of large or difficult problems. The older parallel applications were generally split between large machines at remote locations, where either one machine was not fast enough, or could not hold the necessary amount of data. Smaller and cheaper computers lacked the memory and processing power to participate. The arrival of cheaper, faster, processors, with the capability of accessing very large memories, meant that problems could be split in different ways, because

the physical proximity of such processors permits faster transfer rates, and less time between sending data and receiving a reply.

The increase in the production of microprocessors, and particularly of memory devices, has resulted in large cost decreases for these components, while the components in the remainder of the computer have maintained their price. This means that most of the cost of a computer is in the peripheral devices, such as displays and mass storage devices, which are used. It costs relatively little to add additional processors, and for some problems, the performance increase can be dramatic. The benefits are reduced if the problem is difficult to process in parallel, or if extra peripherals are required to realise the added power.

1.7 The aim of the research.

The aim of the research presented in this thesis is to investigate techniques for improving the speed of the digital simulation of electrical power systems, so that either more detailed models can be used, or so that a larger system can be modelled without loss of detail, than is possible with the current O.C.E.P.S. simulators. The current O.C.E.P.S. simulator can simulate a six generator, thirty node power system in real time in enough detail for the simulation to be used for operator training and the development of national scale control algorithms. It would be advantageous to be able to model a system more representative of a national power system, with the approximately eight hundred node, eighty generator C.E.G.B. transmission system being the target size.

The simulator is aimed at producing results in real–time which are suitable for being passed to the control computer in place of results from the real C.E.G.B. transmission system. Experience with the full, coupled, Newton–Raphson iterative solution algorithm applied to similar sets of equations has shown that convergance is usually achieved within ten iterations, unless a particularly poor initial set of values is present, with five to seven iterations being normal after a reasonable system transient. The target number of iterations per time–step should therefore be set at about ten. Any less than this might cause the simulator to drop out of real–time operation without a severe system transient. A transient resulting from a topology change might still cause problems with this target, depending on the speed of the reconfiguration. The matrix solution only takes part of the time per iteration, because the matrix needs to be formed, which would take slightly

less than half the iteration time, so a target for the matrix solution of approximately 40ms should be satisfactory to achieve ten iterations per second.

The work is part of a larger project which is investigating the development of a simulation language which is particularly suitable for the real–time simulation of transmission and distribution networks, not only for electricity, but also for gas and water. The connectivity of these systems is similar, but the method of analysis and the resulting models differ widely between the systems, so minimal information is presented about the models used foe electrical power systems. The simulation language would process the algebraic and differential equations of the models input by a user, and generate much of the simulator program automatically. The intermediate internal representation used for the equations allows the code produced to be modified for other machines or languages with little additional work.

The special techniques required for the solution of the highly–sparse equations which represent the whole system state must, however, for efficiency, be coded specially for the intended hardware. A considerable proportion of the current simulator's execution time is spent in the formation and solution of these non–linear, sparse, matrix equations. An increase in the simulation speed can be achieved by using faster processors, but this is limited by current technology. It is also possible to increase speed by performing many of the calculations in parallel, but while many parts of the simulator split naturally between procesors, problems exist for combining the results of the different parts, and in particular for obtaining a global solution state for the system. Any use of parallel processors requires that the calculations for the matrix solution should split effectively, otherwise the possible increase in speed would be low.

An increase in the matrix solution speed can be obtained by the use of faster processors, by splitting the solution between processors or by algorithmic improvements in the solution method, both global and specific for the chosen harware. All of these possible paths are considered in the work presented here.

1.8 Layout of the thesis.

The research presented in this thesis may be grouped into four main topics, three of which are closely related to the fast, numerical solution of highly sparse matrices which result from the analysis of power system networks. The fourth topic concerns the identification of what components of the power system are actively participating in the system, and their processing to reduce the number of items to that which is necessary for the remainder of the simulator to produce an accurate simulation. This topic is presented in chapter 9.

Chapters 3 to 8 investigate aspects of the numerical matrix solution, after chapter 2 has discussed the mathematical and modelling techniques which determine the form of the matrix equation. Chapters 3 and 4 investigate the sparsity ordering of the matrix, and the form that the numerical calculations will follow. Chapters 5 and 6 investigate how the calculations interact with computer hardware, and develops a special–purpose processor which is dedicated to the solution of of the equations. Chapters 7 and 8 investigate how these calculations can be split between several processors in order to reduce the time taken to produce a solution. The conclusions of the research are presented in chapter 10.

Chapter 2 discusses the formulation of the equations for the simulation of the power system, and methods for their solution. The other O.C.E.P.S. simulators are described.

Chapter 3 describes the numerical algorithm currently used by O.C.E.P.S. to solve sparse matrices, and develops two new modifications to enhance performance for some processors. The first of these involves the re–packing and re–arrangement of the floating–point matrix terms to reduce memory requirements and permit more natural orders for the calculations to be employed. The second modification is the automatic run–time generation of efficient code dedicated to the factorization and solution of one particular matrix structure, instead of repeatedly interpreting the matrix structure during each solution operation. While this has been tried before, the integration with a special purpose instruction set is new.

Chapter 4 investigates the performance of the ordering criteria which are available for sparse power system matrices, and proposes two new methods which offer significant time savings over those developed elsewhere, and which produce other effects which might be useful in other power system fields of work.

Chapter 5 investigates the hardware on which the simulator could be based, with particular emphasis on the solution of the matrix equation. The strengths and weaknesses of current processor designs is discussed. The desirable features for a processor for sparse matrices are investigated.

Chapter 6 develops a paper design which theoretically offers a much shorter solution time than is available using standard processor configurations. The code produced by the code generator described in chapter 3 is shown to map very well to this architecture.

Chapter 7 describes one method of splitting the matrix solution for parallel processing, and investigates a new variation which attempts to reduce the penalty of network splitting by restricting the connectivity of the areas. A method of generating the matrix splits based on the elimination ordering of the sparse matrix is proposed and its performance is investigated. These results are compared to those produced by the methods of chapter 8.

Chapter 8 describes the use of two optimisation schemes to generate the network splits. An investigation was performed with an optimizer based on simulated annealing, and results from this were compared to splits produced by a new optimizer written on the principal of Genetic Optimisation, which appeared to be well suited to network tearing.

Chapter 9 investigates the affects of these changes to the current O.C.E.P.S. simulator on a tracking topology identification routine, and produces a topology routine that executes in parallel using the same network split as the factorisation routine.

Chapter 10 presents the conclusions drawn from the work contained in this thesis.

Chapter 2.

Power System Mathematics.

2.1 Introduction.

The simulator works by solving the equations which are used to represent the system, subject to the current imposed system states, such as known loads and connectivity. Once a solution which is consistent within the numerical tolerances which are acceptable has been produced, the system time can be incremented, which might impose new external conditions on the system, such as different load values, connectivity changes or set points, and the new solution can be calculated. This incremental solution process continues for future time–steps, but requires an initial system state, which is used to help convergence, and also, for example, to set the starting values for loads, the parameters of the models of which depend on local voltage levels.

The solution phase of the simulator may be iterative, in which case the set points and external conditions are kept constant, while the load, transmission line and generator parameters are adjusted between iterations. If further levels of iteration, within each major iteration of the solution,are present, then the system state is kept constant while convergence of the inner iterations is sought.

Simulators differ in how they represent the equations relating to the algebraic and differential models. The equations are interdependent, since the network solution depends on the generator's output power, voltage and frequency, while the generator depends on the busbar voltage to which it is connected, and the power balance at that node. Two forms for the solution of these equations are in common use, since they may either be solved together in one matrix, or separately, one after the other.

The equations that represent the system state are non–linear, but the basics of the solution of linear equations should be discussed because many methods for the solution

of non–linear equations perform local linearisation about the current solution point to convert the non–linear equations to linear equations which are simpler to solve. The solution to these transformed equations can then be used as the next linearisation point to produce a new set of equations. Successive solutions are compared, and when the difference between successive solutions is less than some error criterion, or tolerance, the solution is considered to be correct.

2.2 Solution of Linear Equations.

A linear equation is an equation in which the independent variables are combined together in a linear manner, so that they are not raised to powers or multiplied together. The equation defining a dependent variable is simply a weighted sum of some, or all, of the independent variables. A set of equations will in general, only posess a unique solution if the number of independent variables is equal to the number of independent constraining equations. A surfeit of variables usually requires a *best fit*; i.e., a solution which minimises some measure of error, while a surplus would usually result in many acceptable solutions. A set of linear equations with the same number of equations as independent variables can be written in matrix form, and either solved iteratively or directly by elimination.

2.2.1 Elimination methods.

The simplest linear equation is:

$$y = ax \tag{2.1}$$

which has a direct matrix counterpart where $\mathbf{A}$ is a square matrix,

and $\mathbf{x}$ and $\mathbf{b}$ are appropriate vectors:

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{2.2}$$

which can be solved in a similar manner to the scalar case

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} \tag{2.3}$$

Here, $\mathbf{y}$ is the vector of known values, or the dependent vector, and $\mathbf{x}$ is the vector of unknown values, or the independent vector. The square matrix $\mathbf{A}$ contains the coefficients which relate each element in $\mathbf{y}$ to elements in $\mathbf{x}$.

The matrix inversion of $\mathbf{A}$ takes a similar rôle to scalar division. Although this form of equation 2.3 is used in matrix algebra, it is not usual to form the inverse and perform the multiplication, because this is inefficient and requires significantly more operations than are required by other methods which obtain the solution. A matrix inverse can be calculated by factorizing the matrix into lower and upper triangular halves, and then solving for a succession of unity vectors to build the inverse. It is more efficient to solve for the independent vector directly once the factorized form is available.

The factored form is derived from the elimination method used to solve small sets of simultaneous equations, and involves similar calculations, although several forms exist, such as Gauss, Cholesky and Crout Elimination [26]. Each of these methods has its advantages for certain situations, e.g., Cholesky is for positive definite matrices, and Crout performs the elimination in–place, and thus requires less memory than the other methods. The Crout method only modifies each term once, and so it was very suitable for hand calculations with calculators or slide–rules, since few terms had to be written down. This might also reduce round–off errors in modern computers, since summations could be performed internally to higher precision than the representation in external storage would allow.

A matrix $\mathbf{A}$ is known as positive definite if it is symmetric about its leading diagonal, and if for any non–zero vector $\mathbf{x}$, equation 2.4 is satisfied.

$$\mathbf{x}^T\mathbf{A}\mathbf{x} > 0 \tag{2.4}$$

The matrices resulting from power system networks are usually location symmetric, but total symmetry is destroyed by phase shifting transformers. Matrices which contain representations of differential equations are not location symmetric, but the relatively few asymmetric terms resulting from the generator models can have dummy mirror elements generated to maintain structural symmetry, if a particular method requires this. If a matrix is positive definite, then the solution will be numerically stable without special measures being taken, otherwise some of the terms can grow very large, with a resulting loss of precision during summations, which results in numerical instability and possible failure to find a solution.

The elimination methods form two or three factor matrices, and as these are formed, the locations in the original matrix are zeroed by combinations of linear operations on

whole rows or columns. The two factor matrices will be triangular in shape, one with non–zero elements above and to the right of the leading diagonal, and the other with them below and to the left. The leading diagonal is handled differently by the methods. Some include it in the lower factor, some in the upper factor, some square root it and include it in both factors, while others keep it separate in a vector, which can save storage space and some indexing calculations.

Taking the simple lower—upper factorisation:

$$\mathbf{A} = \mathbf{LU} \tag{2.5}$$

then

$$\mathbf{b} = \mathbf{LUx} \tag{2.6}$$

introducing an intermediate vector $\mathbf{y}$

$$\mathbf{y} = \mathbf{Ux} \tag{2.7}$$

$$\mathbf{b} = \mathbf{Ly} \tag{2.8}$$

Since $\mathbf{U}$ and $\mathbf{L}$ are triangular, equations 2.7 and 2.8 can be solved by substitution. Consider the simple case of the three–by–three matrix of equation 2.9.

$$\mathbf{A} = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 1.0 & 4.0 & 2.0 \\ 2.0 & 1.0 & 1.0 \end{pmatrix} \tag{2.9}$$

A zero can be placed into the lower two entries in the first column by subtracting the first row from the second, and twice the first row from the third. If these factors are placed in the lower triangular matrix, then:

$$\mathbf{A}_1 = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & -1.0 \\ 0.0 & -3.0 & -5.0 \end{pmatrix} \tag{2.10}$$

$$\mathbf{L}_1 = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 \\ 2.0 & 0.0 & 0.0 \end{pmatrix} \tag{2.11}$$

If one and a half times the new second row is added to the third of equation 2.10, this will place a zero into the second column of the third row without affecting the

first column. This leaves the upper and lower triangular factors of the matrix **A** in equation 2.9 in equation 2.12 and equation 2.13 respectively.

$$\mathbf{U} = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & -1.0 \\ 0.0 & 0.0 & -6.5 \end{pmatrix} \tag{2.12}$$

$$\mathbf{L} = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 \\ 2.0 & -1.5 & 1.0 \end{pmatrix} \tag{2.13}$$

If the solution to $\mathbf{b} = (1.0,\ 2.0,\ 3.0)^T$ is desired, then the same operations are applied to this column vector as were applied to the upper triangular half of **A**.

The top element is subtracted from the second element, and twice the top is subtracted from the bottom. The second element is then divided by two, and three times this is added to the bottom element to give a vector of $\mathbf{y} = (1.0,\ 1.0,\ 2.5)^T$. This can be seen to be equivalent to solving equation 2.14

$$\mathbf{b} = \mathbf{L}\mathbf{y} \tag{2.14}$$

$$\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 \\ 2.0 & -1.5 & 1.0 \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}^T \tag{2.15}$$

The next series of operations is to backward substitute for the elements of **x** using those of **y** as dependent variables. The bottom row of **U** contains only one non–zero value, so there is only one unknown, one multiplier and one known value in the equation represented by this row. This can be solved by dividing the known element by the non–zero term in the bottom row of **U**. Once this element has been found, the next row up only contains one unknown, so this can now be determined by subtracting the product of the rightmost element in this row by the value just found from the corresponding element in the undefined vector, and dividing by the diagonal element. This is repeated for the top row, to give the solution vector of equation 2.16.

$$\mathbf{x} = (\tfrac{20}{13},\ \tfrac{4}{13},\ \tfrac{-5}{13}) \tag{2.16}$$

Which is obtained from:

$$\mathbf{y} = \mathbf{U}\mathbf{x}$$

$$\begin{pmatrix} 1.0 \\ 1.0 \\ 2.5 \end{pmatrix} = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & -1.0 \\ 0.0 & 0.0 & -6.5 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{2.17}$$

2.2.2 Operation counts for elimination methods.

The calculations have the same form for larger matrices, but it will be seen that the operation counts grow quickly, because the rows are both longer on average and there are more of them. With current computer hardware, multiplication and division take significantly longer than addition, so it is the totals for these operations which are really significant. There are also an increasing number of processors which can combine an addition with a multiply in the same instruction for little penalty above a standard multiplication. One division is required per row of the matrix. In the above example, the divisions were performed during the solution phase, but this is undesirable if many solutions are required for the same matrix, so the L and U factors are frequently scaled, and the divisions removed from the solution stage. This is the reason for the division entry spanning the factorize and solution columns in table 2.1. Obtaining the factored forms can be seen to be of order $O(n^3)$, which will dominate the n^2 terms for large n.

	Operation counts for matrices of size n	
	Factorisation	Solution
Additions	$(n^3 - n)/3 + n^2/2 - n/6$	$n^2 - n$
Multiplies	$(n^3 - n)/3$	n^2
Divisions	n	

Table 2.1 Operation counts for factorisation and solution.

The inverse can be calculated from n of these solutions, each on a vector taken from an identity matrix of size n. The multiplication count for the full inverse is therefore $(4n^3 - n)/3$. A full matrix-vector multiply is required for each subsequent solution, which takes the same number of operations as each solution calculated directly from the factors themselves, so calculating the inverse is computationally inefficient. It will also produce less accurate results, because rounding will have occurred during the formation of the inverse.

2.2.3 Accuracy of elimination methods.

Elimination methods are exact. That is, they provide, within the constraints of the precision used for the calculations, an exact result, without any iteration or adjustment. The caveat of numerical precision is important in some cases, where the matrix is poorly conditioned, with large mismatches in element size, resulting in loss of precision during addition and subtraction operations performed using a limited number of digits, as is the case for all computer arithmetic. Much work has been devoted to minimizing these round–off errors [26], and most matrix codes use either pivoting or partial pivoting [20], whereby the largest element in a row/column is exchanged with the diagonal element, which limits the growth of other elements. A record must be kept of these interchanges during the elimination, so that they can be 'undone' during the substitution phase.

If the matrix is positive definite, then no pivoting is required to maintain accuracy. Power system matrices are almost positive definite, and in general, provided that sufficient precision is available to cope with the range of impedances in the system, the numerical matrix solutions will be stable. The representation of the power system equations, coupled with the physical plant characteristics, result in power system matrices which are usually well conditioned, so that pivoting is not required for stability, using standard computer hardware.

2.2.4 Iterative methods.

It is also possible to obtain the solution vector by iteration, just as it is possible to solve a scalar equation such as equation 2.1 by this method, but similar difficulties are encountered with stability and rate of convergence to the scalar case. Iterative methods work by guessing initial values in the independent vector, and using these to evaluate an initial solution vector, which is compared to the actual solution vector, for which the independent vector is desired. The mismatches are then scaled, possibly further adjusted, and fed back into the independent vector. This is repeated until the adjustments are less than a certain tolerance, when the solution is deemed to be complete. The presence of a tolerance means that iteration methods are unlikely to produce an exact result. The simplest method, Gauss Iteration [20], processes all values of x of iteration k, (that is $x_j^{(k)}$),

to form $x_j^{(k+1)}$, before updating the x vector. This can be expressed by equation 2.18.

$$x_j^{(k+1)} = x_j^{(k)} + \alpha_j \left(\sum_{\substack{i \neq j \\ i=1}}^{n} (a_{i,j} x_i^{(k)}) - b_j \right) / a_{j,j} \qquad (2.18)$$

A variant, the Gauss–Seidel [20] Iteration method, defined in equation 2.19, updates the unknown elements as the corrections are obtained, thereby always using the most recent values that are available. This method requires less storage, and usually has better convergence than the Gauss Iteration method. Note that the summation is broken in two in this representation, but in an actual program, this would not be done, since the x^k and $x^{(k+1)}$ elements share the same array, instead of requiring two different arrays, the usage of which is alternated on successive iterations.

$$x_j^{(k+1)} = x_j^{(k)} + \alpha_j \left(\sum_{i=1}^{j-1} (a_{i,j} x_i^{(k+1)}) + \sum_{i=j+1}^{n} (a_{i,j} x_i^{(k)}) - b_j \right) / a_{j,j} \qquad (2.19)$$

Careful adjustment of the feedback by the α_j term is required with both Gauss and Gauss Jordan iteration methods to reduce instability and improve the convergence rate. The adjustment could be made on an individual basis for each row of the matrix, or more commonly, the same factor is applied to every row. If too much correction is fed back, then instability will start, and if too little is applied, then the method would appear to be damped and only converge slowly. Factors between 1.2 and 1.8 are commonly used. The application of these factors slows the algorithm, so their use for speed–ups of less than 1.2 would probably be counter–productive. Lower values may be used to overdamp the iterations if stability is found to be a problem. The feedback may need to be set differently for different scenarios, such as steady–state and transient system states, and might be adjusted depending on the maximum mismatches in the most recent iteration.

An initial guess reasonably close to the solution is required to start the iteration process, which for a large set of equations is not easy to produce without some form of system analysis, using a method which either does not require initial guesses, or will converge from a flat start. An elimination algorithm would probably be used to perform an initial load–flow to set the voltage and current levels to their initial states. Similar, but less extensive initialisation would be required following a change in the network which results in new or changed nodes in the matrix.

Best results are achieved if the matrix is positive definite. If this is not so, then the method may fail to converge. Equation 2.9 is not positive definite, and the iterative solution using Gauss Jordan Iteration completely failed to converge. The elimination method achieved a solution, but probably lost a little precision during the calculations, which would hardly be noticed with reasonable numbers on such a small matrix, but which would become much more noticeable as the errors accumulate for larger matrices.

2.2.5 Operation counts for iterative methods.

The iterative nature of the process makes the determination of operation counts less meaningful than for the elimination methods. The multiplications required to perform each iteration on the matrix of size n is $n^2 - n$, or n^2 if speed–up factors are used. There will also be n^2 additions per iteration. The divisions can be moved so that only n are required for the whole iteration process. The iteration method appears to be of order $O(n^2)$, whereas the elimination method was of order $O(n^3)$. The operation counts for an iterative solution would therefore be expected to increase more slowly with increases in system size than for solution by elimination. The number of iterations would not be expected to grow proportionally with matrix size, but each solution may require the formation of a new matrix, and this could penalise the method as the system grows in size. Flaxman [43] found that iteration was slower than elimination methods, and that convergence was problematical with power system matrices, and reverted to an elimination based solution method. This must be due to the increased number of re–formations of the system matrix due to the increased number of iterations.

2.3 Solution of Non–linear Equations.

A non–linear equation is one which contains non–linear combinations of the independent variables. A non–linear set of equations may be solved by linearising them about the current solution point, and then moving the solution point so that if the linearised equations were re–evaluated at the new solution point, the error should have decreased. Instead, the non–linear system is linearised about the new solution point, and the error in the solution re–evaluated. This process continues until the difference between two successive solutions, or the size of the error, is below the acceptable tolerance limit. The latest solution point is taken as the solution to the non–linear equations.

2.3.1 Newton–Raphson Iteration.

The familiar scalar form of Newton's Method [87] used to solve scalar non–linear equations is given in equation 2.21 which is used to find the values of x for the non–linear function of equation 2.20.

$$f(x) = 0 \tag{2.20}$$

$$\Delta x = -f(x)/f'(x) \tag{2.21}$$

The adjustment is defined in terms of the error $f(x)$ and the slope of the curve at the position of the error $f'(x)$, so that the new approximation will be made where the tangent of the curve from the current estimate meets the x–axis. This is repeated until the adjustment, Δx, is within a tolerance band, so that subsequent adjustments of x would be small.

Matrix equations of the form of equation 2.22 are solved using the matrix form of Newton's Method given by equation 2.23 where $\mathbf{f}$ is a vector of dependent values resulting from the latest state of the system, $\mathbf{J}$ is the Jacobian matrix, and $\Delta\mathbf{x}$ the adjustment of the independent vector to form the next trial solution.

$$\mathbf{f(x)} = \mathbf{0} \tag{2.22}$$

$$\Delta\mathbf{x} = -\mathbf{J}^{-1}\mathbf{f(x)} \tag{2.23}$$

The Jacobian matrix consists of the partial derivatives of the each term in each function with respect to each term in the independent vector $\mathbf{x}$. Like the scalar case, the solutions are iterated until the elements in $\Delta\mathbf{x}$ are within the tolerance band for an acceptable solution.

For power systems, the elements of the Jacobian matrix and the dependent vector depend on the independent vector of the system, so the Jacobian and the dependent vector should be re–formed for every iteration. Each iteration involves using the current estimate of the independent vector to form both the Jacobian matrix and the dependent vector, then solving for the adjustments to be made to the independent vector, and checking for convergence in the adjustments to see whether more iterations are required. The elements of the independent vector are then updated. If more iterations are necessary, the whole loop is repeated. It is also possible to alter the update of the Jacobian matrix for the fully–coupled formulation, either updating a sub–set of the columns each

iteration, or only updating the matrix after several iterations. This reduces the work per iteration, but generally increases the number of iterations until convergence [99]. If the Jacobian changes little between iterations, the degradation in convergence should be small, while otherwise, the convergence may be greatly reduced, or non–existent.

The method produces a similar convergence rate to the scalar case, with quadratic convergence as the solution state is approached. Initial convergence rates can be slower than this, if the trial solution is not close to the final one. The reason for this can be seen by examining the scalar case. It is possible for the slope of the curve at the trial solution point to be either relatively flat, or to be in the wrong direction. This will move the next point away from the actual solution, hopefully to return in future iterations. As the solution point is neared, such dramatic changes in the slope of the curve would vanish, resulting in very fast convergence. This is also apparent in the truncation of the Taylor Series at the second term [87]. Most solutions start with a good initial approximation to the solution, from the previous time–step, and it is rare for the method not to converge, or to converge on a different solution, if one exists. Recent studies of strange attractors have demonstrated interesting behaviour from Newton's Method [47] [107] when investigating certain scalar functions, but no such behaviour is evident with power system studies. The method does sometimes fail to converge from the initial flat start, and a selection of remedies are available [134] to cope with this rare eventuality. This is only important at the start of the simulation, but is more common in load–flow calculations, since flat starts are used more for these calculations than during simulation.

Quadratic convergence approximately doubles the number of correct binary bits in the answer each solution, so once the solution is neared, convergence is extremely rapid to the limit of the floating–point representation used to store the data, subject to small rounding errors present in the formulation of the equations.

2.3.2 Gauss–Jordan Iteration to solve non–linear equations.

It is also possible to solve non–linear sets of equations by iteration using Gauss–Jordan [26] iteration. The initial matrix is formed, thereby linearising the system about the state defined by the current independent vector. This linearised system is then solved, by some method such as elimination or Gauss–Jordan iteration itself, until the solution

is correct to within a tolerance band. These new values for the independent variables are then used to linearise the system about the new state, and a new dependent vector and matrix are produced, which are again solved. This process is repeated until the solutions of the outermost process have converged to within the tolerance band. Similar difficulties exist over convergence and stability as for the iterative solution of linear equations.

2.4 Possible formulations of the system equations.

There are two possible formulations for the network solution.

2.4.1 Current—voltage representation.

The first widely–used representation is based on Ohm's Law, which relates voltages (V), currents (I) and the system admittance matrix (Y), and Kirchoff's Current Law, which states that the sum of all currents at a node must be zero. Each of these quantities is represented by a complex number so equations 2.24–25 represent the system.

$$(I) = (Y) \times (V) \tag{2.24}$$

Where:

$$y_{ij} = g_{ij} + jb_{ij} \tag{2.25}$$

and I and V are defined in figure 2.1.

Currents flowing in a power system can be from connections to other nodes, from generators, to loads and for line–charging. The connection currents are related to the line impedances and the voltage differences between nodes. Line charging is calculated from the node voltage and line susceptance, and is featured in the calculations as a current to ground. Only the load and generator currents appear in the I matrix, so non–zero values will only occur for nodes connected to loads or generation.

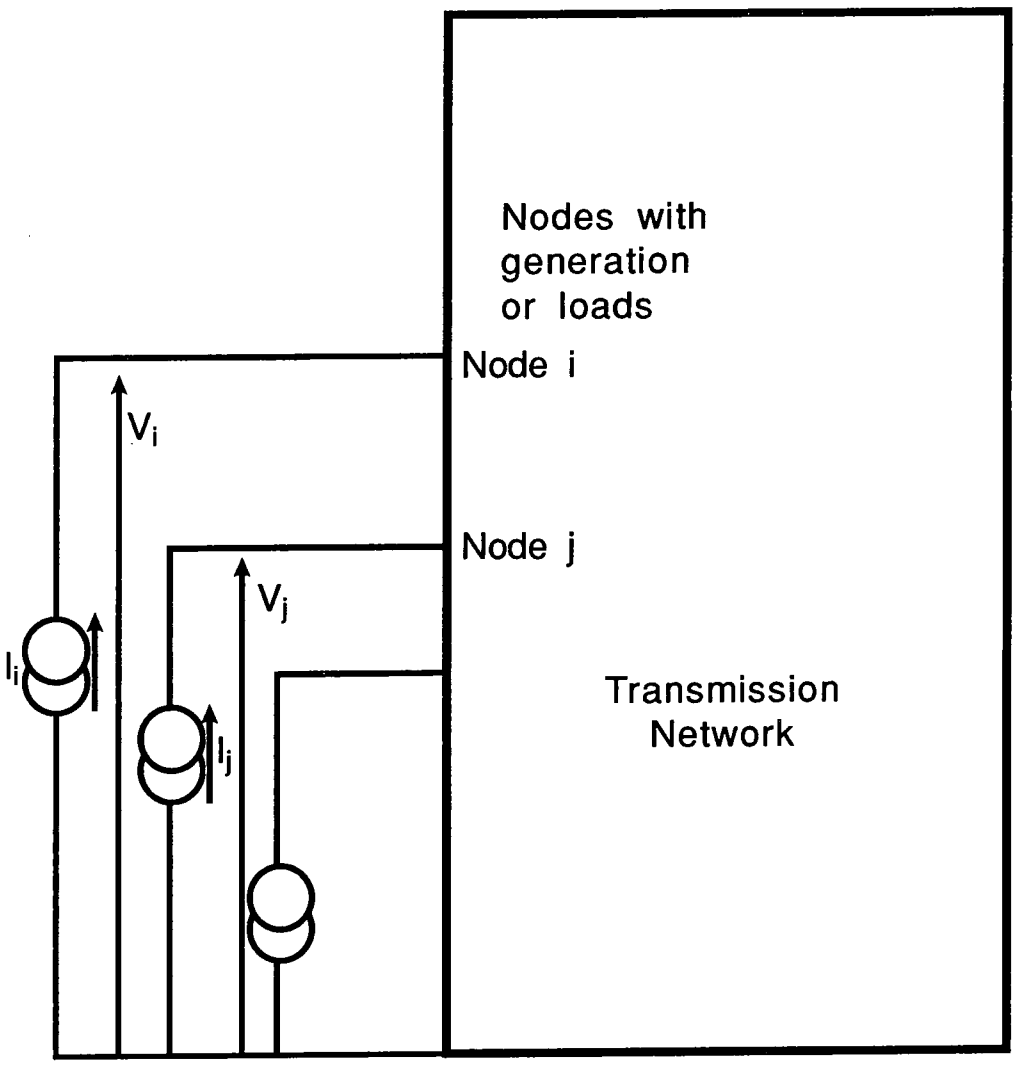


Figure 2.1 Multi–terminal network representation of a power system.

2.4.2 Loadflow representation.

The alternate form for the network equations is the one usually used for loadflows [75], which relates real (ΔP) and imaginary (ΔQ) power mismatches at nodes to voltage magnitude (ΔV) and phase angle ($\Delta\Theta$) changes at every node. Equation 2.26 shows the form of the blocked matrix equation, with the J_x elements being Jacobian submatrices which consist of partial derivatives of either ΔP or ΔQ with respect to either $\Delta\Theta$ or ΔV. The Jacobian sub–matrix which links real power ΔP and phase angle $\Delta\Theta$ is shown in equation 2.27.

$$\begin{pmatrix} \Delta P \\ \Delta Q \end{pmatrix} = \begin{pmatrix} J_1 & J_2 \\ J_3 & J_4 \end{pmatrix} \times \begin{pmatrix} \Delta\Theta \\ \Delta V \end{pmatrix} \tag{2.26}$$

where:

$$J_1 = \begin{pmatrix} \partial(\Delta P_1)/\partial(\Delta\Theta_1) & \partial(\Delta P_1)/\partial(\Delta\Theta_2) & \dots & \partial(\Delta P_1)/\partial(\Delta\Theta_n) \\ \partial(\Delta P_2)/\partial(\Delta\Theta_1) & \partial(\Delta P_2)/\partial(\Delta\Theta_2) & \dots & \partial(\Delta P_2)/\partial(\Delta\Theta_n) \\ \vdots & \vdots & \ddots & \vdots \\ \partial(\Delta P_n)/\partial(\Delta\Theta_1) & \partial(\Delta P_n)/\partial(\Delta\Theta_2) & \dots & \partial(\Delta P_n)/\partial(\Delta\Theta_n) \end{pmatrix} \tag{2.27}$$

The power mismatches at each node are fixed, since unless there is generation or load at a node, there can be no power mismatch. Where loads and generators exist, the power flows are calculated using the current system states, and the the voltage and phase angles throughout the system may then be calculated.

The loadflow representation requires one node to be separated from the system calculations, to act as the reference or *slack* bus. The power mismatch at this bus are allowed to float, so it should have generation attached which can meet this mismatch, and the voltage magnitude and phase angle are defined, usually as 1.0, or a little above, and 0.0 respectively. Larger values tend be used for the voltage since the bus is connected to a generator, so would be expected to have a slightly higher voltage than the remainder of the system buses. Increasing this number would place more of the other buses in the correct voltage range for the system. In order for this to be valid, the bus must be connected to a generator which can fulfil the power mismatch. If this generator becomes inactive, the problem must be reformulated due to the different equations required. If the system splits into several unconnected electrical islands, a separate node must be chosen in each island to act as the slack bus, so islanding causes the representation of some buses to change.

2.4.3 Coupling and Decoupling.

In the loadflow representation, the coupling between real power and phase angle is strong, as is the coupling between reactive power and voltage magnitude, while other couplings are usually relatively weak. The problem size can be halved, and the solution processed in parallel, if these coupling terms are made zero [133]. This results in both the off–diagonal block Jacobian sub–matrices in equation 2.26 being made zero, which results in equation 2.28.

$$\begin{pmatrix} \Delta P \\ \Delta Q \end{pmatrix} = \begin{pmatrix} J_1 & 0 \\ 0 & J_4 \end{pmatrix} \times \begin{pmatrix} \Delta \Theta \\ \Delta V \end{pmatrix} \tag{2.28}$$

$$(\Delta P) = (J_1) \times (\Delta \Theta) \tag{2.29}$$

$$(\Delta Q) = (J_4) \times (\Delta V) \tag{2.30}$$

This is the basis for the Decoupled Newton Loadflow. The convergence is not as fast as the Coupled Newton Loadflow, being more linear, but the overall speed is increased due to the smaller matrices which are faster to solve, and require less computation to generate. As each half is processed, the solution to the other half would be expected to deteriorate slightly, and this must be incorporated into the check for convergence.

2.4.4 Fast Newton Loadflow.

Further savings in time and complexity can be made by noting that the phase difference between the two ends of a transmission line are likely to be small, and that the inductance would be greater than the resistance for most lines, so this dominates in determining power flow [24 82 38 115 135]. If the generators, loads and line charging currents are removed from the matrix, then the matrix values will only change when a topology event occurs. This removes the re–calculation of the matrix terms every iteration of every time–step. The terms removed from the matrix are inserted into the power mismatch vector, so they are still considered. Other changes are introduced to remove terms from the remaining Jacobian matrices where their contribution to the answer is small. In particular, the angle change of phase shifting transformers are removed from J_4, so that this becomes symmetrical, and therefore can be processed more efficiently than the matrix J_1.

Despite the approximations which are made during each iteration, the method is still searching for the correct solution because the power mismatches (the dependent vector)

are evaluated using the full formulae. Convergance rates are similar to the Decoupled Newton Method, so convergance may be slower than the fully coupled method, particularly as the correct solution is neared, or fail completely. The initial convergence rate can be better than the fully coupled method, because the coupled method tends to over–compensate for large errors due to non–linear dependencies between terms. Once the correct solution is approached, however, the Coupled Newton Method converges faster.

2.5 Generator solutions.

The differential equations which represent the generators can be solved together with, or separately from the network algebraic equations. If they are solved together, then the generator equations are combined with the algebraic equations into the same matrix, while if they are solved separately, then the two solutions communicate by elements in the state vectors, which in physical terms could be complex current flows, power flows or voltages, in addition to the island frequency.

2.5.1 Separate algebraic and differential solutions.

If these equations are solved separately, then at each time step, the algebraic matrix equations are solved, usually by loadflow, and then the results from these are presented to the differential equations, which are solved using an explicit integration method, such as Runge–Kutta [20]. This will produce a new set of values at the busbars to which the generators are connected, which will be different from those calculated by the network solution. A further solution of the algebraic equations may then be performed, but the explicit nature of the integration means that it can only be executed once per time–step. The network solution is therefore out of step with the generators, since the solution of the network equations would have altered the values at the busbars to which generators are connected.

The Runge–Kutta integration also requires a short time–step for stability, compared to the shortest event for which modelling is desired. A time step is the interval between solutions in the modelled devices time–scale, i.e., it is the increment in time between solutions. To save computer time, it is advantageous to remove the shorter time–constants from the generator models, since their effects at time–steps of a second or

more is negligible, but the Runge–Kutta method requires a more complex model for the shorter time–step. A small time–step will also limit the size of the mismatch betwen the generator and network solutions. The generators are represented as voltage sources behind impedances.

2.5.2 Combined algebraic and differential solution.

If the differential and algebraic equations are solved together, then the differential equations must first be integrated using an implicit integration method, such as trapezoidal integration [20]. This has proven to be more stable than explicit integration methods, and is capable of accuracy for time–steps which are considerably longer than the shortest time constants in the models [77] [114]. These integrated formulae are then solved together with the algebraic equations. This method also requires iteration of the solution because of the non–linearity in the equations, but there will be no discrepancy between the differential and algebraic solutions. The combined method is more accurate and stable, while the separate method is faster.

A problem encountered when performing loadflows is that the representation of nodes with generation can be different from that of nodes without. If a generator is generating normally, then the known values at its busbar are real power and voltage, hence the node is called a PV node. When the reactive power limit is reached, however, the generator cannot hold its voltage set point, so the voltage is no longer known. Reactive power now is known, since the generator is running at its reactive power limit, so the node becomes a PQ node. This changeover must be handled during loadflow solution, and would also occur during simulation, and causes a re–formulation of the equations for that node and generator. If the generator equations are linked in with the network equations, this difficulty is avoided, since all values are self–consistent, regardless of the operation of the generator.

2.6 Topology.

The modification of the network topology during the simulation, both due to external commands and self–generated signals from plant protection models, requires that the simulator must be able to create and delete modelled elements, and reconfigure connections between them in real–time during a simulation run. The state of elements in the network determine what elements actually exist to be modelled in the numerical simulation, so it is possible for a circuit breaker changing state to generate a new node in the network, possibly creating a complete, new, electrical island. This behaviour is not common to most other simulators, which deal with fixed systems, or complete models being energised and deenergised. Topology determination is described in more detail in chapter 9, and a topology determination routine which will support the developments described in the remainder of the thesis is proposed.

The equations that result from the analysis described in sections 4–5 are formed into a matrix to permit their systematic solution. An example from an actual power system is shown in figure 2.2, which is the matrix representation of the network of figure 2.3. The topology of the network determines the location of the non–zero elements in the matrix, since an element in the matrix is only required where there is a physical link between two quantities. If there is no direct connection between two nodes in the system, then there will be a zero in the locations in the matrix which correspond to the lack of the link. Since there are relatively few direct connections from each node, the portion of the matrix which represents the network is highly sparse. The blocks representing the generators are less sparse, but much smaller, embedded in an otherwise empty block of rows and columns, so sparsity is present throughout the matrix. If advantage can be taken of this sparsity, then the number of calculations can be reduced from those required for a full matrix, because it serves no purpose to calculate with values that are known to be, and remain, zero.

2.6.1 Sparsity.

The solution of sparse systems of equations has been the subject of much research, because of the frequency with which large, sparse sets of equations appear during the analysis of large physics and engineering problems. The type of sparsity which results

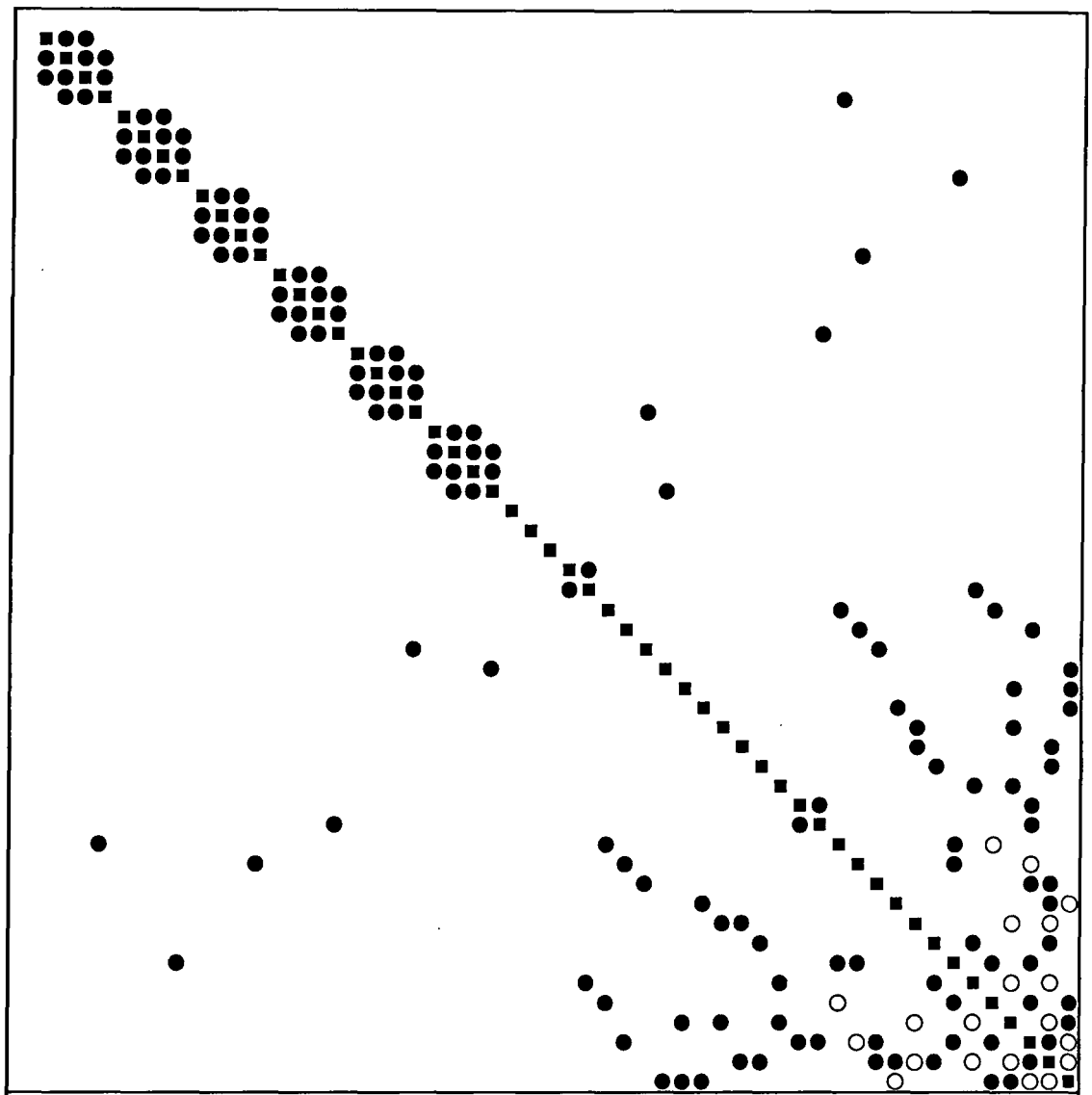


Figure 2.2 Structure of unordered 30 node network with generators.

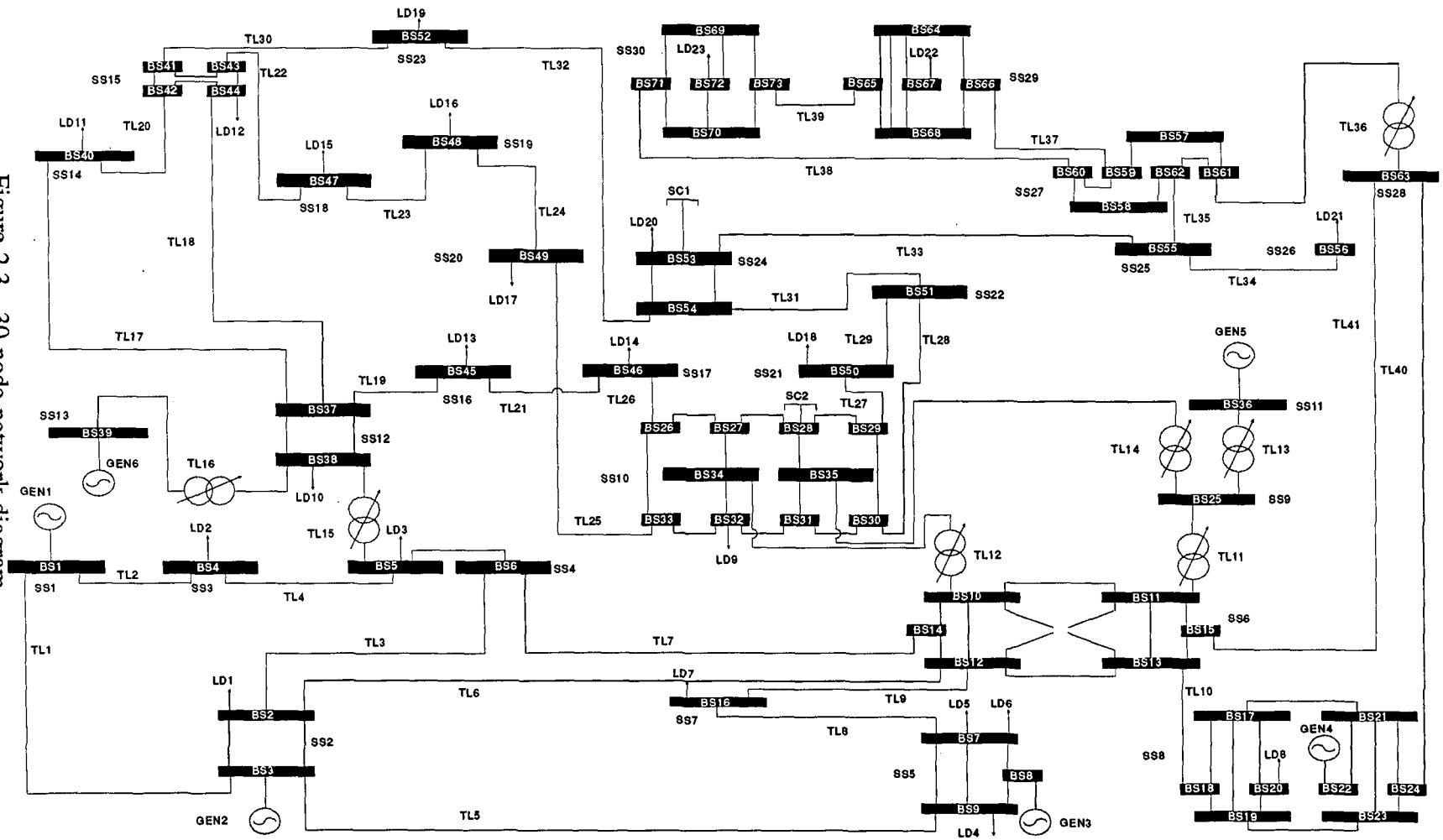


Figure 2.3 30 node network diagram.

from the analysis of networks is governed by the connectivity of the matrix, and requires special methods to maintain sparsity. This is the subject of chapters 3 and 4.

2.7 O.C.E.P.S. Simulation.

The current O.C.E.P.S. simulator [114] is hosted by a Perkin–Elmer 3230 and executes the numeric matrix solution and accompanying calculations on an FPS-5205 array processor. The simulation configuration is shown in figure 2.4. The array processor is a descendent of the AP-120 used in much of the early power system work, and uses a single 38 bit representation for floating–pont data. This representation is non–standard, but carries sufficient precision to ensure numerical stability for most power system problems. The processor gains over processors using standard floating point formats, because the I.E.E.E. 32 bit format does not have sufficient precision, so double precision must be used. This normally involves an extra bus cycle for each data value fetch or store, because of the longer word length.

The O.C.E.P.S. configuration executes the main control algorithms on a VAX-8600, which send commands to the Perkin–Elmer simulator host via an industry–standard S.C.A.D.A. computer. The data from the simulator is sent via the S.C.A.D.A. to the VAX and into the control package. This provides a realistic interface to the control package from the simulator, as in the real system, data would be read in from the S.C.A.D.A. and sent via direct lines.

The current test network is the I.E.E.E. 30 node test network, which is handled in real time, both by the control algorithms and the simulator. Work is in progress towards upgrading to either the 118 node I.E.E.E. network, or a 141 node British network.

The accuracy and stability of the existing simulator were proven in a CEGB case study [74] involving post–processing data from a severe system incident, when good agreement with the actual event, including the tripping of protection, was achieved. The system used contained 141 nodes, including some lumped nodes representing large parts of the United Kingdom, and both local and lumped generation and loads.

An unexpected instability did occur, which forced the use of a small (quarter second) time–step, but the cause was later discovered and rectified. The system state at each node is represented by a real and imaginary voltage, which effectively gives magnitude

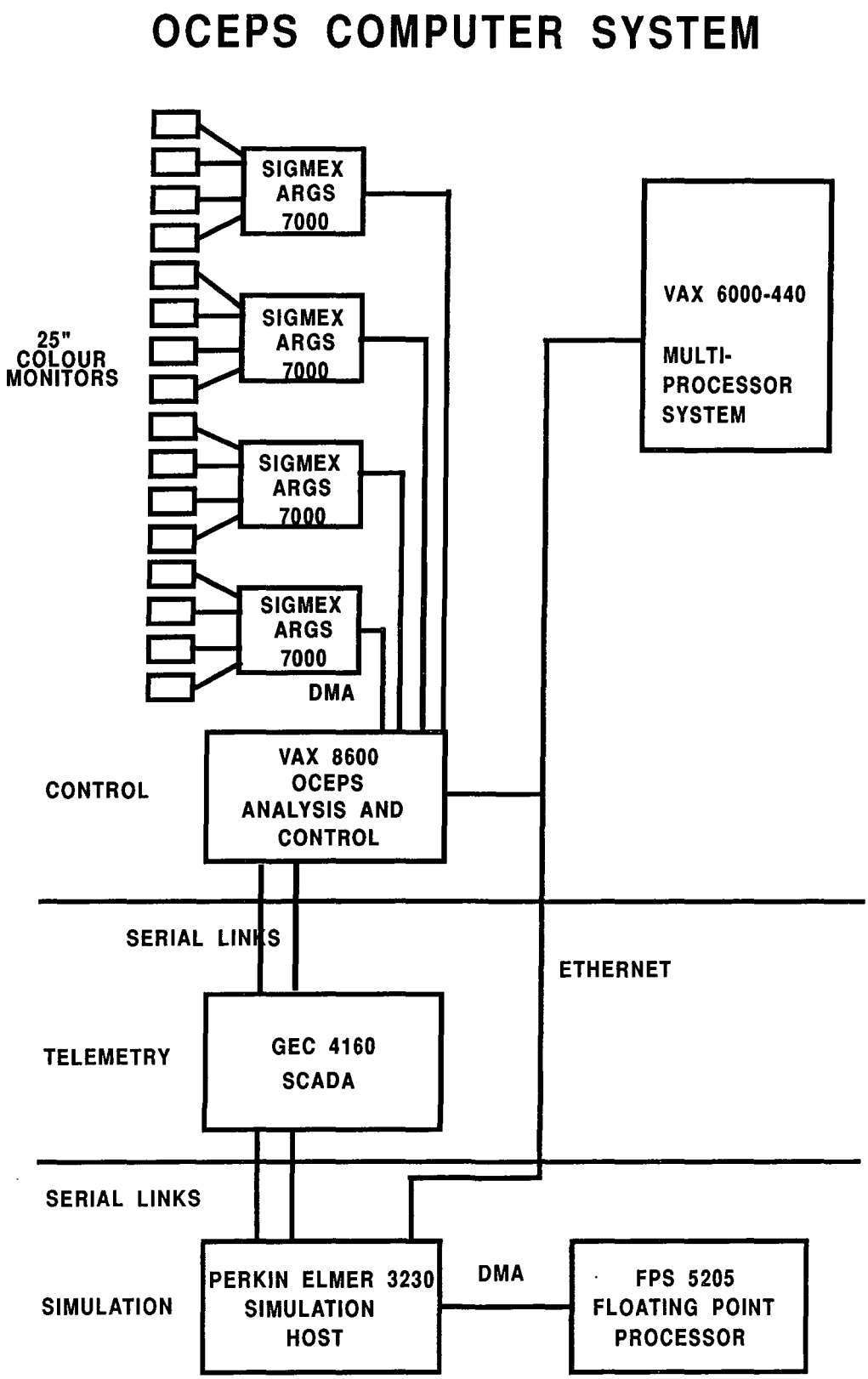


Figure 2.4 O.C.E.P.S. hardware configuration for simulation.

and angle. Several system components are frequency dependent, so the frequency is required at each node. This can be calculated by modulo subtracting the phase angle of the most recent two values, and dividing by the time–step. This formula is correct for small frequency deviations from the set point, but as soon as the angle difference becomes larger than a certain value, it appears that the sign of the frequency deviation is reversed, resulting in a widely oscillating frequency, and hence oscillating network parameters. The solution is to obtain a base frequency from one of the active generators in the island, which contains a frequency term as one of its states, and use an angle difference based on this to calculate the frequency at each node. This frequency base was not being correctly set in the study. Only a severe transient would normally cause this, since an excursion of more than 0.5Hz is required for a time–step of one second. These sudden frequency changes were producing step changes in load powers and line parameters, and were triggering protection circuits.

Development is also proceeding with a distributed simulator, using a uni–processor simulation on the existing Perkin–Elmer and Floating Point Systems hardware. The method uses the matrix inversion lemma [126] for matrix decomposition and modification.

A recent project involved the production of a multi–microprocessor simulator based on Motorola 68020 microprocessors [43] with both 68881 dedicated floating point processors and special Weitek floating point units. This simulator used a reduced model of the elements in the transmission system to decrease the amount of calculation required in each time–step. The method relied upon keeping the network admittance matrix constant unless a topology event occurred. This matrix was inverted externally on the Perkin–Elmer, and held in the simulator. Any small changes in this matrix due to topology changes or load changes were introduced by matrix modification techniques into the inverse. This has two problems, firstly that errors tend to accumulate into the matrix as each change is made, and secondly that accurate load models constantly change their power characteristics, requiring many updates in otherwise steady–state operation. If a major topology event occurs, the matrix must either be re–formed and inverted, or a complex adjustment performed on the existing inverse. The simulator was proven for the I.E.E.E. 30 node system. It was, however slower than the Perkin–Elmer simulator, which provided better and more complete models, and so was not adopted as the O.C.E.P.S. simulator.

A parallel project to this work was also initiated to provide a symbolic front–end and simulator builder to ease the retargetting of the simulator to new hardware, since the rate of hardware innovation was exceeding what was possible for software. This involved the symbolic differentiation and integration of the system models, which were then combined into a machine–generated C program which became the simulator after compilation and linkage. The work in this thesis compliments this work, by investigating the most time–consuming part of the current simulators, to speed both the uni–processor and multi–processor implementations. Special, sparse–aware, code is needed to process the matrices efficiently, and the existing routines did not provide the necessary speed increases to enable large networks to be processed in real–time.

The system models are assumed to be handled by the symbolic pre–processor, and can be of any type, as long as they connect to at most three busbars in the system, with power flow and hence network and matrix connectivity between a maximum of two. Generators and loads are assumed to connect to only one busbar, as it is unlikely that a generator would connect to two separate busbars. The three connection case specifically is to handle transformers with automatic tap changers. All power flow branches must connect to busbars, which can have any number of external connections, and switchable links to other busbars. This simple network model can handle most network configurations, possibly with the addition of a few dummy nodes.

2.8 The options for this simulator.

The combined representation of the generators and network elements was chosen for this simulator, in conjunction with the voltage, current and admittance representation of the network equations. This combination has several advantages over other representations:

1) The formation and destruction of electrical islands is handled semi–automatically by the equations, and no node changes type because of the islanding.

2) The greater numerical stability of the method was felt to be beneficial for a simulator which might be subjected to severe transients during the development of control algorithms. The longer the simulator can cope with the changes in the state of the system, the more information

is available to the control engineer to help to determine the cause and hence reduce future transients.

3) The method is more ameanable to the automatic generation of code by a simulation language from arbitrary system models, since specific knowledge and models of slack buses are not required.

4) The tight coupling between the differential equations and the algebraic equations of the network eliminates any time–lag between their solutions, so every solution is self–consistent.

5) The retention of more information from the models throughout the whole solution, instead of discarding information as with the Decoupled and Fast Decoupled Newton methods was felt to be beneficial.

This approach might be slower than other approaches, specifically those using Fast Newton Loadflows, but these advantages were felt to outweigh the speed penalty.

Chapter 3.

Solution of Sparse Matrix Equations.

3.1 Introduction.

The network structure of power systems dictates that any equations that result from the analysis of the system are highly sparse. A sparse system has relatively few terms in each of the equations which represent the system, compared to the total number of equations. When viewed as a matrix, the matrix has few non–zero terms in each row or column. For large problems, it is wasteful to process and store elements which are known to remain zero, and special techniques have been developed to take advantage of sparsity.

Unfortunately for power systems programming, the sparsity found in power systems is not typical of that found in other naturally sparse systems [140], and the nature of the networks involved reduces any easily usable structure. Most sparse matrices result from either finite element or finite difference analysis in the fields of thermo–fluids or structural mechanics, and usually involve largely regular matrix structures. Because these matrices are mostly machine generated, there is considerable flexibility in the choice of elements and their ordering. The regular structure of the matrix also makes this problem attractive to mathematicians and computer scientists, as the problem is well defined, and any results are widely applicable.

The lack of regular structure makes the power system problem unattractive to workers outside the power systems industry, and differences in the connectivity of each power system reduce the probability that an improvement for one system will improve results for another. Research into solving the equations which result from analysing power systems has largely been conducted by power system research departments, and approached from a power system viewpoint. Detig [28] paints a depressing picture, suggesting that

power system analysis methods are poor because they do not reduce the problem to one which fits well on existing computer architectures, but concludes by saying that the solution lies in faster and more easily usable special purpose computers.

Studies have shown that traditional sparse methods perform poorly when applied to network matrices [93] [117]. Network methods perform well, but not optimally, for non–network problems, but consume much larger amounts of computer time reordering the equations to maintain sparsity. Power system sparsity therefore needs special methods, which are different from those employed for other sparse matrices. The links between nodes which form the non–zero structure of the matrix are undirected, so if a node sees a connection to another node, that node also sees a connection to the first node. This results in location symmetry about the leading diagonal for the network part of the matrix.

The matrices encountered in other fields are usually less well conditioned than those found in power system work, and more effort is required to keep the solution numerically stable, while with power system matrices, the solution will usually be stable, but a poor ordering will result in much unnecessary work. Methods from other fields may still be useful at a higher level, when dealing with groups or clusters of nodes, as they provide a good indication of the optimal overall form of the matrix. It might, for example, be advantageous to arrange clusters in a similar manner to finite element elements, which would simplify the generation of an elimination sequence.

It rapidly became clear that for a simulator to run continuously in real–time, the matrix solution stage was critical. The solution involves a large number of calculations, must be repeated until convergence is achieved, and is difficult to perform in parallel. The initial part of the research presented here was spent on this topic.

3.2 The matrix problem.

Chapter 2 discussed the basics of deriving sets of equations which describe the electrical power system, and the matrix methods which can be applied to obtain a solution. An elimination method was selected for the simulator to retain accuracy and to avoid convergence difficulties which affect iterative matrix solutions. The methods were described primarily for the case of full matrices and vectors, but the power system problem is by the nature of the network, highly sparse.

The possibility of numerical instability and loss of accuracy was mentioned chapter 2 in connection with pivoting, and the benefits of a positive definite matrix were discussed. It was stated that although power system matrices are not positive definite, their elements are of such a size that they almost always do not require any pivoting to retain accuracy and numerical stability. The elements are defined by the physical plant characteristics, and by the system of units which are used to describe them. Most power system calculations are in per–unit notation, and once values have been translated into these units, almost all values fall within a certain range. In particular, the diagonal terms naturally tend to be larger than the off–diagonal terms, which is the object of pivoting elements to retain accuracy. If the diagonal terms are large, then the size of the other elements is limited as the elimination proceeds, whereas if thay are small, other terms can become much larger than the diagonal terms, and accuracy is lost.

Unfortunately, I.E.E.E. 32 bit floating point numbers do not have the required precision to ensure accuracy [55], which forces the use of 64 bit numbers if standard hardware is to be used. The 36 bit numbers used by older Floating Point Systems array processors (AP120s [60 110]) and their descendents, do have the required precision, but this word length is not standard and is unsupported by most computers. An O.C.E.P.S. loadflow program written for the Acorn Archimedes RISC processor was stable using five–byte (40 bit) floating point numbers, and became numerically unstable when a different compiler using four–byte (32 bit) numbers was used.

If sufficient precision is available for stability, the pivoting stage can be omitted, which improves speed, since no record must be kept of pivots, and no floating–point comparisons are required to select pivots. Omitting this stage also makes the calculation order independent of actual values, and therefore constant between successive solutions while the structure of the matrix remains fixed. This allows the operations which are required to solve the equations to be partly or fully determined only once for each change in the matrix structure, which will reduce the work required to produce subsequent solutions.

If no unique solution is available for a matrix equation, the matrix is said to be singular. To be singular, there must be a linear dependence between some rows and columns in the matrix. The sizes of the elements in power system studies make this unlikely, but some problems, particularly loadflows, occasionally delete rows of the matrix as

the type of nodes change, and such programs must remove the zero rows and columns which would cause singularity.

The matrix methods presented in this chapter assume that the leading diagonal will be full of non–zero numbers. This will almost always be the case, with zeros only resulting from cancellation of terms due to lack of sufficient precision. The current methods in use by O.C.E.P.S. monitor the size of the diagonal elements as they are used, and increase their size if they become very small [75]. Such an adjustment will invalidate the solution, but because the matrix solution would form part of an iterated algorithm for the solution of non–linear equations, another iteration should follow, since it is unlikely that the convergence criterion would be satisfied immediately after an such an adjustment. This will usually allow the current solution to proceed, and produce new estimates for the independent variables, so the next iteration of the solution should start with values that do not almost cancel.

3.3 Particular features of sparse elimination.

When a row or column is eliminated from a full matrix, other elements in the matrix are modified to reflect this change [144]. For full matrices, all the elements below and to the right of the eliminated elements are modified, as elimination usually proceeds from the top left to the bottom right. The element to be modified has the product of the two elements in the eliminated row/column which are in the same row or column as the element to be modified subtracted from it, as shown in equation 3.1. If one of these two elements happens to be zero, then no modification is required, since the subtrahend is zero. For full matrices, this property is ignored, since it would occur infrequently, but this is not the case for sparse matrices. In equation 3.1, j is the index of the column which is currently being eliminated, and i and k are indeces of non–zero elements within this column.

$$a_{i,k} = a_{i,k} - a_{i,j} \times a_{j,k} \tag{3.1}$$

In this discussion, a zero element is an element whose value is fixed at zero, while a non–zero is an element whose value is not fixed at zero, but which nevertheless may contain zero. That is, a zero element is fixed to contain zero, while a non–zero element may contain any value, even zero.

If each element must be checked to determine whether it is zero, then little time can be saved, but if the non–zero elements are stored in such a way that only non–zero elements are processed, then this saving becomes very attractive. A difficulty does arise, because when a column is eliminated from a sparse matrix, it is possible that a zero element may become a non–zero. This event is called fill-in, and will occur whenever the eliminated column references two other columns which do not directly reference each other.

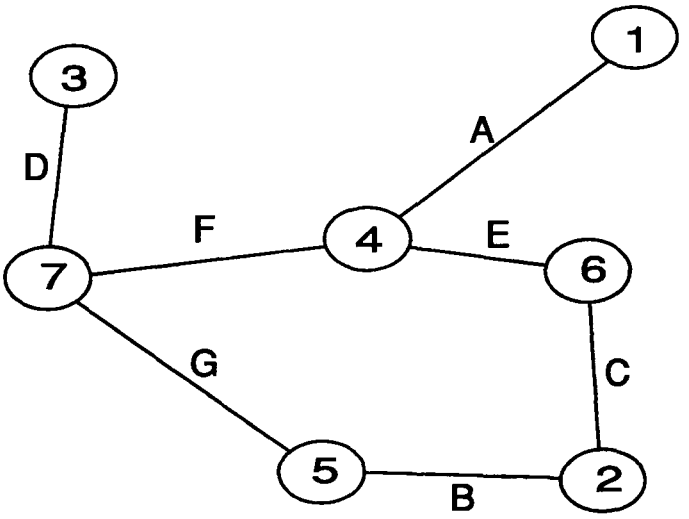
3.3.1 Example symbolic elimination for small network.

Figure 3.1 shows a simple, seven node network which can serve as an example, and the corresponding matrix structure. Figure 3.2 shows the steps in eliminating matrix A in node order. As node 1 is eliminated, the term representing node 4 in A will be updated, since the elements $a_{1,4}$ and $a_{4,1}$, which are labelled 'A' in the matrix, represent the connection between nodes 1 and 4. Node 2, however, is connected to both node 6 and node 5, which have no direct connection between them ($a_{5,6}$ and $a_{6,5}$ are initially zero). To preserve topology when node 2 is eliminated, a connection must be added between them in the matrix (labelled 'H'), and its value will be determined either by the product $a_{2,5} \times a_{6,2}$ or by $a_{2,6} \times a_{5,2}$. This new fill–in element now takes part in further operations, just as if it were an original connection in the matrix. Node 3 is eliminated next, and produces no fill–in, but the elimination of node 4 produces a fill–in element between nodes 6 and 8, which is called 'I'. The remaining network is now fully connected, so no more fill can occur.

3.3.2 Introduction to fill–in in sparse matrices.

Fill–in elements increase the number of calculations which are required to solve the matrix, because they themselves generate arithmetic operations as they are eliminated. They can also increase the number of fill elements which are produced as the elimination proceeds further, because if they reference a zero position, a new fill–in element will be generated. It is therefore important to minimise the number of these fills.

Fill–ins can only occur in the columns and rows which are directly connected to the eliminated column by row entries in that column. The set of elements which must exist after a column has been eliminated is given by every possible pairing of any two of the row entries in that column. If m is the number of non–zeros in a column, called the



Example Small Network

	1	2	3	4	5	6	7
1				A			
2					B	C	
3							D
4	A					E	F
5		B				H	G
6		C		E	H		I
7			D	F	G	I	

H and I are fill--ins

Matrix representation of the network

Figure 3.1 Example 7 node network.

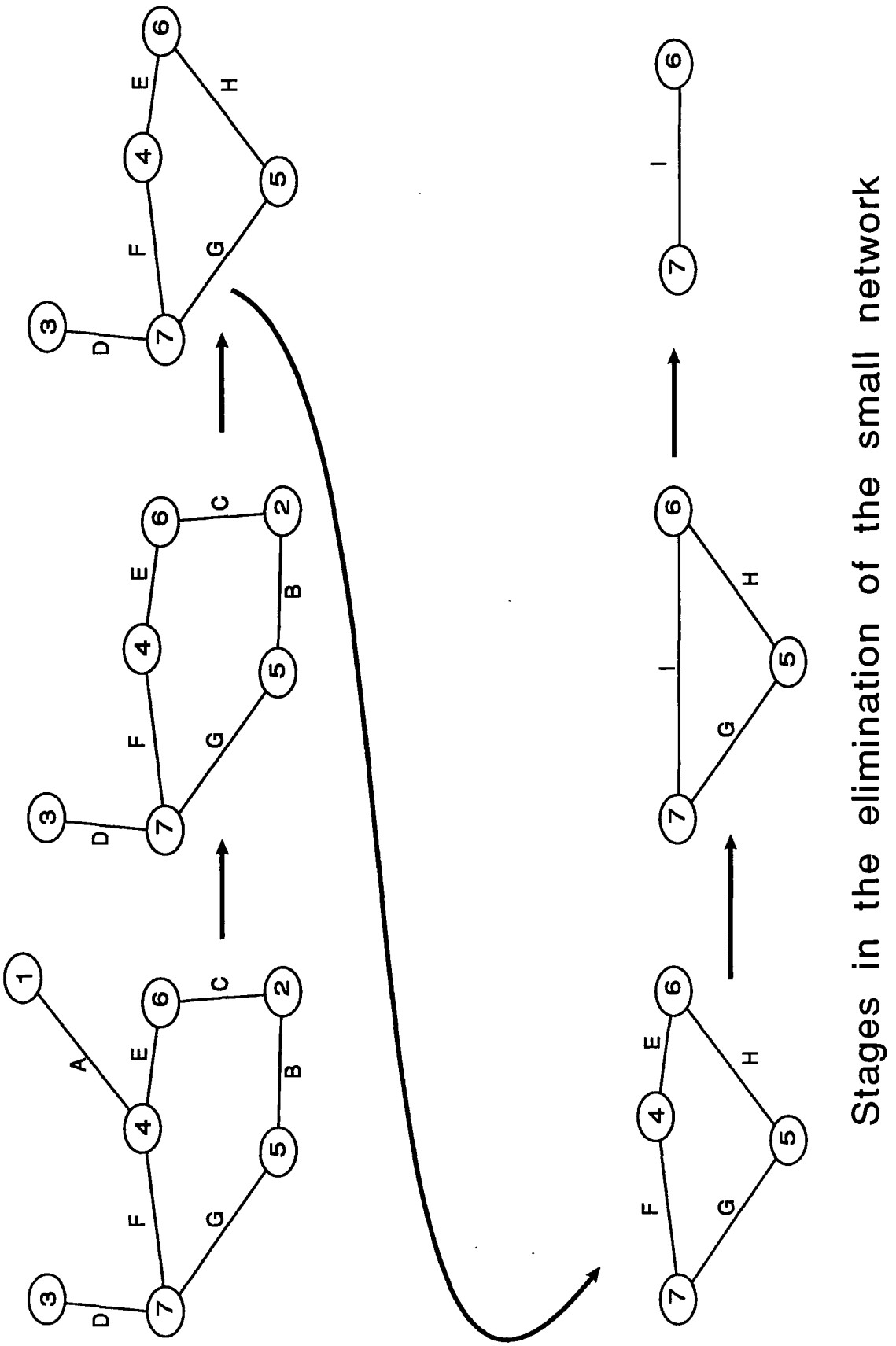


Figure 3.2 Steps in the elimination of the 7 node network.

degree of the column, this yields a full m–by–m sub–matrix distributed in the main matrix. In practice, many of these locations would already be occupied, so fewer new fills would usually be generated. A maximum can therefore be placed on the number of fills which the elimination of a column could produce, bearing in mind that the leading diagonal is assumed full.

$$\text{Possible fills} = (m-1) \times (m-1) - (m-1) \tag{3.2}$$

$$= (m-1) \times (m-2) \tag{3.3}$$

The possible fills can therefore be limited by selecting the column with the smallest number of row entries first, as this will limit the initial number of fills, so the later columns should have fewer entries when they, in turn, are eliminated. Several variations on this theme are investigated in Chapter 4.

3.4 The Crout Method.

An elimination method was selected for the simulator because of the mathematical accuracy of these methods. Several algorithms exist for full matrices, each of which has specific advantages over the others. The simplest to code is Gauss–Jordan elimination, which is simply the systemmatic solution of simultaneous equations. Cholesky elimination efficiently solves symmetric positive definite matrix equations, and Crout elimination is efficient for asymmetric matrices, and works in–place. This property was important when computer memories were small, and large matrices had to be paged to and from slow backing store. It also generates relatively few intermediate results, allowing the use of extended precision in the FPU to retain accuracy during the summations which produce these results. This is also important for hand calculations, since few new results need to be written down, which reduces the opportunity for errors. Following Gustavson [53], an attempt was made to apply the Crout method to power system matrices.

The Crout method starts at the upper left corner of the matrix, and processes the whole of the first row and column, i.e., a right angle of unit width. Each element has the sum of products of the row to the left of it and the column above it subtracted from it. This modified element is then used when the next row and column are processed. Two triangular factor matrices, L and U, are produced, whose elements overwrite the

original locations in A. Variations in the method exist concerning the handling the elements on the leading diagonal, and here they were assigned to the lower triangular factors, while the upper triangular leading diagonal elements were made unity.

Thus:

$$l_{ik} = a_{ik} - \sum_{p=1}^{k-1} l_{ip} u_{pk} \qquad i \geq k \tag{3.4}$$

$$u_{kk} = 1 \tag{3.5}$$

$$u_{kj} = \frac{(a_{kj} - \sum_{p=1}^{k-1} l_{kp} u_{pj})}{l_{kk}} \qquad j > k \tag{3.6}$$

For the first row and column, these degenerate to:

$$l_{ik} = a_{ik} \qquad i \geq k \tag{3.7}$$

$$u_{kj} = \frac{a_{kj}}{l_{kk}} \qquad j > k \tag{3.8}$$

Note that since the l_{kk} term is used to determine all the u_{kj} terms, the column terms should be evaluated first. It would also be usual practice to evaluate the reciprocal of this term, since multiplication is generally much faster than division, so time would be saved overall. This term is only used as the denominator in future processing, so the reciprocal can be stored in place of the actual term. If the vector for which a solution is desired is available at factorisation time, it is often appended to the right of the matrix during the elimination phase, because its forward pass is identical to the processing of the upper triangular part of the main matrix.

It will be seen from these equations that the Crout accesses are localised, with certain areas of the matrix completely processed, and other areas completely unprocessed. In figure 3.3, the thick lines represent the current row and column. The elements which are both above and to the left (L1 and U1) of the diagonal element a_{kk} have been fully processed, and take no further part in the elimination. The elements which are either above and to the right (U3), or below and to the left (L3) have been processed, but are still required as they will be multiplied by elements in the current row or column (L2 or U2). The elements which are below and to the right (A22) have not yet been processed or accessed. The elements (A21 and A12) have just been modified. The access method can be visualised by summing the products of non–zero pairs of column U2 and rows

in L3 into the corresponding elements of A21, and similar pairs of L2 and U3 into A12, but this time multiplying the result by the pivotal term l_{kk}.

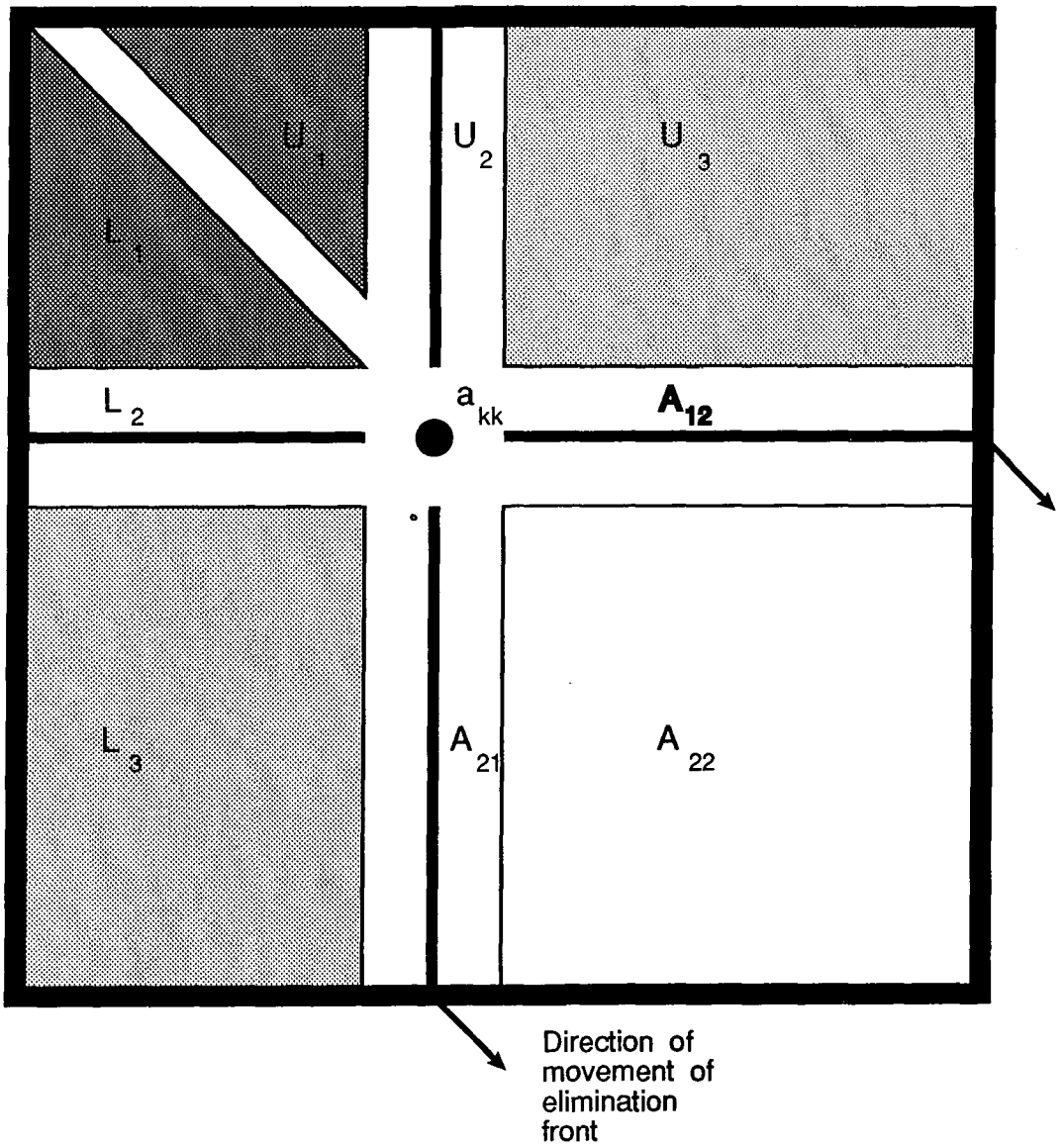
3.4.1 Problems with the Crout Method.

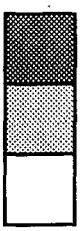
Severe problems were encountered with this method. The number of calculations required were found to depend strongly on the order in which the rows were numbered, and in most cases was excessive. Gustavson ignored the possibility of re–ordering the equations to reduce the number of calculations. If the program was written as a direct, fast replacement to solve a sparse matrix which already has a good form, then no re–ordering is required. This would usually be the case with finite–element or finite–difference solutions, where the matrix is generated by another computer program, but for power systems, this is not the case, and re–ordering is required.

The number of fills depends on the sparsity of U2 and L2, and of L3 and U3. Fill can only occur when a non–zero element in U2 matches a non–zero in any of L3 (or L2 and U3), and will only occur if the corresponding entry in A21 and A12 was previously zero. A reasonable amount of fill–in could be achieved by evaluating every combination at each step and choosing from amongst the best, but this is impractical for a real–time simulator. When a column is selected as the next to be eliminated, nothing can be done about the sparsity of L3 and U3, as these are already fixed, so only U2 and L2 are variable. If the column is chosen from those with a minimum number of non–zeros in U2, this will not guarantee good fill–in performance, but should on average be better than a random selection. If this is combined with counting elements in A21 as a tie–breaker, there will be less randomness at the start of the elimination. Both these counts are easy to maintain during runs, as fills can only take place in the column which is being eliminated, and only the rows which can change have entries in this column.

An alternative strategy is to minimise the bandwidth of the matrix, either globally or within smaller blocks. Since fill cannot occur without matching elements above and to the right of the fill location, forcing blocks away from the leading diagonal to become zero should limit the fills which occur.

Any storage scheme must be able to accept these new elements which are generated during the elimination, and should also be efficient at inserting them and finding specific





Fully processed, no longer used.

Fully processed, still used.

Totally unprocessed.

Crout Elimination.

Figure 3.3 Crout elimination.

elements. These two aims conflict, as fast searching requires well ordered lists and good pointers, while fast insertion is hampered by having to update pointers. The Crout method is particularly problematical because fast access is required to both row and column data structures, which makes reordering the rows and columns difficult once the elimination has commenced. Work with the Crout method was discontinued in favour of the Zollenkopf [162] method, which permits much easier reordering of the columns due to different access patterns.

3.5 Code Generation.

The second part of the Gustavson paper proved more interesting, as it documented an early attempt to produce straight–line code at run time which was dedicated to the solution of one matrix structure. As power system matrices tend to remain constant in structure for considerable numbers of solutions, this is attractive if the initial set–up is reasonably fast. In general purpose code, the branches which occur in loops perform no useful calculations, and viewed from this standpoint, are a waste of time. They not only waste the time they take to perform the branch and test, but may also impede the optimisation stage in compilation, thus reducing speed still further. Some compilers will not rotate loops and move data stores and loads across branches, to effectively hide integer instructions for loop control and address generation behind the longer execution times of floating–point operations.

In–line code does not suffer from these adverse effects, and will allow the compiler more freedom to optimise the code, especially if delayed loads and stores are available. This does, however present different problems. The method proposed by Gustavson produced simplified IBM FORTRAN, which was compiled as normal, with the optimising compiler. This was found to be very slow, because the compiler was attempting to optimise the whole of the straight–line code. A faster compiler was then used, which attempted less optimisation, and this reduced the compile–time considerably. In–line code might also simplify the generation of addresses, and Gustavson produced his program with simple, single–dimension data arrays for this reason. An alternative program was also available, which interpreted the FORTRAN code, for use for one–off solutions, when the overhead of compilation would not be repaid. A low break–even point for the number of solutions of the same form was found, which bodes well for applying the method to a real–time problem.

Some programs have generated assembler code, while others have attempted to generate machine code directly, but the unsuitability of many instruction sets has made both these approaches difficult. As soon as a departure is made from a compiled language, the problems of low–level address generation and instruction scheduling must be tackled. Both these techniques consume large amounts of memory, as each instruction requires a direct address for its operands.

An alternative approach is to form a library of frequently used routines, and generate code which only consists of calls to these routines, and instructions to update pointers to a seperate address list. The updating of the pointers could also be incorporated into the called routines, so the calls could be replaced by an address list themselves, through which a controlling program could step. This reduces memory requirements and provides extra flexibility, but does require an efficient subroutine entry and exit mechanism. The poor performance of many older processors on subroutine entry and exit, coupled with the generalised compiler call interface, made this less attractive, but the more highly tuned RISC call mechanisms [108] may encourage this method [118]. The addresses in both lists could even be placed into linked lists [85] so that whole routine calls could be changed if this proved necessary. Hewlett–Packard used this approach successfully for their Scientific Basic [131] language in the late 1970's, to provide speed without full compilation, and also integrated the reverse capability to reproduce the source code from the machine representation along similar lines.

Invoking complex, pre–defined macro instructions removes the need for either compilation or the production of long machine language programs. In effect, a new instruction set has been defined which is accessed by pseudo–instructions which mainly consist of addresses. The limited varaiety of operations needed by the bifactorisation method makes this approach highly attractive. Modern co–processors have provided a new approach, particularly dedicated FPUs such as the Am29C327. With the increased complexity of these processors, and new programmable devices such as PLAs, it is now possible to define and implement a special processor, with a dedicated instruction set, which can implement arbitrarily complex instructions. As an example, it would be possible to have an instruction which, given a list of addresses, would perform the sum-of-products and subtract this from the correct element. This would be useful for the Crout algorithm. A 'super' instruction could even perform the complete elimination

of a column. This could be achieved with one instruction and an address list, which is very simple and efficient to produce automatically.

Each super–instruction could also be highly optimised, as the basic instruction would not be written and complied during simulations. Special purpose hardware controlled by PLDs could route data as required, including prefetching addresses. Full advantage could also be taken of processor pipelines and delayed loads and stores, as the execution order and location of possible clashes are well known in advance. This would effectively gain all the advantages of Very Long Instruction Word (VLIW) computers, microcoding and high–level compilation, without the drawbacks of providing a full, general purpose environment, which is already provided by the host processor. This approach is similar to the previous method, but with the subroutines replaced by microcode routines. The inter–instruction inefficiencies would occur rarely, as instructions grow more complex and longer. This approach can also be remarkably memory efficient, as only the minimum of data is required, and far fewer instructions are stored.

While the development of special purpose instructions to ease programming or reduce memory requirements is not new, the development of such an instruction set to enable the real–time generation of programs has not been developed elsewhere, probably due to the previous inefficiencies of call mechanisms, and the lack of powerful, general purpose, floating–point processors. The reasons behind the development of the dedicated, very complex instruction set even satisfies the RISC criterion [78], where an instruction should only be incorporated if it would reduce memory requirements, or increase speed by more than 1% in the intended program mixture.

3.5.1 Pseudo–code execution on standard architectures.

This program structure is also amenable to software emulation on a standard architecture, because of the similarity between the 'super' instructions and a subroutine library invoked by either an arithmetic GOTO in FORTRAN, or by C's pointer–to–function. Since the subroutines and main loop are precompiled, full optimisation can be performed, including human analysis of possible conflicts, and action to ensure that all optimisations are possible without affecting results. The routines could even be hand coded, because they would only require coding once. Many of the accesses are guaranteed not to conflict with each other, and others can be scheduled so that any dependencies

are hidden. It is extremely difficult to instruct a compiler where dependencies exist, and hand coding would probably be better for these routines. Once written and proven to be correct, no further work is required unless the main program changes in such a way that the use of the routines is altered, so effort spent in producing efficient routines would therefore be repaid. It would also be possible to code all the routines in one large routine, which would improve efficiency, but conflicts with current computer practice.

This could be used for debugging the algorithm, or alternatively might result in an increase in speed despite the additional step of code generation before the first solution following a topology change. Whether an increase in speed results depends on the relative efficiency of looping and subroutine entry and exit instructions compared to the overhead of repeatedly interpreting the sparse lists directly.

This approach was taken in evaluating and testing the method in C on a VAX-8600. For ease of coding, a single routine was written, which contained variable length loops, instead of multiple un–rolled routines. This would adversely affect speed if sufficient memory were available to hold the whole un–rolled program without paging, but the smaller code size of the approach which was taken might counterbalance this by reducing page faults. The 'instruction' contains information about the size of the block, and what operation is required, and this provides sufficient information to correctly interpret the contents of the stored address array.

3.6 Bifactorisation or the Zollenkopf method.

In his 1969 paper, Zollenkopf [162] introduced a new form for the elimination process. Instead of systematically eliminating rows in the matrix, he built a series of row and column vectors from the original matrix, and showed that by multiplying these together, the inverse could be formed. Given a vector for which the solution is desired, the solution could then be obtained merely by successive matrix multiplications. These multiplications are simple because one of the matrices is just a vector, and the other is just a vector of the same or smaller length, embedded below or to the right of the leading diagonal, in an otherwise unit matrix. This approach is called bifactorisation. The method can be used on either symmetric or location symmetric matrices, with

minor modifications, but an incidence–symmetric matrix structure is required. The form detailed here is for incidence symmetric matrices with possibly asymetric values.

$$I = L^{(n)}L^{(n-1)}\ldots L^{(2)}L^{(1)}\,A\,R^{(1)}R^{(2)}\ldots R^{(n-1)}R^{(n)} \tag{3.9}$$

Where L are left hand factor matrices,

R are right hand factor matrices

and I is the unity matrix.

$$A^{-1} = R^{(1)}R^{(2)}\ldots R^{(n-1)}R^{(n)}\,L^{(n)}L^{(n-1)}\ldots L^{(2)}L^{(1)} \tag{3.10}$$

Since the solution of $\mathbf{Ax} = \mathbf{b}$ is obtained by $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$

the bifactorisation solution is obtained from:

$$\mathbf{x} = R^{(1)}R^{(2)}\ldots R^{(n-1)}R^{(n)}\,L^{(n)}L^{(n-1)}\ldots L^{(2)}L^{(1)}\,\mathbf{b} \tag{3.11}$$

These are simply a series of simple matrix multiplications, one the factor matrices have been determined. The factor matrices may be determined by:

$$A^{(0)} = A \tag{3.12}$$

$$A^{(1)} = L^{(1)}A^{(0)}R^{(1)} \tag{3.13}$$

$$A^{(2)} = L^{(2)}A^{(1)}R^{(2)}$$

$$\vdots$$

$$A^{(n)} = L^{(n)}A^{(n-1)}R^{(n)} = I \tag{3.14}$$

The matrix terms are calculated as follows:

$$a_{j,j}^{(j)} = 1 \tag{3.15}$$

$$a_{i,j}^{(j)} = 0 \tag{3.16}$$

$$a_{j,k}^{(j)} = 0 \tag{3.17}$$

$$a_{i,k}^{(j)} = a_{i,k}^{(j-1)} - \frac{a_{i,j}^{(j-1)}a_{j,k}^{(j-1)}}{a_{j,j}^{(j-1)}} \tag{3.18}$$

$$= a_{i,k}^{(j-1)} - l_{i,j}^{(j)}r_{j,k}^{(j)} \tag{3.19}$$

The factor matrices have the form of equations 3.20 and 3.21 respectively. The constituent terms are formed as described in equations 3.22–25.

The $l_{j,j}$ terms are not actually stored as lower triangular elements, because the same data structures are used for the l and the r elements. If they were stored there, then

$$L^{(j)} = \begin{pmatrix} 1 & & & 0 & & & \\ & \ddots & & \vdots & & & \\ & & 1 & 0 & & & \\ & & & l^{(j)}_{j,j} & & & \\ & & & l^{(j)}_{j+1,j} & 1 & & \\ & & & l^{(j)}_{j+2,j} & & 1 & \\ & & & \vdots & & & \ddots \\ & & & l^{(j)}_{n,j} & & & 1 \end{pmatrix} \tag{3.20}$$

and

$$R^{(j)} = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ 0 & \dots & 0 & 1 & r^{(j)}_{j,j+1} & r^{(j)}_{j,j+2} & \dots & r^{(j)}_{j,n} \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} \tag{3.21}$$

Where

$$l^{(j)}_{j,j} = 1/a^{(j-1)}_{j,j} \tag{3.22}$$

$$l^{(j)}_{i,j} = -a^{(j-1)}_{i,j} \qquad i > j \tag{3.23}$$

$$r^{(j)}_{j,j} = 1 \tag{3.24}$$

$$\begin{aligned} r^{(j)}_{j,k} &= -a^{(j-1)}_{j,k} / a^{(j-1)}_{j,j} \\ &= -a^{(j-1)}_{j,k} l^{(j)}_{j,j} \qquad k > j \end{aligned} \tag{3.25}$$

storage would be wasted because the the $r_{j,j}$ elements are not required. The elements are instead stored as strictly lower and upper triangluar arrays with the same structure, and a single dimensional array containing only the diagonal elements $l_{j,j}$.

As each column is eliminated from matrix A, the $l_{i,j}$ elements are written to L. These will only exist where an element $a_{i,j}$ is non–zero, so the structure of this column in L will be the same as the eliminated column in A. The same argument holds for the eliminated row and the current row in R. The elements in this row and column in A are zeroed after the elimination, so $l_{i,j}$ and $r_{j,k}$ elements can be stored in the locations which were previously occupied by $a_{i,j}$ and $a_{j,k}$ respectively. This saves storage locations, and removes the need for a new set of indexing information.

Since the $l_{i,j}$ elements just have their signs negated, the equations are rewritten so that this trivial but time–consuming operation is not required, moving all negations to where an addition or subtraction would otherwise be performed.

3.6.1 Zollenkopf Program.

The paper also contains program flowcharts for use with the FORTRAN programming language to implement the solution method described in the previous section. The flowcharts incorporate a run–time near–optimal sparsity ordering of the elimination, which is based on selecting the next column from amongst those with the minimum number of non–zero elements. For efficiency, particularly within a paging system memory, the elimination is first simulated, to establish the order for elimination, and to generate the necessary locations for fill–in terms. This stage involves much searching of arrays, and floating point calculations would both complicate and slow the solution. In addition, each off–diagonal element is initially duplicated, with one or other of the pairs being removed as the ordering proceeds, and if floating point calculations were performed during the ordering, then the results would have to be written to two memory locations, one of which would later be discarded, instead of only one which is really required, and this duplication would severely slow the elimination process.

The calculations are performed after the simulation of the elimination, when the matrix has the structure of its eliminated form. This structure is fixed until the structure of the matrix changes, so the similation and ordering does not have to be repeated for each numerical elimination. Each new set of results can be generated by filling the matrix and dependent vector with values, and performing the elimination and solution routines. After the simulation, no attempt is made to alter the representation of the matrix in memory. An investigation is made later in this chapter to determine whether such an alteration would in fact be beneficial.

3.7 Sparse storage.

The size of the power system matrices requires sparse storage techniques. If such techniques were not used, the 734 node network would require $734 \times 734 \times 4 \times 8$ (size, four elements per sub–matrix, eight bytes per double–precision number) which is about 18 Megabytes. This does not include any space for generators, which would make the total $1134 \times 1134 \times 4 \times 8$ or 41 MB, assuming 100 generators. This amount of memory would contain approximately 0.5 MB of actual data, with the remainder being zeros. Accesses to the array would also be inefficient, unless a list of non–zeros was maintained.

If such a list is kept, then the rudiments of a sparse storage scheme exist, and it should be fully implemented. If the matrix structure is fixed, i.e., no elements are created or deleted during processing, then two arrays are required in addition to the array holding the data. One of the arrays gives the row indices of the elements, which are stored in order of increasing rwo number, and the other gives the location of the first non–zero element each column. The former array will contain the same number of elements as the data array, and will be parallel to it, while the latter will probably contain fewer locations, as only one entry is required per column. It is usual to remove the diagonal elements from these arrays, because these are assumed to be present, and can be accessed and stored more efficiently if placed in order into a linear array.

An element can be accessed at random by indexing the column start array with the column number of the element, and using this number to start searching through the linear list of row numbers, until the address of the start of the next column is reached, which would mean that the element had not been found. An alternative indication of reaching the end of a column is to insert dummy elements into the row index and data arrays with row indices of zero as the last elements in each column. This will increase storage requirements, but is easier to implement.

To insert a new element or delete an existing one with this storage scheme, every element after the modification point would have to be moved, and the pointers to the start of each column adjusted accordingly. It would also be possible to copy the affected column to locations after the last column in the arrays, and modify this copy, since this would probably affect less elements. The scheme can accommodate this by altering the pointer to the start of this column. A memory management scheme would have to be

implemented to keep track of vacant memory locations with the aim of reusing them, otherwise storage would quickly become fragmented. The design of such a scheme is non–trivial and closely related to the anticipated pattern of modifications to the structure of the matrix.

These problems can be avoided by placing the elements into a linked list, which requires the addition of another array parallel to the data array. This array is used to locate the element which should be accessed next if a search is made down the column, and replaces the implicit ordering of the simple scheme described above. An element may be inserted by modifying the pointer of the element which it is inserted after to point to it, and by using the original pointer it this element to load its own pointer to the next element. Deletion simply involves loading its own pointer into the pointer of the location which pointed to it. Locations are saved in the data and row index matrices by moving the end of column indication to the linking array, since the last element in a column has no need of a pointer to the next element in the column. The loop control for the scanning loop now becomes `while (( ln = next [ ln ] ) != 0 )` , which is terse and explicit.

There are some complications to this scheme involving storage management and special cases. Storage must be allocated from and returned to a pool of available memory locations, but provided these can be linked together so none get lost, allocation schemes are simple to implement since fragmentation is not a problem. Special cases exist when elements at either end of a list are modified, or where there are only one or even no elements in a list. These special cases present no problems, provided that they are all recognised, so that column start pointers can be updated if necessary.

This scheme is superior to the previous scheme if many deletions and insertions are made to the matrix during processing, but is inferior if the structure remains constant. The Zollenkopf simulation of the elimination and the ordering make many structural alterations, and require the linked list storage scheme. Subsequent routines keep the structure fixed, but make frequent searches and scans through the columns, and would benefit from the simpler storage scheme, since a level of indirection is removed, as is a whole array. The advisability of making the conversion after the structure becomes fixed should be investigated.

This is the main storage scheme used to describe the matrix structure in the Zollenkopf routine, with the addition of a column count array parallel to the column start array. This is required to keep track of how many non–zero elements are in each column, to form the basis for the decision about which column should be eliminated next. It is not part of the storage scheme as such, but is included here as it is often referenced with the other arrays.

3.7.1 Sparse Storage Schemes.

The storage of large sparse matrices can have a significant effect on the remainder of the algorithm, and must therefore be carefully developed. Full matrices present no problems, unless they are too large to fit into available memory at one time, in which case careful consideration must be given to the access patterns which will be used. Sparse matrix storage schemes must not only provide great reductions in memory requirements, but should also require little overhead in accessing an element, and must be able to withstand the creation and deletion of elements.

Many schemes have been proposed, but the most popular for power systems is some form of a linked list. Linked lists are lists of data in which each element contains a link to another element. In the simplest form, this is a single link to the element which should be accessed next if the elements are required in a particular order. Linked lists allow elements to be inserted and deleted at arbitrary points merely by altering the values of a few of these links. Without linking, this would require large blocks of data to be moved about in memory, which is less efficient. For repeated read accesses, linear lists are better, since one level of redirection is removed, and it might be advantageous to convert from one form to the other during processing. Linked lists use memory for links and index values, while linear lists only need index values, but both save memory over storing the full matrix, with all its zero elements.

Some linked lists have extra links to facilitate special, frequently used access methods, but the break–even point where the extra updating required begins to outweigh the advantages of faster access must be considered. The usual limit of extra links is double linking, where each element contains one link to the next element, and one to the previous, which permits bi–directional searching. A more important property is that an

element can be inserted into a list knowing only one of its new neighbours, and can be deleted without searching for either of its neighbours.

To insert an element into a singly linked list, a search must be made through the elements until the desired position is reached. This is usually signalled by having progressed one element too far. For example, a common case is 'insert element x in row k of column j'. This requires a scan down column j, examining the rows of the elements, until an element with a row number greater than k is found, so element x must be inserted before this element. As the links are one–way, the previous element is 'lost', unless its location was stored in a temporary location, which would slow the speed of the search. It is the link of the previous element which must be modified to insert the new element, whose link would point to the element pointed to by the link in the previous element. For deletion, a similar search must be made, this time searching for the element to be deleted, and remembering the previous elements. The link in the deleted element is assigned to the link in the previous element, which used to point to the deleted element.

For doubly–linked lists, it does not matter if the search progresses too far, as a link is available to the previous element. Deletion is simpler, since both affected neighbours are identified by the deleted element's links, and these links can easily be patched without calculation.

It is advantageous if all accesses to the data follow a similar path, for example scanning down columns searching for a row entry. This limits the links which are required, and with ordered row entries, can provide a fast indication of whether an element exists. For this access pattern, access to columns could be provided by links to the first non–zero element in each column, from an array indexed by column number. The rows could be held in a doubly–linked list, with the end of column being indicated by a zero link.

3.7.2 Sparse solution methods against full methods.

The introduction of fast vector processors, such as the Cray-1 supercomputer, raised the question as to whether sparse equations could be solved faster using vector arithmetic on long, assumed full, vectors, than by exploiting sparsity. Studies showed that the break–even point for the Cray-1 was matrices which were about 5% full. For matrices with more non–zero elements, the full vector method was faster, while with fewer elements, sparse methods were better. This compares to typical power system matrices

which are between 0.5% and 3% full. The I.E.E.E. 118 node network has 476 nonzeros, which is 3.4%, while the 734 node C.E.G.B. matrix has 2696 nonzeros, which is 0.5% full.

Super–fast vector computers are presently therefore no substitute for sparse matrix methods. Assuming the vector calculation speed approached the asymptotic maximum of 100 MFlops, the sparse matrix speed achieved was about 5 MFlops, which is only a factor of 10 faster than the VAX-8600 in use by O.C.E.P.S., and on a par with the FPS-5205 currently used for solving the sparse matrices during simulation.

The memory requirements are also prohibitive when large matrices are used, unless the vectors are formed immediately prior to use by scatter operations, and compacted on completion of the calculation by gather operations. These operations make heavy use of the memory interface, and would therefore slow calculations.

3.8 Representing sparse complex numbers.

Quantities found in power system analysis are frequently complex, that is they possess both magnitude and angle, such as voltage, power and current. FORTRAN and ADA, with their direct support of complex numbers provide two possible representations for these quantities, while other programming languages without this support provide only one possible representation. The direct support of complex numbers has not been used by the O.C.E.P.S. group for two reasons, firstly that the standard for FORTRAN-77 only defines single precision complex numbers, and while VAX–FORTRAN-77 provides double precision, this would cause portability difficulties if the code were moved to other execution platforms, such as SUN or IBM. Double precision variables are required because single precision values are not sufficient to ensure the numerical stability of the solution of sparsity ordered power system matrices. The second reason is that the modulus of a complex number cannot be differentiated, which causes probelms in the definition of differential models for generators. The approach used by O.C.E.P.S. is to split all equations in two, with a real part and an imaginary part. This increases the size of the matrices, but removes stability or standardisation problems, removes dependence on FORTRAN and ADA, and also cures differentiability difficulties. The introduction of the C++ object oriented [137] programming language might provide a different solution

by allowing a complex class to be defined to act as desired, while retaining the benefit of similarity to the widely–used C programming language.

Representing complex numbers in this way results in a four–fold increase in the number of matrix elements, since the number of equations doubles, and each equation has double the number of terms. The adverse effect of this on the sparse matrix structure can be nullified, at the expense of some inefficiencies in the solution of generator equations, by dealing with two—by—two submatrices [31]. These matrices are made up from real and complex pairs of equations, and occur naturally in the matrix when the real and imaginary equations are placed side by side. This returns the number of elements seen by an ordering routine or by code generation to the original value, and also affects the order of calculations. Processing small matrices instead of individual values results in more pattern in the calculations, which can be used to improve processor pipeline performance or reduce branch statements.

Matrix operations such as multiplication are governed by more restrictive associativity and commutivity rules than those which apply to scalar mathematics, so care must be taken when performing the conversion between scalar and 2–by–2 formulations to ensure that these rules are followed. If the rules are broken, then the desired operation may be impossible, due to incorrectly matched matrix dimensions, or might just produce an incorrect result, which is more difficult to detect and correct. The representation of the equations in this form, seems to increase numerical stability slightly, and the O.C.E.P.S. fully–coupled loadflow programs which use this formulation appear to be more stable than others which do not.

3.9 Memory usage.

The original Zollenkopf algorithm initially duplicates all off–diagonal elements of the matrix. As the simulation of the ordering proceeds, one of each pair of duplicates is deleted. New elements (fill) are created, also as duplicate pairs, and are then processed as if they were original matrix elements. This duplication of elements generally causes the final number of used locations to be less than the number initially used, but usage peaks about two–thirds through the simulation. The creation and deletion of elements not only creates problems of list management, but also of memory allocation.

The original algorithm places empty slots into a singly linked list, from which space for new elements is obtained, and to which empty locations are returned. Empty locations are placed into the linked list in the order in which they are freed, i.e., the list is used as a push–down stack. As memory usage initially increases, as more fill occurs than duplicate deletions, the upper bound of the memory table is increased. Later, when memory is freed, some elements will remain in high locations, while gaps will be created lower down in memory, which makes inefficient use of the high–speed memory likely to be used in special purpose hardware. The simulation phase could be altered by storing the free locations in memory in a singly–linked list in order of increasing memory location, instead of in the order in which they were freed. This would ensure that when a new location is needed, the lowest available address would always be chosen. Although it is conceptually simple, this scheme has several disadvantages. It will not eliminate the problem of fragmentation, because the need for new locations is not matched to the generation of free locations, so some locations low in the memory structure could be freed late in the simulation, when the need for new locations has disappeared. The scheme is also very inefficient, as considerable time would be required to search through the list of locations for the correct position for insertion, and many such searches would be required since many elements are deleted and re–used.

Fast processors generally require fast memory, and floating point operands consume large amounts of memory. Fast memory is not only expensive, but generally of low packing density, which reduces the amount which can be placed on a single circuit board, along with the processor. The difficulties involved in transmitting electrical signals along connections between boards mean that fast memory is restricted to being on the same physical card, or a dedicated daughter card, to the processor. It is therefore important to maximise the use of on board memory by packing data into it as tightly as possible. This is not only a financial decision, but could also determine the physical possibility of the solution using current technology.

The Zollenkopf algorithm does not perform any memory packing after simulating the elimination, but a new subroutine could not only perform packing to improve memory utilisation, but could also change the form of the lists and re–order elements to facilitate improvements in the calculation phase. The simple scheme proposed does not alter the method of accessing the elements in memory, and does not render any data unnecessary by altering the implied data in the storage structure. The next section investigates what

alternative strategies exist, and whether they offer advantages over the simple methods outlined in this section.

3.10 List repacking strategies.

Several alternative repacking strategies were evaluated. Moving elements around in memory requires the provision of a mapping between the old and new locations, to enable the set–up routines to place their data into the correct locations. The creation of new elements during the ordering phase, about which the set–up routines have no knowledge, and which occupy locations initially used for deleted duplicates, causes problems for their correct initialisation. A set of tri–state flags, with states: old, new and empty, and with associated pointers was found to provide sufficient information for the correct initialisation of the matrix elements. The usage entries are made and modified during the simulation of the ordering, which will slow the ordering, but this is the only time when such a list can be made.

Each equation which participates in the generation of the initial matrix must know where to place its result. If the matrix remained un–packed, either simple array indices or pointers would be used to indicate the destination locations. A similar approach can be used for the packed matrix, if these pointers are modified according to the mapping vector. It would be best to perform this modification once, instead of using indirection through the mapping table, since this would be more efficient. Thereafter, there would be no computational cost resulting form the mapping.

A list may also be needed of locations which were freed and reused during the simulation, as these locations will need to be zeroed between solutions, as they would otherwise receive no initialisation. The list would not be required if all locations are zeroed, but this might not be the case for some initialisation methods. The generation of such a list is trivial, but for efficiency considerations should be avoided if possible.

As a first attempt at improving memory useage, elements were simply re–packed by creating a new list structure, empty but otherwise identical to the original. The usage list was then scanned for locations which contained either original or fill elements. When such a location was found, it was inserted into the new list, and a mapping entry was made in the mapping table. A pass must then be made through the array containing the

location of the first element in each column, since the pointers to the first element in each column must be updated. The mapping vector can be used to assist this conversion.

It was soon realised that memory accessing could be improved if column entries were placed into contiguous memory locations. This has the benefit of causing less thrashing in virtual memory, caches or 'special' memory access modes (particularly static column DRAM access), since more accesses would be made to local data. It would also remove the need for the linked–list data structure, since column elements are guaranteed to be in consecutive memory locations. The parallel array of row indices is still, of course, required, since the arrays still contain a packed sparse matrix, and some indication of end of column is also required. Nevertheless, about one third of the memory locations are saved by this reordering. Scanning the original list structure in this order is not difficult, since it is the order in which the lists are intended to be scanned, and might be faster than a linear scan through the complete list, since fewer locations would be accessed, but each access would involve an extra level of indirection.

If the row entries in each column were additionally sorted into the order of the ordering, then much more variety is possible in the calculation phase. The original method kept the elements ordered according to their original row or column number. With this storage, the structure of the matrix was unclear, with no well defined pattern. Ordering the row entries according to elimination order brings out the patterns, and permits different calculation orders to exploit the patterns. The whole matrix now appears lower triangular, and if only the referenced columns of any single column are viewed at one time, the resulting submatrix is full lower triangular. If a list of the scattered addresses is provided, with preferably automatic address generation, then any of the calculation orders applicable to full matrices can be used to eliminate this column.

Sorting the elements takes longer than simply packing them together, because an intermediate list must be formed, along with a mapping vector. This list must then be sorted, and the elements assigned to locations in the new list structure, and the overall mapping vector produced as before. Because most columns would contain very few rows, a simple sort routine can be used, which might out–perform a more complex sorting method which requires an initial set–up stage. For the 734 node network, the maximum degree of any column during elimination is sixteen, and the maximum degree of any column after the simulation of the elimination is ten, with the average remaining

between three and four. The number of entries which must be sorted is one fewer than this, because these degree totals include the diagonal elements which are not stored in the off–diagonal lists. A simple sort method with an execution time proportional to n^2 will not be significantly bettered by one with an execution time proportional to $n \log n$, which is believed to be optimal for large n. Nevertheless, a Quick Sort was used, because one is provided in the Standard Library of the C language.

A re–numbering of the elements into the elimination order could also be applied, but this would serve no useful purpose, because during the numerical elimination, a search is only made for a particular element if it is known to be present in a column, so there is no need for size comparisons to detect failure, only for equality comparisons to signal that the target element has been found. It is possible to remove the indication of end of column, because with the elements of each column in contiguous locations, the start location and the degree of each column defines where the last element must be located. This is preferable to removing the array of degree totals, because each of those elements is an integer, while each of the elements in the main arrays consists of eight, double precision values, which take space in the special purpose hardware, unlike the degree totals, which are private to the main processor. No further savings on the information which must accompany the main element list are possible, since row indices and pointer to the start of each column must still be provided.

Calculations could then be scheduled to suit the numeric processor, for example if this processor has limited internal registers, then either of the column or row should be held constant for as long as possible, so that some of the registers could store a set of entries from either the principal row or principal column, which reduces access to external memory.

Storing in elimination order also largely eliminates any problems with processor stalls due to data dependencies. For reasonable pipeline lengths, these will only occur when a column references a single other column, which is also the next column which is to be eliminated. If there are references to other columns in addition to the next one, then the next column must be referenced first, because of the ordering of rows based on elimination order. Calculations will then naturally fall between its modification, and its first usage as principal column, which will ensure that these calculations have passed completely through the FPU pipeline before the results are required. These intervening

calculations will not be present for a column which references no other columns apart from its successor.

3.10.1 Pipelined processing and data dependencies.

A pipelined processor may have several calculations underway at any one time, in order to increase its overall processing speed. Pipelined processing becomes impossible where the serial stream of calculations contains data dependencies. Some processors ban or place severe restrictions on the pipelining of such instructions, while others detect the dependencies, and reschedule the pipeline to satisfy them, which unexpectedly slows down the overall rate of calculation. It is therefore beneficial to remove as many of these dependencies as possible, while still retaining sparsity and other good properties of the ordering. Taoka and Abe [141] discuss the importance of keeping a pipeline busy.

Most FPU pipelines are relatively shallow, usually of about four stages, but this might be considerably extended by all the functional blocks between memory and FPU. A pipelined FPU is not necessary before problems are encountered with data dependencies. RISC processors, for example, use a load / store architecture in which data fetches and stores are separated from the instructions which process the data, and use similar loads and stores to access floating point data and units. Pipeline problems can become evident with these processors by incorrect results, or by the processor placing itself in a paused state, waiting for data to become avaialable, which will show up as an increase in calculation time. An element would usually only be available for use once it is written back into memory, and an attempt should only be made to access it once it is guaranteed to have been written, unless register optimisation is attempted.

There are three forms of data dependency which are encountered during the processing of sparse matrices. The first, and easiest to analyse is wholly internal to the processing of one of the two–by–two submatrices. Here, several pairs of products must be summed, and it is not possible to perform the summation until both products are available. The useage of multiply–add instructions highlights the scheduling required, but on–chip registers would usually make the pipeline as shallow as possible. This dependency is completely predictable, and will be present for every matrix multiplictaion.

The second dependency is also straight forward to analyse, since it is between the submatrices which must be processed during the elimination of any one column. The

leading diagonal matrix must be inverted before the matrices in the leading column can be adjusted, and these must be adjusted before they can be used to adjust the other elements in their rows. Both these dependencies can be handled by storing results in on–chip register files, and scheduling calculations so that they take advantage of this.

The third dependency is more difficult to analyse, because it is between elements which are shared between the elimination of two successive columns. The elimination of a complete column is regarded as the basic building block of bifactorisation processing, so inter–block dependencies are more problematical than intra–block dependencies. Analysis can be helped considerably if the elements within these building blocks are procesed in a regular, well defined order which is dependent on the elimination order, rather than the initial numbering of the nodes. An investigation into reducing the number of possible inter–block dependencies will be discussed in more detail in chapter 4.

3.10.2 Remedies for pipeline dependencies.

Two different instructions could be provided for the case of a column containing a single reference, one of which processes at full speed, while the other contains a delay to allow the pipeline to be flushed. This is more efficient than providing duals of each instruction, but does require some look–ahead during code generation. A similar problem occurs during forward and backward substitution, and here the reduced number of calculations per column might cause problems with columns referencing two other columns, and not just one. Similar remedies are equally applicable to this case.

Re–arranging the calculation order does not affect the actual calculations which take place, since the only necessary calculation orderings are still maintained, and all other calculations on a single column are independent of each other. The inversion of the leading column entry must be performed first, and the multiplication of the first entry in each row by this matrix must be performed before any elements from that row are processed. The correctness of solutions calculated by this method was assumed to be unaffected.

Array names used in the Zollenkopf program.			
Name		Expansion	Explanation
ITAG	I	Index Tag	Contains the row index of the entry. The list in which this element is located defines the other column index.
LNXT	L	Next Location	Contains the link to the next element in the linked list. May be either an index into this array and ITAG, or a pointer if these are combined into a structure.
NSEQ		Sequence	Initially contains a list of all the participating columns in the matrix, and is used to store the elimination order as it is determined. After ordering NSEQ(1) contains the column index which will be eliminated first.
LCOL		Locate Column	Used to find the first pair of ITAG and LNXT for each column, i.e., it points to the start of each linked list.
NOZE		Non–Zeros	maintains a count of the number of non–zeros ein each column, for use by the optimal ordering.
	M		Memory loaction in the arrays.
	(C)		Column number for entry
	end		Is this entry the last in a column?

Table 3.1 Array names used in Zollenkopf program.

3.10.3 Illustration of data repacking.

Repacking the floating–point arrays aimed to achieve two goals, firstly to reduce the amount of memory required, both to hold the matrix structure and to hold the floating–point data, as this is likely to be at a premium, and secondly to bring out all the available structure in the calculations. The I.E.E.E. 30 node network is used as an example to illustrate these two aims. Re–packing is best illustrated by taking snapshots of the memory arrays and matrix structure at three important points during the elimination process; before ordering, after ordering, and after re–packing. The arrays in memory do not show the full advantages of re–packing which are found with the larger matrices, beacause the maximum length of the arrays does not increase during the ordering, but

the small network was chosen so that important points would not be lost in a mass of data.

Unordered 30 node network ITAG and LNXT by memory location											
M	I	L	(C)	M	I	L	(C)	M	I	L	(C)
1	2	3	(1)	31	10	33	(9)	61	20	0	(19)
2	1	7	(2)	32	9	37	(10)	62	19	0	(20)
3	3	0	(1)	33	11	0	(9)	63	22	0	(21)
4	1	11	(3)	34	9	0	(11)	64	21	65	(22)
5	5	9	(2)	35	20	39	(10)	65	24	0	(22)
6	2	17	(5)	36	10	62	(20)	66	22	68	(24)
7	4	5	(2)	37	17	35	(10)	67	24	0	(23)
8	2	12	(4)	38	10	58	(17)	68	23	69	(24)
9	6	0	(2)	39	21	41	(10)	69	25	0	(24)
10	2	14	(6)	40	10	63	(21)	70	24	71	(25)
11	4	0	(3)	41	22	0	(10)	71	26	73	(25)
12	3	13	(4)	42	10	64	(22)	72	25	0	(26)
13	6	15	(4)	43	13	45	(12)	73	27	0	(25)
14	4	19	(6)	44	12	0	(13)	74	25	79	(27)
15	12	0	(4)	45	14	47	(12)	75	30	0	(27)
16	4	43	(12)	46	12	51	(14)	76	27	82	(30)
17	7	0	(5)	47	15	49	(12)	77	29	75	(27)
18	5	20	(7)	48	12	52	(15)	78	27	81	(29)
19	7	21	(6)	49	16	0	(12)	79	28	77	(27)
20	6	0	(7)	50	12	57	(16)	80	27	0	(28)
21	8	25	(6)	51	15	0	(14)	81	30	0	(29)
22	6	29	(8)	52	14	53	(15)	82	29	0	(30)
23	28	0	(6)	53	18	55	(15)	83	0	84	(0)
24	6	30	(28)	54	15	59	(18)	84	0	85	(0)
25	9	27	(6)	55	23	0	(15)				
26	6	31	(9)	56	15	67	(23)				
27	10	23	(6)	57	17	0	(16)				
28	6	32	(10)	58	16	0	(17)				
29	28	0	(8)	59	19	0	(18)				
30	8	80	(28)	60	18	61	(19)				

Table 3.2 ITAG and LNXT for unordered network.

Unordered 30 node network Entries for each column			
COL	LCOL	NOZE	NSEQ
1	1	3	1
2	2	5	2
3	4	3	3
4	8	5	4
5	6	3	5
6	10	8	6
7	18	3	7
8	22	3	8
9	26	4	9
10	28	7	10
11	34	2	11
12	16	6	12
13	44	2	13
14	46	3	14
15	48	5	15
16	50	3	16
17	38	3	17
18	54	3	18
19	60	3	19
20	36	3	20
21	40	3	21
22	42	4	22
23	56	3	23
24	66	4	24
25	70	4	25
26	72	2	26
27	74	5	27
28	24	4	28
29	78	3	29
30	76	3	30
31	0	1	31

Table 3.3 LCOL, NOZE and NSEQ for unordered network.

Unordered 30 node network ITAG and LNXT by column														
(C)	M	I	L	end?	(C)	M	I	L	end?	(C)	M	I	L	end?
(1)	1	2	3		(10)	37	17	35		(22)	64	21	65	
(1)	3	3	0	end	(10)	35	20	39		(22)	65	24	0	end
(2)	2	1	7		(10)	39	21	41		(23)	56	15	67	
(2)	7	4	5		(10)	41	22	0	end	(23)	67	24	0	end
(2)	5	5	9		(11)	34	9	0	end	(24)	66	22	68	
(2)	9	6	0	end	(12)	16	4	43		(24)	68	23	69	
(3)	4	1	11		(12)	43	13	45		(24)	69	25	0	end
(3)	11	4	0	end	(12)	45	14	47		(25)	70	24	71	
(4)	8	2	12		(12)	47	15	49		(25)	71	26	73	
(4)	12	3	13		(12)	49	16	0	end	(25)	73	27	0	end
(4)	13	6	15		(13)	44	12	0	end	(26)	72	25	0	end
(4)	15	12	0	end	(14)	46	12	51		(27)	74	25	79	
(5)	6	2	17		(14)	51	15	0	end	(27)	79	28	77	
(5)	17	7	0	end	(15)	48	12	52		(27)	77	29	75	
(6)	10	2	14		(15)	52	14	53		(27)	75	30	0	end
(6)	14	4	19		(15)	53	18	55		(28)	24	6	30	
(6)	19	7	21		(15)	55	23	0	end	(28)	30	8	80	
(6)	21	8	25		(16)	50	12	57		(28)	80	27	0	end
(6)	25	9	27		(16)	57	17	0	end	(29)	78	27	81	
(6)	27	10	23		(17)	38	10	58		(29)	81	30	0	end
(6)	23	28	0	end	(17)	58	16	0	end	(30)	76	27	82	
(7)	18	5	20		(18)	54	15	59		(30)	82	29	0	end
(7)	20	6	0	end	(18)	59	19	0	end					
(8)	22	6	29		(19)	60	18	61						
(8)	29	28	0	end	(19)	61	20	0	end					
(9)	26	6	31		(20)	36	10	62						
(9)	31	10	33		(20)	62	19	0	end					
(9)	33	11	0	end	(21)	40	10	63						
(10)	28	6	32		(21)	63	22	0	end					
(10)	32	9	37		(22)	42	10	64						

Table 3.4 ITAG and LNXT for unordered network.

Unpacked 30 node network ITAG and LNXT by memory location											
M	I	L	(C)	M	I	L	(C)	M	I	L	(C)
1	2	3	(1)	31	10	0	(9)	61	20	45	(0)
2	3	74	(0)	32	24	0	(27)	62	19	0	(20)
3	3	0	(1)	33	11	83	(0)	63	22	0	(21)
4	2	11	(3)	34	9	0	(11)	64	15	69	(0)
5	7	12	(0)	35	10	58	(19)	65	24	0	(22)
6	2	17	(5)	36	10	62	(20)	66	22	5	(0)
7	4	9	(2)	37	10	19	(0)	67	24	0	(23)
8	2	66	(0)	38	10	52	(17)	68	23	39	(0)
9	6	0	(2)	39	21	61	(0)	69	27	8	(0)
10	24	0	(6)	40	10	63	(21)	70	24	73	(25)
11	4	0	(3)	41	24	0	(10)	71	2	20	(7)
12	3	2	(0)	42	10	65	(22)	72	25	0	(26)
13	6	15	(4)	43	13	33	(0)	73	27	0	(25)
14	12	10	(6)	44	12	0	(13)	74	25	25	(0)
15	12	0	(4)	45	14	30	(0)	75	30	82	(0)
16	6	28	(0)	46	12	51	(14)	76	27	0	(30)
17	7	0	(5)	47	24	0	(12)	77	29	68	(0)
18	5	43	(0)	48	12	55	(15)	78	27	81	(29)
19	6	16	(0)	49	12	53	(0)	79	12	41	(10)
20	6	0	(7)	50	12	57	(16)	80	27	0	(28)
21	8	18	(0)	51	15	0	(14)	81	30	0	(29)
22	6	29	(8)	52	12	0	(17)	82	29	77	(0)
23	6	32	(27)	53	10	37	(0)	83	0	84	(0)
24	6	80	(28)	54	15	59	(18)				
25	9	75	(0)	55	24	0	(15)				
26	6	31	(9)	56	15	67	(23)				
27	10	14	(6)	57	17	0	(16)				
28	6	64	(0)	58	15	0	(19)				
29	28	0	(8)	59	19	0	(18)				
30	8	21	(0)	60	10	48	(15)				

Table 3.5 ITAG and LNXT for unpacked network.

Unpacked 30 node network Entries for each column			
COL	LCOL	NOZE	NSEQ
1	1	3	11
2	7	3	13
3	4	3	26
4	13	3	1
5	6	3	5
6	27	4	8
7	71	3	14
8	22	3	16
9	26	3	18
10	79	3	20
11	34	2	21
12	47	2	23
13	44	2	29
14	46	3	30
15	60	4	9
16	50	3	25
17	38	3	3
18	54	3	7
19	35	3	28
20	36	3	17
21	40	3	19
22	42	3	22
23	56	3	2
24	0	1	27
25	70	3	4
26	72	2	15
27	23	3	6
28	24	3	10
29	78	3	12
30	76	2	24
31	0	1	31

Table 3.6 LCOL, NOZE and NSEQ for unordered network.

Unpacked 30 node network ITAG and LNXT by column									
(C)	M	I	L	end	(C)	M	I	L	end?
(1)	1	2	3		(16)	57	17	0	end
(1)	3	3	0	end	(17)	38	10	52	
(2)	7	4	9		(17)	52	12	0	end
(2)	9	6	0	end	(18)	54	15	59	
(3)	4	2	11		(18)	59	19	0	end
(3)	11	4	0	end	(19)	35	10	58	
(4)	13	6	15		(19)	58	15	0	end
(4)	15	12	0	end	(20)	36	10	62	
(5)	6	2	17		(20)	62	19	0	end
(5)	17	7	0	end	(21)	40	10	63	
(6)	27	10	14		(21)	63	22	0	end
(6)	14	12	10		(22)	42	10	65	
(6)	10	24	0	end	(22)	65	24	0	end
(7)	71	2	20		(23)	56	15	67	
(7)	20	6	0	end	(23)	67	24	0	end
(8)	22	6	29		(25)	70	24	73	
(8)	29	28	0	end	(25)	73	27	0	end
(9)	26	6	31		(26)	72	25	0	end
(9)	31	10	0	end	(27)	23	6	32	
(10)	79	12	41		(27)	32	24	0	end
(10)	41	24	0	end	(28)	24	6	80	
(11)	34	9	0	end	(28)	80	27	0	end
(12)	47	24	0	end	(29)	78	27	81	
(13)	44	12	0	end	(29)	81	30	0	end
(14)	46	12	51		(30)	76	27	0	end
(14)	51	15	0	end					end
(15)	60	10	48						end
(15)	48	12	55						end
(15)	55	24	0	end					end
(16)	50	12	57						end

Table 3.7 ITAG and LNXT for unpacked network.

Packed 30 node network ITAG by memory location							
M	I	(C)	end?	M	I	(C)	end?
1	9	(1)	end	31	2	(18)	
2	12	(2)	end	32	6	(18)	end
3	25	(3)	end	33	27	(19)	
4	3	(4)		34	6	(19)	end
5	2	(4)	end	35	10	(20)	
6	7	(5)		36	12	(20)	end
7	2	(5)	end	37	15	(21)	
8	28	(6)		38	10	(21)	end
9	6	(6)	end	39	10	(22)	
10	15	(7)		40	24	(22)	end
11	12	(7)	end	41	4	(23)	
12	17	(8)		42	6	(23)	end
13	12	(8)	end	43	6	(24)	
14	19	(9)		44	24	(24)	end
15	15	(9)	end	45	6	(25)	
16	19	(10)		46	12	(25)	end
17	10	(10)	end	47	10	(26)	
18	22	(11)		48	12	(26)	
19	10	(11)	end	49	24	(26)	end
20	15	(12)		50	10	(27)	
21	24	(12)	end	51	12	(27)	
22	30	(13)		52	24	(27)	end
23	27	(13)	end	53	12	(28)	
24	27	(14)	end	54	24	(28)	end
25	6	(15)		55	24	(29)	end
26	10	(15)	end	56	0	(30)	
27	27	(16)		57	0	(30)	
28	24	(16)	end				
29	2	(17)					
30	4	(17)	end				

Table 3.8 ITAG for packed network (LNXT implied).

Packed 30 node network Entries for each column			
COL	LCOL	NOZE	NSEQ
1	4	3	11
2	41	3	13
3	29	3	26
4	45	3	1
5	6	3	5
6	50	4	8
7	31	3	14
8	8	3	16
9	25	3	18
10	53	3	20
11	1	2	21
12	55	2	23
13	2	2	29
14	10	3	30
15	47	4	9
16	12	3	25
17	35	3	3
18	14	3	7
19	37	3	28
20	16	3	17
21	18	3	19
22	39	3	22
23	20	3	2
24	0	1	27
25	27	3	4
26	3	2	15
27	43	3	6
28	33	3	10
29	22	3	12
30	24	2	24
31	0	1	31

Table 3.9 LCOL, NOZE and NSEQ for packed network.

Packed 30 node network Row numbers in elimination order							
M	NSEQ(I)	(C)	end?	M	NSEQ(I)	(C)	end?
1	15	(1)	end	31	23	(18)	
2	29	(2)	end	32	27	(18)	end
3	16	(3)	end	33	24	(19)	
4	17	(4)		34	27	(19)	end
5	23	(4)	end	35	28	(20)	
6	18	(5)		36	29	(20)	end
7	23	(5)	end	37	26	(21)	
8	19	(6)		38	28	(21)	end
9	27	(6)	end	39	28	(22)	
10	26	(7)		40	30	(22)	end
11	29	(7)	end	41	25	(23)	
12	20	(8)		42	27	(23)	end
13	29	(8)	end	43	27	(24)	
14	21	(9)		44	30	(24)	end
15	26	(9)	end	45	27	(25)	
16	21	(10)		46	29	(25)	end
17	28	(10)	end	47	28	(26)	
18	22	(11)		48	29	(26)	
19	28	(11)	end	49	30	(26)	end
20	26	(12)		50	28	(27)	
21	30	(12)	end	51	29	(27)	
22	14	(13)		52	30	(27)	end
23	24	(13)	end	53	29	(28)	
24	24	(14)	end	54	30	(28)	end
25	27	(15)		55	30	(29)	end
26	28	(15)	end	56	0	(30)	
27	24	(16)		57	0	(30)	
28	30	(16)	end				
29	23	(17)					
30	25	(17)	end				

Table 3.10 Row number given in elimination order for packed network.

The arrays for the 30 node network are presented in tables 3.2–10. The array names are defined in table 3.1, where the asingle character abbreviations used in the wider tables are also siven for LNXT and ITAG. The initial form of the arrays (table 3.2) shows the duplicate elements required by the Zollenkopf program. Table 3.11 shows how table 3.2 and table 3.3 can be used to determine the non–zero elements in a column. The example chooses column 9, so the first step is to find the first off–diagonal element of column 9, so LCOL(9) is examined, which shows the first element is stored in location 26 of the ITAG and LNXT arrays. Examination of ITAG(26) shows that this element is located in row 6, and examination of LNXT(26) shows that the next element is stored in location 31 of these arrays. This is examined, and is found to be in row 10, and point to another element in location 33. This in turn is in row 11, but has an LNXT value of zero, so there are no mere elements in column 9. The degree of column 9 can now be examined (NOZE(9)), and is found to be four, which is correct, because three off–diagonal elements were found, in addition to the implied diagonal element. Access to the other columns, and to the other tables follow a similar pattern.

After ordering, (tables 3.5–7) all the duplicate elements have been removed, but additional (fill) elements have been created. There are several empty locations in the arrays, where duplicates were deleted, but unusually the upper bound of the matrix did not move, because initially more than twice as many duplicates were deleted than fills generated, so enough emtied locations were available to accomodate the fill elements and their duplicates. The repacked structure (tables 3.8–10) shows that the memory space required to hold this matrix structure can be reduced, both by packing the elements closer together, and by removing redundant information, such as links to the next element and indications of end of column. In particular, LNXT() is not required, since all column entries are in consecutive locations, and the end of column is indicated by the degree of the column NOZE() and the start of column LCOL(). A final table is given where the row entries have been numbered in their elimination order, so that the structure can be seen to define the matrix in figure 3.6.

Figure 3.4 shows the structure of the matrix in its uneliminated state, while figure 3.5 shows the matrix after ordering and elimination, with the nodes still numbered in their natural order, so that node 1 is still in the top left of the matrix. The large circles represent connections which are represented in the array structure (i.e., are in the lower triangular factor matrix), and the small circles represent their mirror images. It will be

Example of the use of the sparse tables.		
Access	Value	Interpretation.
`LCOL( 9)`	26	First element is in location 26
`ITAG(26)`	6	First element is in row 6
`LNXT(26)`	31	Next element is in location 31
`ITAG(31)`	10	Second element is in row 10
`LNXT(31)`	33	Next element is in location 33
`ITAG(33)`	11	Third element is in row 11
`LNXT(33)`	0	No further elements in this column
`NOZE( 9)`	4	Three off–diagonal elements, so four in column 9 overall

Table 3.11 Example of the use of sparse arrays.

seen that there is a mixture of elements on both sides of the leading diagonal. Once the matrix is repacked according to the final algorithm, so that all entries are placed into the elimination order, the structure becomes much clearer, and the full sub–matrices become readily identifiable in figure 3.6. Figure 3.7 shows the fill–in locations as hollow circles, and this can be compared to figure 3.6, which has the same structure.

3.11 Handling Generator Equations.

The differential equations governing the behaviour of generators are special in certain respects. Since they do not change structure with time, unlike the remainder of the matrix, it would be beneficial if they could be removed from the re–ordering and code–generation processes. If the generator only connects to one busbar, as most would, then this can be achieved. If the generator feeds one busbar while taking some reference values from another, then problems would arise if these two busbars became part of different nodes, because an extra term would be generated in the matrix. If the generators are placed before any nodes in the matrix, then because each can only connect to one node, no fill in terms are generated, and the network elimination can proceed as if they were not present, as long as the feed terms have been included.

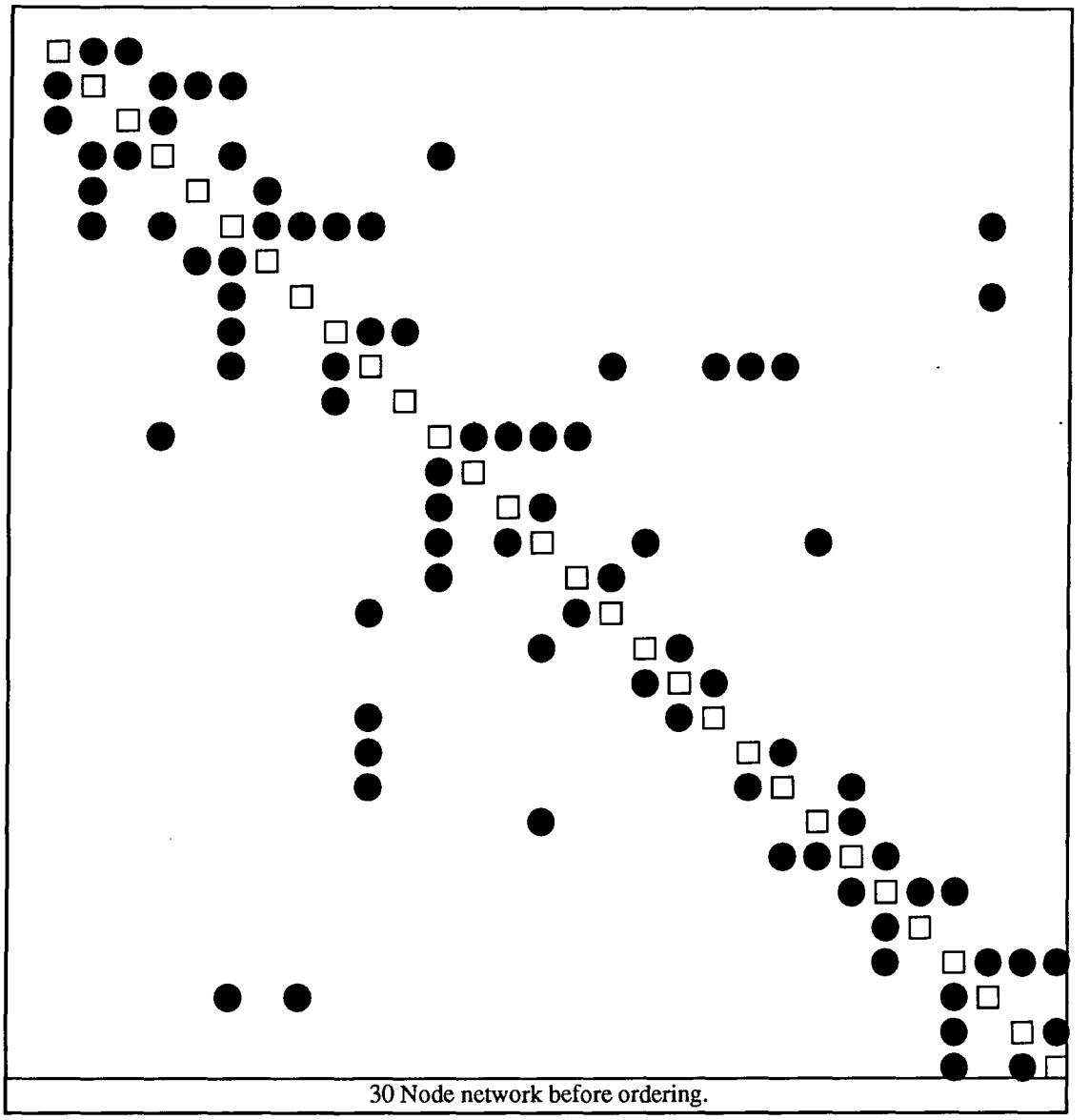


Figure 3.4 Unordered I.E.E.E. 30 node network.

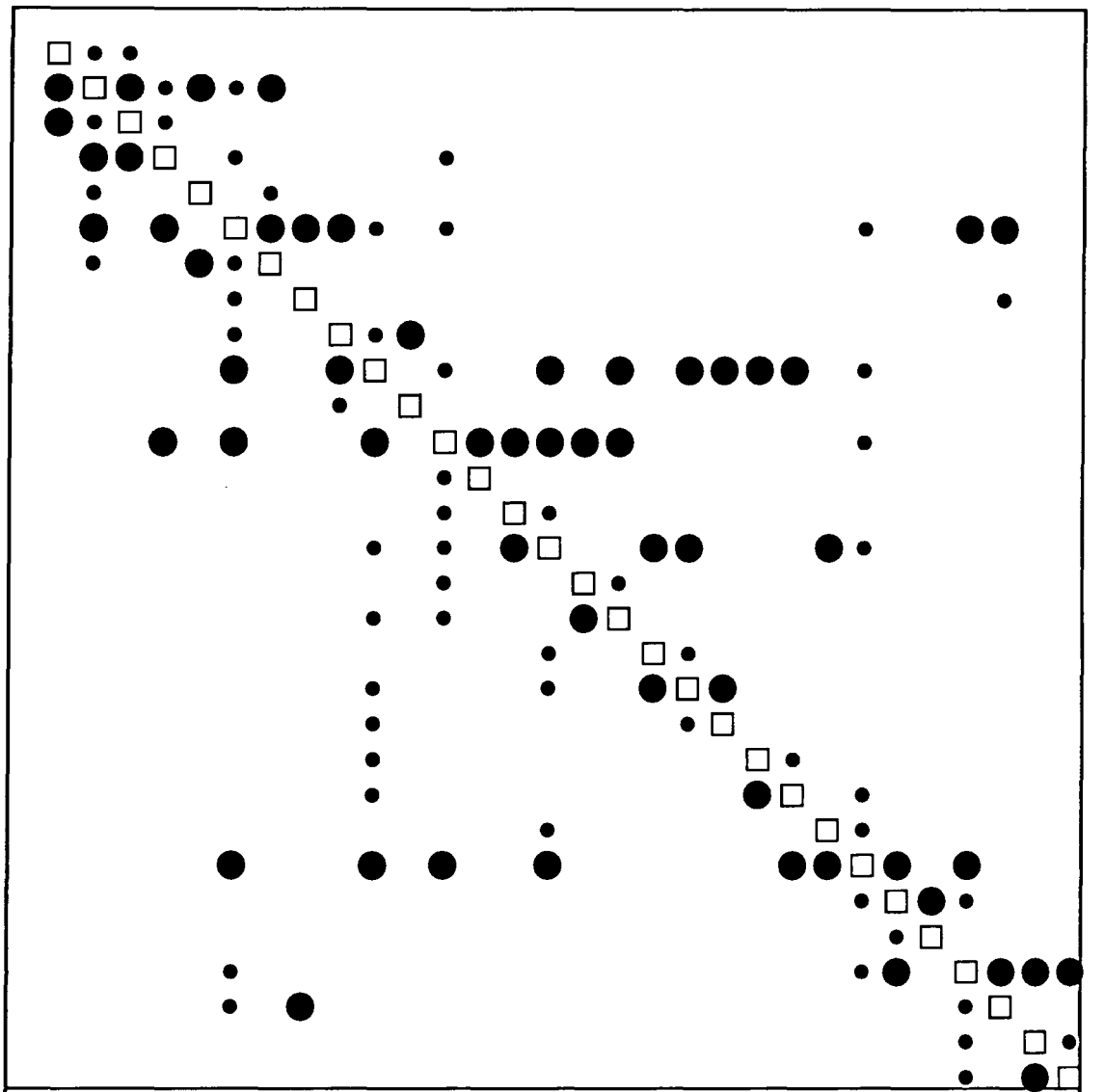


Figure 3.5 Ordered but unpacked I.E.E.E. 30 node network.

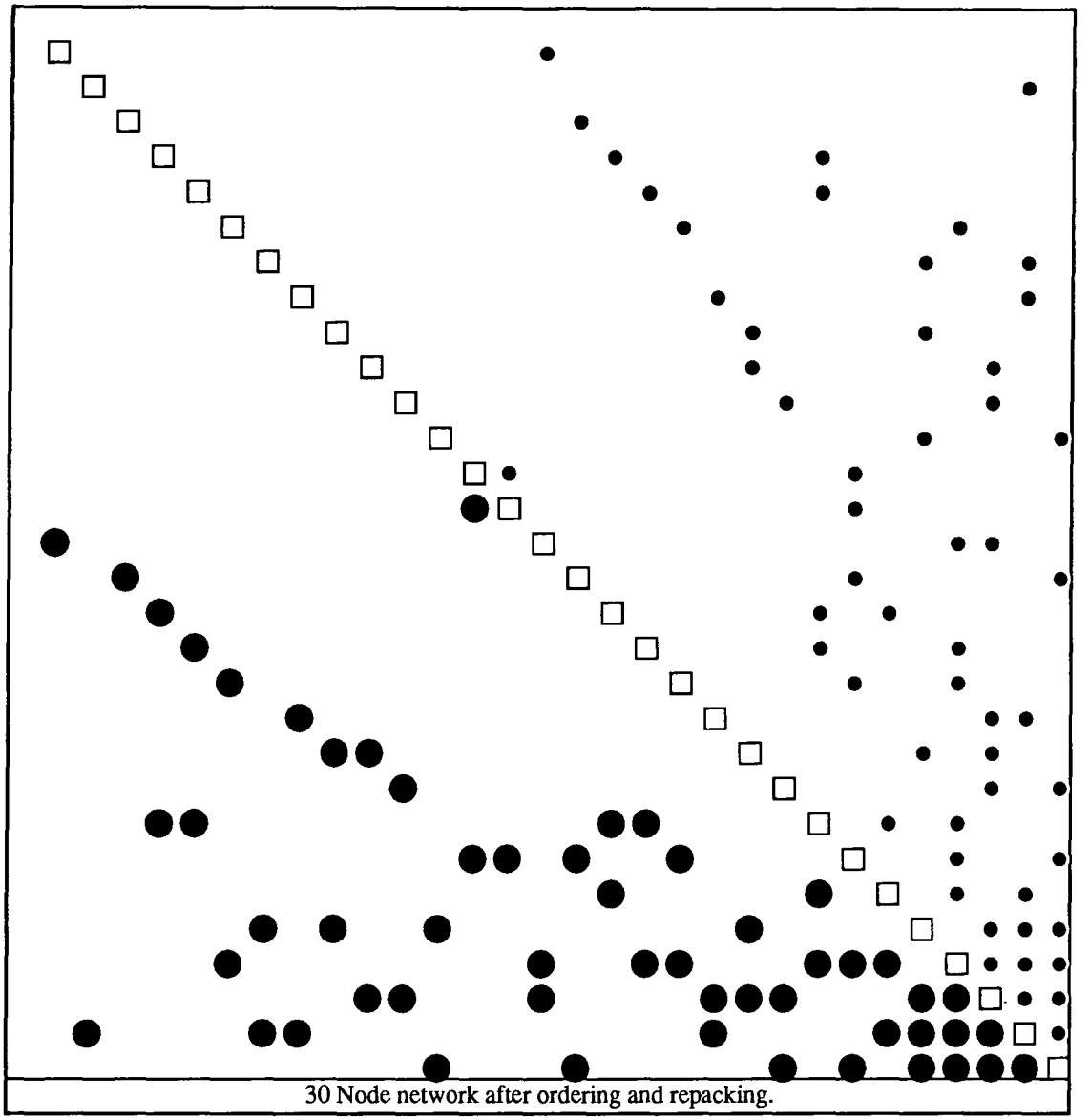
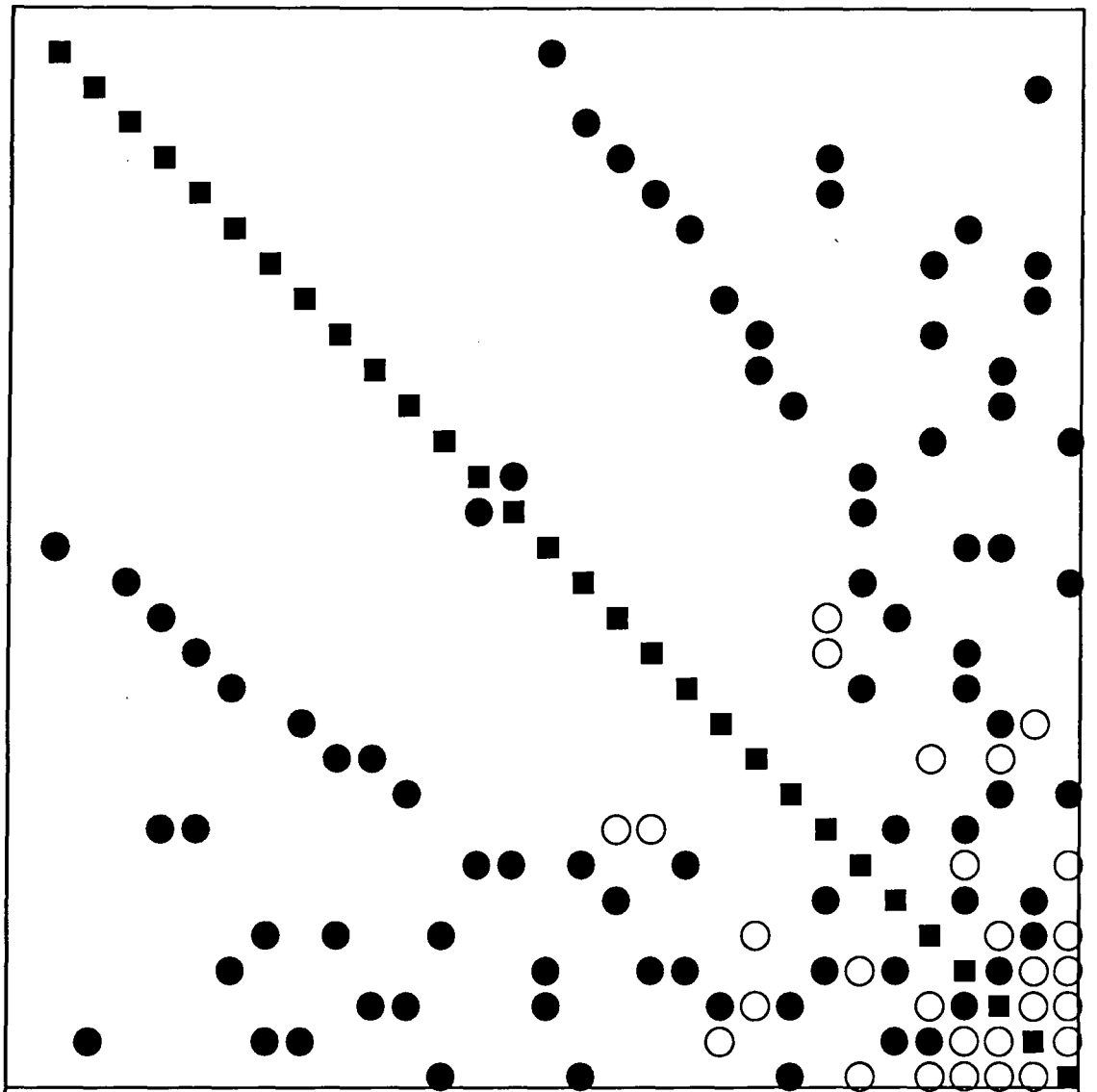


Figure 3.6 Ordered and packed I.E.E.E. 30 node network.




```
Elements before : 112
Elements after  :  85
Fill            :  14
Percentage Fill :  34.1%
```

Figure 3.7 I.E.E.E. 30 node network showing fill–in.

The forward substitution process has similar data dependencies, and by positioning all the terms first in the matrix, all network variables are solved before any generator values, so processing is again separate.

It would be possible to generate eliminated matrix values directly form the generator equations, instead of generating matrix terms, which are then inverted, if the ordering of the terms is known. Because each generator is independent, it is the internal ordering of the equations of each generator which is important, and not whether the equations for different generators are interleaved. It is simpler to conceptualise if each generator is represented by a block of equations, but it might be more efficient to process all similar equations of similar generators together.

Using two–by–two submatrices is not efficient for the generators, since their equations do not naturally split into convenient two–by–two blocks, and indeed, their structure is not well suited to the Zollenkopf method, since they do not have location symmetry. Dummy elements have to be added to make them incidence symmetric, and still more elements to make them incidence symmetric at the sub–matrix level are required. For a given structure, it would be possible to calculate which operations are actually required, and produce an optimal code to perform just these operations. For a general purpose simulator, it should be possible to add new generators without this optimisation, but once a generator type becomes fixed, it should be optimised for efficiency. It would also be possible to produce the code automatically, but off–line by symbolic manipulation in the simulator generator.

Since all internal references within a generator block are fixed, most of the code can use a fixed set of pointers and indeces. The only addresses which are unknown are those of the node to which the generator is attached, and the address of the location in the solution vector corresponding to the node. These can be supplied before the elimination is performed.

3.11.1 Parallel Generator Blocks.

The same argument holds for multiple blocks as for a single matrix. If the generator only connects to one node, then its position is unimportant, provided that all its equations come before the node in the matrix. Each small block can therefore be made up of a block of generator equations, followed by a block of network equations.

3.12 Code generation Results.

3.12.1 Size of address lists for pseudo–code.

Distribution of sub–matrix sizes for 734 node network.										
M	1	2	3	4	5	6	7	8	9	10
N	1	248	264	128	49	15	13	5	8	3

Table 3.12 Distribution of sub–matrix sizes for 734 node network.

The number of addresses required to define the bifactorisation for the floating–point processor depend on how much address calculation can be performed locally to the processor and on the connectivity and size of the matrix in question. The distribution of post–ordering column degrees is shown in table 3.12 for the 734 node transmission network. Four possible schemes are investigated, ranging from the briefest to one which requires no calculation to form the sub–matrix addresses. It is assumed that the processor posesses logic to retain the operand address where a write–back is required. Figures 3.8–11 show addresses that are required by the four schemes, with *a* representing a general element, *d* a diagonal element, *c* an element in the principal column, and *r* an element in the principal row.

This scheme is called *scheme a*, and the addresses required are shown in figure 3.8. Note that since only two input values are required for each sub–matrix operation, two memory channels are sufficient to define the operations. *Scheme b* requires the same data as *scheme a*, except that the addresses of the elements below the leading diagonal are ommitted, since they can be formed without arithmetic from the locations of the elements above the diagonal. The above–diagonal addresses are guarenteed to preceed the below–diagonal reference, but the order is different, which requires considerable storage area and non–sequential accesses. This is not difficult to arrange, but would require the addition of another memory device. It would be possible to under–utilise a large memory device by storing the addresses in triangular form, and then recalling them, but this would require a considerable number of bits in the microcode instruction word, and the trade–off might not be worthwhile. Only one access, either read or write, is required per sub–matrix operation, so the device can be quite slow to save cost.

$$\begin{pmatrix} d & r,d & r,d & r,d \\ & c,d & c,a & c,a \\ & c,a & c,d & c,a \\ & c,a & c,a & c,d \end{pmatrix}$$

Figure 3.8 Addresses for scheme a.

$$\text{Total}_a = \sum_{i=1}^{na} \left(1 + 2\,(m-1) + 2\,(m-1)^2\right) \tag{3.26}$$

$$= \sum_{i=1}^{na} (1 + 2m\,(m-1)) \tag{3.27}$$

$$\begin{pmatrix} d & r,d & r,d & r,d \\ & c,d & c,a & c,a \\ & & c,d & c,a \\ & & & c,d \end{pmatrix}$$

Figure 3.9 Addresses for scheme b.

$$\text{Total}_b = \sum_{i=1}^{na} (m\,(m+1)) \tag{3.28}$$

Scheme c assumes that the processor can retain the address of the pivotal diagonal element and the address of the first element in the pivotal row, which can be used to form the addresses of all elements in the pivotal row and column, which lie in contiguous memory locations. This requires a simple arrangement of latches and two address counters. Note that now only one new address is required per sub–matrix operation, so that only one address list is required. Scheme d is a combination of *schemes b* and *c*, so that the bare minimum of addresses are required, and in particular, that only one address list is needed.

Table 3.13 lists the number of addresses that are required for each of schemes *a–d* for the MDLRU (chapter 4) ordering of the 734 node C.E.G.B. transmission network. These totals are for the emimination phase only, and would increase when the solution phases are added, but the number of addresses for *scheme a* is not excessive, and *schemes c* and *d*

$$\begin{pmatrix} d & r & & \\ & d & a & a \\ & a & d & a \\ & a & a & d \end{pmatrix}$$

Figure 3.10 Addresses for scheme c.

$$\text{Total}_c = \sum_{i=1}^{na} \left(2 + (m-1)^2\right) \tag{3.29}$$

$$\begin{pmatrix} d & r & & \\ & d & a & a \\ & & d & a \\ & & & d \end{pmatrix}$$

Figure 3.11 Addresses for scheme d.

$$\text{Total}_d = \sum_{i=1}^{na} \left(2 + m\left(m-1\right)/2\right) \tag{3.30}$$

Addresses required to define 734 node limination.	
Scheme	Addresses
Scheme a	14170
Scheme b	10722
Scheme c	6551
Scheme d	4827

Table 3.13 Number of addresses required to define 734 node elimination.

produce very small address lists, which should not require much time for transmission to the floating–point processor.

3.12.2 Speed of pseudo–code on a VAX-8600.

Tests were performed using the Zollenkopf Bifactorisation with the standard degree based ordering to determine the effect on performance of generating code instead of direct interpretation of the matrix structures. The results are summarised in table 3.14, and are correct to the nearest 10 milliseconds, the granularity of the VAX-C process clock, averaged over several program runs. The times for ordering are given for comparison with the floating point calculation times, and were for identical orderings between the two programs with the same network.

	Code generation times in milliseconds					
	734 nodes		234 nodes		118 nodes	
	in–line	code	in–line	code	in–line	code
Ordering	160	160	90	90	70	70
Generation		80		30		20
Solution	290		110		50	
Calculation		230		90		30
First Solution	290	310	110	120	50	50
Subsequent Solutions	290	230	110	90	50	30

Table 3.14 Comparison of code–generation and in–line execution times.

The results show that generating pseudo–code and then interpreting it on the VAX-8600 shows a slight performance degredation for the first iteration of the first solution after a topology change, but that subsequent iteration steps and solutions will be faster. After a topology change, several iterations would almost certainly be required to achieve convergence, so the code–generation method would almost certainly be faster. The time taken to produce the results for n iterations is given by:

$$\text{Time} = \text{First Solution Time} + (n-1) \times \text{Subsequent Solution Tim} \quad (3.31)$$

The break–even point is where the two times are equal.

$$\text{Time for code}_n = \text{Time for in–line}_n \quad (3.32)$$

$$n = 1 + \frac{\text{First } (\text{Sol}_{code} - \text{Sol}_{line})}{\text{Subsequent } (\text{Sol}_{line} - \text{Sol}_{code})} \quad (3.33)$$

For the 734 node case, this becomes:

$$n = 1 + \frac{310 - 290}{290 - 230} \tag{3.34}$$

$$n = 1.333$$

And for the 118 node case, the break–even point is:

$$n = 1$$

The code generation technique breaks–even part way through the second iteration after a topology change for the 734 node network. The 118 node system has identical performance for the first iteration, so the gains start at the start of the second iteration. It should be remembered that these times are for interpretation on a VAX-8600, by a program which uses loops instead of unrolled code, so better performance gains would be expected if a special–purpose processor were constructed.

3.13 Conclusion.

This chapter has discussed two of the fastest methods for the solution of the sparse matrix equations which result from the analysis of electrical power systems. It has shown that, for maximum efficiency, the elimination order for the matrix need only be determined once per change of matrix structure. A conflict between a sparse storage method which would allow efficient access to elements during the processing of the matrix was found to conflict with the requirement to reorder the terms for the Crout method, so it was discarded in favour of the Zollenkopf bifactorisation approach.

Several new modifications were made to the existing Zollenkopf program in an attempt to improve its performance. The Zollenkopf program makes no attempt to repack the elements after the matrix ordering, which leaves them scattered in the arrays. This has several important consequences, namely, that valuable floating–point memory space is wasted due to empty locations, that accesses are mailny non–local, which defeats many memory enhancement schemes, and finally, that the matrix structure is not evident from the arrays.

The removal of empty locations was combined with a reordering of the columns into their elimination order, and with the placement of individual row entries into the same order. This makes the matrix structure much more readily apparent, and contains considerable implied information, which can be used to remove some of arrays which are no longer necessary, and also improves the locality of reference. The packing of elements in this

way also reveals much more structure in the floating–point calculations, which can be used to alter their local order to further improve memory access and remove pipeline stalls.

The second main investigation of this chapter was the development of real–time code generation for code dedicated to the solution of one matrix structure, so that after every structural change, new code is produced to solve that matrix. Other research has tried this, with limited success, but a new technique was developed here, whereby a special–purpose processor was proposed which implemented the required instructions for matrix solution. The use of high–level instructions greatly speeds and simplifies the generation of real–time code, since all that is required is a list of addresses and a simple instruction. This fits well with the repacking work, because the implied information can be used to reduce the number of addresses which define the operations, and also permit more regular instruction sequences to be coded into the large instructions. The hardware implementation of such a processor is discussed in chapter 6.

The pseudo–code was found to provide speed benefits when the 'program' was interpreted on the VAX-8600, over direct interpretation of the Zollenkopf arrays themselves. This is an architecture which is not well suited to the execution of pseudo–code, so the gains on a more suitable architecture should be significant.

The alterations proposed in this chapter have resulted in several important improvements in the original programs, in speed, memory efficiency and flexibility,

The experiment of re–packing and re–ordering the matrix elements after the ordering phase of the Zollenkopf program was shown to decrease the number of addresses which uniquely define the operations which must be performed by a floating–point processor, which reduces the code–generation and transmission times if real–time code generation is attempted. There are also reductions in the memory required for the storage of floating–point operands and for the storage of the indexing information required to access the packed storage, because several lists are rendered unnecessary by the improved storage scheme.

The re-ordering also allows the calculations to be performed in a more natural order, which has the effect of making pipeline stalls much more predictable, so that specific measures can be taken to minimise the performance degredation which these cause.

The calculation order is also more easily adjusted to accommodate different processor architectures which might strongly favour a particular order.

Chapter 4.

Optimal Elimination Ordering.

4.1 Introduction.

The introduction to chapter 3 showed that the sparsity of the power system matrix must be used, and that the ordering for a power system matrix only needs to be determined when the topology changes. This results from the numerical stability which is almost assured, given that the calculations during the solution and the intermediate results maintain a certain accuracy.

This property reduces the amount of computation which is required in the simulator, except when it is most desireable to decrease it. Immediately after a topology change, the matrix representing the system must be re–built, and then symbolically re–factorised, before any further processing is possible. This work comes in addition to the extra iterations of the matrix solution which would probably be required due to the change in network state.

The re–ordering process must therefore attempt to produce a near–optimal ordering to minimise fill–ins and therefore floating–point calculations, while taking as little time as possible itself. Some simulators attempt to get round this problem by adjusting the existing matrix for a few time–steps, until the solution state has steadied sufficiently that the number of iterations has fallen. This approach restricts the options in other parts of the simulator, and could cause the simulator to loose real–time processing if several changes follow each other.

4.2 Elimination Orderings.

Experience with the sparse Crout code in the previous chapter emphasized the importance of using and preserving sparsity. A maximum was derived for the number of fill–ins which the elimination of any single column could produce, which was shown to depend on the number of elements in the column, or the *degree* of the column. Ordering methods which are successful with other sparse systems perform poorly with power system sparsity [93], and early work showed that the relatively random sparsity could best be handled by explicit minimisation of fill–ins on a column–by–column basis [27 102 136].

4.2.1 Tinney's Orderings.

Tinney and Walker [147] proposed a series of methods for minimising the number of fills, which vary in effectiveness and complexity. For each method, it is possible to construct a matrix for which they will produce poor results, but results for typical power systems are reasonably consistent.

4.2.2 Tinney's first ordering method.

Tinney–1 is the simplest method to implement, but generally provides the least satisfactory solution. The method performs a single ordering before any elimination steps are simulated, based on the number of non–zero elements initially in each column. Those columns with the fewest non–zeros are eliminated first, with arbitrary selection from amongst columns which have the same number of initial elements. While the ordering phase is fast, the time saved during the ordering will be wasted during the actual calculations due to the poorer fill performance. This method clearly ignores the fill effects of previously eliminated columns.

4.2.3 Tinney's second ordering method.

Tinney–2 has proved to be the best compromise of any of his methods between ordering speed and minimising fill–in. The order in which columns are eliminated is again determined by counts of non–zero elements in the matrix columns, but the selection is performed immediately before each column is eliminated, and after the element counts

have been updated after the elimination of the previous column. This method clearly involves more work in the ordering phase, but produces far fewer fill–ins, and therefore saves time during subsequent calculations. The choice between columns with equal numbers of elements is again arbitrary. The elimination is usually implemented in two stages, the first of which is called simulation, and the second elimination. The first stage performs no floating–point calculation, and just generates fill–in locations and fixes the ordering, for the calculations which take place during the second stage.

4.2.4 Tinney's third ordering method.

Tinney–3 selects a node from those which would introduce the minimum immediate fill. This clearly involves trial simulations for several, if not all columns, which greatly increases the ordering time. This could clearly be developed further to multiple layers of look–ahead, with the global optimum being reached when the scans, to select the first node to be eliminated, completely eliminate the whole matrix. This method has factorial complexity which is clearly beyond the capabilities of present computer technology. The column selected by Tinney–3 with single column look–ahead, will generally be amongst those with a minimum number of non–zeros, i.e. will be one of the columns from which Tinney–2 makes its final selection. Tinney–2 can therefore closely approach the performance of Tinney–3, but a good supplementary selection criterion is required to choose the correct entry from amongst those with a tied number of entries.

4.2.5 Comparison of Tinney's methods.

Tinney does not specify which of the columns should be selected in the event of a tie of the number of non–zeros between two or more columns, and various schemes have been proposed to select the best without incurring much additional work. These schemes take the form of sub–criteria, as the main criterion of minimum degree still selects first. The sub–criteria in use range from the simple, such as select the first or last column found, or even random selection, to complex, such as selecting a column which has the minimum number of already eliminated columns referencing it. Two new sub–criteria are proposed later in the thesis, and take much longer to achieve this poorer result.

There are likely to be many such ties during an elimination, because with most orderings, the degree of any column would probably be limited to about 20. For the 734 node CEGB system, the maximum degree produced by the Tinney–2 method at any time during the ordering was 16, and the maximum degree of an eliminated column was 11. With 734 columns, about 630 would probably have a degree of four or less, and 720 or more less than eight. The correct sub–criterion is therefore a very important choice, since it may be invoked on 80% or more of ordering decisions.

4.3 The ordering used by the Zollenkopf Method.

Chapter 3 stated that the Zollenkopf program used an ordering based on minimum degree. The actual ordering method used is an implementation of the Tinney–2 ordering. All the columns are initially listed in their natural order (although this need not be so) in an array, which is modified during elimination to give the order in which the columns were eliminated. Each time a column must be selected, a scan is made through the uneliminated columns, to find the column with the least degree. Zollenkopf selects the first column found which posesses the minimum degree, since this is the simplest search to implement. The selected column is swapped with the column in the current position in the order array, which means that only those columns with entries after the current position in this array need to be scanned during the search. The search method is similar to a bubble sort, with a simulation of the elimination occurring between iterations of the sort. The performance is therefore of order $O(n^2)$ where n is the number of columns in the matrix.

4.3.1 Decision when equally suitable columns are found.

Although Tinney does not specify which column should be selected when several are found with the minimum degree, most algorithms choose the same one, which indicates that they all use the method of storing uneliminated columns used by the standard Zollenkopf program. For computational simplicity, the list of degrees of the uneliminated nodes is scanned linearly, searching for a lower degree than the best found so far. If the degree is lower, then this node is selected as the best, and the search continued. Selecting only nodes with a lower degree than the best currently found selects the first node that is found with the lowest degree. This is a *first amongst equals* selection

criterion. This is faster than selecting the last node with the lowest degree because fewer updates of the *best* node are made. The choice of which tied node should be chosen does not affect the number of nodes which must be scanned, because in both cases, the degree of every uneliminated node must be inspected.

4.3.2 Implications of the Zollenkopf ordering array.

The Zollenkopf program initially places all nodes into an array, in the natural numbering order, but any order could be used, if each node is listed only once and there are no 'holes'. The first location in the list is made the current position. A linear scan is made through the list, starting at the current position, to find the 'best' node to eliminate next. The node in the current position and this node are then swapped, and the current position incremented, so that the next scan will start at an an unused node. The start of the list will build up as list of the nodes in the elimination order, while the uneliminated nodes will be concentrated in the latter part of the list, which aids scanning.

This scan method also introduces effects in the ordering which are difficult to describe algorithmically, since when the decision goes against the node in the current position in the list, it is moved to a pseudo–random position in the list. This affects how it will be considered in future comparisons, since if the newly selected node is near the front of the list, it would be moved to this location and stand a better chance of being selected in future scans, than if it were moved near the end of the list, if the *first amongst equals* selection method is used.

This method of selecting nodes for elimination has an important influence on how the overall topology of the network can be represented. Only the nodes listed in the ordering array are candidates for ordering, so only these node have to be defined, which means that the node numbers do not have to be contiguous. This means that after a nodality change, only directly affected nodes have to be renumbered, which speeds topology changes and keeps more information valid between topology changes. The node numbers can also be used to provide implied information.

This property is also used in the development of a parallel bifactorisation process in chapter 7, where elimination must be restricted to within geographical groups of nodes at any one time, and it would be difficult to renumber the whole network to reflect these

splits. It is much simpler to group the nodes together and limit the area of the array which is searched at any one time.

4.4 Path length.

There has been recent interest in producing orderings which can be used to efficiently solve for either a few elements in the independent vector, or for just a few non–zero terms in the dependent vector [23] [119] [146]. The number of columns in the factorisation path, or the path length, determines how efficiently these calculations can be performed. Such problems might occur in security analysis, when only a few nodes are likely to be made critical by the change. The path length is also important for methods which involve the partial refactorisation of the matrix between solutions, such as with contingency analysis, where many small changes are made to the same basic matrix.

The path of a column is the list of the columns which are modified either directly or indirectly by the diagonal element of the column in question during the elimination. The path is constructed by obtaining the first row entry in the elimination ordered list of the column in question, and then using this to select the next column, where the procedure is repeated. The list of all the columns visited in this way is the path of the column at which the process was started.

Since the path defines which entries in the elimination matrix must be changed if an entry in a column is changed, minimising the average path length without increasing the number of fill–ins would decrease the number of calculations required to adjust the matrix values.

The path length is not so well defined, with two definitions in use, unfortunately used indiscriminately by Betancourt [14]. The more obvious is simply a count of the number of nodes in the path for each node. This visualises the path tree as a rooted tree with a column at each level in every branch, until there are no more columns in that particular path. The alternative definition makes every path start at the same level, determined by the path containing the most columns, and finish at the root. The columns are pushed as far from the root as possible, so the slack in most of the paths is inserted at the join between the path and a more critical path. This method of calculation is most useful for parallel processing.

The simple seven node network used in chapter 3, and shown in figure 4.1 can be used as an example. The path shown in figure 4.2 a) shows the elimination tree as a rooted tree, while figure 4.2 b) shows the tree with all branches staring at the same level. Both trees show that the alteration of the value for node 1 does not affect the elimination of node two, but does affect the elimination of node 6, to which it is not connected.

The latter definition is of couse useful during the elimination phase itself, if the path length must be monitored, because neither the root column nor most of the other columns in the path of a column are known when the column is eliminated. The columns referenced in the column's entry list will clearly be in the path, but their positions are not known. The path from the top of the tree can be constructed by initially assigning the length of all columns to be zero, and then incrementing the length of each column selected, and setting the length of any adjacent, uneliminated columns to be the maximum of their own length and the length of the eliminated column.

Some ordering routines attempt to minimise path length directly, by keeping track of the path length of each node during elimination, while others concerned with path length, make selections on other criteria which tend to reduce path length.

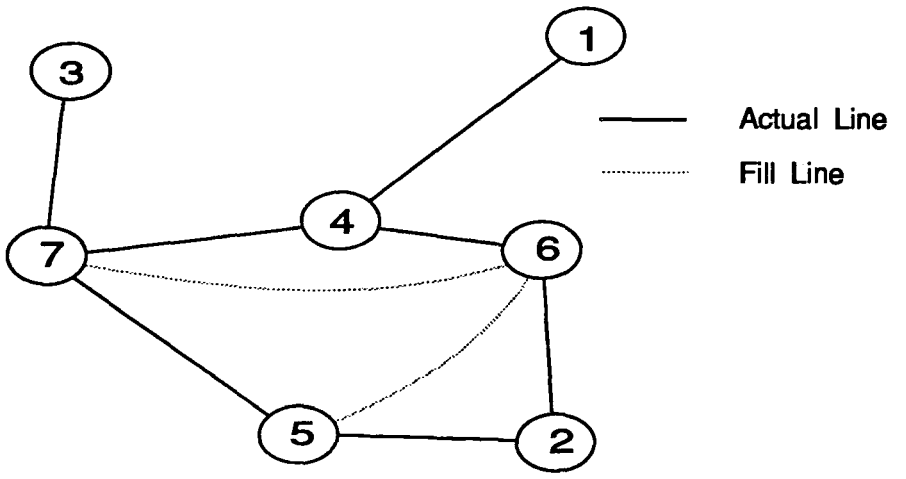
4.5 Alternative ordering criteria.

4.5.1 Gomez and Franquelo

Gomez and Franquelo [49] [50] proposed three sub–criteria for the Tinney–2 algorithm, aimed at sparse vector routines, which attempt to provide optimum solution times when only subsets of elements of the independent vector are required, or when only a few non–zeros are present in the dependent vector. These methods attempt to minimise the path length of each node. The algorithms are called either GF–x or A–x where x takes the values 1...3.

4.5.2 GF–1 ordering.

GF–1 monitors the number of eliminated nodes adjacent to each uneliminated node, and selects a node with minimum degree with the least number of adjacent previously eliminated nodes.



Small Network described by the matrix

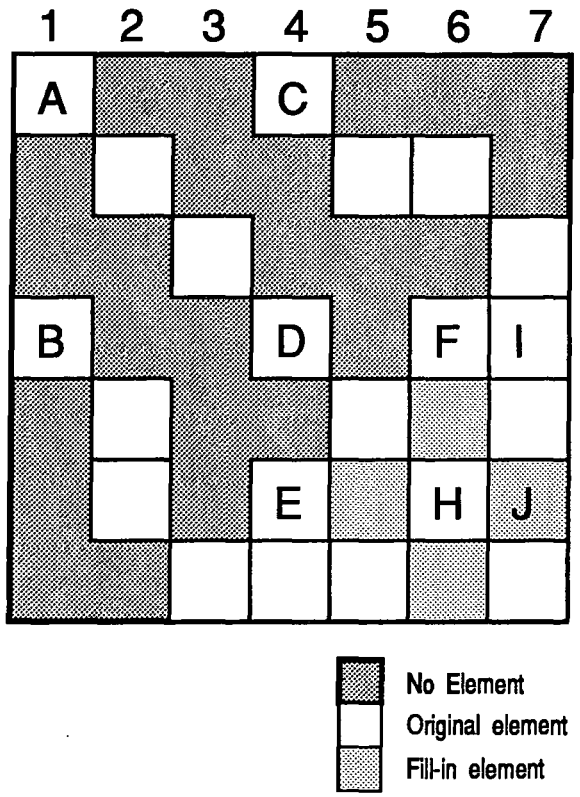
First column zeroed by subtracting a multiple of the first row from the fourth row

Multiply factor is -B/A, which is placed in the lower triangular matrix, and becomes the new value for element B

Diagonal element D is adjusted by subtracting the product of the new value for B and element C from elment D

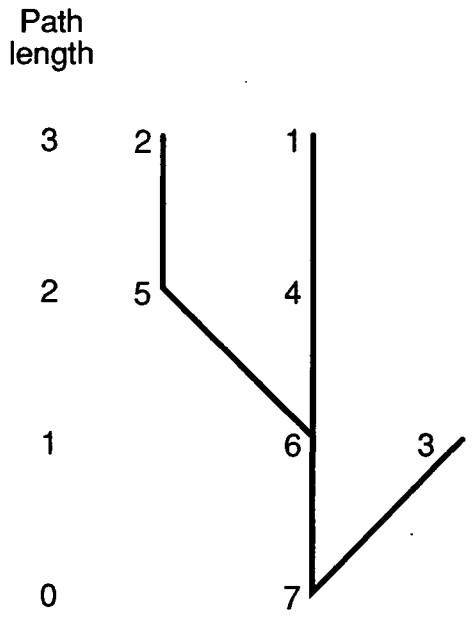
The same process to eliminate column D
$H = H - E_{new} F$

A new element J must be created at the intersection of E and I
$J = E_{new} I$

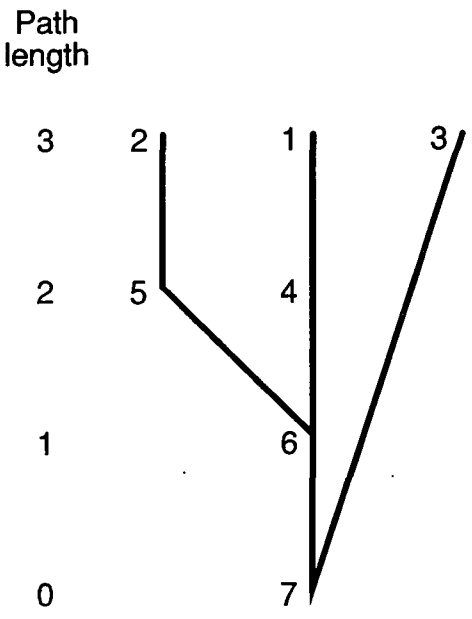


Matrix representation of the network

Figure 4.1 Example 7 node network.



A) Path from base of tree
Total path length 16.



A) Path from top of tree
Total path length 18.

Figure 4.2 Elimination paths for 7 node network.

4.5.3 GF–3 ordering.

GF–3 performs the same algorithm, but modifies the Tinney–2 slightly, so that the degree of each node is never reduced, but can increase during an ordering. This places it somewhere between Tinney–1 and Tinney–2, since Tinney–1 ignores all modifications to degree during the elimination process. GF–3 should discourage the successive elimination of electrically adjacent nodes, since the elimination of the first node of the possible pair can not make the second node more attractive by decreasing the degree which the ordering routine sees, but will actually make it less attractive by increasing its count of adjacent nodes eliminated, in addition to possible increase in degree from fill–in. Conversely, this routine should produce a decrease in sparsity since nodes with higher actual degree might be selected ahead of lower degree nodes.

4.5.4 GF–2 ordering.

GF–2 implements a least recently used subcriteria by time–stamping the adjacent nodes with the time at which the eliminated node was processed, and choosing the node with minimum degree (full Tinney–2) and the least time value.

4.5.5 Decision when equally suitable columns are found.

All these methods leave the method used to break any ties which remain to the actual implementation, in a similar manner to the basic Tinney–2 algorithm, but for computational simplicity, the same tie–breaking rules are used as the Zollenkopf program. The list of nodes is scanned, this time searching for either a lower or equal degree to the best found so far. If the degree is lower, then this node is unconditionally selected as the best, and the search continued. If the degree is the equal, then the sub–criteria is examined, and a the node with the better sub–criterion selected. If the nodes are still tied, the new node is forgotten, so that changes are only made when necessary. The methods also leave the order in which the nodes are examined undefined, and for simplicity, the same order as used by the original Tinney–2 algorithm included in the Zollenkopf program was used.

4.5.6 Minimum Depth, Minimum Length.

Two methods were proposed by Betancourt which attempt to minimise the path length directly, within the constraints of minimising fill. They both maintain and monitor the path length from the top of the tree during elimination, and make ordering decisions based on minimum length. One of the methods (Minimum–Degree, Minimum–Length (MDML)) uses the path length as a tie–breaker to the standard Tinney–2 algorithm based on degree. The other, (Minimum–length, Minimum–Degree (MLMD)) uses degree as a tie–breaker for the primary length criterion. The nomenclature therefore indicates which is the dominant criterion in deciding on the next column.

It would be expected that MDML should produce a smaller number of fill–ins and a taller tree then MLMD, and while the former was true, the additional fills generated by MLMD nullified the decrease in path length, and resulted in a much worse overall ordering. The paper concluded that MDML was the best compromise between execution speed, number of fill–ins and tree shape. The performance of MLMD was not investigated for this thesis due to poor fill performance.

Betancourt demonstrated that MDML performed significantly better than the original method (Tinney-2), producing a much flatter tree, and also reducing fill–ins. The standard deviation of these properties also decreased, showing that the results are more predictable, and that the method is less likely to suddenly produce bad ordering. The results are, however, confused by the two methods used to calculate the path length, and there would seem to be an error either in the table of results, or in the text which describes it. If the path length is calculated as described, then it is not possible to produce a mean path length which is less than half the maximum path length.

This can be seen most easily by examining the 'full' branches of the tree, which are branches that have a column at every level between the top and the root of the tree. If one of these is taken as the critical path, then this must have an mean path length of half the distance from top to root. The other full branches are calculated from where they join either the critical path, or an already calculated path. The averages of these must be either on or above the mean of the critical path. The branches which are not full, must have more columns at the top of their paths than at the bottom, and so their means must be greater than this mean value. The mean path length must therefore be equal to or greater than half the maximum path length.

It must also be noted that this method will be slower than the original method, because additional data must be maintained, in addition to further complicating the selection process. This may affect its inclusion in a real–time simulator, because if it causes the simulator to drop out of real–time during topology changes, any benefits in the actual ordering would be nullified.

4.6 New orderings.

The ordering search for the column with minimum degree was shown to be of order $O(n^2)$ as it is a bubble sort [137]. The degrees change between the selection of the minimum and the next search, so one of the more efficient sorting methods, such as Quick Sort [86], can not be used. It was found that as the matrix size was increased from the small I.E.E.E. test networks towards networks more representative of real systems, that the ordering and simulation time began to dominate the remainder of the matrix solution calculations. The elimination and factorisation appeared to be of order $O(n^{1.2})$ [40], which is based on $O(n \times m^2)$ where m represents the sparsity and connectivity of the matrix. While the initial m remains reasonably constant with increases in matrix size, an increase in fill–ins towards the end of the elimination increases the average value, which gives the slightly non–linear behaviour with respect to n.

A test was performed to investigate which of the simulation or ordering was responsible for the increase in time. The test involved timing the elimination and simulation process combined, and then removing the Tinney–2 ordering code and replacing it with the ordering list produced by a previous Tinney–2 ordering and recording the time taken to just simulate the elimination. The time spent in each part of the code could then be calculated. The ordering code was found to be responsible for the drop in performance with larger networks, with the simulation phase showing a lower speed decrease than expected. This provoked an investigation into alternative ordering methods which would eliminate the linear scans of the node degrees.

Some speed–up had already been achieved in O.C.E.P.S. by attempting to keep track of the column with minimum degree during the ordering, but at best this could only result in a 50% improvement. Once the stored value is used as the next column, a new search must be performed for the next column after that, because another column could have had its degree decreased to become equal to the previously eliminated column. If none

of the columns referenced in this column have the same degree as this, then if the search was not performed, this desireable node would be overlooked. The effectiveness of this adjustment did, however, correlate with the previous findings, and indicated that speed increases were attainable by altering the method used to identify the next column for elimination.

4.6.1 Use of a data structure to increase speed.

Due to the unsuitability of the problem to more complex search algorithms, an attempt was made to use a data structure to reduce the time spent scanning through the large array. It would clearly be desireable to store the columns so that the next to be eliminated would always be at the front of the queue, and it was thought that a data structure might be devised which would achieve this. The columns would therefore have to be stored in the order of their degrees. A storage method was required that keeps all columns of similar degree in lists or queues, and allows a column to be removed from one list and placed in another with a minimum of effort. From the discussion of lists in chapter 3, it will be seen that a doubly–linked list [85] would be the most suitable data structure for these requirements, particularly since the removal of a particular column can be achieved without any searching through any lists. Every time a column is referenced by another column as it is eliminated, it will need to be removed from a list, and inserted into another.

The necessary extra arrays of pointers or links were created, and the un–eliminated columns were placed into a series of doubly–linked lists. The link pointers were placed into two arrays which could be accessed by indexing with the column number. Each list contained columns of the same degree, so that searching for the column with the minimum degree became a simple ordered scan of the linked list headers to find the first non–empty list.

The actual elimination process needs to be examined closely to find the optimum method of using linked lists. As each column is eliminated, all its row entries effectively become lower triangular. Each of the row entries has its column examined. Each of these columns has the row entry for the currently eliminated column deleted, thereby deleting the duplicated element which is required by the Zollenkopf simulation process and decrementing that column's degree. The column is also scanned to see whether any

fill–ins have been generated, and if necessary, these are inserted into the data structure, and the column's degree is incremented.

This process means that during the elimination of one column from the matrix, the degree of other columns can decrease by at most one, but can also remain constant or increase by several due to fill–ins. This has several important implications:

Firstly, it is only necessary to scan a limited number of headers in the degree lists. The column which has most recently been selected for elimination must have had the lowest degree at that time, and so must have have been in the first non–empty list. The next column can have at most one element less, therefore the scan can start at the queue whose elements have one fewer degree than the previous column, without the danger of missing any columns. This can save a significant number of comparisons.

Secondly, it is more efficient to adjust the linked degree lists after each column has been completely eliminated, than after each element has been dealt with, because several modifications could occur to the same column while eliminating a single column.

Thirdly, only those columns which are referenced by a column can be changed, so after its elimination, that column provides a ready–formed list of columns which require updating. This removes an otherwise messy requirement of keeping track of which columns must be altered.

4.6.2 Implementation of the linked–lists.

To keep track of the queues for each number of non–zero elements in a column, two extra arrays are added in parallel to the column start array. One contains a pointer to the previous element in the queue, and the other contains a pointer to the next element. These links are only required during the simulation and ordering phase, and so can be local to that routine. This allows the memory consumed to be reused, and keeps the change to data structures local.

It would be possible to use pointers (actual virtual addresses) as links to point to the next element in the lists, which would speed accesses by eliminating index to address calculations, but the advantages of keeping the lists in human readable form during development and testing was felt to out–weigh the speed increase. Avoiding pointers also permits easier translation back into FORTRAN, so the method could be

incorporated more easily into other O.C.E.P.S. software. Avoiding pointers results in further complications, as some notation is required to distinguish between indices to header entries and actual list elements, and negative indices were used to indicate references to headers.

These header pointers are used only during the removal of an element from an unknown list, before its placement into a new list. The old list which contains the element is unknown, because the element's degree has already been updated. If the element is not at either end of the list, then its removal is simple, because its links indicate its neighbours. However, if the element is at either end of the list, the list header must be correctly identified and updated. The most efficient method is to use negative indices to indicate a reference to a header entry.

Since the only difference between a header entry and an entry for an actual element is that the header cannot be deleted or moved between lists, such a distinction is unnecessary for links implemented with pointers. Here, an empty list would be indicated by either zero pointers from the header, or by identical, self–referential pointers in the header's links. The former case would result from the header entry being specially identified during link modifications, while the second would result from the header being treated as a normal element during these modifications.

4.6.3 Insertion of column entries in the ordering lists.

A decision was required about where the altered elements should be added to their new lists. The obvious places were at either end, i.e., adjacent to the header entry. The overall performance of the method depends on this decision, as the elimination order would be drastically affected. Adding to the front of the list would place the modified element ahead of less recently modified elements in the queue for elimination. Placing the element at the back end of the list would place it behind less recently accessed elements.

The former method would tend to produce a narrow but tall elimination tree, while the latter would tend to produce a flatter, wider tree. The latter should also produce an ordering with fewer columns referencing the next column to be eliminated, which is desirable, because it reduces the number of pipeline stalls if a pipelined processor is used. Such references will still occur, but where an alternative exists, the latter method

will automatically choose it. Other methods were thought to introduce too much delay for no apparent gain in performance, since a search through the lists would be required if any other location was desired. Abe and Taoka [141] also based an elimination ordering on minimum–degree, dependancy avoidance.

Columns which experienced no change in degree despite being referenced, that is they had one fill–in which countered the deletion of the duplicate element, also presented a choice as to whether they should be moved or not. It was decided that they should be moved to make the method uniform, maintain the pipeline benefits and so that no special code was required to detect when no change had occurred.

4.6.4 Pipelined processing and data dependencies.

Chapter 3 discussed some aspects of minimising the impact of data dependencies when using pipelined processors and memory interfaces. An internal calculation order was developed for each of the eliminated columns that was regular and internally consistent. Columns were also processed in such a way so that the columns which would be eliminated the soonest would be processed as soon as possible, thereby placing as many other diagonal calculations between the modification and useage as possible. It is assumed that the matrix inversion will flush the processor and memory queue of any pending results, so that only the diagonal elements really need to be considered. Avoiding pipeline problems should not be allowed to degrade performance in other areas, such as register optimisation within a block, since stalls would be expected to be reasonably rare.

If the operations are ordered correctly within the blocks, then only successive elimination where the first column is of degree two should be problematical, because otherwise there should be enough operations between alteration and useage of an element that it can safely be written to memory and re–read. If a column of degree two is processed first, the final element to be processed is the first that is required for the processing of the next column. This can most safely be handled by delaying the processing of the next column, but correct behaviour could also be achieved by reading the required data directly from the register file. This is problematical because the data might be in the wrong registers, so that register operations would clash with other building blocks. A separate instruction or special logic would be required to ensure correct operation. A

column of degree three or more being processed first would update the first element of the following column, followed by one other in either the leading row or column of the next column, followed by three other submatrices, only the second of which affects the leading row or column of the next column. This places at least two submatrices between alteration and useage, which is safe.

The ordering routine should therefore try to minimise the successive elimination of columns when the first is of degree two. If there is only one island in the system, then there must be at least one such elimination, concerning the final two columns in the elimination matrix. More islands allows the endings to be interwoven, but in practice this will not be the case, since one island will be fully eliminated before the full blocks at the ends of other islands would be processed, because once these blocks become full, the smaller block has nodes of lower degree, so these will be preferentially eliminated. There will, therefore be at least one such elimination per island.

These eliminations could also occur when a line of doubly–connected nodes is being broken, or when a node of degree two connects to another node, which, due to the elimination of the second degree node, becomes the node of lowest degree. While such eliminations cannot completely be removed while keeping minimal fill as the primary criterion, they can be reduced by eliminating nodes according to how long they have been left since their last modification, if several are found to have the same degree.

4.7 Definition of the new ordering method.

A new ordering method for ordering the elimination of sparse matrices has been proposed, which uses implicit information contained in a data structure to speed the selection of the next node for eliminiation. Other methods rely on linear searches through an array containing all the uneliminated nodes left in the network, comparing the principal criterion of each against the best found until then. Other sub–criteria are then compared until a difference is found. This process dominates the remainder of the operations for large matrices.

The new method stores the uneliminated nodes in lists, a seperate list being used for each value of the primary criterion, here the degree of the node. This removes the long searches and replaces them with more complex operations of list management, but far fewer of these operations are required than comparisons in the original methods. All

that is required to find the next node to be eliminated is a search through the lists to find the first, non–empty, list, and this search has been optimised so that no unnecessary comparisons are made.

The lists are made doubly–linked so that nodes can quickly be moved from their previous list without any searching, when their degree is changed, and inserted at the end of the new list. The elimination process was examined, and rules were developed which minimise the number of movements and optimise the use of the lists. No movement of node entries is involved, so that the list entry for a particular node is available simply by indexing with its node number, and the double linking identifies its neighbours without searching.

Although the method still selects primarily on minimum degree to maintain sparsity, a second implied criterion is added, of least recently modified, since nodes are taken from the front of the list of minimum degree for elimination, and nodes affected by the elimination of another node are placed at the back of the queue for their new value of degree. The performance of the new method should be fully investigated and compared to the performance of the ordering methods derived by other workers in the field of power system analysis.

4.7.1 Implications of the new ordering method.

The changes from the existing Zollenkopf method were restricted to the ordering phase of the simulation process, with no externally visible alterations, except for a different elimination order. The correctness of the remaining modules was therefore unaffected.

The ordering algorithm can be seen to implement a minimum degree, least recently used (MDLRU) selection criterion. This extra criterion would be expected to alter the existing ordering, but its precise effects were not clear. One of the effects was discussed during the reasoning behind the decision to place the altered elements at the rear of their queues. This was to reduce the number of successive eliminations to the minimum possible, consistent with the limited look–ahead of the method and keeping the number of fill–ins small. The effect of the method on the number of fill–ins was not certain.

4.7.2 Effect of the new ordering on Path Length.

The Minimum Depth, Least Recently Used (MDLRU) method, while attempting to avoid successive eliminations, produces an elimination tree which tends to be fat and hence flat. The method attempts to eliminate as many nodes as possible, given the precedence of the degree criterion, between the reference to a node by a previously eliminated node and its own elimination. It would appear that this method should also reduce path lengths in the elimination tree, as a by–product of the increase in ordering speed. Preliminary tests indicated that not only did this method reduce path lengths, but tended to produce smaller average paths with less variance than the MDML algorithm itself. The MDLRU method also does not require the additional book–keeping of the MDML method, so the speed–up should be even more marked than when compared to the Tinney–2 ordering.

4.8 Combination of ordering methods.

It was also noted that the least recently used criterion did not conflict with the minimum length criterion, so they were combined to produce a Minimum Depth, Least Recently Used, Minimum Length (MDLRUML) ordering. Due to the slightly different nature of the criteria, it was not possible to try them both ways round, because the lists are formed in chronological order. The selection process was therefore minimum degree, with ties in path length being broken by the first occurrence of the minimum length in the first non–empty time–ordered queue.

Since the decision process is affected by the path length of each node, the overhead of maintaining the path lengths for each node remains from MDML, but savings can be made in the decision phase by observing how path lengths are related to the past history of the node and its position in the time–ordered degree lists.

All columns with a length of zero must come before all other columns in that list. This is because all columns are added to the back end of lists when referenced, and for a column to have a length of zero, it must never have been referenced, and must therefore have been in the list from the start. If the column has a length of zero in the list, no further scanning is necessary. This also implies that if a length of one is found, no further searching is required, because no shorter lengths will be later in the list.

Because these rules eliminate much of the searching, no further trials were performed to test whether performance could be improved by adding extra links to the queues based on length, which would effectively create a matrix–like structure of header entries.

These special properties are not applicable to the MDML algorithm as originally proposed as a modification of Tinney–2, because of the method used by the Tinney–2 algorithm to keep track of its uneliminated columns. Each time a column is selected, every remaining column must be examined, because even if a length of zero is found, it is possible that a node remains later in the list with a lower degree. The worst that can happen with the MDLRUML search is that a whole list of constant degree must be searched. The list containing the lowest degree nodes should contain relatively few nodes compared to the total which the MDML routine must scan, especially when all columns of length zero have been eliminated.

4.9 Other variations on Least Recently Used ordering.

Two variations on the new ordering method were tested. The first placed elements onto the front of the queues, thereby making the ordering a Minimum–Depth, Most Recently Used, or Least Recently Used Reversed, (MDLRUR). This ordering should produce long elimination paths, and narrow and tall elimination trees. There should also be a large number of potential pipeline stalls and clashes. The reversed method requires no additional computation over the plain method, since elements are added to the queues in the order that they occur in the linked column lists, but are added to the other end of the queue.

The second modification was to remove some of the well defined structure of the ordered matrix by reversing the order of addition to the end of the queue. This modification breaks up some of the inevitability of the plain ordering, because the nodes are initially added to the columns in their numbered order. The plain method produces an ordering in which very few nodes are initially out of their numbered sequence. Many nodes are missed out, but few that are listed at first are out of position. All the nodes which are adjacent to these nodes which possess the same nodality will be processed next, and there will be a similar ordering of these nodes, which will be passed down to the next level of lists. Once low degree nodes have been ordered, the unmodified higher degree nodes will be used next, and will still be in their original order.

If the order of addition to the queues is reversed, this breaks up the sequence a little, without altering the least recently used part of the algorithm. The average results should not differ greatly from the plain method, but some orderings may be very different. The method requires a little extra computation, as the singly linked list must be traversed in the opposite direction to its links. The easiest method to achieve this is to traverse the list, copying the node numbers found into a standard array, and then running through this in the opposite direction. The low degree of the nodes ensures that this array is not large. This modification was made to both the MDLRU and MDLRUML orderings to produce MDLRUA and MDLRUMLA orderings respectively.

4.10 Comparison of ordering times.

Extensive tests were performed on the new ordering method with several of the standard O.C.E.P.S. test networks. These included the I.E.E.E. 30 node and 118 node networks, a 734 node representation of the CEGB's transmission network, and a 234 node simplification of the CEGB transmission system. The CEGB networks exhibit a higher connectivity with more meshing than the American networks, and are therefore much more difficult to order, as fill–in is higher. The speed of the ordering and simulation of the elimination phase is not only affected by the computational complexity of the decision phase, but also by the amount of fill–in, since fills slow the simulation part of the combined process. The MDLRU based methods are additionally slowed by increases in fill–in, because this increases the number of columns which must be moved between lists after the elimination of each column, but this effect would appear to be small. Generally the results for the 30 node network are not reported because there was too little to distinguish between the results for the different orderings.

All the orderings were combined in the same subroutine with the simulation code, and selected every loop with a case statement. This slightly slows the ordering because of the conditional branch, and the presence of other ordering blocks using different variables will also affect the optimisations performed by the compiler. Unless specifically stated otherwise, the queue based orderings are grouped together in the discussions, and specifically the reversed order of additions are grouped with the normal orders (i.e., MDLRUA is grouped with MDLRU).

Several tests were performed to assess the speed of the orderings. The simplest series of tests involved the repeated manual invocation of the program, and the recording of the times displayed, which were then averaged. These times include the creation and management of virtual memory pages, some of which might not be used for the particular ordering, due to the code having to cope with the eventuality of larger networks and the provision of slack space for poorer orderings.

The test was then re–run with a minimal ordering routine using the MDLRU ordering read from a file (outside the timed loop), to determine the time spent in simulating the elimination process. This time could then be subtracted from the combined time for ordering and simulation, to give the time taken to determine the order of elimination. These times are again affected by the management of virtual memory, but give a clearer indication of the speed of the actual ordering part of the process. The simulation time was assumed to be independent of the ordering method used, and since the number of fills was reasonably independent of ordering, this should be a reasonable assumption. Only Gomez–Franquelo–A3 generated a significant excess of fills, which would tend to degrade its result when calculated in this manner.

4.10.1 Timing results for single orderings.

The results presented in table 4.1 show that all the methods except Tinney–3 are comparable in speed for the 118 node network. The improvement in the combined time for ordering and simulation on the 234 note matrix show that the linked list based methods are about 40% faster than the methods which use a linear search. The difference is much more marked once the simulation time has been subtracted, since the simulation time is almost independent of the ordering method. The linked list based methods are about four times faster than the search based methods for the 234 node system, with the times showing little scope for further improvement.

The results for the 734 node system were much more marked, with the queue based methods providing almost a four–fold speed–up over Tinney–2, and a larger improvement compared to the more complex ordering methods. When the simulation time is removed, the linked list orderings can be seen to be over ten times faster than Tinney–2, and fifteen times faster than MDML. The MDLRUML combination of MDLRU and MDML is over twelve times faster than MDML alone. The degradation imposed by using the length

	Ordering times in milliseconds					
	734 nodes		234 nodes		118 nodes	
	All	Order	All	Order	All	Order
Tinney–2	540	440	110	40	60	10
Tinney–3	13770	13670	1710	1650	420	370
MDML	710	610	130	60	70	20
GF–1	680	580	120	50	70	20
GF–2	680	580	130	60	60	10
GF–3	690	590	130	60	70	20
MDLRU	140	40	70	10	60	10
MDLRUR	140	40	70	10	60	10
MDLRUML	150	50	80	20	60	10
MDLRUA	140	40	80	20	60	10
MDLRUMLA	160	60	80	20	60	10
Simulation alone	100		60		50	

Table 4.1 Comparison of ordering times and with and without simulation.

sub–criterion on the MDLRU algorithm can be seen to be negligable, which backs–up the premise that the special cases identified and the relatively short lists should save a significant amount of time. These times also show that there is little further time to be saved by altering the ordering routine from one based on MDLRU implemented using doubly–linked lists, even on these large networks.

More important than the speed–up factor is the actual time that is saved by using a queue based ordering routine instead of a standard Tinney–2 routine. 400ms are saved for the 734 node system, which is four tenths of the simulation time–step saved. This increases the time available for the remainder of the simulator to produce a result inside the first time–step after a connectivity change from 460ms to 860ms, or a little less than the full second normally permitted when no topology changes have occurred. The quadratic convergence of the Newton Method ensures that each solution will be much better than the previous one, so this time is valuable in at least getting a good approximate solution, even if absolute accuracy is not achieved in the first time–step after a topology change.

The times for Tinney–3 are included for comparison, primarily because the orderings were required for comparison on other criteria. This ordering is clearly far too slow to be used, and it is little more optimal than many of the other methods used, which can be over three hundred times faster.

The test also shows that the simulation time does not increase rapidly with network size. There should be at least a linear dependency between the simulation time and network size, with a dependency of order $O(n^{1.2})$ expected. The reason for the departure from the expected behaviour could be that the time for simulation also includes the fixed overhead of the routine, which accounts for about 40 milliseconds. The subtraction of this time gives the dependency expected.

4.10.2 Timing results for repeated orderings.

	Ordering times in milliseconds Averages of 10,000 orderings.		
	734 nodes	234 nodes	118 nodes
Tinney–2	384	66	29
Tinney–3	6940	951	218
MDML	396	70	30
GF–1	410	70	30
GF–2	421	74	34
GF–3	455	72	30
MDLRU	103	42	24
MDLRUR	102	41	27
MDLRUML	107	44	25
MDLRUA	106	43	25
MDLRUMLA	112	44	25

Table 4.2 Comparison of ordering times for repeated runs.

The times for 10,000 orderings presented in table 4.2 were taken from repeated orderings of random initial node numberings of the fully connected networks. These runs were used primarily to assess the fill–in and path length performance of the ordering algorithms, and some of the monitoring instructions affected the timings. Specifically,

the path length calculations were inserted into the simulations of the elimination for all methods, as some methods require them, so the times for these methods would be expected to benefit compared to the other methods which would have their times inflated. The large number of orderings should make the effects of the initial demand on virtual memory negligable, and this was found to be the case, as paging completely ceased after the first few orderings. These times are more representative of the performance of the orderings than the one–off tests, but the length of some of the runs will be seen to be prohibitive. Each Tinney–3 ordering took about seven seconds for the large network, and the whole test about twenty hours of CPU time on a 8 MIPS VAX.

The times for repeated runs presented in table 4.2 show that the queue orderings are faster than the linear search based methods, but not by as much as for the single runs. The queue orderings are still faster by a considerable amount, and it is clearly still worthwhile to use them on a time basis. Even on the faster VAX-6440, the MDML method saves 250ms over Tinney–2, and more compared to MDML for the 734 node network. This saving is a quarter of the simulation time–step.

4.10.3 Differences between single and repeated runs.

The times for the single orderings are not directy comparable with those of the repeated orderings, since they were taken on different machines. The single orderings were made interactively on a VAX-8600 which executes at about 4.2 MIPS, while those for the repeated runs were made on one processor of a VAX-6440 in batch mode, which executes at about 8 MIPS, to save time. The VAX-6440 was unloaded apart from three similar batch files (one per processor), which were memory and processor intensive, and there was no page–faulting after the initial program load. The VAX-8600 was in use by other users, and page faults occurred during the runs. The times executed on one machine should be comparable with other times from that machine, and the overall rankings should also be consistent.

	Effect of ordering routine on the successive elimination of connected nodes					
	734 nodes		234 nodes		118 nodes	
	Successive	Stalls	Successive	Stalls	Successive	Stalls
Tinney–2	120	12.2	50	2.16	27	2.47
Tinney–3	148	15.5	65	3.19	40	2.12
MDML	74	1.5	29	1.08	16	1.18
GF–1	83	2.1	33	1.07	16	1.16
GF–2	69	1.5	24	1.00	11	1.00
GF–3	18	1.0	14	1.00	8	1.00
MDLRU	67	1.5	24	1.00	10	1.18
MDLRUR	230	23.0	103	3.34	59	4.60
MDLRUML	72	1.5	27	1.00	14	1.00
MDLRUA	75	1.5	24	1.00	11	1.00
MDLRUMLA	72	1.5	26	1.00	14	1.00

Table 4.3 Successive Eliminations and Stalls for 10,000 orderings.

4.11 Effect of orderings on pipeline stalls.

The number of electrically adjacent successive eliminations was also monitored during these tests, as it is these which could cause pipeline stalls in a pipelined floating point processor. One aim of introducing the MDLRU ordering was to reduce this successive elimination. The most damaging case is when the first node is of low degree, so that there are few elements to be processed between the update of a matrix element belonging to the following node, and when it is required for processing the following node. Every ordering must have at least one successive elimination after a node of degree 2, which will occur with the last two nodes of a fully connected network. It will also occur with the last couple of nodes for every electrical island, when the network becomes split. Successive eliminations of higher degree are less important, as there are more intervening calculations which can flush the pipeline. Two figures are given for each combination of network and ordering routine; the total number of clashes, and the number of clashes where the first node is of degree 2.

The results of table 4.3 show that the *forward* MDLRU orderings do tend to decrease the number of clashes. Gomez–Franquelo–2 is included in this category, since it implements selection based on time–stamping. The *reverse* MDLRUR method increases the number of clashes by eliminating an adjacent node first if it has an acceptable degree, which verifies the decision to place altered nodes at the back of their queues.

The Gomez–Franquelo–3 performs exceedingly well, ignoring most of the possible successive eliminations, but its performance on other criteria make it less attractive. The bare Tinney algorithms both perform poorly, and it seems that a secondry criterion is especially important in reducing the number of clashes produced by an ordering.

4.12 Comparison of operation counts.

	Effect of ordering routine on Operation Counts and Fill								
	734 nodes			234 nodes			118 nodes		
	Ops	sd	Fill	Ops	sd	Fill	Ops	sd	Fill
Tinney–2	6457	138	634	2817	88	298	917	4.8	85
Tinney–3	6059	39	597	2604	9	279	913	3.7	84
MDML	6423	112	631	2845	92	300	918	2.9	85
GF–1	6450	168	632	2898	101	305	920	2.8	86
GF–2	6523	96	641	2860	34	303	919	3.0	85
GF–3	9110	354	840	3725	107	376	1117	54.5	108
MDLRU	6720	51	654	2867	43	304	920	2.8	86
MDLRUR	6352	91	626	2747	23	293	920	4.6	86
MDLRUML	6459	97	634	2958	26	310	920	2.8	86
MDLRUA	6435	45	635	2839	19	301	918	2.8	85
MDLRUMLA	6368	50	628	2915	26	307	918	2.8	85

Table 4.4 Comparison of Operation Counts and Fills for 10,000 orderings.

The ordering methods were applied to the 118, 234 and 734 node test networks used previously, in addition to the 30 node I.E.E.E. network. The results for this network are not reported as fully as for the other networks, because it produced negligable differences in performance between the methods, The operation counts which each method required

to eliminate and solve the various test networks are presented in table 4.4. Only GF–3 produced any variations in the number of operations over 1000 ordering attempts, and all the other methods produced the same number of operations. The orderings were different, which is demonstrated by variations in path length, both between methods and between individual orderings.

The results show that Tinney–3 generally produces the lowest number of operations and the lowest number of fills, and GF–3 produces the most of both by a considerable margin. These two extremes apart, all the other methods produce results within 5% of Tinney–2. The MDLRU based methods seem slightly inferior to the linear search methods. The standard deviation of the operation counts have also been included in the tables, as the variability of the number of operations must also be considered, since the worst case depends on both the mean and this figure. Again, Tinney–3 produces the least variability, and GF–3 the most, but now the MDLRU based methods have less variance, which is to be expected because their selection method is more rigidly defined than the linear search methods with their element swapping.

4.13 Structure of the eliminated matrices.

Diagrams of the structure of the eliminated matrices for all the ordering strategies discussed are presented in appendix A. To aid comprehension of these diagrams, both upper and lower triangular parts are shown, while in reality only one or the other is represented inside the computer after the elimination has been simulated. The columns are shown in the elimination order, so the first column eliminated is at the left, and the last at the right. For the smaller networks, fill–in and original elements are shown differently, with the fill–ins being hollow, and the original elements solid. The large number of columns makes this impractical for the larger networks. Here, seperate plots are provided for the fill–ins, in addition to the plots showing all the matrix elements present after elimination.

The results, shown beneath the plots, are different from those presented in the tables, since they only reflect the results of one particular ordering instead of an average of 10,000 orderings with the nodes in random initial orders. They are included, so that the patterns in the plots, can be compared to numerical measures of the ordering performance. The first set of results show the times taken by each part of the Zollenkopf

program, the ordering and simulation times are combined, followed by the repacking, elimination and substitution times. These last two times are for the generation of address lists and operation counts only, since no numerical calculations were performed. This is indicated by the last time, that for calculation, being zero. Then follows the totals of the above times. The next line of four numbers which show how many elements the matrix contained before elimination, after eliminetion, the number of fill–in elements, and finally the percentage of fill which took place. The number of elements would normally decrease during the elimination because of the deletion of the mirror–image elements. The fill–in percentage is calculated as the increase in the number of off–diagonal elements in the lower triangular part of the matrix during elimination.

The first column of the next group of results show how many submatrix operations would be required to perform the elimination, and the second column gives the totals for the forward and backward substitutions. The totals for addition and multiplication are given seperately, and combined as multiply–additions. Each sub–matrix operation would require eight multiply–additions (eight multiplies and eight additions), although the modification of the sub–matrices in the principal rows would require four fewer additions. The substitution phase uses two–by–one vectors, so the counts are reduced to four multiplies and four additions per sub–matrix. The final line shows how many columns directly reference the following column, and the second number shows how many of these first columns have only one off–diagonal term, and so would be the most troublesome columns to process with a pipelined processor.

The distribution of elements is remarkably consistent between the different matrices for the same ordering methods. The orderings which select on either time or to a lesser extent history of adjacent nodes, have several bands of non–zero elements converging on the leading diagonal. These are shown most clearly in the MDLRU ordering, which is very similar to that produced by GF–2. These plots are markedly different to those produced by the Tinney–3 ordering, which appears to prefer adjacent off–diagonal terms to be horizontally or vertically adjacent, instead of diagonally with the MDLRU based methods. The MDLRUR method produces more horizontally adjacent off–diagonal terms, and generally performed well on the number of operations and fill counts. It would seem that diagonally adjacent non–zero terms increase fill–in and operation counts, while a horizontal or vertical arrangement of adjacent non–zero terms reduces fill–in. This also appears to result in an increase in pipeline stalls and path length.

The plots showing just the fill elements and the leading diagonal highlight the type of fill which each ordering generates. It should be noted that the orderings which give preference to nodes whose neighbours have not been eliminated, tend to push fills to the end of the matrix by preferentially choosing unaltered nodes ahead of altered nodes. Most of the methods tend to push filled columns in this direction, because a fill would increase the degree of the column, and make it less attractive, but this influence is not as strong as, for example, time–stamping. MDLRUR is the exception, since it actively encourages the selection of altered nodes.

A diagram showing the structure of the 118 node network ordered by the Tinney–1 algorithm is also included for reference. The fill–in performance is exceptionally poor, and this ordering failed on the larger networks, producing more than 30,000 elements part way through the elimination. This is clearly not a viable option despite its apparent simplicity. The increase in the simulation time produced by the increased fill–in can be seen to outweigh the time saved by the simpler ordering, even without considering the numerical calculations which follow.

4.13.1 Estimate of the number of operations.

An estimate of the total number of floating–point operations for the solution of the 734 node network can be obtained from the operation counts shown below the matrix plots. Each multiply–addition in the first column would consist of eight multiplies and eight additions, so the elimination would require over 110,000 double precision operations. Similar calculations for the substitution phase show that a further 30,000 operations are required, so a total of 140,000 operations are required to produce a solution. The aim of obtaining the solution in under 50ms would therefore require sustained double precision calculations at the rate of 3.5 MFlops or more. If it is assumed that two operands are required from memory for each calculation, and that a quarter of operations produce a result which must be written to memory, then the memory must be able to sustain over 7½ million double precision transfers per second, or, for a 32 bit bus, 15 million operand bus cycles per second.

4.14 Path length results.

Table 4.5, table 4.6 and table 4.7 give path length statistics for the various orderings from the 10,000 orderings used in the other tests. The standard deviation of mean path lengths is the standard deviation of the 10,000 mean path lengths reported. This figure shows the consistency of the average path length produced by the ordering. The mean of the standard deviations of the path length is the mean of the 10,000 standard deviations of the distribution of path lengths for each ordering attempt. This figure shows the variation between the path length of individual nodes in each ordering. A low figure is best for both results.

The path length results show that the Gomez–Franquelo, the MDML and the MDLRU methods produce very similar path lengths, which are generally better than those produced by Tinney–2 alone. Tinney–3 and the reversed ordering MDLRUR both produce longer path lengths. The exceptional performance comes form the MDLRUML combination, which produces by far the shortest paths for all three networks. This ordering produces elimination orders which have lower than average variability between the path length for each node in an ordering, and between the average path lengths of many individual orderings, so it should provide consistently good performance for different configurations of these networks. Particularly noteworthy is its performance on the 118 node network standard deviation of mean path lengths, where almost no variation on average path length was to be found over 10,000 orderings.

4.15 Path diagrams.

Path diagrams are included in appendix B for all the standard orderings for the 118 node and 234 node networks. It was not possible to produce diagrams for the 734 node network due to the size of the lettering which would have been required. The path diagrams are not as distinctive as the matrix plots, but the ordering methods which select the least recently used nodes tend to produce flatter, better balanced trees, while the other methods often produce lob–sided trees, which have some very long elimination paths. MDLRUR as expected produced very unbalanced trees, as did Tinney–3, GF–1 and GF–3. MDML also produced unbalanced trees, with some path lengths much longer than necessary, which indicates that although primarily intended to minimise path length, it does not work as well as the new methods proposed in this thesis.

	Path Length for 118 node network			
	Mean Length from base	sd of mean path lengths	mean of sds of path lengths	Mean Length from top
Tinney–2	9.9	0.63	3.4	15.2
Tinney–3	10.5	0.62	3.9	16.2
MDML	8.9	0.42	2.8	12.5
GF–1	8.5	0.21	2.8	12.4
GF–2	8.9	0.21	2.8	13.0
GF–3	9.0	0.50	2.6	11.8
MDLRU	8.7	0.22	2.7	12.6
MDLRUR	11.2	0.36	4.7	18.3
MDLRUML	8.0	0.02	2.3	10.2
MDLRUA	9.0	0.05	2.9	13.4
MDLRUMLA	8.0	0.04	2.9	10.2

Table 4.5 Comparison of Path Lengths for 118 node network.

	Path Length for 234 node network			
	Mean Length from base	sd of mean path lengths	mean of sds of path lengths	Mean Length from top
Tinney–2	16.4	0.8	5.2	23.1
Tinney–3	19.2	1.1	7.8	29.1
MDML	15.9	0.7	4.9	21.5
GF–1	15.8	0.6	4.8	22.2
GF–2	15.5	0.5	4.6	21.3
GF–3	15.5	0.8	3.6	20.1
MDLRU	15.6	0.6	4.9	22.0
MDLRUR	16.5	0.3	5.4	24.7
MDLRUML	14.6	0.2	4.2	19.5
MDLRUA	14.6	0.1	4.0	18.8
MDLRUMLA	13.6	0.3	3.2	17.0

Table 4.6 Comparison of Path Lengths for 234 node network.

	Path Length for 734 node network			
	Mean Length from base	sd of mean path lengths	mean of sds of path lengths	Mean Length from top
Tinney–2	21.4	1.2	7.2	35.3
Tinney–3	23.5	1.0	9.8	43.0
MDML	20.5	1.1	6.7	31.8
GF–1	20.6	1.1	6.8	33.1
GF–2	20.7	1.0	6.8	33.4
GF–3	20.3	1.2	4.4	26.4
MDLRU	20.2	0.9	5.7	30.3
MDLRUR	21.6	1.0	8.0	38.0
MDLRUML	17.6	1.1	4.8	25.6
MDLRUA	20.5	0.7	6.9	32.9
MDLRUMLA	18.2	0.7	5.8	29.1

Table 4.7 Comparison of Path Lengths for 734 node network.

The work in chapter 7 on solving the matrix equation in parallel also references these path plots, and trees which can be broken up into regular sized main branches, with a short common trunk are most desirable. Again, the trees from the new methods are well suited, as, surprisingly, are those from the GF–3 ordering, despite the larger number of operations than the other methods.

4.16 Overall performance of the ordering routines.

The results presented for the ordering routines have shown the performance of the algorithms on a variety of criteria, from ordering speed through to the average path length produced in the elimination matrix. Each of the ordering routines has its own particular strengths and weaknesses, and since only one can be selected for the simulator, some form of ranking must be devised.

Because the results vary greatly in size, a weighted geometric mean was used to rank the ordering methods. A weighted geometric mean is the product of the all the relevant values, each raised to a weighting power, and then rooted by the sum of the individual powers. The properties which are most important for the simulator are the number of fills generated (or number of operations, they are linked) and the time taken to order the

matrix. Other properties which should be considered are the average path length and the number of stalls and successive eliminations. The weightings reflect the importance of these, with the operation counts and the time taken being squared, the fill total, the path length and the number of stalls included with unity power, and the number of successive eliminations raised to the quarter power. The ranking is therefore of the form given by equation 4.1.

$$\text{RANK} = \sqrt[7.25]{\text{Operations}^2 \times \text{Time}^2 \times \text{Fills} \times \text{Stalls} \times \text{Path} \times \text{Successive}^{0.25}} \tag{4.1}$$

The rankings are based on the results from the other tests, where a small number indicates good performance, so again, a smaller number indicates better performance in this table. Two relative results are provided in table 4.8 for each combination of ordering routine and network, to provide a clearer indication of performance. The first gives performance relative to Tinney–2, a widely used algorithm, while the second gives performance relative to the best result for that network, which shows how much of a compromise must be made if a method other than the best is selected.

The MDLRU and MDLRUML (and their slightly altered forms) performed particularly well with this measure of performance. They were clearly the best for the larger systems, mainly because of their speed, but also with their good path length performance. The distant third place was taken by Gomez and Franquelo's second method, which used a similar sub–criterion, and so had similar performance to the MDLRU method, except that it was much slower.

Either the MDLRU or the MDLRUML should therefore be used in the simulator, depending on the balance desired between ordering time and number of operations generated. The remainder of the orderings used in this thesis are based on the MDLRU because it is less complicated to code and modify for special purposes.

	Overall ranking of the ordering methods					
	734 nodes		234 nodes		118 nodes	
	Relative to T–2	Relative to Best	Relative to T–2	Relative to Best	Relative to T–2	Relative to Best
Tinney–2	1.00	1.98	1.00	1.30	1.00	1.25
Tinney–3	2.28	4.52	2.21	2.86	1.74	2.18
MDML	0.77	1.53	0.91	1.19	0.88	1.11
GF–1	0.78	1.54	0.92	1.19	0.88	1.10
GF–2	0.75	1.47	0.89	1.16	0.85	1.07
GF–3	0.77	1.53	0.98	1.28	0.96	1.20
MDLRU	0.51	1.02	0.78	1.01	0.79	1.00
MDLRUR	0.77	1.53	0.95	1.23	1.12	1.40
MDLRUML	0.51	1.00	0.79	1.03	0.81	1.01
MDLRUA	0.51	1.02	0.77	1.00	0.81	1.02
MDLRUMLA	0.51	1.01	0.78	1.01	0.81	1.01

Table 4.8 Overall ranking of the ordering methods.

4.17 Conclusion

A completely new methopd of ordering the elimination of sparse matrices has been proposed and several variations on the theme were tried. The method replaces the standard search for an optimum column with a data structure, which, with little additional work, produces a node with minimum degree with almost no searching. This results in similar ordering times to the original ordering methods for small networks, but provides dramatic speed increases for larger networks, particularly those which represent the C.E.G.B. transmission network. The reduction in ordering time doubles the time remaining in the first time–step after a topology change for the remainder of the solution to produce a converged result. At the target floating–point solution speed, this would allow approximately four additional complete iterations of the solution, or eight in total, which should be sufficient to ensure real–time convergance for most network events. The times taken for the new orderings are shown to form a small part of the overall ordering time, so there is little speed to be gained from the further development of the ordering method.

The method also alters the order in which nodes are selected for elimination by adding a least recently altered sub–criterion to the minimum degree normally used, and this has been shown to reduce the number of pipeline stalls in a pipelined floating–point processor, and also to reduce the average path length, which is important in some applications, although it is not required for the uni–processor. It is, however, very important for the multi–processor simulator, because the analysis of path length provides a great deal of information about how to split the system to minimise the parallel solution time. A path which is well balanced and branches quickly from the root is highly desirable for a good split. Much effort has been devoted to this aim by other workers, but the new method proposed here produces equal or better results, in a much shorter time than the best of the other methods. When the new method is combined with one of these other methods, a dramatic reduction in path length is achieved, with little penalty in ordering time over the fast new ordering method used alone. This is achieved through optimisations in the searches which are not possible with the standard ordering methods.

A major comparison of the standard ordering methods and the novel methods proposed here was carried our, concentrating on ordering speed, the avoidance of pipeline stalls, fill–in and path length. While a few individual orderings could occasionally outperform the new methods on one task, the new ordering methods were shown to be greatly superior overall to the established ordering methods, and even performed much better than some, most notably MDML, at their intended improvement to the standard Tinney–2 ordering. This was achieved with a considerable time–saving at a critical time for a real–time simulator.

The new method proposed in this chapter not only provided an important speed increase over methods developed elsewhere, but also improved performance on other criteria which are of importance to the remainder of the simulator.

Chapter 5.

Hardware.

5.1 Introduction.

Since the introduction by the main four microprocessor designers and vendors (Intel, Motorola, Zilog and National Semiconductor) of 16 bit processors in the late 1970s, the computing power of microprocessor systems has approached and in some cases exceeded that of mini–computers, but at a much reduced cost. At about this time, a new group of super–computers was developed, for use in specialist fields requiring fast numerical calculations, the most notable of which were the CRAY family. These provided good performance on scalar arithmetic and non–arithmetic operations, and also possessed special vector instructions to pipeline repetitive calculations on vectors, to give a large performance boost.

The mini–computer manufacturers responded to the microprocessor challenge by introducing some of these features into their more expensive machines. Although microprocessor systems boasted good 'straight line' speed and good benchmark results, mini–computers, with their more complex input/output structure still handled multiple users and multitasking better than microprocessors, and so retained a market share. Some of these new features have even been incorporated into microprocessor systems.

On the Cray, the vector instructions can be concatenated, or chained, so that one vector instruction can use the results from the previous instruction, without the results being written to memory. These instructions can process data either from contiguous memory locations, or from regularly spaced locations in memory, such as would be necessary for vector products. More complex memory accessing methods require the formation of an address vector, which is used in a scatter operation which converts from the packed, sparse storage scheme to the expanded, vector storage scheme required by the vector

processor. The gather operation performs the converse operation, reading the values of specific elements of the mainly zero vector, and writing the contents of these locations into the packed storage locations. If the basic packed storage method is used, consisting of three parallel arrays, or an array of structures of three elements each, which contain the elements value, index and link to the next element in the vector, the simplest scatter implementation is given by equation 5.2, and the gather implementation is given by equation 5.3.

The structure for packed sparse matrices is (5.1)

```
struct packed_list_def {
    double value;
    int index;
    int next;
} pl[SIZE];
```

The scatter operation is given by: (5.2)

```
do {
    vec[pl[i].index]=pl[i].value;
    i=pl[i].next;
} while (i!=0);
```

The gather operation is given by: (5.3)

```
do {
    pl[i].value=vec[pl[i].index];
    i=pl[i].next;
} while (i!=0);
```

Accessing such data structures involves relatively random access to memory, with one level of indirection, which would clearly be slower than accesses to regular matrices.

5.2 Requirements for fast bifactorisation.

Studies have shown that the architectures of general purpose computers are not well suited to the solution of power system sparse matrices [141] [56]. This task requires little memory or process management and no interface to the outside world, but does require fast double precision calculations and fast access to operands stored in memory. The only computer architecture to support this is that of array processors [110], but these are provide poor programming environments and are weak performers for general purpose numerical calculations, such as are required to set up the matrix equation for solution. There is therefore much communication between the general purpose host computer and the array processor. The memory architecture of the array processor is also commonly optimised for typical digital signal processing applications, such as Fast Fourier Transforms, and could be more suited to bifactorisation.

The fast solution by bifactorisation of sparse matrices resulting from power system analysis requires fast double precision arithmetic, a fast memory interface, and the ability to generate addresses quickly. Chapter 3 showed that code can be produced in which all memory addresses are pre–determined, so address calculation is not a problem. The difficulty lies in the design and utilisation of the memory interface, through which all floating point data, floating–point and integer instructions and address data must pass. The memory interface must support high bandwidth access to and from relatively random memory locations, which is exacerbated by the double precision data required to ensure numerical accuracy. Accesses to the address list would conflict with the operand accesses for memory accesses with a standard, one data bus, processor. The problem is worse for older processors, since instructions also contend for these accesses, and while the newer RISC processors provide seperate instruction and data buses, full in–line code for all operations would take a significant amount of time to generate and use considerable memory space for the program. A brief tutorial on the hardware of memory accesses is required.

5.2.1 The microprocessor memory cycle.

To access memory, the processor must provide the full address for the access and an indication of the direction of the transfer, and an indication that the address is valid for the transfer, i.e., that a transfer is required. The full address must then be decoded

by the memory interface to determine which memory devices should participate in the transfer, as there are likely to be many such devices, each responding to a different, possibly non–contiguous range of addresses. The remainder of the address is buffered and passed to each device, along with transfer direction signals, and an enable signal from the address decoding circuitry. If data is to be written to the memory, data is also supplied and a strobe is used to show that it is valid and can be latched by the chip. All the signals are kept constant for a specified time, and can then be removed, normally in a specific order. A read cycle involves a wait after all signals have been applied, before the data is guaranteed to be valid. It would then be latched, buffered and sent to the processor. At the end of a completed transfer, the processor would either get some acknowledgement that data transfer was complete, and that the the cycle could end, which would also latch the data for a read cycle, or assume that the transfer was complete if no *wait* signal had been applied earlier to slow the memory cycle.

The control of the buffers and latches is complex, because some need to be bidirectional, and both ends cannot be allowed to drive the line at the same time, so time must be allowed for direction reversal. Passing a signal through a buffer or latch used to add a deley of about 20ns to the signal, with an similar delay for direction reversal. Taking an Intel i8088 [70] as an example, a memory access consisted of a minimum of four processor cycles, each of 200ns, so that a memory access took 800ns (figure 5.1). Of this, about 300ns was wasted in set–up times and inactivity in the processor, so the time from the address becoming valid to the data transfer being complete was 500ns. Due to the multiplexed address and data bus, further time was wasted with additional latches, so the memory cycle at the memory became more like 300ns. The logic chips then available had propagation delays of 10ns, which limited the granularity of the application of timed signals.

Older processors generated most of the signals required for memory interface, and some even provided special signals specifically for the control of bidirectional buffers, but rarely supplied address decoding (The Intel i186 [71] and Inmos Transputer do this too). Although this simplified the design, it also reduced the possible performance of the memory interface, because the most common current chips were taken as the model for the transfers, so little benefit was available from better, more costly, devices.

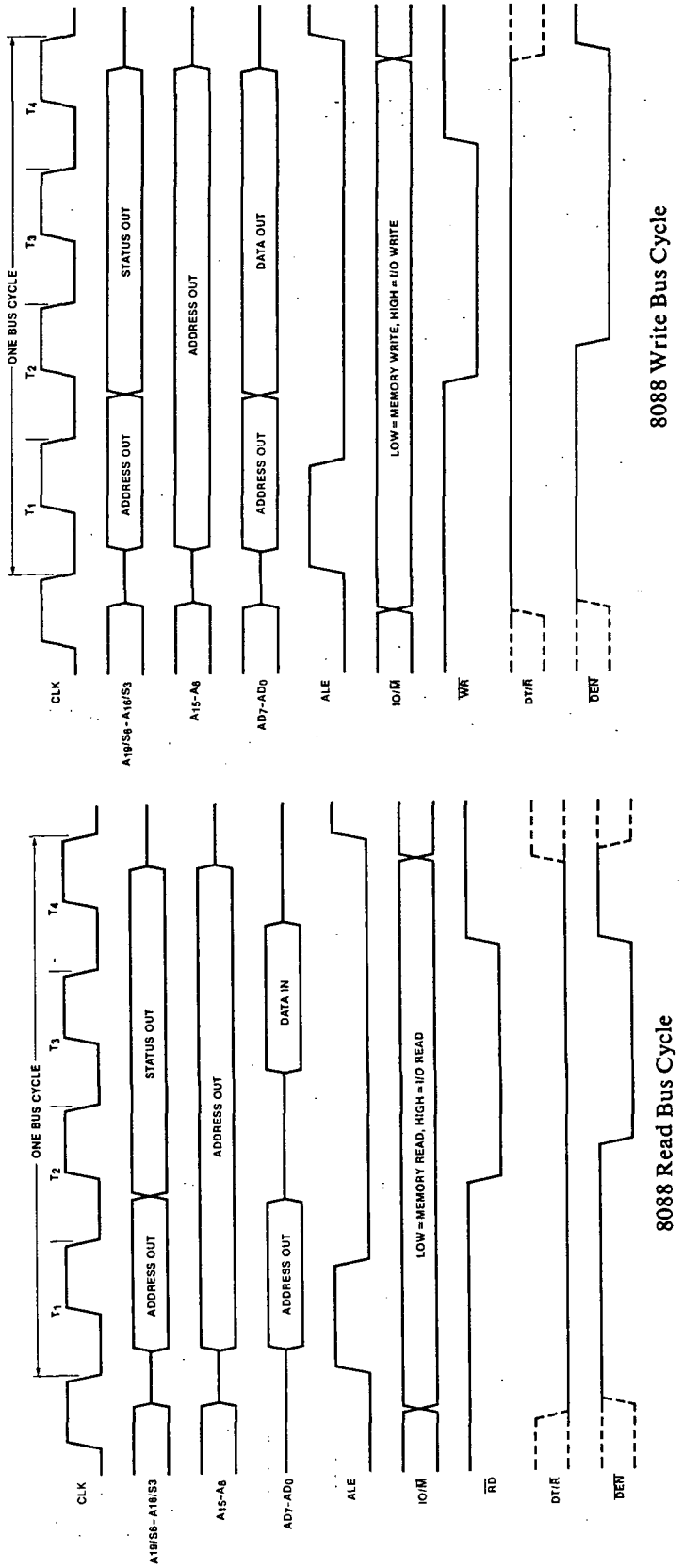


Figure 5.1 Intel i8088 memory cycle.

5.2.2 A higher performance memory cycle.

The basics are still the same for more modern memory designs, in that addresses and control signals must be provided to the memory chip, and a certain time later, the data transfer will be complete. More recent processors have faster cycle times, some, particularly RISC processors as short as 40ns, so buffering delays have become much more significant, although they too have decreased to approximately 5ns for the fastest devices. Memory speeds have increased, but fast memories are much more expensive than slower ones, and per–chip densities are also lower, which results in more chips and more buffering problems. Designers realised that they could produce better memory cycles if the timing was seperated from the description of the required cycle, so that instead of controlling the transfer, the processor states that it requires data from a certain address, and waits for the memory interface to provide the data, and acknowledge the completion of the transfer. This leaves the designer much more freedom to optimise his memory interface.

This has become particularly important with RISC processors, which consume vast amounts of instructions, which come largely from consecutive addresses, and such cycles are easy to predict, so the data can be obtained from memory in advance of the request from the processor, and provided with the minimum of delay when it finally is requested. It also enables the processor to issue requests for memory transfers, and then continue with other instructions until the data is actually required, or another cycle is needed, so the slow memory cycles can be hidden from the program.

5.2.3 Locality of reference and caches.

Many programs only require access to relatively small areas of instructions and data for significant periods, and it would be advantageous if these accesses could be made as fast as possible. This can be achieved without upgrading the whole memory by the use of caches, which are small blocks of fast memory, placed between the main memory and the processor, or more recently, inside the processor itself. Each location has a transparent dual location, which contains the address of the data which is currently stored in the location, and information about the validity of the data and the main memory location that is hidden behind it. As data is first read from main memory, a slow cycle must be used, during which the data is passed to both the processor and the

cache memory, and the cache dual is updated to show the hidden address, and that the data is unmodified. Future reads from that location will be satisfied quickly from the cache, where a parallel comparison of all the dual addresses is performed to determine whether the value is in the cache or not.

On processor write operations, the data is written to the cache, and may also be written to the external memory. If it is not written immediately, then a flag is set in the dual, to indicate thet the cache value is more recent than that in external memory. When an access occurrs to a location which would cause the changed value to be replaced in the cache, and it has been modified, then it must be written to external memory before it is replaced. Caches cause problems of data coherency between multiple processors, because the value in main memory may not represent the latest value, which might be cached locally. Some processors perform *bus snooping*, where the multi–user bus is monitored for cache conflicts, while others insist that the designer must ensure that such data is not cached. Adding additional processors can slow some cached systems.

Caches can produce impressive speed increases for some programs, but can also slow others if their access patterns interfere with the operation of the cache. This can be particularly problematical where the size of the problem leads to cache problems, especially where certain sizes of full matrix cause many cache elements to be invalidated, while a similar size does not. This is especially noticeable for matrix sizes near a power of two, such as 1024, because many caches consist of an integral power of two locations.

The access pattern during the bifactorisation algorithm can be seen to be largely random, but a cache might be beneficial for holding the values in whichever of the principal row and principal column are varied fastest during the elimination phase. It would certainly speed the access to the these elements, but might slow other accesses due to the increased complexity, and a fast memory interface is required in any case to read these values at the start of the processing of each block.

5.2.4 Alternatives for increased transfer rates

An alternative to speeding each access is to permit more than one access to be active at any one time. This can be seen in many designs which use multiple, interleaved memory banks to speed accesses to different banks, which occur frequently in instruction

fetches and auto–increment array accesses. Dynamic RAMs require time after the actual access cycle to recover, and also require periodic refresh cycles, and a predominately alternating access pattern can hide these delays. Some regular access schemes cannot take advantage of this, as accesses conflict (e.g., the CRAY multiple memory banks [60]), and in some cases it is possible to rewrite the code, or alter the arrangement of data in memory [91]. It is also possible with these designs to preempt cycles, but this might slow the next cycle if the guess were wrong, as the desired cycle might not be able to be started immediately. In particular, any accesses to data or the stack would disrupt the instruction fetch sequence.

An improvement would be the provision of multiple buses, one for each type of access, so that largely sequential instruction fetches could run without interruption, and would also not be slowed by possible reversal of bus direction. This also opens the possibility of multiple banking being used to preempt all instruction fetches, drastically improving the sequential transfer rate, or the use of memory devices with special serial shift–out modes, such as video dynamic RAMs.

Placing all data accesses on a seperate bus also permits these to be detatched from actual program execution, unless the program is waiting for the result of the data transfer, because instructions can still reach the processor on the other bus. Data transfers would usually take longer to complete, even using the similar memory devices to the instruction stream, because they cannot be accurately preempted, and the bus must be able to reverse direction. Most modern processors have a seperate instruction bus, because of the fast cycle times that are now common, but the only processor which has more than a single data bus is the Novix NC4000 [149], which is a dedicated FORTH language processor without floating–point hardware. Here, the instruction and data buses are combined, but seperate stack and parameter buses are provided to cope with the abnormal FORTH transfers.

It is possible to build a general purpose processor with more than two buses using components from microprogrammable chip sets, but the complexity of such designs was considered to be beyond this project. The main difficulties lie in the routing of the multiple wide buses, and the provision of logic to initialise the chips, and provide test vectors and monitor results should problems be encountered. The O.C.E.P.S. VAX-8600

uses an LSI-11 mini–computer exclusively for testing, power–up booting and a system–operator interface. If many of the general–purpose computer facilities are not required in the processor, then the design could be greatly simplified, and this was investigated, during a survey to find the most suitable hardware for sparse matrix processing. The strengths and weaknesses of the contending processors are discussed, and a paper design produced for how the problem could be mapped onto one of the processors.

5.3 Comparison of available processors.

There are currently three main types of micro–processor which should be examined in a survey to find the optimum choice for bifactorisation. These are the complex instruction set processors, reduced instruction set processors, and dedicated floating point processors from microprogrammable chipsets. In order to perform the sparse bifactorisation efficiently and quickly, the processors must have a minimum external data bus and internal ALU width of 32 bits, since 64 bit double precision data needs to be moved, a hardware floating point processor for the basic arithmetic operations, and a memory interface which can sustain high transfer rates. It would be beneficial if some support were provided for inter–processor communications, but this is not of paramount importance.

5.3.1 Complex Instruction Set Computers.

The derivatives of the first 16 bit processors which are represented by the Intel i386 and i486, and the Motorola MC68040 processors, have a complex instruction set and are referred to as CISC processors. The instruction sets are characterised by having a plethora of addressing modes, which most instructions can access, mainly two–operand instructions, with one operand possibly coming from memory, and variable length instructions to incorporate all the addressing modes. These processors have collected all the features of their predecessors, and many of the features of their competitors, and provide specialist instructions such as string movement and searching, table translation and single instruction subroutine entry and exit. The latest versions are fully integrated, with on–chip caches and floating–point units, and now support special fast bus cycles for improved memory access rates.

5.3.2 Motorola 68000 series processors.

The MC68020 [95] was the top processor in the 68000 series of 32 bit processors at the start of my research, and so it was already in use as the main processor in a parallel processing simulator designed at Durham University by John Flaxman and Jim Swift [43]. Floating–point operations were provided by the 68881 [97] numeric coprocessor and by a custom board using Weitek FPUs, which are discussed later. No support is provided for parallel processing.

This solution was discarded because the 68020 does not have a bus which is fast enough to transfer data at the required rate for larger systems, and because the floating–point performance was poor. The interface of the 68881 was designed so that it could function as both a dedicated coprocessor and as a standard memory driven processor for older 68000 processors which lacked a coprocessor interface. Instructions must therefore be specially written to the processor, and all communication is by bus cycles passing information. This contrasts with the far superior Intel interface, where the coprocessor watches the main instruction stream for its special instructions, and communicates with the main processor using dedicated signals. The result of the Motorola interface is that about five processor cycles are wasted in each data transfer.

Like the Intel processor, extended precision is used internally, and time is also wasted for all loads and stores in converting between this and the external 64 bit format. No parallelism is provided, so every load is penalised. This has now been alleviated in the 68882 [98], but although its performance has been improved, it is still not good. For general purpose mathematical code, the processors internal registers provide sufficient temporary storage, but for sparse matrices, almost every value would require to be loaded every time it is referenced.

An attempt was made to provide better performance by using FPUs from Weitek, which have a sparser instruction set, and calculate faster using the shorter I.E.E.E. standard formats. The chipset consists of two processors; one for addition, subtraction and format conversions, and the other for multiplication and division. The theoretical performance on simple arithmetic is good, but no support is provided for more complex functions, and the attainable performance is governed by the speed of the processor interface. Unfortunately, the 68020 interface can only provide about 4% of the maximum calculation rate. This is clearly not worthwhile.

The Weitek FPUs were configured as a memory mapped processor, so that writing data to a particular address initiated a certain operation and supplied part of the data. In effect, part of the address bus was used to generate the function inputs to the processors. All data must pass through the CPU on all transfers to and from the FPUs, and no temporary registers are provided internally, so if registers are required, they must be constructed from external logic.

The newest member of the series, the MC68040 [37] appears to have rectified many of these problems, but the design was revealed too late for consideration for the simulator.

5.3.3 Intel i486.

The Intel i486 [25] is a 32 bit, fully–integrated CISC processor. It is upwardly software compatible with its predecessors, which gives it a vast, general software base, but also retains inefficiencies, however it contains an improved internal FPU and faster instructions. There is now little to choose between the Intel and Motorola CISC processors.

5.4 RISC Processors.

5.4.1 Why RISC?

The advent of RISC (Reduced Instruction Set Computer) processors has made a great impact on computers. Prior to this, the instruction sets of computers were becoming increasingly more and more complex, with improvements in technology being used to implement more features instead of necessarily improving speed. Much effort was being expended into making the assembly language of each new processor as similar to high level languages as possible, with complex addressing modes to mimic array and matrix access modes, and special purpose instructions to perform what were seen as common operations in compiled code. The compiler writers found these instructions and addressing modes too complex to use, and that in many cases they actually decreased performance by inhibiting optimisation.

A research project at the University of California at Berkeley [106], found that a carefully selected instruction set could perform better than a more complex one, and took less die–space and was easier to develop. The criteria used to decide which instructions should be included were whether the instruction would produce an overall decrease in

code–size or execution time for a standard mix of software. This removes many less frequently used instructions, since they have little effect on the overall piece of code. It ws also found that by reducing the number of variations in each instruction, that the remainder could execute faster, because less decoding and intervening logic was required.

5.5 RISC characteristics.

There is considerable variety amongst the current RISC processors, but certain features are more or less common to them all [92].

Registers: Most RISC processors have a large on–chip register file, with some locations reserved for special purposes, such as processor control or state description. Registers are costly in terms of chip space, but are easy to design, and fit well with the instruction format.

Zero register: Most processors have a register whose contents are guaranteed to be zero, which can be used to provide comparisons against zero. The register is also used in address computations, so that the designers did not have to provide a special decoding circuit and addressing mode for a zero offset. The large register files minimise the effects of the loss of this register for general use.

Three operand instructions: In order to reduce register move instructions, most operations have two source locations and one destination, which may all be different. This helps significantly in address generation, since an address can be formed in a 'scratch' location with a succession of adds and multiplies, without a single data move being required.

Load / Store instructions: The only access to memory is through load and store instructions. All other instructions access register or immediate data contained in the instruction. This makes the load and store instructions easier to move at compilation time, which is beneficial if other instructions can proceed while external memory cycles are in operation.

Addressing modes: Most RISC processors have a single, simple addressing mode, based on register indirection from an immediate (hence fixed) base. This can be used to

implement direct, indexed and register indirect addressing by choice of address fields, and use of the zero register and a zero constant.

Instruction length: All instructions are the same length, usually 32 bits. Since there are relatively few instructions, this leaves most of the word available for other purposes, such as immediate constants or addresses (both register file and external).

Virtual memory support: All processors support a full 32 bit virtual address space, and most provide paging registers to manage the translation between this and a smaller real address space. The model is more flexible than provided by most CISC processors, but less transparent for the user.

Delayed memory accesses: Because the cycle time is so short, most memory architectures cannot respond in a single processor cycle, so the designers have implemented a delayed bus interface, so that the processor can continue executing code, until the value that was read from from memory is required by an instruction, or until another external data access is requested. This allows a slowish external data memory to be partially hidden behind the bus interface.

Cache control: Most processors provide cache control linked to the virtual memory support, or the manufacturers produce dedicated cache components to be linked to their processors. These provide another way to shield the processor from slow external memory, by attemptng to store frequently used data in a smaller amount of fast memory.

Multiple buses: The high cycle rate and thirst for an instruction every cycle, means that a single bus to external memory would saturate, so most processors provide seperate instruction and data buses. The address bus may, however, be shared, since most instruction fetches would be sequential. Two buses also permit long data access cycles to coexist with instruction fetches. Accesses to data would normally be longer than accesses to fetch instructions, since the cycles are more random, so the whole memory cycle must be performed within the processor's cycle, whereas the instruction fetches are regular, so multiple banking can be used. Multiple banking allows a different cycle to be in progress in each memory bank, so while one access is almost ready to pass its data to the processor, another has just passed its data, and is beginning another cycle for future data.

Pipeline: The regular form of each instruction permits more parallelism, which is used to increase speed. Each instruction is broken down into four or more parts, such as fetch, decode, get operands, excecute, store, so different parts of several instructions can be executing at once. By–pass ltches are provided so that sequential operations on the same registers do not have to wait for results to be written to the register file before they can be re–used.

Delay instructions: The pipeline means that there are wasted instructions after branch instructions, because the instruction pipeline must be flushed and refilled from new program locations if the branch is successful. Most processors execute the instruction following the branch instruction, so this can be used for useful work, if some can be rescheduled. If not, a NO–OP can be constructed to fill the location.

Branch target cache: This is a major RISC innovation. It is a block of on–chip memory used as a cache, but only for the first few instructions of recent branch (or trap) destinations, so that when a non–sequential instruction fetch occurrs, particularly in a loop, the processor does not have to wait for external memory to supply the next instruction. Without this, the pipeline would stall, which would reduce looping performance. Some processors have different branch instructions depending on the probability of the branch being taken, so that a pre–fetch unit would fetch along the normally taken path, rather than the sequential one.

Traps: The RISC processors have continued with the current CISC practice of having a large interrupt table, most of which is reserved for user 'traps'. The reduced stacking of traps [108] has made them much more efficient, and they can be used to implement commonly occurring groups of instructions to reduce code size. Many processors have a floating–point emulation library which use this interface, and, due to the simplicity of many of the hardware FPUs, the more complex floating point operations are implemented in software, even with a hardware FPU present.

Deletions: Many of the features included with almost all other processors have been deleted. These include the ALU status word, with its zero, carry and overflow flags, which have been replaced either by instructions which store the status into registers, or by branch instructions which perform tests. This allows the compiler much more scope in code optimisation inside loops, by making the

loop control seperate from the real work performed by the loop. The complex interrupt system has also largely gone, replaced by a single layer of interrupts and traps. This has reduced the need for a stack of addresses, so the stack has been removed, replaced by a small set of special purpose registers. The call and return mechanism in normal code can be predicted, and is left more to the compiler or user to implement as desired. Since high level languages make heavy use of stacks, the operation of the stack can be optimized for the needs of the compiler. Languages such as C use dynamic, stack based variable allocations, while Fortran uses the stack only for subroutine calls, with no dynamic variables.

5.5.1 Suitability of RISC processors for sparse matrix code.

The sparse matrix code can make little use of many of the addressing modes and complex instructions offered by CISC processors. All that the actual matrix processing requires is control transfer, simple address formation, operand transfer and floating point arithmetic operations. RISC processors certainly provide these operations, but the load/store architecture can restrict the transfer of data to and from the floating point processor, particularly for double words. The 32 bit bus can cause problems because most processors can only move 32 bits with one instruction, so that double–precision transfers need two instructions, which can also cause the pipeline to stall while the first transfer completes. The addressing modes are sufficient for no actual address arithmetic to be required on most processors, and the delayed branch instructions can hide most of the control transfer instructions in conjunction with a cache of either all instructions or just the first instructions after recent control transfers. Although the required tasks are handled well by the RISC features, their combination can saturate even these processors.

5.5.2 The RISC Options.

Each major processor manufacturer, and several new companies have all produced incompatible RISC designs, which have reduced the overall impact of RISC technology, by dividing the effort of software companies, and thereby reducing the software which can be run on any one processor. The processors which seem destined for success are the Intel i860, the MIPS R3000 series, the Motorola 88000 series, the SPARC designed by Sun Microsystems, and the Inmos Transputer. The Advanced Micro Devices Am29000 was withdrawn from the main market place, but has done well as an embedded controller.

Most computers are still based on CISC processors, primarily because most software still runs on them, but major manufacturers such as IBM, DEC and SUN are now producing RISC based systems, which should ensure their success.

5.5.3 Am29000.

The Am29000 [2 3] is a 32 bit RISC processor which eschews normal caches in favour of a streamlined dual–bus interface and a large branch target cache. The philosophy behind this architecture is that since the seperate instruction bus is pipelined, once the pipeline is started, the memory structure should be able to cope with regular transfers at the processor's clock frequency (which is 25MHz, giving a 40ns cycle time), otherwise cache misses would be devastating. The branch target cache minimises performance loss after branches, and other non–sequential instruction fetches.

The address bus is shared between data and instruction transfers, since most of the instruction transfers would be expected to be sequential. The data bus can support two transfers at any time, so a second transfer could start before the first completes, if the memory is partitioned to permit this. This design complicates the memory interface, but is only required for full processor speed. The pipeline makes possible the use of relatively slow and inexpensive DRAM for the instruction memory, or even the newer video DRAMs (VDRAM [100]), which have a seperate serial output mode, which would allow almost transparent data and instruction accesses to the same RAM chip.

Delayed branches, and delayed loads with full scoreboarding are implemented. The pipeline is generally logically invisible to programs, and the remainder of the processor facilities are all general purpose, so that most architectures could be implemented with little overhead. There is, however, no external data stack, as is found in almost every other processor. The designers opted to use the large internal register file to speed accesses, and force the software writer to implement the stack himself if one is required.

The Am29000 also has a companion floating point processor, the Am29027 [4], which is very similar to the Am29C327. The Am29000 communicates with the Am29027 over a 64 bit wide bus formed from the address bus and the data bus, using the standard memory access instructions with a special memory space. The interface can transfer 64 bits per cycle, so a multiply–accumulate of the form of equation 5.4 would take four cycles of data transfer, or 160ns. The data would still have to be moved to and from

memory, which requires the processor bus for the transfer, which can only move 32 bits per cycle.

$$I' = I - AE \tag{5.4}$$

The high rates of data flow into the processor require that the memory be physically close to the processor, since wide buses at high clock frequencies are prone to, and generate electrical interference. The fast cycle times coupled with the complex memory architecture for full performance, require fast PLDs in the interface between the processor and its memories. Problems were also initially encountered with the processor mask, which made the behaviour of the first production samples erratic, as successive register file accesses could corrupt unrelated registers.

5.5.4 MIPS R3000 series RISC procesor.

The MIPS R3000 [69] [79] RISC processor was not considered for the project because of its more extensive hardware requirements for a minimal system, and because of the tight coupling of the cache. The processor was intended more as a main processor for a mini–computer type system, and the architecture was tailored to suit. Full performance could only be achieved with large data and instruction caches and write buffers. These are fast FIFO buffers which accept data and addresses from processor write instructions, and then perform the write to memory more slowly at a later time, during otherwise free memory cycles. They work by accepting data at the processor clock frequency, and writing it using slower memory cycles, when the processor is not accessing memory, assuming that the processor cannot or is unlikely to sustain a high rate of memory writes.

In common with the other RISC families, there are several ideosyncracies in the design. The external signal tolerances are extremely tight, some timed to within 2 nanoseconds, with the sampling of input signals being most tightly defined. Special sampling circuitry was developed to eliminate the usual requirement for long set–up and hold times. This is a welcome contrast to Intel processors, which frequently have tolerances greater than a whole processor cycle on the address bus signals.

The processor has a fairly standard RISC instruction set. Complex aditional capabilities are provided by up to four possible coprocessors, the first of which is defined to be the on–chip virtual memory controller, and the second would usually be the external FPU.

The R3010 FPU [69] [120] is pipelined, contains 16, 64 bit registers and has the same cycle time as the main processor. The FPU is split into 4 functional blocks, which may work independently. These are the register–file and bus interface, an adder, a multiplier and a division unit. Multiplication and division operands both pass through the adder before entering and on leaving their functional units. These instructions can only be started if the adder is free, and will only complete when the adder is idle, but an addition can be *hidden* inside a multiplication, because it is a faster instruction. In a similar way, a multiplication and an addition could be hidden inside a division. This novel form of parallel execution could provide significant speed–ups, or could be completely useless, depending on the problem. Fortunately, this is handled relatively transparently, in that no special scheduling is required, but degradation of coupled performance could result if coprocessor instructions are placed at unfortunate points in the main instruction stream. The register file can be used to reduce the number of operand transfers to and from memory, and a special cache could be implemented to store the two sub–matrices for the current program block.

A tight schedule for sparse–matrix calculations must also be made to fit with the main processor instruction stream, as it will make heavy use of the load, store and address generation functions of the processor. The FPU requires that loaded and stored registers be sacrosanct for two cycles after the relevant instruction, which in addition to the delayed load instruction, provides a four–cycle delay on loading data into the FPU. An example schedule is shown in figure 5.2. This schedule shows that the processor can sustain sufficient operand transfers to keep the floating–point processor busy, with only address calculation instructions slowing calculations. Each submatrix would require 48 processor cycles for processing, so approximately 7,000 submatrix operations for the network elimination would need 340,000 cycles. The 25 MHz execution speed of this processor combination shows that the processor is sufficiently fast to achieve the target time of 50ms per matrix solution, if this schedule can be followed without pipeline stalls resulting from slow memory accesses. It is difficult to anticipate the sub–matrices that would be needed to permit the operands to be pre–fetched from slow memory into a small amount of faster memory, because that information is under the control of the main processor, and to provide the full benefit of cache usage, the whole sub–matrices would need to be fetched into the cache before any attempt was made by the processor to access them.

Instruction				Load	F–Point	Store	Integer
LWC1	FGR8	(R10)	0	A_u			
SUB.d	FPR2	FPR14	FPR2		$L - CF$		
LWC1	FGR9	(R10)	4	A_l			
ADD	R0	R0	0				NO–OP
MUL.d	FPR6	FPR10	FPR22		DH		
LWC1	FGR12	(R11)	0	I_u			
LWC1	FGR13	(R11)	4	I_l			
SUB.d	FPR4	FPR0	FPR4		$K - CE - DG$		
LWC1	FGR14	(R11)	8	J_u			
ADD	R0	R0	0				NO–OP
MUL.d	FPR0	FPR8	FPR16		AE		
LWC1	FGR15	(R11)	12	J_l			
LWC1	FGR10	(R10)	8	B_u			
SUB.d	FPR6	FPR2	FPR6		$L - CF - DH$		
LWC1	FGR11	(R10)	12	B_l			
ADD	R0	R0	0				NO–OP
MUL.d	FPR2	FPR8	FPR18		AF		
SWC1	FGR4	(R12)	16			K'_u	
SWC1	FGR5	(R12)	20			K'_l	
SUB.d	FPR0	FPR12	FPR0		$I - AE$		
SWC1	FGR6	(R12)	24			L'_u	
ADD	R0	R0	0				NO–OP
MUL.d	FPR4	FPR10	FPR20		BG		
SWC1	FGR7	(R12)	28			L'_l	
LWC1	FGR8	(R10)	16	C_u			
SUB.d	FPR2	FPR14	FPR2		$J - AF$		
LWC1	FGR9	(R10)	20	C_l			
ADD	R13	R13	4				R13 += 4
MUL.d	FPR6	FPR10	FPR22		BH		
LWC1	FGR12	(R11)	16	K_u			
LWC1	FGR13	(R11)	20	K_l			
SUB.d	FPR4	FPR0	FPR4		$I - AE - BG$		
LWC1	FGR14	(R11)	24	L_u			
ADD	R12	R11	0				R12=R11
MUL.d	FPR0	FPR8	FPR16		CE		
LWC1	FGR15	(R11)	28	L_l			
LWC1	FGR10	(R10)	24	D_u			
SUB.d	FPR6	FPR2	FPR6		$J - AF - BH$		
LWC1	FGR11	(R10)	28	D_l			
ADD	R10	R10	32				R10 += 32
MUL.d	FPR2	FPR8	FPR18		CF		
SWC1	FGR4	(R12)	0			I'_u	
SWC1	FGR5	(R12)	4			I'_l	
SUB.d	FPR0	FPR12	FPR0		$K - CE$		
SWC1	FGR6	(R12)	8			J'_u	
LW	R11	(R13)	0				R11=(R13)
MUL.d	FPR4	FPR10	FPR20		DG		
SWC1	FGR7	(R12)	12			J'_l	

Figure 5.2 MIPS R3010 instruction schedule for bifactorisation.

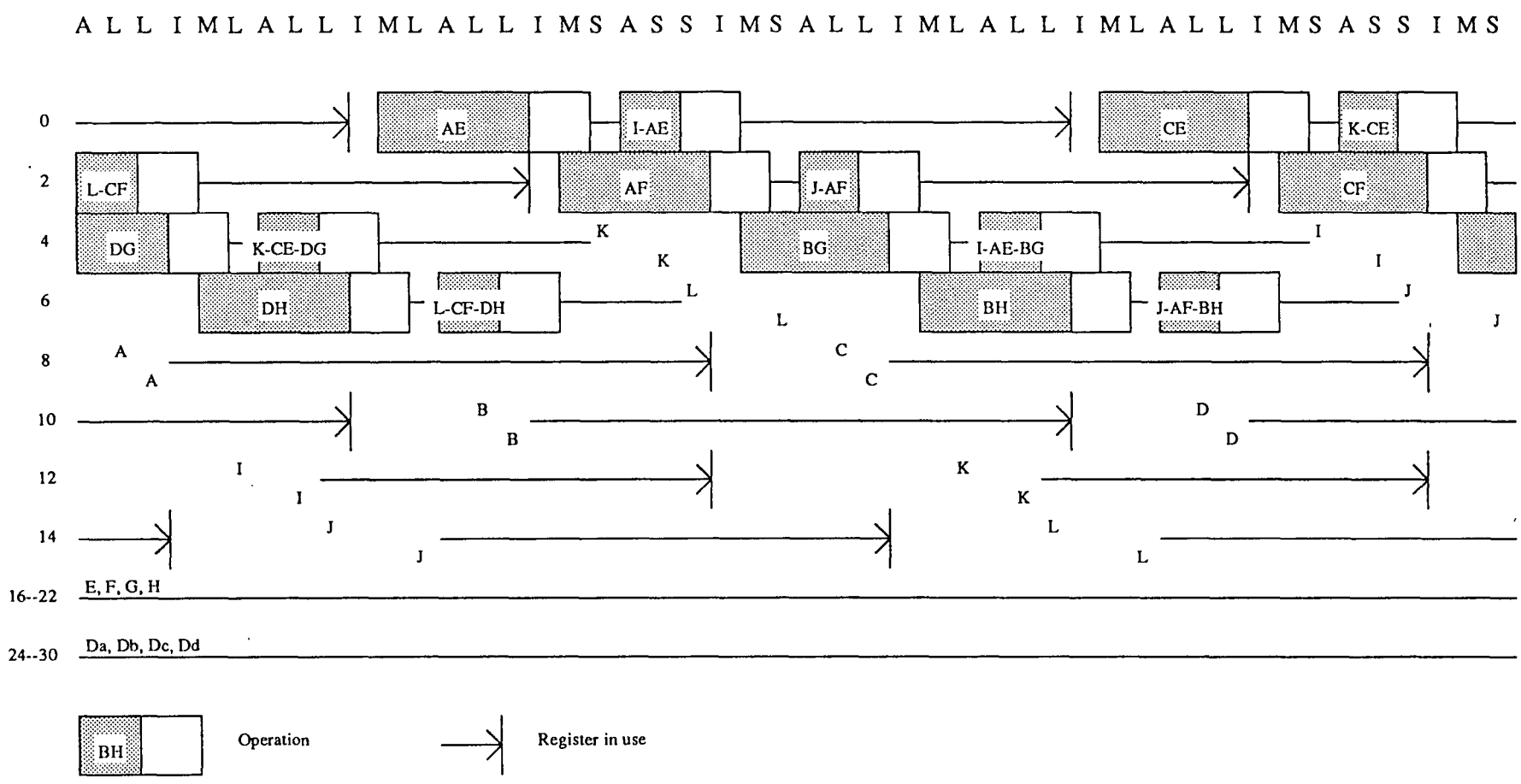


Figure 5.3 Registers for MIPS bifactorisation.

A regular schedule has been found, but the difficulties of scheduling interwoven calculations of different lengths in a pipelined environment, with possible delays in loading data, and limited registers (figure 5.3 should not be underestimated. The feasibility of this solution depends on whether memory delays become significant, in particular, whether two successive memory cycles can be instigated without stalling the processor.

The main processors *load and store* architecture is extended to the FPU, with all data transfers being between FPU register and either a processor register or memory, addressed in the usual manner. Software support for the processor is good, with DEC and MIPS both using it in commercial computers running variants of the UNIX operating system. This has also provided a good, unbiased speed comparison [68] between VAX (CISC medium performance mini–computers) and the new RISC processors, because DEC market both systems.

5.5.5 Intel i860.

The Intel i860 [72] appeared too late to be used for the simulator, but has several new features which make it suitable for sparse matrix processing. It contains a 32 bit RISC core, a 64 bit FPU and caches on chip, which is nothing unusual for RISC chips. Where it does differ, is that the FPU can produce two results per cycle, and work completely in parallel with the main CPU. The processor can be switched from 32 bit to 64 bit instructions, where half is fed to each processor unit, thus enabling complete parallelism of the processors, without having to include FPU instructions in the main instruction stream, or generate them with register moves. Two processors executing in parallel will consume many instructions, and require vast amounts of data, so all on–chip buses and the external data bus are 64 bits wide, except for the internal data cache bus which is 128 bits, allowing two 64 bit transfers per clock cycle.

The external data bus provides for transfer rates up to 320 megabytes per second, which is sufficient for sparse matrix algorithms. The floating–point instructions can only access the register file, however, so if many new values are required for processing, the integer processor will be saturated by data transfers to and from the floating–point unit, in addition to its own calculations for the addresses. A particularly useful feature is the software interaction with the data cache. In most processors, hardware must be used to determine whether memory accesses should be cached, usually on a block–by–block

basis. The i860 allows individual instructions to signal whether or not the data should be cached, both on reads and writes. Much data is read into the chip during sparse matrix solutions which will never be needed again, and it is therefore pointless to cache it, possibly overwriting cache data that will be needed again. Judicious use of this capability could speed the factorisation and solution operations, and permit the use of slower memory. The most efficient instructions for transferring floating–point data assume that it is in regularly spaced locations in memory, so they cannot be used effectively.

Special instructions are provided to evaluate an addition and a multiplication in parallel, but there is less flexibility in the choice of operands than with some other processors, since at least two temporary registers must be used as sources, since there are only two FPU read ports on the FPU register file.

5.5.6 Inmos Transputer (T414).

The original Transputer, the T414 [66] [111] had several unusual features which set it apart from most other processors. Most unusual is its in–built support for parallel processing, in both hardware and software, with process queues and message passing instructions, and DMA (Direct Memory Access) driven serial data links. While it is undoubtedly a RISC processor, with specific instructions added to facilitate parallel programming and multiprocessor communications, its programming model is completely different from other processors [101]. The user registers consist of a three–deep, unchecked stack, an operand register, and a base pointer. These registers are not preserved across certain instructions, such as loop control. The T414 provides two priority levels, with automatic saving of the user registers during a priority transition. The operand register is cleared after every main instruction, and is used to form long data values, and less frequently used instructions.

The instructions are broken down into two groups, based on frequency of use. The most commonly used instructions are encoded into a single byte, with less frequently used instructions being manufactured by combinations of special instructions, encoded as single bytes. The single byte instructions use the first four bits to determine the instruction, and the last four as data. Commonly used instructions are load, store, add, prefix, negative–prefix, execute, jump, call and load pointer.

The prefix instructions append their four data bits to the operand register, shifting any previous contents left first. This register can be used to form larger constants than can be accommodated in four bits, and also to form instruction words for the execute operation.

A further set of sixteen instructions can be formed in one byte by encoding them into the data part of the execute function. These instructions cannot have immediate data, and therefore operate wholly on the stack. Memory references are all register indirect, either locally from the base register, or non–locally using the stack to form and hold the address. This use of the main stack requires careful coding, so that address calculations do not force the real result off the stack.

This instruction format means that many instructions are required to perform simple operations, but each word fetched from memory contains four instructions, and the most common instructions are represented completely by one byte, so coding is quite efficient.

Multiprocessing on chip is performed by having two process queues, one for high priority processes and the other for low priority. If a high priority process is ready, then one will run, and will not release control until it finishes or cannot proceed. It will then be descheduled, allowing another high priority process to run. If none are ready, then a low priority process can be started. These are handled slightly differently, in that they may be descheduled at some branch or loop instructions, while still able to proceed, and will also be suspended if a high priority process can run. Communication and control processes should therefore be high priority, and calculation processes low, so that data is always passed between processes as quickly as possible. A process will also be descheduled when it requests either incoming or outgoing communications, since these operations are handled concurrently by hardware. The actual communications used are hidden from the process, and processes running on the same processor communicate via blocks of memory, which are handled in exactly the same way as hardware links

The T414 also contained a small amount of fast static RAM on chip, and could be run as a single–chip processor without other RAM or ROM for small programs with little constant data. The processor provides the option of loading programs from one of its communication links, instead of from ROM on power–up.

The performance of the T414 was impressive [159], because even without a dedicated floating–point processor, floating–point emulation routines could out–perform floating point coprocessors form Intel and Motorola. The lack of hardware floating–point support and the ideosyncratic native programming language called OCCAM [111], and the Transputer Development System, limited its acceptance.

5.5.7 Inmos Transputer (T800).

Instruction		operation	clocks	time	memory
genaddr	A		8	8	
fpldnldb			4	4	2
genaddr	E		8	8	
fpldnlmuldb		AE	22	22	2
genaddr	B		8	0	
genaddr	G		8	0	
fpldnldb			4	0	2
fpldnlmuldb		BG	22	22	2
genaddr	I		8	0	
dup			2	0	
fpadddb		$AE + BG$	10	10	
fpldnldb			4	4	2
fprev			2	2	
fpsubdb		$I - AE - BG$	10	10	
fpstnldb	I		4	4	2
TOTALS			124	94	12

Figure 5.4 Partial bifactorisation instruction schedule for T800.

Hardware floating point support was provided with the T800 [61] [65], which combined an almost identical integer unit with an on–chip, I.E.E.E. compatible floating point processor, again arranged on the three register model. The size of the on–chip RAM was doubled, and the speed of the links was also improved, allowing message bytes to be acknowledged on receipt of the first bit rather than the last. The FPU was arranged so that it could process independently of the integer unit, but it must receive all its instructions through it, and access memory via the contents of the integer register stack.

Instructions are formed using prefix instructions to load floating–point the instruction onto the top of the integer stack, and then a special instruction is built up in the operand register using prefix instructions, which launches the floating–point instruction selected by the top of the integer stack. The formation of instructions in this way makes the scheduling of calculations on the integer stack more difficult and ties up the integer processor while it could be performing other useful work.

The speed of the FPU is well matched to that of the integer unit, in that the address of a two dimensional array reference can be calculated in the time the FPU takes to perform a multiplication, thereby hiding the address calculations for matrix multiplication. A schedule for a quarter of the elimination process for a sub–matrix is shown in figure 5.4. The limited stack space means that no overlapping of blocks of calculations is possible, and the generation of addresses is taken on average to need eight processor cycles. Each sub–matrix would need approximately 400 cycles, including extra cycles for the fastest possible external memory accesses, which, at 20 MHz, is approximately five times too slow for the target time. This would result in a solution taking approximately a quarter of a second, instead of 50 milliseconds. It would be possible to split the task between four processors, but the scheduling would be difficult, particularly for small sub–matrices.

The processor was still plagued by a slow but programable memory interface [51], which cannot take advantage of current fast static RAM technology, being at best three times slower then the on–chip RAM. If the processor is executing code from the on–chip RAM, then the bus interface is fast enough to maintain an average data flow, but it cannot fetch double precision floating–point operands at the full rate at which the FPU can process them, because each access requires two cycles, and the previous result must be stored before any new operands can be read from memory because of the push–down floating–point stack. There are no instructions which can exchange data between the floating–point stack and the integer stack. The constant interruption of the calculations for memory access would severely degrade performance when solving the sparse matrices, as many fetches and stores are required, both for data and for information about the location of data, and the matrix data is much too large to fit in the on–chip RAM. Table 5.1 shows that it is better to have the program off–chip, and retain data on–chip if possible. The optimal solution would be to store the frequently

Speed degradation for external memory accesses.			
Program Located	Data Located	Data Intensive	Comp. Intensive
On–chip	On–chip	1.0	1.0
External	On–chip	1.3	1.1
On–chip	External	1.5	1.2
External	External	1.8	1.3

Table 5.1 Speed degradation of use of external memory.

used data, such as pointers and loop counters, in on–chip memory, and put as many of the inner loops into the on–chip memory as possible.

Since only the matrix data needs to be represented in double precision, a seperate dedicated processor could be used to actually solve the matrix, while the transputers set up the matrix and process the solution in parallel.

5.5.8 Inter–Transputer communication.

The four links [121] provided by the transputer provide a flexible and simple model for inter–process communication [84]. The flexibility is improved by the provision of a link exchange device, the IMS C004 [67], which can switch 32 links, and thus make 16 connections between transputers under software control from a control link. A small time penalty exists, due to the delay in passing each bit through the device, but this is most noticeable with the T414, which only acknowledges bytes after the whole byte has arrived, whereas the T800 sends the acknowledgement at the arrival of the first bit. The acknowledge message should still, therefore, arrive in time to permit uninterrupted transmission. Modifying links in real–time is dangerous, and must be carefully synchronised with processor activity, otherwise processors could dead–lock, or incorrect messages could be received.

The links have been optimised for ease of use, and not for throughput, which is about 2 MegaBytes per second at maximum. Each eight–byte floating point value would therefore take 4 microseconds for transmission, but since each block that would have to be transmitted would consist of four such values, $16\mu s$ would be required. This rate

is clearly unsuitable for the transfer of much floating point data, but is sufficient for message passing, semaphores and the like, and for integers, which can often be reduced in size to half–words or even bytes.

A direct bus link is therefore required between each of the transputers, which could either be DMA (direct memory access) or processor driven. The block move instructions of the T800 can move data at the maximum bus rate, and other instructions could continue internally if no other external accesses are required. Each processor supports standard external bus arbitration (BUS–REQUEST, BUS–GRANT), which can be used to force it from its own bus, while data accesses take place. Link messages can be used to synchronize these transfers. The main use for these bus transfers would be to pass the Zollenkopf matrix addresses and structure to each local processor, and to collect system data at the end of each time–step.

5.6 Special purpose processors.

5.6.1 Digital Signal Processors.

The processing of digitised analogue signals to extract information or remove noise has long been an important computational task, and the poor performance of general purpose processors was quickly realised. The type of calculations required by typical DSP algorithms such as FFT and filters are very similar to those required for the solution of matrix equations by bifactorisation, except that the required precision is much lower, because the accuracy of sensors and analogue to digital conversion is relatively poor. Both tasks share complex memory access modes, with operands being reqired from at least two seperate arrays, and with relatively little calculation per operand access. Many of these processors therefore had either two external buses, or a large internal 'constant' store for one set of operands. All possessed complex internal data paths to optimise routing for the most common applications.

The original market split in two, with most applications using either integer or fixed–point arithmetic, while applications that required floating–point accuracy and range used a new type of add–on computer called an array processor. These, typified by the Floating Point Systems AP120, were computers intended to be hosted by a general purpose computer which provided I/O and storage, and which were dedicated to the

fast solution of memory and computationally expensive calculations on arrays of data. They contained parallel address generation logic and multiple banks of memory so that memory access patterns for the most common applications did not cause the processing rate to decrease.

The developments in general purpose processors have also affected the single–chip DSP chips, with cycle times decreasing, and the provision of I.E.E.E. standard floating–point processing [35 36 45 105 132]. Currently, only single precision is available, but when double precision is incorporated, these should be considered for the solution of power system matrices. These chips are now being used for audio output form general purpose computers, and for graphics and video programming. Less dedicated processors with good double precision performance are available in the form of FPUs from microprogrammable chipsets. These are often used for DSP applications where more general calculations are also required, but require more external logic to produce a complete processor.

5.6.2 Weitek FPUs.

As this project began, Weitek manufactured a series of two–chip floating–point units, optimised for different tasks. Some were aimed at minimum–latency applications [153], while others were for vector–processing [150 151 154] applications. One chip of each pair was dedicated to addition, subtraction and scaling, while the other performed multiplication and division. These chips provided tight external control over their operation, and the separation of functionality allowed the ratio of adders to multipliers to be varied, and specific routings between multiple chips to be tailored to the algorithm. The processors could, for example, be laid out to effectively perform one convolution of the Fast Fourier Transform (FFT) in one cycle. This layout would not be efficient, and might not even be able to perform, other operations.

The chips provide no register file on–chip, so all operands which have changed from those of the previous cycle must be loaded for the next cycle, but a multiport register file is provided in the series [152]. This increases the amount of bus traffic around the processors, which causes electrical interference. Weitek later produced a new range, the XL series chips [158], with FPU [157], with more functions and registers on a single chip, and with a microprogrammable sequancer [155] and integer processor [156]. These were not as well suited to the bifactorisation algorithm as the AMD chips, and so were not used.

5.7 The Am29C327 Floating Point Unit.

The Am29C327 [8] [9] is a high performance, double precision floating point processor intended for use with other Am29C3xx components. These are VLSI micro–code building blocks [5], and include a sequencer, an integer ALU, a fast integer multiplier, a multi–port register file and a single precision FPU [7]. These can be used, with latches and memory, to build 32 bit processors with a user–defined micro-programmed instruction set. The cost and complexity of designing with these components is higher than using *off the shelf* microprocessors, principally because of the wide buses which must be routed between the individual chips, and the high bandwidth required of these buses and the surrounding memory.

These costs can be reduced by reducing the flexibility of the design, and therefore removing several of these building blocks. In particular, if the addresses of all memory references are known in advance, then no address calculations are required, so the ALU is not needed, which decreases the possible number of simultaneous accesses to the register file etc.

If a fast processor is required just to solve the sparse matrix factorisation, then, of these building blocks, only the FPU needs to be present. With the exception of errors, the control flow is fixed at code generation time, and all addresses are known in advance. A simple two–level controller capable of stepping through a series of calls to microcode routines involving simple looping could control the processor. The processor contains a small number of internal registers so that an external register file is not necessary with judicious coding.

5.7.1 The Am29C327 programming model.

The processor has a simple instruction set which provides comparison and format conversion operations in addition to three of the four basic arithmetic operators; add, subtract and multiply. Division is not supported, as it is complex to implement, and relatively rarely needed compared to the other operators. The chip area required to implement efficient division was felt to be better employed in providing more registers and bypass routes around certain functional blocks in the processor. The processor contains an adder and a multiplier curcuit, a set of I/O registers, a register file, a set of constants and an input multiplexor.

The multiplexer allows complete flexibility in routing operands from the input registers, the constant registers or the register file to the inputs to the functional units. This flexibility, combined with the small set of frequently needed constants (particularly 0.0, 1.0 and 2.0), permit many variations on the basic instructions provided, and also can help to maintain performance in pipe–lined mode. The units are also fitted with sign change blocks on inputs and output, which can negate data or produce absolute values on the fly. The units can also perform multiply addition instructions, which involves multiplying two inputs and adding the result to another number, thus forming the *sum of products*. This result is then fed back to the register file in the usual manner. This operation is of little benefit in normal operation, but becomes very useful when the processor is placed in one of its pipelined modes of operation.

5.7.2 Pipelined Operation of the Am29C327.

Pipelining involves breaking successive operations down into stages, and executing a different stage of several instruction in parallel. This can boost performance by reducing the cycle time of the processor, since less work needs to be performed per cycle, and by performing some operations in parallel. Simple instructions, such as addition, benefit relatively little from pipelining, as they cannot be broken down, but multiplication can be, and more complex composite instructions such as multiply–addition split well.

The add and multiplication instructions are split in two in pipelined mode, with the first cycle evaluating the result, and the second normalising or scaling this result. Multiply–adds do not require intermediate mormalisation, so they can be executed in three cycles.

Pipelining inherently delays the production of the result from when the operation is initiated, but the results are produced at a faster rate, once the first result has been produced. A trade–off must be found between obtaining results quickly, or at a high rate. Clearly the data dependencies in the problem will determine what approach is best, since if an operation needs a result which is still in the pipeline, in order to proceed, time is lost waiting for it to emerge, while in normal operation, the result would already be present. Switching from normal operation to pipelined operation is immediate, while the reverse operation requires dummy operations or cycles to flush results from the pipeline.

A doubly pipelined mode is available for multiply–addition instructions, which permits both adder and multiplier to execute at their full speed. This mode adds a two cycle delay to the normal execution time, but results are returned every cycle. There is also a two cycle penalty for switching out of this mode, so it is only worthwhile for relatively long sequences. This mode is made more efficient by delaying the direct input to the adder by one cycle, which permits the total from a sum of products operation to be used more quickly than would otherwise be the case.

Only the basic instruction must remain the same in this mode, so the sign change blocks can be used to vary the instruction, and different registers can be used as inputs, and therefore constants can also be used if desired. It is therefore possible to execute any of the basic operations using the multiply–add instruction, and thereby keeping the processor in double–pipelined mode.

5.7.3 Data flow through the processor.

An FPU performing two operations per clock cycle will consume vast amounts of data, and must, if execution is to continue at full speed, have a high–bandwidth data bus and memory interface. The processor has a very flexible bus interface, capable of attatching to either two 32 bit or one 64 bit bus, and being clocked either synchronously or at twice the clock speed.

5.7.4 Systolic arrays.

Although not a particular processor, systolic architectures should be considered, if only to be refuted for sparse matrix factorisation. A systolic array is a collection of processors arranged in a regular manner, which execute instructions on the data which passes between them. Each processor is usually relatively simple, out of necessity because there are so many of them. They have achieved good results for operations on full matrices, and for some digital signal processing applications, such as Fast Fourier Transforms, but this application, the massive data transfer requirements, the sparsity of the data, and the double–precision data format rule out their use. Their use for Gauss–Seidel iteration is discussed in [30].

5.8 Floating point performance.

There are several benchmarks which attempt to assess the floating–point calculation rates which are achievable for certain problems. While these results provide useful information, their application is fraught with problems, because even small modifications to the problems can result in large deviations in performance. Possibly the best example is reported in the IDT report [68] of the LINPACK benchmark, where results were drastically influenced by cache interaction.

Many of the benchmarks involve the evaluation of relatively complex expressions, involving regular address accesses and relatively few write operations. Processors which can perform address calculations in parallel with the floating–point operations can gain considerably from this type of calculation, and many optimisations are possible, such as unrolling loops and incrementing pointers instead of forming the addresses each iteration. This type of expression also hides many memory access problems, and can make very good use of caches which incorporate read–ahead or burst fill on cache misses.

The solution of sparse, power system matrices is unlike most of the benchmark problems, since it contains an above average ratio of memory writes to reads from memory, and can be written to require almost no operand address arithmetic, but does require many external bus cycles to read in operand addresses, in addition to those required for access to double–word floating–point data.

5.9 Selection of the optimal processor.

The CISC processors perform format conversion on all data transfers, which slows algorithms which require much loading and storing of operands. This penalty is in addition to the poor bus interface and over–complex instructions. RISC processors perform floating–point calculations in the external formats, so no format conversions are necessary on loads and stores, and indeed, several processors can perform operations on mixed single and double precision operands. Only the MIPS processors, the Intel i860 and the T800 support direct transfers between floating–point registers and memory, while the other architectures require the integer processor's registers to be used as an intermediate step, which will slow operand transfers.

The seperate instruction and address busses of the RISC processors, execpt for the Transputer, provide faster and better access to operands because instructions do not conflict for bus cycles with data transfers. A similar optimisation can be achieved with the Transputer for small programs, by packing the program into the internal RAM, but this precious resource must be shared with memory–mapped communications links and frequently used variables. The RISC processors do, however, rely on caches and special memory interfaces which may perform poorly with the unusual memory access patterns required by the bifactorisation process. Back–to–back accesses for double–words may cause pipeline stalls because many processors can only have one outstanding read operation at a time, and the unusually large number of writes combined with many operand fetches could nullify the MIPS write buffers, in addition to the caches filling up with data which is unlikely to be re–used, such as contents of the address list, and fully–processed operands. Only the Intel i860 provides good cache control, but the on–chip cache is small.

The MIPS processors are unique in the provision of arbitrarily overlapped floating–point instructions, but these present scheduling difficulties, particularly when coupled with long external operand access times and deep processor pipelines. The generation of addresses on the processor chip prevents effective pre–fetching of operands, either into on–chip registers or into an external cache. The Am29000 and Am29027 provide fast scalar arithmetic, and a novel utilisation of the processor interface for operand and instruction transfer, but suffer badly from the lack of direct transfers to and from memory.

All the general purpose processors, while providing reasonable to good floating–point performance on operands which are already loaded into the processor, have severe problems accessing externally stored operands, particularly when these operands are not arranged in a regular manner in memory. The high optimum performance of the Intel i860 is only achievable for regularly spaced vectors, because auto–increment transfer modes can cater for the address calculations transparently during operand transfers. The Intel i860 provides the best memory interface, with its wide external bus and extra–wide internal data paths, but the data paths and limited register access for the FPU limit the calculation speed. Support for inter–processor communications is severely limited in all except for the Transputers, which are hampered by their slow memory interface.

Even though all operand locations are predetermined, the large number of references and the two–by–two storage scheme limit the references produced by the Zollenkopf program (chapter 3) to the sub–matrix level, so the actual address of each individual operand must be calculated by the processor itself. This is trivial, and can easily be accomplished by *register plus offset* addressing, but can waste instruction cycles on some processors.

The above summary has shown that all the general purpose processors suffer from an operand transfer bottleneck, either due to insufficient bandwidth or data paths, or the inability to prefetch operands sufficiently early to guarantee that no processor stalls will occur. It would seem to be necessary to seperate out operand transfer, address formation and instruction generation, and this is only possible with a custom design using one of the floating–point units with additional logic controlling its operation.

The flexibility of using the Am29C327 can be used to alleviate most of these difficulties, except for the communication between parallel processors, but at the cost of increased system complexity. At the time when the choice was made, this option was the most promising, but the introduction of the Intel i860 has made this decision less clear, and for the relative simplicity that it provides, would probably have been chosen, had it been available.

Chapter 6 develops a design for a processor dedicated to the solution of power system sparse matrices by bifactorisation, using the Am29C327.

Chapter 6.

Dedicated elimination processor.

6.1 The Am29C327 Floating Point Unit.

Chapter 5 presented an overview of the available processors, and discussed their strengths and weaknesses with particular emphasis on the operations required during the solution of sparse matrix equations. It concluded that none of the available general purpose processors (possibly excepting the Intel i860 which was a late contender) possessed the required blend of characteristics, and that the best choice was to construct a dedicated processor from microprogrammable building blocks. The design of a general purpose computer from these components was considered to be beyond this project, so a dedicated design was attempted.

The Am29C327 [8 9] was chosen because it incorporates a fast, double precision processor, with an extremely flexible external interface and internal routing options. In particular, very high rates of data transfer into and out of the chip are possible, and high performance, flexible pipelined operations are provided. The chip provides a limited number of register locations, so that an external register file is not required, which considerably reduces the complexity of the design.

6.1.1 Programming the Am29C327.

The processor has a simple instruction set which provides three of the four basic arithmetic operators; add, subtract and multiply. Division is not supported, as it is complex to perform, and relatively rarely needed compared to the other operators. The chip area required to implement efficient division is employed in providing more registers and bypass routes around certain functional blocks in the processor, and in the provision of useful constant values. The processor contains an adder and a multiplier curcuit, a set

of I/O registers, a register file, a set of constants and an input multiplexer. The more complex functions can be calculated using power series or iteration [6] [57], which gives the programmer control over the time—accuracy compromise.

The multiplexer allows complete flexibility in routing operands from the input registers, the constant registers or the register file to the inputs to the functional units. This flexibility, combined with the small set of frequently needed constants (particularly 0.0, 1.0 and 2.0), permit many variations on the basic instructions provided, and also can help to maintain performance in pipelined mode. The units are also fitted with sign change blocks on inputs and output, which can negate data or produce absolute values on the fly. Subtraction can, for example be obtained by $A + (-B)$. The possible combinations are shown in table 6.1.

Sign Change Blocks	
Pass Through	A
Negate	$-A$
Absolute	$\|A\|$
Negative Absolute	$-\|A\|$

Table 6.1 Sign Change Optiions on Arithmetic Units.

The units can also perform multiply addition instructions, which involves multiplying two inputs and adding the result to another number, thus forming the *sum of products*. This result is then fed back to the register file in the usual manner. This operation is of little benefit in normal operation, but becomes very useful when the processor is placed in one of its pipelined modes of operation.

6.1.2 Pipelined Operation of the Am29C327.

Pipelining involves breaking successive operations down into stages, and executing one stage of each instruction in parallel. This can boost performance by reducing the cycle time of the processor, since less work needs to be performed per cycle, and by performing some operations in parallel. Simple instructions such as addition benefit relatively little form pipelining, as they cannot be broken down, but multiplication can be, and more complex composite instructions such as multiply–addition split well.

The add and multiplication instructions are split in two in pipelined mode, with the first cycle evaluating the result, and the second normalising or scaling this result. Multiply–adds do not require intermediate normalisation, so they can be executed in three cycles.

Pipelining inherently delays the production of the result from when the operation is initiated, but the results are produced at a faster rate, once the first result has been produced. A trade–off must be found between obtaining results quickly, or at a high rate. Clearly the data dependencies in the problem will determine what approach is best, since if an operation needs a result which is still in the pipeline, in order to proceed, time is lost waiting for it to emerge, while in normal operation, the result would already be present. Switching from normal operation to pipelined operation is immediate, while the reverse operation requires dummy operations or cycles to flush results from the pipeline.

A doubly pipelined mode is available for multiply–addition instructions, which permits both adder and multiplier to execute at their full speed. This mode adds a two cycle delay to the normal execution time, but results are returned every cycle. There is also a two cycle penalty for switching out of this mode, so it is only worthwhile for relatively long sequences of multiply–additions. Standard sum of products algorithms would appear to have a data dependency in the sum term, but this can be removed by forming two sums, and adding them together to produce the total result. This is possible since the direct input to the adder is multiplexed to it one cycle after the instruction is initiated. This allows the result from the previous addition on that sum to be read directly fom the output of the adder, bypassing the write to the register file, and reducing the effective latency of the adder stage.

$$S_{\text{total}} = \sum_i A_i \times B_i \tag{6.1}$$

$$S_{\text{even}} = S_{\text{even}} + A_{\text{even}} \times B_{\text{even}}$$

$$S_{\text{odd}} = S_{\text{odd}} + A_{\text{odd}} \times B_{\text{odd}} \tag{6.2}$$

$$S_{\text{total}} = S_{\text{even}} + S_{\text{odd}} \tag{6.3}$$

Only the basic instruction must remain the same in this mode, so the sign change blocks can be used to vary the instruction, and different registers or constants can be used as inputs, if desired. Table 6.2 shows how the basic instructions can be formed from the multiply–add instruction. It is therefore possible to execute any of the basic operations

while actually executing multiply–add instructions, and without needing to exit from the doubly pipelined mode. The results will still be delayed relative to the start of the instruction, but other calculations may be inserted to nullify this penalty.

Degenerate Cases	
Main Instruction	$A + B \times C$
Addition	$A + B \times 1.0$
Subtraction	$A - B \times 1.0$
Multiplication	$0.0 + B \times C$

Table 6.2 Base Instructions obtainable from Multiply–Add.

6.1.3 Data flow through the processor.

An FPU performing two operations per clock cycle will consume vast amounts of data, and must, if execution is to continue at full speed, have a high–bandwidth data bus and memory interface. The processor has a very flexible bus interface, capable of connecting to either two 32 bit or one 64 bit bus, and being clocked either synchronously with, or at twice the frequency of the processor clock. Two input operands can therefore be loaded every instruction cycle, and a seperate output bus can produce one data value per instruction cycle.

There is no direct route from the input registers to the register file, so all inputs have to pass through either the adder or multiplier before being stored, so pre–loading data is not possible. This only becomes a problem for the multiply–add instruction, as otherwise all instructions can be sourced entirely from the two input registers. Instructions must be ordered so that only two external data accesses are required by each operation. A multiply–add instruction can actually use three external inputs, but only at the expense of the following instruction, since the add term is delayed.

6.2 Application of the Am29C327 to Bifactorisation.

Chapter 3 outlined the benefits which are obtainable from ordering the sub–matrix calculations in different ways, and also introduced the idea of generating pseudo–code for these calculations. Pseudo–code requires that all data addresses be known in advance of execution, and for efficiency of program generation and storage, that macro instructions should be used, which in turn, invoke the basic instructions of the processor. These macro instructions are similar to the *call* instructions in high level languages, and offer similar benefits of compactness.

Using a small number of different macro instructions allows more time to be spent optimising each one for speed and/or memory usage. For the factorisation, the obvious basic instructions are to factorise sub–matrices of size m given an ordered list of all the addresses of the elements. Ideally, all these instructions should fit together seamlessly, so that no time is lost between them. This particularly important for the smaller sub–matrices, because these both occur more frequently and the proportion of the inefficiency loss would be greater as the instruction stream would be shorter.

The factorisation instructions for a condensed sub–matrices of order m in position j are of the form of equation 6.4–7.

$$a[j,j] = a[j,j]^{-1} \tag{6.4}$$

$$a[j,k] = a[j,k] \times a[j,j]^{-1} \tag{6.5}$$

$$\vdots \quad m-1 \text{ times with } k \text{ changing.}$$

$$a[i,j] = a[i,j] \qquad \text{No change.} \tag{6.6}$$

$$\vdots \quad m-1 \text{ times with } i \text{ changing.}$$

$$a[i,k] = a[i,k] - a[i,j] \times a[j,k] \tag{6.7}$$

$$\vdots \quad (m-2)(m-2) \text{ times with } i,k \text{ changing.}$$

Where i and k are indices of the non–zero elements in column (and row) j. The elements marked *no change* do not require processing. The inversion of the pivotal element $a[j,j]$ must be performed first, as all other operations depend on it. As the processor provides eight registers, two, two–by–two matrices can be resident in the register file at any one time, one of which must be the currently processed two–by–two matrix. The other could be either of the matrices in the principal row or column. The

order of element processing must therefore keep one of the matrices fixed for as long as possible to keep the register contents valid.

A decision must be made as to when the elements in the principal row should be modified. They could all be modified directly after the diagonal inversion, keeping this constant, or they could be modified at the start of each column, which makes them automatically resident for the remainder of that column, but would require the repeated reading of the elements of $a[k,k]$. The former makes relatively little use of memory during the modification of these elements, but causes the bus to become overful at the start of each column, because elements of all three matrices are required to be read in for the first element of each column. The latter alternative keeps memory accesses at a more even level, and is therefore to be preferred.

Each individual modification must now be written. Given the three general two–by–two matrices of equation 6.8, the following operations given by equations 6.9 to equation 6.12 must be performed

$$\begin{pmatrix} I' & J' \\ K' & L' \end{pmatrix} = \begin{pmatrix} I & J \\ K & L \end{pmatrix} - \begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} \tag{6.8}$$

$$I' = I - A \times E + B \times G \tag{6.9}$$

$$J' = J - A \times F + B \times H \tag{6.10}$$

$$K' = K - C \times E + D \times G \tag{6.11}$$

$$L' = L - C \times F + D \times H \tag{6.12}$$

Generally, it can be assumed that E, F, G and H in equation 6.8 are already loaded into the register file at the start of the operation, from the processing of the first element in each column.

Equations 6.9–12 can be split for processing in several different ways, but it is clearly necessary to evaluate each line in at least two or possibly three stages, which cannot be executed consecutively because of the latency of the pipelined operations. To make the structure more regular, it would be best to perform a similar operation on each line before proceeding with the next operation. This leaves operations of the form of equation 6.13, followed by operations of the form of equation 6.14.

$$I' = I - A \times E \tag{6.13}$$

$$J' = J - A \times F$$

$$
\begin{aligned}
K' &= K - C \times E \\
L' &= L - C \times F \\
I' &= I' - B \times G \\
J' &= J' - B \times H \\
K' &= K' - D \times G \\
L' &= L' - D \times H
\end{aligned}
\tag{6.14}
$$

If elements $E \ldots H$ are preloaded, then only two external elements are required per line, so no input/output restrictions are violated. A schedule for both the multiplication of the first element in each column by the leading diagonal sub–matrix, and the for the further processing of the sub–matrix in the following rows is shown in figure 6.3. This shows how the eight registers can hold two sub–matrices, and that no conflicts arise with the transfer of operands, either internally to the processing of a sub–matrix, or between such operations.

The first element in each column is a special case of this process, where only a multiplication is performed, without the subtraction, so the operation is of the form of equation 6.15, and can be broken down in a similar way to produde equations 6.16 and equation 6.17.

$$
\begin{pmatrix} I' & J' \\ K' & L' \end{pmatrix} = \begin{pmatrix} I & J \\ K & L \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} \tag{6.15}
$$

$$
\begin{aligned}
I' &= I \times E + J \times G \\
J' &= I \times F + J \times H \\
K' &= K \times E + L \times G \\
L' &= K \times F + L \times H
\end{aligned}
\tag{6.16}
$$

$$
\begin{aligned}
I' &= 0.0 + I \times E \\
J' &= 0.0 + I \times F \\
K' &= 0.0 + K \times E \\
L' &= 0.0 + K \times F
\end{aligned}
\tag{6.17}
$$

$$
\begin{aligned}
I' &= I' + J \times G \\
J' &= J' + J \times H \\
K' &= K' + L \times G \\
L' &= L' + L \times H
\end{aligned}
\tag{6.18}
$$

Bifactorisation schedule for Am29C327.								
Input		Multiplex			Function	Store	Out	Equation
R	S	P	Q	T	Op	Reg	F	
First element in column.								
R_a	D_a	R	S	0.0	T+PQ	R0	I_a	$R0 = D_a \times a$
R_a	D_b	R	S	0.0	T+PQ	R1	I_b	$R1 = D_b \times a$
R_c	D_a	R	S	0.0	T+PQ	R2	I_c	$R2 = D_a \times c$
R_c	D_b	R	S	0.0	T+PQ	R3	I_d	$R3 = D_b \times c$
R_b	D_c	R	S	R0	T+PQ	R0		$R0 = R0 + D_c \times b$
R_b	D_d	R	S	R1	T+PQ	R1		$R1 = R1 + D_d \times b$
R_d	D_c	R	S	R2	T+PQ	R2		$R2 = R2 + D_c \times d$
R_d	D_d	R	S	R3	T+PQ	R3		$R3 = R3 + D_d \times d$
Other rows.								
C_a		R	R0	S	T–PQ	R4	R_e	$R4 = i - R0 \times a$
C_a	I_i	R	R1	S	T–PQ	R5	R_f	$R5 = j - R1 \times a$
C_c	I_j	R	R0	S	T–PQ	R6	R_g	$R6 = k - R0 \times c$
C_c	I_k	R	R1	S	T–PQ	R7	R_h	$R7 = l - R1 \times c$
Cb	I_l	R	R2	R4	T–PQ	R4		$R4 = R4 - R2 \times b$
Cb		R	R3	R5	T–PQ	R5		$R5 = R5 - R3 \times b$
Cd		R	R2	R6	T–PQ	R6		$R6 = R6 - R2 \times d$
Cd		R	R3	R7	T–PQ	R7		$R7 = R7 - R3 \times d$
C_a		R	R0	S	T–PQ	R4	I_a	Store
C_a	I_i	R	R1	S	T–PQ	R5	I_b	previous
C_c	I_j	R	R0	S	T–PQ	R6	I_c	results
C_c	I_k	R	R1	S	T–PQ	R7	I_d	

Table 6.3 Bifactorisation schedule for Am29C327.

Again, only two new terms are required per line, so multiply–add instructions can be used without bus conflicts. $I' \ldots L'$ become $E \ldots H$ respectively in equations 6.16 to equation 6.18 for the row in question. Table 6.3 shows how these instructions fit together, and that the output of each instruction arrives in the register file in time for when it is required for further processing. Since both instruction blocks use similar instructions and use similar data, the join between them should also be smooth, without inefficiencies. The results from the output latch are fed back into the register file and are available for re–use in the third instruction after the issuing instruction if the data is needed by the multiplier, and the second if it is used by the adder. Since all these instruction sequences involve sets of four calculations, no difficulties should arise. Two blocks are shown, the upper shows the multiplication of the row entry by the inverse of the principal diagonal sub–matrix, and the lower shows the adjustment of the first sub–matrix after this operation, and part of the next adjustment.

6.3 IEEE floating point data formats.

There are three forms which are used represent floating point numbers in computers. They are based on exponential notation, where a mantissa, in a predefined range is multiplied by a constant raised to a variable power to give the number desired. This is generally used in the base ten number system most commonly used by humans, for example 1600 can be represented by 1.6×10^3. Computers are much better at handling binary, (base two) numbers, or a number system of some power of two. The above number could also be represented by 1.5625×2^{10}, and also by 6.25×16^2.

The base ten representation is less convenient for the computer to process, and provides less precision for the same number of bits than the other methods, and is therefore less widely used. It can be advantageous where accuracy or simplicity in calculations is required, since the number system is so similar to that in common use by humans. Numbers produced by and for humans tend to be rounded to base ten, and humans tend to think in base ten, so for example, money is frequently expressed in some decimalised form, which is difficult, and in some cases impossible (e.g., $1/10$), to represent accurately in binary form, just as $1/3$ is impossible in base ten. The base two and sixteen representations pack more information into memory, but are less good at representing some frequently used numbers. The extra precision can counter this problem.

The base two notation is the most widely used, and was adopted by the IEEE in the standard 'ANSI/IEEE Standard 754-1985 for Binary Floating Point Aritmetic' [63]. It is also used by DEC in a slightly different form in their VAX computers. The IEEE representation will be described in more detail because it is more widely used by microprocessors.

6.3.1 IEEE floating point representation.

The IEEE standard defines several lengths for floating point data, ranging from 32 bits, through 64, 80 to 128 bits. The first two of these lengths are widely implemented, the third is used internally in some coprocessors to increase accuracy, and the fourth is little used, because of the increased calculation time, storage space, and most importantly, the difficulties of moving such data around. The numbers are stored in three parts; a sign bit, a biased exponent and a normalized mantissa. An explicit sign bit instead of two's complement negation simplifies the processing of the mantissa, while a biased exponent removes the need to detect negative numbers while shifting and scaling the mantissas, so that they are always in the range $(1.0 \ldots \{2.0 - \epsilon\})$ where ϵ is the weighting of the least significant bit. The bias applied to the exponent is usually approximately half the exponent range. The double precision format uses eleven bits to store the exponent, so a bias of 1023 is used.

A number of embelishments are added to provide more information about any special events which might have occurred. The largest representable exponent is reserved to signal special events, such as infinity, or that the operator or function invoked could not return a value for its argument. Special 'numbers' called NANS (Not A Number) are also provided to signal special events, and are handled by defining that any operation which processes a NAN produces a NAN of equal or increased seriousness, so they propagate through to the final result. At the other end of the scale, there is a sudden break in representable numbers between zero and 1.0×2^{-1023}, so for the smallest exponent, the mantissa is assumed to be denormalised, to increase the range, at the expense of precision. The memory format is slightly optimized, by removing the most significant bit of the mantissa, which is assumed to be one, unless a denormalized number is present, in which case it must be explicitly stated, so one bit of precision is lost. Zero is represented as a number with all bits zero (except the sign bit, which can be either

positive or negative). If s is the sign bit, e the exponent and f the stored mantissa, then equation 6.19 gives the possibilities which are available.

$$\text{value} = \begin{cases} \text{Reserved, NAN} & e = 2047 \text{ and } f \neq 0 \\ (-1)^s \infty & e = 2047 \text{ and } f = 0 \\ (-1)^s (1.f) 2^{e-1023} & 0 < e < 2047 \\ (-1)^s (0.f) 2^{-1022} & e = 0 \text{ and } f \neq 0 \\ (-1)^s (0.0) & e = 0 \text{ and } f = 0 \end{cases} \tag{6.19}$$

The VAX representations are similar, except that the number of bits allocated to the mantissa and exponent are different, and a normalized exponent is in the range $(0.5 \ldots \{1.0 - \epsilon\})$.

6.3.2 Division on the Am29C327.

The processor does not provide a division instruction, so this must either be performed externally, or an iterative algorithm used internally. The best approach is a combination of these, using a pair of ROMs to generate an initial seed for the denominator, which is then iterated internally to produce a more accurate value. This number is then multiplied by the numerator to complete the division.

Using the Newton–Raphson approximation of equation 6.20.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{6.20}$$

on the formula for the reciprocal:

$$f(x) = 1/x - B = 0 \tag{6.21}$$

where B is the number for which the reciprocal is required.

This becomes:

$$x_{i+1} = x_i - \frac{1/x_i - B}{(-1/x_i^2)} \tag{6.22}$$

Rearranging produces:

$$x_{i+1} = x_i \times (2.0 - B \times x_i) \tag{6.23}$$

For binary floating point data formats which use a biased exponent, an approximation for the exponent for the reciprocal is obtained by subtracting the exponent from the maximum value of the biased exponent, and applying equation 6.24 for the mantissa

$$\text{mantissa} = \frac{2.0}{1.0 + \text{Frac}(x)} - 1.0 \tag{6.24}$$

The exponent and mantissa are easily seperable, and the sign bit takes care of itself. All that is required is an external data path round the processor which incorporates two PROMS, which can be switched in when required. Values can then be output, passed through the approximator and re–input.

Convergence of this method is good, approaching quadratic as the estimate improves, so the number of correct bits should approximately double every iteration. If thirteen bits in the mantissa can be correctly predicted, then a result correct to about 50 bits should be available after the second iteration. This is not implausible, as 64K ROMS are available, and two would give the 16 input bits and 16 output bits required. Clearly, not all the 16 bits would be accurate, but IEEE double precision should be attainable within two iterations. Convergance would be expected to slow slightly as the limit of the processor's precision is reached, but because power system matrices only require about 30 bits of precision, the possibility that the lowest few bits might be incorrect is not important.

If 16 bits are used as inputs and outputs from the mantissa PROMS then effectively 17 bits of the denominator are used, because one is implied in the representation. The error likely in the mantissa estimate can be calculated from the Taylor Series for $y = {}^1/_x$, which is given in equation 6.26, where a is a number $1.0 \leq a < 2.0$ and h is the maximum error in rounding x to the precision of a.

$$y \approx f(a) + \frac{h}{1!}f'(a) + \frac{h^2}{2!}f''(a)\ldots \tag{6.25}$$

Since $h = 2^{-17}$ and $f''(a) < 2$, h^2 term is negligible.

$$\approx f(a) + hf'(a) \tag{6.26}$$

The maximum error in the full–precision reciprocal approximation is therefore:

$$\text{error} \approx \left\lfloor \frac{-h}{a^2} \right\rfloor \tag{6.27}$$

$$\approx 2^{-17} \tag{6.28}$$

Equation 6.28 gives the error in the mantissa which results from the truncation of the input mantissa to 16 bits. A similar error will occur due to the limited precision of the output mantissa, which will be of the same magnitude, so therefore at least 15 and possibly 16 bits, including the implied bit, will be correct before the first iteration using a lookup table with 16 bit input and output. This should achieve the fullest accuracy within two iterations.

The performance of this configuration was tested on the VAX-8600 using DEC–D format, which provides 55 mantissa bits for extra precision, but fewer exponent bits for less range. The PROMS were simulated by performing rounding on the generated random number, and using this in the above equation to produce the seed, which was also rounded before being iterated. The results were as expected, with a minimum accuracy of 55 bits, the limit of DEC–D.

Trials were performed with other seed generators, and using different numbers of bits for input and outout, but the 64K by 16 approach was found to be the best compromise between speed and hardware requirements. Removing one iteration would require an 8 Giga–word lookup table of 32 bit words, which is beyond a power system simulator using 1990's devices.

A problem was found with mantissas in the range $1.0 < x < 1.0 + 2^{-16}$, because these were truncated to 1.0, which gave a reciprocal mantissa of 1.0 instead of 1.999. Unfortunately, the exponent calculation was correct, so the seed was approximately half of what it should have been. Five corrective methods were simulated. The first corrects the exponent when all the explicit mantissa bits are zero. The other four adjusts the mantissa input to the PROMS, either setting the least significant bit (LSB) of the input, or the bit below the LSB. This is difficult to do in hardware, because this bit is not actually passed to the PROM, but is easy to implement in software. The adjustments of the input to the mantissa PROMs require no additional hardware, because the value stored in the PROM could be altered to simulate the altered input. This is particularly easy, because only the function used to generate the values needs to ba altered. It would be possible to adjust all values in the PROM, or only the one giving trouble. This would be in the first location in the PROM, since the external part of this mantissa is zero, which makes the full mantissa unity. The first option is more difficult as it requires incrementing the exponent, which not only requires a comparison against zero on the mantissa, but also an adder, and could give overflow or wrap–around conditions.

The results presented in table 6.4 are for ten million random numbers in the range $(1.0 \ldots \{2.0 - \epsilon\})$ and show that correcting the exponent or the mantissa only when necessary are the most accurate methods, with no difference in accuracy between them. Correcting every entry decreases the accuracy, and is therefore undesirable.

Number of correct bits during divide			
Correction to input to seed PROMs	seed	iteration first	second
Increment exponent when mantissa = 1.0	16.0	32.0	55.0
Set LSB when mantissa = 1.0	16.0	32.0	55.0
Set LSB-1 when mantissa = 1.0	16.0	32.0	55.0
Always set LSB	15.4	30.9	55.0
Always set LSB-1	15.7	31.4	55.0

Table 6.4 Number of correct bits during reciprocal approximation.

The division can be performed wholly in double–pipelined mode using multiply–addition instructions, and can be made quite efficient by careful analysis of the instructions required. The main operation $(2 - BX_0)$ fits well with the basic instruction, especially as the processor has the constant 2.0 available directly from the multiplexer. The equations are rearranged to the form of equation 6.30, and then if two iterations provide sufficient accuracy, substitutions can be progressively made (equation 6.31) to produce the final version of the expression which is given in equation 6.32.

$$y = \frac{A}{B} = A \times \frac{1}{B} \tag{6.29}$$

Inserting last approximation for reciprocal

$$= A \times x_1(2 - Bx_1) \tag{6.30}$$

Inserting first approximation for reciprocal

$$= A \times x_0\,(2 - Bx_0)\,(2 - Bx_0\,(2 - Bx_0)) \tag{6.31}$$

Rearranging to express parallelism using pipelined multiply–add instructions

$$= [(Ax_0)\,(2 - Bx_0)] \times \{2 - B\,[x_0\,(2 - Bx_0)]\} \tag{6.32}$$

Where the term $(2 - Bx_0)$ only needs to be evaluated once.

If the processor is kept in doubly–pipelined mode, then the instructions given by instructions 6.33–37 are required to perform a division. The divisor is B, the seed x_0 and the numerator A in equations 6.29-32.

$$R2 = 2.0 - \text{divisor} \times \text{ seed} \tag{6.33}$$

$$R1 = 0.0 + \text{seed} \times \text{ numerator} \tag{6.34}$$

dummy cycle

$$R0 = 0.0 + \text{ seed} \times R2 \tag{6.35}$$

$$R1 = 0.0 + R1 \times R2 \tag{6.36}$$

dummy cycle

$$R0 = 2.0 - B \times R0 \tag{6.37}$$

dummy cycle

dummy cycle

$$R0 = 0.0 + R1 \times R0$$

6.4 Hardware implementation of Bifactorisation.

Performing fast calculations on sparse matrices consumes a large amount of data very quickly, so a large, fast memory array is essential. The Am29C327 is capable of clock cycles of 120 nanoseconds in doubly–pipelined mode, and maximum performance clearly requires that the processor should be run at this rate. To keep the processor executing at its maximum rate also requires that data values must be transferred when the processor wants to transfer them, and that memory restrictions must not affect these transfers. Each processor cycle can require two data fetches and produce one data write to memory, which can all be to and from any location. Interleaving cannot therefore be used, as eventually a conflicts between accesses to the same bank would occur, so the memory must permit three transfers per processor cycle. The problem is made more difficult by the data bus having to change direction during each processor cycle, so extra time is required to prevent bus contention during the change of direction. Currently, memory capable of reads in less than 30ns is expensive and only available in low densities. To obtain more time, a double–banked memory design was investigated, where the reads take place in parallel from two identical banks, and the data is written to the same location in both banks during the remainder of the cycle. This gives a memory cycle of 60ns, enabling a memory access time of 40–45ns to be used.

As double precision data is required, each access must be 64 bits wide, which would result in severe electrical transients on the card, requiring special grounding techniques, and restricting device lay–out. Since most of the card is synchronous with respect to the processor clock, these transients would be especially severe, and may prevent the design from working. Swift found that on the Weitek FPU boards used by Flaxman [43],

that the calculations were correct at low speed, but when full speed was attempted, the calculations failed due to voltage collapse accross the board, because so much switching was taking place. A major rewiring was required. The wide bus restricts the design to a single card, as it is not possible to transmit wide data reliably between cards without special hardware, such as is found in the CRAY-1. The memory used must therefore be fast and high-density.

IDT provide such memory modules [64] for use as memory caches with their RISC processors, and these are especially useful as they are designed to support double accesses per cycle, and also support the very tight tolerances of their processor interface. The double cycle interface incorporates latched address buffers and seperate data pins for input and output, which permit much more overlap of accesses, particularly when one is guaranteed to be a read, and the other a write. These modules are available in 32K by 64 bit packages, so four should be sufficient for the CEGB problem. It should be bourne in mind that at £1000 each, compared to a floating-point processor priced at £100, these memory modules are not cheap.

The actual program and address data is required at a much slower rate, and could use normal memory devices and a standard data bus. Data for the sequencer is more difficult, and at this point work on the design was discontinued in favour of a parallel approach.

6.5 Microcode.

Programming and designing with in microcode is unlike programming in either assembler or high level languages, because each instruction can perform many different tasks. The instructions are generally very long, and consist of many sub-instructions which are possibly decoded into a longer instruction work, parts of which are passed to specific hardware locations. These sub-instructions control specific functions of the overall processor. The microcode for important tasks should ideally be defined in conjunction with the hardware design, so that the necessary hardware features can be provided.

Examination of the bifactorisation process shows that for each floating-point operation, one floating-point instruction and three operand addresses must be formed, three operand transfers must take place, and the next microcode instruction must be fetched.

There is little flexibility in the floating–point instruction, because the instruction set is fixed, so it would probably be best to provide this directly from the micro–code instruction. Parts of this instruction, such as input/output port controls and register control have close links with address generation and buffer and memory control, which must also be under microcoded control.

Fetching the next micro–code word would be expected to be relatively simple, because loop control is not required, so no conditional branches are needed. Transfer instructions could read an address from one of the input queues, and transfer this to the microcode instruction counter to fetch the next instruction from a different location in the microcode store.

The address calculations are trivial, but hardware must be provided to perform the required actions, which depend on the form of the addresses which were stored by the pseudo–code generator. If schemes b or d of chapter 3, section 11.1 are used, then some addresses need to be latched for later use, two sets of incrementing addresses are required, based on a latch value, and some addresses are needed directly from the address list. All these addresses need modification to access individual sub–matrix elements, but if the sub–matrices are aligned on quad–word boundaries, this can be accomplished by directly setting the lowest two bits. Latching addresses and incrementing them (actually by four) are simple operations which can be accomplished by single chips.

A delay is required for the address of the data value which needs to be written, because of the finite calculation time and the multiple pipeline stages that are in use. It should be noted that only one sub–matrix is altered per set of three addresses, but the new destination address will appear before the previous data is completely written, so at least a two–stage latch is required. This can be acomplished by using the pipeline register. This chip provides clocked internal transfers, and the very valuable feature of access to any internal pipeline stage, so that variable delays can be used. The individual element addresses do not need to be latched, since the microcode address modifiers could be delayed in the microcode. The well defined microcode sequence, even across major instruction boundaries allow some parts of the first microcode instructions from major instructions to complete the processing of the previous major instruction, most notably controlling the memory write operations. The regularity of the instruction set is very important here.

The pipeline register chip could also be used as a dual, sixteen bit latch to hold the address of the main diagonal element and the location of the first element in the major row, which is used to reload the counter used for stepping down the main column.

Once the required operations in each microcode cycle have been defined, the microcode word can be coded. The more difficult process can then begin, of providing the correct timing for all the signals. This is not discussed here, as it is too specific, not only to the logical design, but also to the individual devices used, and to some extent, external influences such as the operating temperature.

6.6 Address generation.

The addresses which are produced by the Zollenkopf program can be output in several forms, and the elements can be arranged in memory in various forms. The Zollenkopf process works in–place, so that the solution vector elements are overwritten as they are processed, so a duplicate storage scheme suggests itself, where the memory is split into two equal parts, one of which holds the lower factor matrix and the diagonal elements, while the other holds the upper factor matrix and the solution vector, with similarity between their addresses.

If the minimal number of addresses are passed to the matrix processor, then more complex address generation is required, and a compromise must be reached. The addition of offsets inside the two–by–two submatrices is clearly easy to organise automatically, but there is some implied information in the structure of the matrix which could be used to halve the addresses which must be transferred. The 'mirror image' storage scheme outlined above only needs the addresses of the lower half of the matrix, if these addresses can be captuted, altered slightly and used in a different order for the mirror image elements when their time is due for processing. This requires extra circuitry in the scheduler and address generator, but allows the Zollenkopf routine to be made more efficient. The address table for the 734 node network can be made to be about 5 kilowords long by this method, which possess no storage problems. It would almost certainly be easier to produce all addresses in the Zollenkopf program, in their correct order, except for the addresses in the main column.

The availability of the addresses ahead of the time when the memory accesses are required, and the regular nature of the instructions permits calculations to be carried

out in advance of the data cycle itself, and also permits the cycle to be initiated slightly early to allow the address to filter through whatever buffers and control signals are required. All this is under the control of the microcode which controls the execution of the floating–point unit.

6.7 Performance of the proposed design.

The processor is capable of a clock frequency of 8⅓ MHz, which gives a peak performance of 16⅔ MFLOPs, From this possible maximum, there are cycles wasted in the division routines and in the processing of the row headers. Nevertheless, sustained useful execution rates of over 14 MFLOPs are attainable. This would solve the CEGB transmission network in about 14ms, well inside the 40ms envelope allowed to sustain real–time operation.

The processor to memory bandwidth is sufficient to transfer all the operands that the processor can produce or accept. Each processor cycle can transfer two double precision operands to the processor, and one from the processor to memory, so that 50 million operands can be transferred per second, or 400 million bytes per second. The externally visible address generation logic permits effective prefetching of operands, and also allows cycles to be started early, so that the full half cycle can be used for the memory access, which makes the best use of the memory speed.

Data transmission to and from the card must next be examined. Since the other processors in the system would have a natural bus size of 32 bits, and double precision is only required during the matrix solution, it would be better to transmit all data in 32 bit from. The conversion between single and double precision is straightforward, even in hardware in the bus interface. A bus based on 32 bit transfers can therefore be used.

About 64K words of 32 bits each must be transmitted to the processor for each iteration of the matrix. Uniprocessor backplane buses can currently run at about 12 MHz, but rates decrease as soon as arbitration is required, and about 4 MHz would be fast for a multi–processor bus [18] [42] [103] [141]. taub. Bus arbitration between bus masters is shown to be particularly problematic, particularly if this is frequently required. To achieve good performance, FIFO buffers should be used between each processor and the bus, so that if the bus is busy, the processor can continue with its own local calculations, but this requires the FIFO to have bus control signals added to it, which greatly increases

the complexity of the design. This would give a time of 16ms to transfer data to and from the card, assuming full bus utilisation. This is unlikely in practice so the transfer time would probably increase to over 25ms per iteration. This time is considerably longer than the total calculation time for the sparse elimination and substitution, which demonstrates both the problems of inter–processor communication and the processing speed of the proposed solution.

6.8 Conclusion.

A paper design has been produced which is capable of sustaining high rates of floating–point operations for the duration of a sparse matrix solution. The attainable rates were easily sufficient to achieve real–time operation for the C.E.G.B. transmission network. The memory architecture has been tailored to the type of memory accesses, and the availability of operand addresses before the access cycle commences was shown to ease the design of the memory system. The board was intended to execute the pseudo–code produced by the modified Zollenkopf routine (chapter 3), and this code was shown to be a good match for the facilities available within the processor. The construction of the board was expected to be difficult with the facilities available, in particular the problems of interference and electrical noise due to the fast switching speeds and wide buses, which are best suppressed by the use of multi–layer circuit boards. The difficulties experienced with the much slower Weitek boards highlighted this problem. The loading of the matrix and dependent vector data into the on–board memory was also shown to be inefficient, and to take considerably longer than the actual matrix solution.

Chapter 7.

Parallel Simulation.

While most of the simulator splits easily onto parallel processors, the solution of the system equations does not split simply between parallel processors. Many good algorithms have been developed for processing full matrices in parallel [34] [59], but these rely on the regular data patterns to keep processors busy and to hide most of the communication times. The regular communication patterns also permit the processors to be arranged in the form most suitable to the communications which take place.

The lack of structure in power system sparse matrices makes such methods unattractive, as processors are poorly utilised, either free–wheeling or performing useless calculations. Routing data also becomes much more difficult to determine in advance, because the data must be passed to the processors which will require that data, and the transfer of data between processors that are not adjacent requires a full message routing scheme, and increases communications overhead.

The volume of data transfer is also large. Each matrix element which must be passed between processors consists of some form of identification, in addition to the four, eight–byte words of data. Due to the location symmetry of the matrix, there would probably be a 'mirror' element, which gives 64 bytes of data in addition to any identification bytes. The identification would probably require 2 bytes, possibly per data block. This will be seen to be a non–trivial data transfer, particularly if many such blocks must be transferred, and to maintain high transfer rates, a parallel bus would be needed between each communicating processor.

7.1 Parallel Sparse Matrix Solutions.

Previous work in O.C.E.P.S. [116] on parallel matrix solution had applied the matrix inversion lemma [126] to form a set of disjoint matrices, which communicated by solving for unity right hand vectors, combining these togther in a coordination matrix which was then inverted, and used to correct the individual solution vectors for each area. J_b is the matrix of all the intra–block Jacobian terms without the contributions from the tie–lines, which are added by the T matrix. matrices U and V are used to select the columns which have tie–lines from J_b, and also can be used to expand the condensed tie–line matrix back to the size of the J_b matrix. They perform a similar operation to scatter and gather in sparse storage. They are shown here for one tie line between node k in area a, and node l in area b.

$$
\mathbf{U} = \begin{array}{c} \\ \vdots \\ e_{ak} \\ f_{ak} \\ \\ \vdots \\ e_{bl} \\ f_{bl} \\ \\ \vdots \end{array}
\begin{pmatrix}
[0] & [0] & [0] & [0] \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & -1 & 1 & \vdots \\
\vdots & 0 & -1 & \vdots \\
\vdots & [0] & [0] & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & 1 & 1 & \vdots \\
\vdots & 0 & 1 & \vdots \\
\vdots & [0] & [0] & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
[0] & \cdots & \cdots & [0]
\end{pmatrix}
\tag{7.1}
$$

$$
\mathbf{V} = \begin{array}{c} \\ \vdots \\ e_{ak} \\ f_{ak} \\ \\ \vdots \\ e_{bl} \\ f_{bl} \\ \\ \vdots \\ \\ \end{array}
\begin{pmatrix}
[0] & [0] & [0] & [0] \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & -1 & 0 & \vdots \\
\vdots & 1 & -1 & \vdots \\
\vdots & [0] & [0] & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & 1 & 0 & \vdots \\
\vdots & -1 & 1 & \vdots \\
\vdots & [0] & [0] & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
[0] & \cdots & \cdots & [0]
\end{pmatrix} \tag{7.2}
$$

$$
J = J_b + UT^{-1}V^T \tag{7.3}
$$

$$
\begin{aligned}
J^{-1} &= \left\{ J_b + UT^{-1}V^T \right\}^{-1} \\
&= J_b^{-1} - J_b^{-1}U \left\{ T + V^T J_b^{-1} U \right\}^{-1} V^T J_b^{-1}
\end{aligned} \tag{7.4}
$$

Where T is a square, diagonal matrix of tie–line parameters which is given by equation 7.5.

$$
T^{-1} = \tag{7.5}
$$

$$
\begin{array}{cccccc}
\multicolumn{2}{c}{\text{tie line 1}} & \multicolumn{2}{c}{\text{tie line 2}} & \cdots & \text{tie line 1} \\
\begin{pmatrix}
2B_1 & B_1 - G_1 & & & & \\
G_1 - B_1 & G_1 & & & & \\
& & 2B_2 & B_2 - G_2 & & \\
& & G_2 - B_2 & G_2 & & \\
& & & & \ddots & \\
& & & & 2B_m & B_m - G_m \\
& & & & G_m - B_m & G_m
\end{pmatrix}
\end{array}
$$

If F_{DN} are the state variables for each block, then:

$$J^{-1}F_{system} = J_b^{-1}F_{DN} - J_b^{-1}U\left\{T + V^T J_b^{-1}U\right\}^{-1} V^T J_b^{-1}F_{DN} \quad (7.6)$$

Defining T' as:

$$T' = \left\{T + V^T J_b^{-1}U\right\}^{-1} \quad (7.7)$$

$$\Delta S = \Delta S_{area} + J_b^{-1}UT'V^T\Delta S_{area} \quad (7.8)$$

Where ΔS_{area} is:

$$\Delta S_{area} = J_b^{-1}F_{DN} \quad (7.9)$$

This method can be seen to have much in commom with the Diakoptic method of solution [10 12 54 130], and shares many of its problems, particularly with respect to the modifications that are required to the equations which represent the system. The method has the advantage of permitting all the area processors to complete most of their calculations before having to wait for the serial routine to process its data, and also makes the ordering of each area completely independent of the other areas.

Extra computation is required to generate the information which must be passed to the central processor, since the inverse matrix of the communicating nodes is required. This can be formed by solving for each unity vector corresponding to a communication node, and passing this block to the central processor. The inverse tie–line matrix of each block then combined with those from the other blocks, solved, and the modifications passed back to each area. This method is proven to be inefficient for sparse matrices by Fong and Pottle [44].

7.1.1 Reference values for each area.

The application of this method is not transparent to the formation of the equations, because each area is considered seperate until the communication inverses are combined, and in particular, a separate solution vector is formed for each area. Each part of each island in each area must therefore have its own voltage and frequency reference if the solutions are to be stable, so some equations must be reformed to provide these references, some of which are later eliminated by the correction process.

7.2 Alternative Formulations.

7.2.1 Uni–processor matrix solution.

The parallel processor scheme assumed in chapter 6 used multiple general–purpose processors to set up the matrix for solution, and to process the results of the solution after it had been obtained. The actual matrix solution was performed on a single processor, even though it was performing multiple operations at any one time (address generation, two floating–point operations), but this was shown to involve a large volume of data transfer to the board, which would take much longer than the actual matrix solution itself.

Chapter 5 showed that two of the RISC processors could sustain the necessary calculation rates to achieve the full solution well within the target time if no memory induced pipeline stalls were generated. These would be good candidates for processors which could both generate their own matrix for the nodes assigned to them, and solve it within the time envelope. A scheme to permit this style of parallel processing was therefore investigated.

7.2.2 Parallel processor matrix solution.

If, however, the flow of information in the uni–processor case is inspected, an alternative approach can be developed. The bifactorisation technique proposed by Zollenkopf dictates that the basic operation during the elimination phase and before its own column is eliminated is given by equation 7.10.

$$a[i,k] = a[i,k] - a[i,j] \times a[j,k] \tag{7.10}$$

Over the 'life' of element $a[i,k]$, this becomes

$$a[i,k] = a[i,k] - \text{term}_1 - \ldots - \text{term}_m \tag{7.11}$$

where 'terms' are the products of the row and column entries which reference element $a[i,k]$. If these terms are independent of each other, then the order in which they are determined is irrelevant, as is the order in which the summation is performed.

Tinney and Walker [147] showed that if the matrix can be ordered with a predominantly block structure, with all connections between the blocks grouped at the right and bottom of the matrix, then there are no data dependencies between the blocks during the

elimination process, so the blocks can be processed independently. The resultant matrix is in doubly bounded block diagonal form. Because there are no connections directly between the blocks in the matrix (all inter–block connections are via the borders and the connection block in the bottom right) the elements of every block are numerically independent of each other, and there is therefore no fill in these blocks generated from the elimination of any other block. The same argument holds for the communication borders; any fill in these must come from the elements internal to the block. The communication block in the bottom right will probably be full, or almost full, because by the time it is processed, the large sparse network has effectively been condensed into a small, dense network by the previous eliminations. Fill will almost certainly be higher than in the standard orderings because of the ordering constraints imposed by the blocks.

Each of these block can therefore be processed completely in parallel with the others. The results must, however be combined into the communication block, which must be eliminated serially after all the other blocks have been eliminated. The independence of the summation order can be used to add the values passed from the seperate blocks as they are ready, without waiting for specific results to appear. The elements in the borders can be processed in the seperate processors, with only the numbers which must be summed being passed to the coordination routine. The final summed results from each area could be passed, instead of their constituents, which would reduce the volume of data transfer, but would delay the formation of the communication matrix until one area had completely finished its elimination.

A similar argument holds for processing the right–hand vectors, since their data dependencies are similar to the matrix itself. Here, the forward substitution can be performed in parallel, until the elements in the communication matrix are reached, when a switch must be made to serial solution to complete the forward substitution, and begin the backward substitution. This can proceed in parallel when the processing of the communication block is completed.

This method can be used to permit parallel processing if a suitable matrix ordering can be found. If a set of blocks can be found which has a small number of interconnections, then the border should be small, and the final submatrix small. This matrix would be expected to be relatively full, and may even be full enough to be processed more

efficiently as a full, rather than as a sparse matrix. It is therefore important to minimise the size of this matrix, as the computational effort required to solve it will tend towards the cubic power of the matrix size.

7.3 Selection of Blocks.

A method has been proposed which can be used to solve the simulation matrix on parallel processors. This method will only be applicable to the simulator if the matrix can be split into a suitable set of blocks without incurring too much penalty in the ordering. If too many elements are forced out of near–optimal order, then a greatly increased amount of fill–in will occur, which will prevent any speed increase from the use of parallel processors. This fill will occur primarily in the coordination matrix at the end of the matrix. This is doubly problematical, firstly because this matrix is solved serially, and its elimination will almost certainly form part of the critical path, while its forward and backward substitutions are by definition in the critical path, and secondly because of the non–linear growth in solution time with decrease in sparsity.

In Chapter 2, it was stated that the solution of a full matrix of size n requires calculations of order $O(n^3)$, while a sparse matrix with on average m non–zero elements per column, requires $O(n \times m^2)$. As fill–ins occur, m will increase, so the solution time will tend to increase quadratically with the number of fills. This can be limited by reducing either or both of n and m. The ordering routines were developed to minimise m, but are impeded in this by the imposed blocks. An attempt should therefore be made to reduce the size of the coordination matrix. The size of the coordination matrix depends on the number of columns which are referenced by columns in previously eliminated blocks, since each of these is moved to the coordination area. The blocks should therefore be chosen to minimise the number of these boundary nodes. The variability of the ordering methods reduces the probability of being able to determine the amount of fill–in which will be generated by a particular partitioning without actually performing the ordering on the split.

7.3.1 Linearising the block connectivity.

An alternative in reducing the number of calculations required to solve the coordination matrix is to limit the fill–in which can occur in the coordination matrix, at the expense of an increase in coordination matrix size. The method as outlined so far allows any column in any block to have a direct connection to any other column in any other block. After all the blocks have been eliminated, a less sparse network will remain, consisting of the nodes which connect to nodes from more than one area. As this matrix is eliminated, fill is possible anywhere within the matrix, which will tend to become almost full.

If a restriction were applied to the connectivity of the columns, such that the columns in any block could only connect to the columns in two other blocks, then an effectively linear network would have been produced, with a block tri–diagonal coordination matrix. This would ensure that certain regions of this matrix must remain zero, and would give more structure to this matrix than would otherwise be the case, which might be useful in reducing the number of calculations or rearranging them for faster solution. The greater difficulty in finding such partitions would be reflected in the increase in size of the coordination matrix, due to the larger number of columns which would probably have to become boundary columns.

To correctly linearise network connectivity, a further restriction on inter–block connectivity is required, in that no node can be connected directly to nodes in two different external areas. This restriction would affect the assignment of nodes to areas, but is probably best ignored during the main assignment, and either incorporated at the end, by slightly adjusting the boundaries of each area where this is possible, or by the addition of a small number of dummy nodes, one per tie–line connected to the problematical nodes. This can be made invisible to the external simulator interface.

7.3.2 Degradation in choice of blocks.

The advantage which can be gained from linearising the connectivity of the blocks will vary considerably from network to network. A tightly meshed network, such as is to be found in the C.E.G.B transmission network should be affected relatively little, particularly if it is possible to add dummy nodes to correct some important violations of the partition rules, where otherwise good split lines would have to be ignored. Networks that have

grown from the weak connection of local utilities, as is found in the United States of America, would probably be affected more, because such networks split naturally into utility–sized blocks, which will almost certainly be arranged in a non–linear fashion.

Linearisation of these networks would therefore require either many fewer processors than possible good blocks, or that the good blocks which exist are split. The former would result in a smaller performance increase due to the use of parallel processing than would be possible using other block formation criteria, while the latter would involve many columns in the coordination matrix, which would make poor use of the parallelism achieved by partitioning the system into areas. It might be possible to solve the matrix in a three–tier process, by subdividing each of the primary blocks into actual utility areas, but such subdivisions were not investigated further.

7.3.3 Partitions within blocks.

Once the blocks have been determined, it is possible to split them further according to the connectivity of the nodes allocated to each block. The general (non–linear) case could be split into two partitions consisting of nodes which only connect to other nodes in the area, and those which connect to nodes in other areas. The nodes which connect completely internally can be processed seperately, and appear to coalesce into a reduced network involving only the other nodes when viewed from the nodes in another area. The nodes completely within an area can be processed independently, while the nodes with external connections must be processed in conjunction with other externally connected nodes from the other areas.

The more restricted connectivity of the linearised network permits three partitions to be used, with the externally connected partition being split to seperate nodes which connect to the preceeding area from those which connect to the successive area. If the nodes which connect to the preceeding area are placed either last in each block, or are moved to the bottom right of the matrix, then the remaining two partitions in each block are independent of all the other blocks.

There is no benefit from seperating the wholly–internal nodes and the nodes which connect to the succeeding area during the processing of each block, because even if the succeeding nodes are processed first, values cannot be passed to the coordination matrix because fill will occur in the coordination partitions as wholly internal nodes are

processed, so the final values which must be passed are only fully determined after the last node in each block has been eliminated.

There might, however be a benefit in relaxing the block ordering of the coordination matrix, as a better sparsity order might be possible by removing this constraint, but this would also depend on the method adopted for eliminating this matrix.

7.3.4 Non–sparse processing.

The sparsity of the coordination matrix was expected to be poor, for the tightly–meshed C.E.G.B. network, and this was usually proven to be correct. The natural break between the sparse areas and the almost full coordination matrix means that this matrix could be stored and processed using a different technique to the areas themselves, as long as the basic bifactorisation algorithm is followed. The matrix would be expected to be full to the diagonal element after the first non–zero in each row, and it would probably be more efficient to process any zeros which occur in these positions than to try to pass over them. The matrix would then be viewed as a full, row banded matrix.

This makes this particular matrix much more amenable to parallel processing than the individual areas, and a transputer grid could usefully be applied to its solution. The regular structure could be used to determine operand routing and hide most of the communication times. The structure could also be processed very efficiently by a variation on the bifactorisation processor proposed in chapter 6, because with the addition of a slightly more general instruction scheduler, a looping capability could be provided, with the parallel formation of operand addresses in hardware. The high–speed processor would also perform the summations necessary to form the matrix more quickly than the other processors investigated in chapter 5.

7.4 Modification of the Ordering routine.

The method was initially derived for a serial matrix solution, with the interesting property that many of the calculations could be performed in parallel. The method does not involve the application of matrix algebra to split the matrices, so from the theoretical viewpoint, whether the solution is performed in parallel or serial after the ordering has been made is irrelevant. Small modifications of the standard serial Zollenkopf code can therefore be used to generate and test the new ordering.

7.4.1 Alteration of the Tinney–2 ordering.

The Tinney–2 ordering routine was chosen for initial modification, because of its simplicity. The Tinney–2 method, as implemented in the Zollenkopf program, has the useful property that only the columns listed in the ordering vector are inspected for ordering, while any column which is referenced during the simulation of the elimination is handled correctly, whether or not it is available for ordering. The only columns which are moved as a result of the ordering search must be within the search array, so if the search area is restricted, only these columns can be affected. The elements in all columns are created and deleted as necessary, and a valid count is maintained on its number of non–zero elements, so that when a column does participate in the search, all the information about it is correct.

The partitions can therefore be ordered by placing the columns into the ordering array in groups, corresponding to the partitions. Restricting the searches to within these partitions forces the routine to order every column within each partition before progressing to further partitions. The restriction of the searches to within the partitions means that only a column within the partition can be selected, so if the column at the current position in the ordering array is not selected, it can only be moved to another location in the ordering array which is in the current partition.

After all the nodes in a partition have been eliminated, the current position is incremented as usual, and the end pointer of the array that is used by the search routine is moved to the position of the last column in the next partition. A new sub–ordering is then started on this new partition. It should be noted that as the nodality is maintained for the whole matrix, and these values are used in the ordering selection, the ordering should be fairly optimal, within the restraints of maintaining the matrix partitions.

The only differences from the standard Tinney–2 method are the restriction of the search to a small part of the ordering array at any one time, and the updating of the section of the ordering array which is being searched.

This process can also help the speed of the ordering, despite the added complexity, by reducing the number of elements which must be scanned to select the next column. The dependency of the time for the Tinney–2 ordering on the number of nodes is quadratic, so reducing the number of nodes which must be scanned should greatly reduce the ordering time. This will be offset by the increased number of fill–ins which

the restrictions on ordering will generate, and the slightly more complex loop control. The ordering time for each partition should decrease quadratically with the number of partitions, so the overall decrease would be linear, which would be reduced by the added complexity and the increased fill.

7.4.2 Alteration of the Least Recently Used ordering.

The ease with which the partitions were introduced into the Tinney–2 ordering encouraged an investigation into whether they could be introduced into the MDLRU ordering. This was expected to be more difficult because of the more complex ordering method involved. The solution used is not as good as that for Tinney–2, but provided acceptably fast ordering times.

The queues must be re–formed with the new columns at the start of every partition, and the partitions of the altered columns must be inspected after the elimination of every column, to determine whether the column is part of the current partition, and therefore needs to be moved between queues, or whether it is currently not in any of the queues. This modification introduces more extra work than the modification to Tinney–2, and does not result in an overall speed increase, but this method was still used in the parallel Zollenkopf routine because it provides a more consistent ordering with a lower mean path length. The additional work involves the reinitialisation of the queues between each partition, and a check on each altered column to determine whether it is in the current partition, and therefore whether it's queue entry needs to be updated. The former involves only the resetting of the header entries in addition to the work involved in the uni–processor case, and the latter will find relatively few columns which are not in the current partition because the only columns which are reachable and not in the current partition are those in the coordination matrix, and there should be few connections to those from the minimisation of the number of tie–lines.

7.4.3 Special processing options.

Several options were added to the ordering so that several variations on the basic ordering could be investigated. These are shown in the title line of the plots in appendices C and D. The main options used are as follows:

Merge. This option instructs the ordering routine to merge the partitions together in the individual areas, and to merge all the 'previous' partitions of the areas together in the communication matrix. This option is almost always used, because it gives the ordering routine more freedom to control fill–in, and therefore improves its performance.

Reverse. This option reverses the areas, so that a different area is eliminated first. The main effect of this is an alteration in the fill–in pattern in the coordination matrix. It is sometimes used in the examples provided.

Path. This option instructs the program to by–pass the ordering routine and use the ordering read in from a file, for use with a path analysis derived split. This may be used with merge, in which case, only the communication matrix is reordered.

There are other options which inform the optimiser that the split is a path based split, so that the last area is assumed to form the coordination matrix, and is unaffected by any reverse command. If this is not the case, the coordination matrix is eliminated first, with disasterous fill–in performance. It is also important to prevent the ordering routine from adding dummy nodes to these splits in order to enforce linear inter–block connectivity.

7.5 Measures of Parallel Performance.

The main test for a parallel routine is to run it on the target parallel processor, and record the individual and overall execution times. This was not an option which was available, for two reasons. Firstly, the only parallel processor which was available was the VAX-6440 four processor system, which did not provide the flexibility or number of processors to try different configurations of partitions. Secondly, the compiler support, although purchased with the machine, was not installed until too late in the project to be of use. The option of moving the code to a transputer system was also considered to involve too much work in porting the code from one non–standard implementation of the C language to another.

The performance was calculated instead by executing the code on a single processor, and summing the operation counts along the critical path of the program. The critical path varies from split to split, depending on the times taken to forward substitute the

each block, and the elimination phase of the coordination block. The steps involved in producing the solution are given in table 7.1.

Parallel Solution	
Area Processors	Coordination Processor
Eliminate	WAIT for data
SEND Data	WAIT for data
Forward Substitute	Eliminate
WAIT for Ready	SEND Ready
SEND Data	WAIT for data
WAIT for data	Forward Substitute
WAIT for data	Backward Substitute
WAIT for data	SEND Data
Backward Substitute	Done
Done	

Table 7.1 Steps involved in parallel solution.

The elimination of the coordination matrix would almost always take longer than the longest forward substitution phase of any of the areas, so the critical path would almost always be Eliminate Area, Eliminate Coordination, Substitute Coordination, Backward Substitute Area. Each area can continue processing as soon as it has completed its own backward substitution, with the only remaining communication being the transfer of voltage and frequency information between areas for the reference generators in each island. The number of operations in the substitution phases are closely related to the number of operations in the elimination phase for each area, so the critical path would not be altered by allowing areas to proceed with forward substitution as soon as they have transferred their elimination results to the coordination area.

No allowance was made for communication times, but these should be relatively small if buffered parallel links or an arbitrated bus are used for the data transfers. Once the critical path has been identified, the number of effectively serial calculations which must be executed by the parallel processor can be calculated. Two comparisons can usefully be made with this figure, firstly against the total number of calculations performed by the parallel processor, and secondly against the number of calculations which a single

processor would have had to perform without any ordering constraints. The former result gives the percentage of useage made of the processors, while the second gives the speed–up of using multiple processors instead of a single processor. The normal case for the number of operations for the parallel case is given by equation 7.12.

$$\begin{aligned}\text{Critical operations} = &\lfloor\text{eliminate area}\rfloor + \\ &+ \text{eliminate coord} + \text{solve coord} + \\ &+ \lfloor\text{backward solve area}\rfloor\end{aligned} \tag{7.12}$$

7.6 Network configuration for splits.

The partitioning methods were tried on test networks, either with all their possible connections active, or with the network in its usual operational state. This should provide a good test for the partitioner and the ordering routine, since they both must break lines in order for the network to be partitioned, and most network configurations are likely to be small variations on the standard connectivity. If lines are removed from the network, this effectively reduces the lines which might have to be broken. If nodes were split, so that what in normal operation would be parallel transmission lines became connected to different end nodes, the number of lines which might have to be split would increase. An attempt could have been made to find the configuration with the densest connections, but the fully–connected case was felt to be sufficient. For the C.E.G.B networks, the 'normal' connectivity was used, because this would be encountered for most of the simulation time.

7.6.1 Effect of inactive circuits.

The possible changes in network configuration should be considered, to determine the effects that these changes would have on the simulator. Three cases can arise; inactive lines, severe islanding and node splitting. Node splitting can be handled by assigning the new nodes to the same area as their predecessors, such that no circuit elements may change areas. All models are therefore confined to the area to which they were originally assigned. Node splitting is the most troublesome modification if a node which is in the communication set splits, because this can alter the factorisation in every block in the matrix, and alter the amount and structure of the transfers to and from the coordination

block. Before a final split is chosen, trials could be performed to investigate the impact of the splitting of each node in the communication set. A line which becomes inactive will help the simulator, because the connectivity of the system has been temporarily reduced, while a line which is normally inactive, which becomes active will adversely affect the simulator, particularly if it affects the inter–block connectivity.

7.7 Production of network splits.

The parallel processing solution proposed is of little real use if the network splits badly, within the constraints of the matrix solution. The size of the networks involved makes the determination of good splits, and their evaluation, a tedious process without automation. To determine the suitability of the proposed method, many possible splits must be generated, and each must be evaluated to determine how well the simulator would perform using that split. This must be repeated for different possible processor configurations, to determine the optimal hardware for the simulator.

It was shown in chapter 4 that the ordering of the matrix is critical in determining how much fill–in occurs as the matrix is processed, and it is possible for the constraints on ordering produced by poor splits to nullify any gains from parallel execution.

There are four methods available for splitting networks into minimally connected areas. These are linear programming, optimisation, clustering and factorisation path analysis. Interest was expressed in the O.C.E.P.S. group concerning the Simulated Annealing algorithm [73] [88] for optimisation, and this and another optimiser based on Genetic Optimisation [129] [143] ideas are discussed in chapter 8. The other three methods are discussed in the following sections.

7.7.1 Linear programming.

The linear programming approach proposed by Undrill and Happ [148] passes forwards and backwards through the network. At each stage, a every combination of each pair of nodes is tried, and, for each allocation of the current node, the optimum allocation of the previous node is recorded. The next two nodes are then examined, until no more nodes remain. The minimum cost allocation of the last node is chosen for this node, which defines the allocation of the penultimate node, which in turn determines the allocation of the next prior node, until the first node is reached. A new system state

has now been defined, which is used as a base for furure trials. This process would appear to be very dependent on the node numbering scheme, and would appear to be susceptible to thrashing, but the authors claim otherwise. The more random nature of the optimiser trials were felt to give more chance of finding an optimal solution.

7.7.2 Clustering Methods.

Cluster analysis [21] [124] [125] tries to find dips in a curve showing the number of connections between nodes which have already been examined, and those which have not. When a dip is found, all the examined nodes and their connections are removed from the system, and the process starts again on the remaining nodes. A seed node is initially defined, and nodes adjacent to this are examined one by one, so the cluster grows. As a node is examined, its connections to examined nodes are subtracted from the connection total, and its connections to unexamined nodes are added to the total, and its unexamined neighbours are added to the list of nodes to be examined. When the neighbours of the seed node have been exhausted, neighbours of the neighbours are examined, until either a dip is found, or some other criterion is matched. One such criterion could be a restriction on the maximum size of a cluster, which would force the algorithm to look back along the curve and try to select the optimum compromise between connectivity and block size. It is also possible to ignore dips until a certain block size has been reached.

Cluster analysis was discarded because it seems to be very dependent on the seed nodes which are used to start the identification of each cluster, and the ordering of the nodes in the adjacency tables for each node. Unless these are varied between runs, some possible configurations would be impossible to find, because the configuration is wholly defined by the seed node, neighbour ordering and dip selection criteria.

7.8 Splits based on the Factorisation Path.

The production of factorisation path diagrams to aid the understanding of the ordering algorithms presented in chapter 4, prompted the investigation of another method of splitting the matrix into areas. Use has been made of path trees for sparsity preservation and the allocation of tasks to processors [11 13 15], and for improving the speed of convergence [22]. The factorisation path of a column gives the list of all columns which would be affected by the elimination of that column. A column could be affected directly, through being referenced in the column, or indirectly by being referenced by one of the columns in the path of the first column. Any column not on the path of a particular colunm can be processed in parallel to it, until their paths join. The factorisation path is described more fully in chapter 4 with reference to orderings attempting to maintain sparsity while minimising path length.

The path diagrams of all the ordering methods showed a natural tendency of columns in the matrix to group predominantly into their geographic areas during ordering, in a similar way to the splits found using genetic optimisation. If a path diagram could be found with paths which contain an approximately equal number of columns, then this would be a candidate for splitting the network between processors. The paths should ideally have a short common stem, and then branch out quickly, because any columns in paths which are shared by different areas have to be processed serially. If the paths branch quickly after the common stem, then few extra columns are placed in the communication matrix above those used by every area.

None of the path diagrams for the standard (non–randomised) orderings discussed in chapter 4 produce good splits for more than eight areas, with GF–1 producing the most fragmented diagram. The common stem length for this ordering, would however, be excessive at about 23 columns for the 234 node network, which is a tenth of the columns, and between one quarter and one half of the matrix elements. This large coordination matrix would severely limit the possible speed–up, and may even slow the solution compared to the uni–processor case.

The path diagrams for the 234 node network were examined, and three were selected as having the desirable properties of a short common trunk, and evenly populated, short main branches. These three diagrams were used to allocate nodes to areas for the 234 node network. These were the GF–3, MDLRUML and MDLRUMLA orderings. The

partitions generated from the analysis of the paths were used in two ways; firstly, the original ordering within each block was maintained, and secondly, the MDLRU ordering was applied to each block and the communication matrix in a similar fashion to the ordering after Genetic Optimisation.

The first method keeps the nodes in the same relative order to each other within each block, and unless the merge option is applied to the coordination matrix, the number of fill–ins and the operation totals must be the same as for the uniprocessor case. This provides a check on the splitting method. The second case required adjustments to the MDLRU algorithm to make it recognise block boundaries correctly, if they were presented with or without the cut–set nodes being allocated to a different area. In particular, dummy nodes were not added to these splits to force a linear, inter–block connectivity.

The analysis of factorisation paths is not as flexible as the optimisation methods because it is not possible to weight either nodes or the connections between them to reflect the amount of computation involved with a particular node on the one hand, or the desirability of making a particular connection a tie–line. The former would allow better load balancing between the processors, because the circuit elements which do not alter connectivity, such as loads and compensators, but which require computation can be included in the partitioning. Generators can be handled by both methods, because they contribute terms to the matrix which are attached to only one node, and are therefore unlikely to be torn into a different area by the elimination ordering. They would be candidates for the optimiser if they were included as simple nodes, so a nodal weighting should be applied to the hostnode instead. The weighting of tie–lines would enable good break–points to be indicated, and also penalise some lines so that they would not be broken, which might be useful if stability became a problem.

It is also not possible to apply the soft constraint of linearising the area connectivities, or impose other constraints which might be valuable in the investigation of improved splitting criteria. Partitioning the network using the elimination path tree does, however, take fill–in into consideration, which is shown to be too computationally expensive for the optimisers to consider in chapter 8.

7.9 Automation of path analysis.

The output of the optimisation programs can be passed directly to the ordering analysis routine. This is only possible because the optimiser is completely automated, so to make the analysis of the factorisation path competitive, a method is needed to automatically partition the path tree into suitable blocks, and label these for processing. It would be possible to display the tree graphically, and use human intervention to select the cut points, but the large number of nodes is very difficult to represent using the restricted resolution of current graphics displays.

The method used to display the trees does not make the most efficient use of the available resolution, but it does represent the tree in a form which is easy to visualise. The method counts the number of terminating branches, and divides the vertical size by this number. The diagram is then started with the root node(s), with each node at each level being centred in a space, the size of which is the sum of the terminal branches which are to the right of it. The nodes are scanned in the tree order, so no paths have to cross. It would be possible to pack the nodes more efficiently, but this requires much more complex algorithms, and makes it less easy to understand the diagram.

The path tree diagrams produced by this method for the 234 node system tax the resolution of current PostScript laser printers; 300 dots per linear inch in both directions, which compares to a graphics screen with a total resolution of 1000 by 800. On such a screen it would not be possible to display node numbers, which removes one aid to combining areas.

An automated method for splitting the path tree should try to minimise the number of columns in the common trunk, and try to equalise the number of columns in each sub–tree. It is possible to prune any sub–tree and move it higher, as long as all the the columns in its original path are still in the path after the move. Small sub–trees can therefore be combined with other trees in the same branch, which can be used to equalise the totals in the main sub–trees, or remove the small sub–trees and thus reduce the number of areas. This provides considerable flexibility in determining splits, but this flexibility must be used carefully if the common trunk is not to grow too large.

The full automation of splitting the path tree has so far not proved possible. This seems to be one of the tasks for which the human eye and brain are much better

than computer analysis. If one could be produced, then the method would probably be significantly faster than network optimisation, because fewer trial solutions would have to be assessed. It is doubtful whether real–time operation could be achieved, but run–time partitioning would be a possibility.

7.10 Results from the path analysis.

The paths for the 234 node network, which are presented in appendix B, were inspected to find which diagrams displayed the required features; a small common trunk, short branches, and well balanced major branches. Three paths were selected as being the most promising, those of GF–3, MDLRUML and MDLRUMLA. These were manually split, and the splits were entered into the computer using the graphical interface of the Genetic Optimiser. The nodes in the trunk were added as an extra area, to ensure that they would be processed last, and the parallel ordering routine was modified so that the restriction on area connectivity was relaxed. A further option was added which allowed the matrix to be processed following the NSEQ ordering which generated the path tree. This keeps the matrix elements in the same order relative to any node to which they are connected, but allows the nodes to be formed into their blocks. Two results are presented for each split; this ordering, and one obtained by applying the MDLRU ordering to the blocks defined by the path tree. The former should produce the same number of fill–ins, and operations, as the uni–processor case, since the overall ordering is unaffected by the split, unless the merge option is used on the connectivity matrix. The latter option may be better or worse, depending on how the MDLRU performs on the matrix blocks.

The results which were obtained for these three paths are presented in table 7.2 and diagrams are shown in appendix C. Each path was tried with several different numbers of areas, and the best speed–ups were selected.

A similar investigation was made using two 118 node paths, the MDLRU and the MDLRUMLA. These were identified as being the most balanced path trees in the diagram. The speed–ups are identical, and the trees were almost equivalent. The results, in table 7.3, are slightly inferior to the split produced by Sasaki, Aoki and Yokoyama [126], but the differences between the splits were small.

Name	path	N–P	Operations		Time	Speed
			Total	Critical	%	%
GF–3	path	5	5392	1871	43	235
GF–3		5	4926	1723	39	256
MDLRUML	path	5	4518	1513	34	291
MDLRUML		5	4532	1526	35	289
MDLRUMLA	path	5	4446	1326	30	332
MDLRUMLA		5	4470	1343	30	328

Table 7.2 Path splits for the 234 node network.

Name	path	N–P	Operations		Time	Speed
			Total	Critical	%	%
MDLRU	path	3	579	1513	37	271
MDLRU		3	579	1513	37	271
MDLRUMLA	path	3	579	1580	37	271
MDLRUMLA		3	579	1580	37	271

Table 7.3 Path splits for the 118 node network.

7.11 Comparison of Partitioning methods.

Two methods have been developed and have been shown to produce good, usable splits for the parallel solution of the matrix equation. A comparison of the splits produced by the two methods should be made, so that the strengths and weaknesses of the methods can be examined. The development of two methods which produce very similar results by two independent processes gives a great deal of confidence that good results were achieved. Before the MDLRUMLA path was tried for the 234 node network, so many splits with a speed–up of around 2.8 were found, that it was believed that this was the global optimum. The exceptional result from this split proved otherwise, and no other split has been found which comes close to this speed–up.

Although both optimisers found very similar clusters; and found the same parts of the country difficult or impossible to break, such as Central London, Wales, and South Yorkshire, the actual cuts were very different. The cost function of the optimiser

penalised area total mismatches and the number of cut lines, while the path analysis minimised the cut–set nodes and performed area equalisation as a subsidiary process. This will result in the communication matrix being small and full, while the optimiser tries to make it sparser, but it will be larger due to the increase in cut–set nodes. The slightly superior results from the path analysis indicate that a reduction in size is better than an increase in sparsity. The optimiser should therefore be tried with a cost function which minimises the cut–set, instead of the number of tie–lines.

The path analysis also incorporates the fill–in into the split, while this was far too computationally expensive for the optimiser to consider.

7.12 Speed of the parallel solution.

The best splits found produced a three–fold speed increase for between five and seven processors, with the exceptional result of a speed increase of 3.3 for six processors. The speed increase would be expected to decrease to between 2 and 2.5 when data transfer times are included. An efficiency of usage of processor power of about 50% was achieved for the matrix solution, which would appear to be a poor return for increasing the number of processors, but this figure is for the most difficult part of the solution process to split. The remainder of the simulator would be expected to show almost a linear speed increase with additional processors, which would improve the overall speed–up of the complete time–step. This is because there is almost no interrelation between the areas until the matrix solution is underway, so each processor is almost entirely independent during the set–up and post–process phases of the solution. This contrasts with the single matrix processor, solving a matrix produced in parallel, which was proposed in chapter 6, because only the terms in the communication matrix need to be passed between processors if each processor solves its own area.

It was estimated that the matrix solution required approximately half the time required for each iteration of the solution on a single processor, with the remainder being taken by setting up the matrix equation, and processing the results of the solution. Equation 7.13 shows how a typical overall speed–up is related to the individual speed–ups from the two main parts of the iterative solution process. It is assumed that four processors take part in the area calculations, and that one processor controls them and solves the serial

part of the solution. The matrix solution speed–up is two–fold, while the remainder is assumed to show a four–fold speed increase.

$$\text{Time} = \text{Time}_{\text{matrix solution}} + \text{Time}_{\text{setup}} \tag{7.13}$$

If each part is assumed to take 50ms serially

then processing in parallel on five processors will take:

$$\begin{aligned} &= 0.5 \times 50 + 0.25 \times 50 \\ &= 37.5ms \end{aligned} \tag{7.14}$$

compared to 100ms serially.

This shows that the overall speed–up is almost three–fold, for the use of five processors, which is over 50% efficiency.

7.12.1 Comparison with other research.

The speed increases obtained with this method are similar to, or better than, those obtained by other research. Most work has shown a reletively small speed improvement for the matrix solution with the use of parallel processors. Use of the Cm* architecture [46] [77] shows a speed–up by a factor of between two and three [32] [33], although the parallel simulation of a single transmission line is not a representative system.

A more general study [58] claimed that a linear speed increase was attainable, but the hardware design is unrealistic, requiring multiple buses per processor to multiple banks of shadow memory to provide a large, *n port* register file. The predicted speed–ups take no account of memory conflicts caused by the update of these memories. This is of vital importance, because every processor must have the most recent data, and the more processors which exist, the more frequently new results must be written into every memory bank. Chapters 5 and 6 of this thesis showed that a uni–processor could completely saturate an advanced memory interface using the fastest (non ECL) memory devices available, so the write–back of results from several processors would leave no time for the processor to read data from the memory. The paper also assumes that as networks are split between more and more processors, the number of *cut set* variables does not increase; i.e., that making more splits in the network neither increases the number of nodes attached to tie lines, nor the number of tie–lines themselves. The paper

does, however, confirm the belief that the Newton–Raphson convergence is unaffected by the parallel processing of the solution.

Abur [1] obtained a three– to four–fold speed increase for the 118 node network, but used between eighteen and fifty–seven processors, which is very inefficient. larger systems are used, but consist of contrived replications of the 118 node network, requiring over two hundred processors. If this number of processors is considered necessary for a five–fold speed increase, it might be better to return to analogue computers, if only to save space.

Yu and Wang [160] [161] claim better speed–ups through the use of data–flow methods on a hyper–cube, and even spend considerable time incorporating communication deleys into their models. Unfortunately, the premise on which this analysis is based is incorrect, as, correctly, an average transmission time from any node, to any other node in the N cube is shown as $0.5 \times T \log_2 N$, where T is the time to transmit from one node to a neighbour. This is correct for a single transfer, but the solution of sparse matrices is transfer limited on a single processor, and a data passing architecture will saturate with transfers, even if a direct link is available between every processor pair. As soon as several simulataneous transfers are required, conflicts will occur, which make the communications network appear even more heavily loaded than it actually is, by increasing the average time for a transfer. The network will soon slow dramatically, invalidating the predicted speed increases. This method also uses large numbers of processors.

7.13 Conclusions.

Three methods for producing splits of two representative power systems were investigated, with an optimiser based on Simulated Annealing performing poorly, with very erratic results for the larger matrices. The other two methods, path analysis and Genetic Optimisation performed well and independently produced very similar splits, which gives a great deal of confidence that the splits produced were good. Most of the best splits were so similar, both in geographic layout and performance, that they were felt to be the global optimum splits for the networks, until the analysis of one of the paths produced by a variant of the new ordering method proposed in chapter 4 produced a

split which was greatly superior to the other splits. No other splits have achieved similar performance gains.

The splits differed slightly due to the differences between the cost function used by the optimiser and the ordering rule used to produce the elimination order, and hence the elimination tree. The optimiser tried to equalise the number of nodes between areas, while minimising the number of inter–area tie–lines, which should equalise the calculations between each processor, and restrict the size and hence number of calculations to solve the communication matrix, which must be solved serially. The aim of the path analysis depends on the ordering method used, but the most successful orderings tried to minimise fill–in and eliminate as many nodes as possible between the elimination of connected nodes. This had the effect of making the path tree wider and flatter, which reduced the number of nodes in the common trunk, and hence the serial part of the solution. Equalising the processor workload was done by eye, with only approximate, and in some cases poor, equalisation possible. The results indicate that the optimiser cost function should be changed to reduce the number of cut-nodes instead of the number of tie–lines.

The results show that the new restriction imposed on the connectivity of the areas to which nodes are assigned has little beneficial effect for the C.E.G.B. networks, and has a detrimental effect on the splits for the more radial American network. This restriction cannot be applied to the path analysis, and the performance of the path splits was equal to, or slightly better than the optimised splits, so a small coordinayion matrix would appear to be preferable to a sparse, slightly larger one.

The three–fold speed–ups which were obtained for the matrix solution alone were promising, but a relatively poor return on the five– to seven–fold increase in processor power was achieved. It should be remembered, however, that the matrix solution would only be expected to need about half of the total time per iteration on a single processor, and that, due to the almost entirely independent nature of the remainder of the calculations, these should show a linear speed–up with increased number of processors. This gives better overall speed increases than those achieved for the matrix solution alone, with over 50% efficiency for six processors, producing a three–fold speed increase. The speed–ups were as good as, or better than those achieved by other research using a small number of processors, and rivalled results from much larger networks of processors.

Chapter 8.

Optimised Partitioning.

The parallel processing solution proposed in chapter 7 is of little real use if the network splits badly, within the constraints of the matrix solution. The size of the networks makes the automation of the splitting process desirable, particularly during the many trials required to match the simulator hardware to the chosen split. It was shown in chapter 4 that the ordering of the matrix is critical in determining how much fill–in occurs as the matrix is processed, and it is possible for the constraints on ordering produced by poor splits to nullify any gains from parallel execution.

There are three methods available for splitting networks into minimally connected areas; optimisation, clustering and factorisation path analysis. Optimisation is the subject of this chapter, while the other two methods were discussed in chapter 7. Interest was expressed in the O.C.E.P.S. group concerning the Simulated Annealing algorithm for optimisation, and it was felt that network tearing would be a suitable test of its capabilities. Joint work was therefore started on coding the optimiser and testing its performance for this task [73].

8.1 Optimisers.

Optimisers are programs which attempt to produce an optimal solution for a problem, based on a set of cost rules for aspects of its state. This involves the generation of new, trial states, the evaluation of their cost, and the decision on whether the new state is suitable for future use. Optimisation is usually progressive, with the search for the optimum being based on the current optimum solution which has so far been found. When a more suitable solution is found, this replaces the current state and is used as the base for future searches.

The method employed to generate these new states varies widely between optimisers [26], and is influenced by the problem type, its granularity and constraints. Optimisers which are used on problems with continuous cost functions and continuous variables can evaluate the slope of the cost function and move the solution state in the direction of steepest slope, until all slopes are zero. For multidimensional problems, the vector of these partial derivatives $\partial C/\partial x_i$ shown in equation 8.1 is used to give the slope.

$$\begin{pmatrix} \partial C/\partial x_1 \\ \vdots \\ \partial C/\partial x_n \end{pmatrix} \tag{8.1}$$

Lagrange Multipliers [52] are used for constrained problems of this type, where the minimum cost is located where this vector is perpendicular to the constraint boundary.

For problems where either the permitted states or the cost function are discrete, this is difficult to apply, and more random methods are used to generate new trial states. The spliting of a network falls into this category, as the nodes and the areas to which they are assigned are discrete.

The evaluation of the cost of a solution uses a cost function, which assigns numerical weightings to important aspects of the system state, which are summed to produce the overall system cost. Cost functions are usually written so that a minimum cost is optimal, which makes most optimisers minimisers. The cost function is clearly problem specific, but due to its frequent evaluation, should be as simple as possible while still representing what is desirable or undesirable in a solution.

The decision phase would at first sight seem to be the simplest, with only a yes/no answer required, based on the cost produced by the cost function. The new solution can only be the same, better or worse than the old one, so a decision should be simple. This is usually not so, as a major problem in optimisation is the avoidance of local minima in the search for the global minimum. A local minimum is a point which is surrounded by points with higher cost, but which is itself not the minimum cost which the system can achieve. Other points also cause problems, and an analogy with three dimensional surfaces is useful to explain what types of points are problematical, and why.

A three dimensional surface, such as a landmass, can have slopes, flats, domes, saddles and depressions. Of these, flats occur rarely in good cost functions, because they indicate sets of adjacent states which are indistinguishable by cost, but the others can

be common. These all have zero slope in all directions at at least one point, which can trap an optimiser if its permutations do not take it outside the locality of the point. Depressions, or local minima are obviously the most problematic, since all surrounding points have higher cost, so the permutations would have to be larger to escape, because they would have to move the current state to one with a lower cost than the local minimum.

An optimiser can use two methods to escape from local minima; it can either hope that a permutation will eventually move the solution to one with a lower cost directly from the local minimum, or it can accept some changes which increase the cost of the solution, in an attempt to escape. A simple yes/no criterion for choosing whether to accept the new state is therefore undesirable and rarely used. Care must be excercised when accepting such changes, because they can rapidly undo the good that the optimiser has performed, if they are used too often. Watch–dog monitors can be added so that if a better solution is not found after a certain number of trials, the solution reverts to the local minimum, which is used as the base for further trials. This can trap the solution near a local minimum, but will also prevent the solution from drifting away from a good solution.

8.2 The Network Problem.

The critical part of the simulator for partitioning purposes is the matrix solution, and the cost function should therefore weight the factors which most influence the suitability of a split to these algorithms. Ideally, actual operation counts and counts of memory locations consumed would be used to cost each split, but the size of the networks make this too computationally intensive for realistic run times, because each trial configuration would require a trial factorisation. Simpler criteria must be found, which predict the performance of a split to a reasonable degree of accuracy.

Parallel processing requires that inter–processor data transfer should be minimised, that no processor should be idle while another is working, if this can be avoided, and that any serial phase should be as short as possible if other solution phases depend on its results. Reducing idle times is clearly a matter of equalising processor workloads, which in its simplest form, could be a count of the number of nodes which are assigned to each processor. More complex schemes could be used, where weightings are given for

generators, loads, lines etc., which are connected to each node, and as these do not change during a run of the optimiser would add little to the computational cost of the optimiser. The optimiser was kept simple, because it is simpler to test the method with simpler models.

The time taken to process the coordination sub–matrix depends on the initial sparsity structure, the amount of fill–in and the number of columns in the sub–matrix. The structure and sparsity are not known unless the factorisation is simulated, so only the number of columns is directly available to the cost function. This is the number of receiving (or sending) nodes connected to tie lines (where line direction refers to a link to a higher processor area). A different measure could also be used if the number of tie–lines is counted. This gives a mixture of the size of the matrix and its sparsity, and also gives some indication of the number of terms which must be transmitted to the coordination matrix. It was therefore used in preference to the number of nodes in the cut–set. No information is available about memory usage for each area, but this is assumed to be primarily dependent on the number of nodes in each area, while in reality, it would also depend greatly on their sparsity and structure, which could be incorporated by counting the transmission lines at each node and weighting the nodes accordingly.

The above discussion leads to a cost function involving two quantities; number of nodes per partition and number of lines between these partitions, of the form shown in equation 8.2.

$$\text{Cost} = f(\text{nodes}, \text{tie lines}) \tag{8.2}$$

The cost function is still not fixed, however, as there are many ways in which these quantities could be combined. Most cost functions are additive, whereby each factor makes its own contribution to the total, which are then summed to give the final total cost. Most cost functions also weight these contributions, and may raise them to a power, to compensate for their relative size and stiffness. If this is not done, a small relative change in the number of nodes allocated to an area would swamp any change in the number of tie lines since there should be many more nodes than tie lines. This leads to a cost function of the form of equation 8.3.

$$\text{Cost} = f(\alpha \times \text{nodes}^{\alpha_p}, \beta \times \text{ tie lines}^{\beta_p}) \tag{8.3}$$

The term representing tie lines is simple to calculate, as it is the number of tie lines in the system. But how should the nodes term be calculated? Should this be the largest number of nodes assigned to a processor, the difference between the largest and the smallest numbers so assigned, or the total of the totals for each area? This last case is clearly ridiculous, as it is just the total number of nodes, unless the program is faulty and looses nodes. It could become useful if the individual totals are raised to a power before the summation, as this performs a calculation similar to variance. This is preferable to the other methods, which ignore any changes unless they alter the number of nodes assigned to the areas with extreme sizes, while many beneficial changes can be made that affect only the distribution of nodes between tha areas which are not extreme. The extreme cases would still be favoured by the variance method, because their changes would receive higher weightings. Costings which only inspect the extreme sizes would also fail if two areas have the same extreme sizes, because even when one is improved, the cost function will remain unchanged because of the other will retain the same extreme value.

The power to which the totals are raised alters the sharpness of the cost change due to the size of the participating areas. Given a cost function of the form of equation 8.4, and that a group of δ nodes is transferred between two of the areas with initial node totals μ and ν, without affecting the number of tie–lines, then the following possibilities arise for different integer powers to which the node totals are raised before summation. If the area totals are cubed, and an over–large area looses or gains nodes, then this will involve the difference of two large cubes, while if a small area undergoes the transfer of the same number of nodes, this would result in the difference of two smaller cubes, which is smaller. The cost function would therefore encourage the reduction in size of the larger area, and the growth of the smaller, thereby tending to even out the sizes of the areas. This results in a similar nodal cost function to Undrill and Happ [148], but their cost function did not attempt an explicit reduction in the number of tie–lines.

$$\text{Cost} = \sum_{i=1}^{areas} \alpha \times \text{nodes}^{\alpha_p} + \beta \times \text{tie lines}^{\beta_p} \tag{8.4}$$

Case 1: Linear power.

$$\Delta\text{Cost} = ((\mu + \delta) + \nu) - (\mu + (\nu + \delta)) \tag{8.5}$$

$$\Delta\text{Cost} = 0 \tag{8.6}$$

There is no cost change.

Case 2: Quadratic power.

$$\Delta\text{Cost} = ((\mu + \delta)^2 + \nu^2) - (\mu^2 + (\nu + \delta)^2) \tag{8.7}$$

$$\Delta\text{Cost} = 2\delta(\mu - \nu) \tag{8.8}$$

which will be positive if $\delta > 0$ and $\mu > \nu > 0$, and is linear with respect to δ.

Case 3: Cubic power.

$$\Delta\text{Cost} = ((\mu + \delta)^3 + \nu^3) - (\mu^3 + (\nu + \delta)^3) \tag{8.9}$$

$$\Delta\text{Cost} = 3\delta(\mu^2 - \nu^2) + 3\delta^2(\mu - \nu) \tag{8.10}$$

which will be positive if $\delta > 0$ and $\mu > \nu > 0$ and quadratic with respect to δ.

8.2.1 Introduction of constraints to the optimisation.

A further investigation was made to determine the effect of restricting the connectivity of the areas to two. This effectively breaks the network into slices, and gives the coordination matrix a block tri–diagonal form. The tri–diagonal form is desirable as it effectively limits the fill–in pattern in the communication matrix, and gives some usable structure to this bottleneck process. Two methods are available for introducing this into the optimiser; hard or soft constraints. A hard constraint would reject any solution which contained an area which linked to three others, regardless of its cost, while a soft constraint would permit such a state, but make it unattractive by allocating a large penalty to it. A soft constraint brings several advantages, and this approach was adopted. A hard constraint places restrictions on the available states through which the optimiser can pass in its attempt to find the global optimum, and also limits the initial state to one which does not violate the constraint. In order both to prove the optimiser, and not to influence to solution by the initial state, random initial states should be possible. By using a soft constraint, the importance attatched to the violation of the constraint can also be varied, because a solution which only slightly violates the constraint might be better than one which does not, as it might be possible to insert dummy nodes to remove the violation.

The soft constraint used was to multiply the number of lines between non–numerically adjacent areas by a scaling factor before adding them to the number of tie–lines between numerically adjacent areas. This total was then raised to the appropriate power and

scaled. This weights the undesired lines, and gives a smooth transition to the original unpenalised cost function, so that very small weightings can be used if needed. In particular, the unpenalised function is regained by making the γ weighting unity. The cost function therefore becomes equation 8.11:

$$\text{Cost} = \sum_{i=1}^{areas} \alpha \times \text{nodes}^{\alpha_p} + \beta \times (\text{tie lines} + \gamma \times \text{far tie lines})^{\beta_p} \qquad (8.11)$$

Generally the values of α_p and β_p in equation 8.11 should be fixed by the relative influence of the properties on the actual solution, but with some cost functions the values need adjustment. Here, the solution time is expected to grow with approximate order of $O(\text{nodes}^{1.4})$ [40], and $O(\text{lines}^{2.5})$. The latter term is derived from the relatively full coordination matrix, which requires n^3 calculations to solve if full, and less if sparse. The former constant, which comes from the performance of the Zollenkopf routines, was increased slightly due to the lower relative sparsity because of fewer nodes being in each area, with most of their connections remaining intact. This constant was incremented by one because a constant of unity would not differentiate between different area totals due to the summation method used (equation 8.6). A quadratic power gives an approximately linear differentiation between area sizes (equation 8.8), so a power of approximately 2.4 would be appropriate. Experience showed that the optimiser performed well when both quantities were cubed, because it provided enough non–linearity in the cost function to differentiate between changes affecting the extreme values and the others, while still permitting some degradement of these extreme values in order to escape form local minima.

The value for the multiplier γ must be fixed to reflect the relative undesireability of far tie lines. A value of unity treats these lines as normal tie lines, so this multiplier should be always be greater than or equal to one, because these lines should never be more desirable than a local tie–line. A value of three was found to be most suitable, so that one far tie line is worth three ordinary tie lines, or, from another viewpoint, if a far tie–line replaces a local tie–line, then the cost increase is double the addition of another local tie–line.

Typical values which were used for these parameters are given in table 8.1.

Typical cost changes at or near equilibrium can be calculated using the weightings given in table 8.1, and are presented in table 8.2. The first gives the cost change for a single

Typical Weighting Factors	
α	4.0
β	40.0
γ	3.0
α_p	3.0
β_p	3.0

Table 8.1 Typical Weighting Factors used in the Optimisers.

Cost Changes near a Good Minimum	
ΔC_{node}	2800
$\Delta C_{\text{tie line}}$	66280
$\Delta C_{\text{far tie line}}$	138320

Table 8.2 Cost changes near a good minimum resulting from these Weighting Factors.

node changing area without altering the number of tie lines, when all areas initially have the same number of nodes. The second case is when a local line becomes a tie line without affecting the area totals, and the third case is for a non–local tie–line becoming a local tie–line, again without affecting the area totals.

Once the cost function has been defined, it can be integrated into the remainder of the optimiser. Two optimisation methods were attempted; Simulated Annealing and Genetic Optimisation.

8.3 Simulated Annealing.

This method distinguishes itself from other optimisers by the statistical model used in the decision mechanism. An attempt is made to simulate the phase transitions which occur when a liquid is cooled through its solidification temperature [89]. Initially, if the temperature is high enough, all the substance will be liquid, and in a constant state of change. As heat is removed from the liquid, small parts will tend to solidify as crystals, and will therefore become fixed. Each crystal orientation is coded differently, and different orientations do not combine together well, resulting in relatively little energy being required to effect a phase transition. The crystals can and do remelt, if sufficient

energy is present to permit the transition to a state with higher specific energy, as the temperature of the solid and surrounding liquid is the same. The statistical nature of the energy distribution determines which parts will solidify and which remelt.

The optimiser tries moving each constituent to a new state, and evaluates the cost change that would result, which is presented as an entropy value. If the entropy of the change is negative, then the change has resulted in a part of the system liberating energy and becoming more structured, while, conversely a positive entropy change indicates the absorbtion of heat energy, and a decrease in order. Clearly, liberation can always occur, while absorbtion can only occur if enough heat energy is available. This availability is defined as an inverse exponential function (equation 8.12), known as the *Metropolis Criterion* [88] involving the temperature and the size of the entropy change required.

$$P_{(\text{Accept})} = \begin{cases} 1 & \Delta E \leq 0 \\ e^{-\frac{\Delta E}{T}} & \Delta E > 0 \end{cases} \tag{8.12}$$

Examination of equation 8.12 shows that for high heat and small entropy increase, the probability of acceptance is approximately unity, while for low heat and large entropy increase, it is small. As heat is being removed from the system, the probability of accepting a state change from solid to liquid decreases, so that slowly, solidity and order will prevail. A transition of a node to a crystal orientation similar to that of its electrical neighbours will produce more order, and will reduce the number of tie–lines and therefore will probably therefore result in a cost reducion.

The method relies for success on determining the correct starting energy level and cooling rate. A sufficient initial heat is required so that virtually every change is accepted, while if this is too high, time is wasted on many, purely random changes. The cooling rate should be set so that sufficient time is allowed during the critical solidification phase, while still producing a final solution in a reasonable time. If the cooling rate is too fast, then defects will be frozen into the structure, as the changes required to escape from the defect involve an improbably large entropy barrier. If the cooling is still more rapid, then quenching will occur, which will result in many non–optimal areas, but this can be useful at late stages in the optimisation, to prevent the solution wandering away from the optimum one.

Once an area has solidified with reasonably low entropy, it will probably remain solid unless a modification is found which reduces its entropy still further, as any other

changes would probably require the entropy to be increased by an improbably large amount. This is seen as a collention of nodes all assigned to the same area, with no tie–lines between them. The method will therefore tend to preserve or improve good sections of the system. Parts of the system adjacent to these settled areas will tend to solidify too, if they can effectively merge with the solidified area, so this would encourage groups of nodes belonging to the same area to cluster together, and merge into larger clusters. Less well organised areas will tend to have higher entropies, with less dramatic changes due to small modifications, and so are more likely to be in flux, with more chance of finding a lower state of entropy.

This optimiser uses a simple permuter to generate its trial states, based on the *Monte Carlo* [88] method, which takes each node in turn and evaluates the change in entropy that would result from moving it to a random new area. A decision is then made using the Metropolis Criterion (equation 8.12) to determine whether the change should be accepted or not, before proceeding to the next node. This is repeated until either the heat energy in the system reaches zero, or virtually no changes are accepted. For low temperatures, very few entropy increasing changes will be accepted, and if the algorithm has performed well, then the system should be in a low entropy state by this stage, so the probability of finding an entropy reduction will also be low. Even if the solution state is considerably non–optimal, it will almost certainly be at a local minimum, so few changes would be accepted, either up or down in cost.

8.3.1 Implementation.

The optimiser as described was coded in the C programming language. The cost function incorporated variable weights, and the constraint on locally–connected areas was in–place, but could be nullified by adjusting its weighting. The structure of the optimiser is given in figure 8.1.

One modification was made to the optimiser to enhance speed. Because the optimiser modifies only one node per evaluation of the cost function, a simplification in the calculation of the cost is attractive. This would save a considerable amount of time, because many such small changes are required per iteration, and many iterations for the total solution. Instead of re–calculating the system cost from scratch, an incremental

```
BEGIN
Initialise
WHILE temperature not minimum
    FOR each node
        Generate new area for node
        Evaluate cost change
        IF increase in cost
            Obtain random acceptance level
            Evaluate acceptance probability
        IF cost change is accepted
            Make change permanent
            Update cost
    NEXT node
    Update temperature
    EXIT IF very few changes
END WHILE (temperature)
END
```

Figure 8.1 Structure of Simulated Annealing Optimiser.

method (equation 8.13) can be used whereby the new cost is calculated from the old cost and changes from the previous state.

$$\text{new cost} = \text{old cost} + \Delta\text{nodes} + \Delta\text{lines} \tag{8.13}$$

The movement of only one node between evaluations of the cost makes this calculation simple. The area totals of the current and previous areas change, and the number of tie–lines changes by the difference between the number connected to the node in its original area and in its new area. These are both simple calculations. If multiple permutations were permitted, then difficulties arise over lines being counted twice, which can invalidate the adjustment if an incremental approach is used.

8.4 Results for Unconstrained Optimisation.

The optimiser was tried on test networks, either with all their possible connections active, or with the network in its usual operational state. This should provide a good test for the optimiser and the ordering routine, since they both must break lines in order for the network to be partitioned, and most network configurations are likely to be small variations on the standard connectivity.

The optimiser was initially used without the adjacency criterion on tie–lines on the I.E.E.E. 30 node test network, and produced good results. Examples of the splits are given in Irving and Sterling [73] This network was chosen for the first runs because of its small size, so that the splits could be validated by independent human calculations. It is clear that the optimiser has correctly identified good clusters and the weak links between them.

Results for the larger 118 node network were not as promising, with more erratic behaviour evident. It would be expected that the solution should become better with slower cooling rates, as this allows more time for defects to be removed before being frozen in, but although a trend in this direction was evident, the results did not fit a trend line. Behaviour was highly erratic between runs with different random number seeds for the same cooling rate and starting temperatures. The high initial temperature effectively makes the start random, since just about any move is initially permitted. This behaviour did not bode well for the larger, more tightly meshed British networks for which partitions were desired.

An attempt was made to improve performance by making the cooling rate dependent on the number of successful changes made during the previous pass through the network, with slower cooling for many changes, and faster cooling if few changes were made, so that a promising solution would be frozen to prevent the drifting of the final result. This produced a small improvement in the results, but erratic behaviour was still evident. Such an improvement would be expected, from the identification of the trend line between final cost and cooling rate, but few accepted changes could also be an indication that a local minimum had been found. The results for the 734 node C.E.G.B. transmission system were much more erratic than for the I.E.E.E. 118 node system. No node identification was available for this network, so it was not possible to visualise the splits and thereby see how good or bad they actually were, and how they were failing.

8.4.1 Results for Constrained Optimisation.

The optimiser completely failed when the soft constraint on the inter–area connectivity was activated by increasing the weighting from unity. It failed by completely emptying most of the areas, usually leaving just two areas occupied, but occasionally for larger numbers of initial areas, several disjoint pairs of areas would remain. Examination of the permutation process and the cost function reveals the reason for this unfortunate behaviour.

The cost function effectively penalises the outer two areas, as these have only one numerically adjacent area, while all the remainder have two. Given a random flat start, there will be approximately the same number of nodes in each area, and on average each area will have the same total number of tie lines, and also the same number of tie lines to each other area. The average cost of each tie line for the end areas is however, larger than the cost for the other areas, as one more of its sets of tie lines is counted as being non–local. For n areas, most areas have $n-3$ out of their $n-1$ sets of tie lines as non–local, whereas the end areas have $n-2$ sets of non–local tie–lines. The average cost of being a node in these areas is therefore higher than being in one of the other areas with two neighbours, so the end areas tend to be emptied. This process is countered by the cost increase of having the other areas larger than necessary, but aided by the fragmentation introduced by the random start and the trial movement of single nodes.

When the end areas have been largely emptied, the probability of a node being moved into them having more connections to that area than to other areas will decrease, due to the smaller size of the area, so fewer nodes will be moved into the end areas. Nodes within these end areas will lose more and more of their intra–area connections, as other nodes are moved out, so pressure will increase to vacate the area as well. As more nodes are moved out of the end areas than are moved back in, the adjacent areas will develop an extremely biased set of tie–lines, so the process will repeat, this time emptying these areas, then the next pair, until only two connected areas remain.

8.5 Genetic Optimisation.

Genetic Optimisation [129] [143] is also an attempt to simulate nature's methods in arriving at an optimal solution. This method however simulates the random modification of genetic information as it is passed from generation to generation, and either survival of the fittest or selective breeding to decide which of the new states are suitable for keeping for future development.

The system state is represented as a strand of data, analogous to the way information is represented in DNA. The information in DNA is stored in coded form, with the position of the information determining its meaning and influence, i.e., whatever value is in a certain position in the strand is used to determine part of a characteristic, there is no information in the strand which determines what effect a particular value will have apart from its position. For network partitioning, each localtion in the strand contains the area number to which the corresponding node is assigned. This matches the natural interpretation of individual values according to their position. There can be one or many such strands in the parent population, each representing a different state with different properties and cost.

To form new system states, a variety of permutations are performed at random on some of the set of current strands. Most of these permutations are similar to those occurring in nature. For example, DNA strands being brittle can break and reform in different configurations, thereby swapping small segments of the strands, or reversing the order of the information bits, and therefore their meanings.

This description of one of the basic permutations highlights two of the main advantages of genetic optimisation over simulated annealing, one of which is that several characteristics can be altered by one permutation, and therefore the cumulative cost change for several changes is considered, instead of the individual stages which must otherwise be costed. The other advantage is that changes can be made to the system state without altering the overall composition of the system. This allows nodes to be moved between areas without affecting the area totals, so that distributional changes can be made unimpeded by any cost changes involved in passing through intermediate stages which would otherwise be necessary.

The method places no restrictions on the number of permutations which differentiate each child from its parents, on the number of parents from which each child can draw genetic material, or the number of children which are produced in each generation before a selection is made on which should survive. There is also no restriction placed on whether parents should survive for several generations until they are bettered by one of their offspring, or whether they should only last for one or a limited number of generations. By varying these parameters, a whole spectrum of optimisers can be formed, ranging from single parent, single child with one permutation, to many parents cross–producing many children, each of which has undergone several modifications from its parents.

In order to bring more randomness into the optimiser, other types of permutations were provided, such as that individual values can be set to new random values, which will alter the overall composition of the state and not just the distribution of nodes between areas. This will therefore modify the area totals, and will also alter the distribution of nodes, as it is not possible to alter totals without changing the area to which some nodes are assigned. Examples of the permutations which were used are defined in table 8.3.

8.5.1 Program structure of Genetic Optimisation.

The structure of the program that implemented the genetic optimisation algorithm is given in figure 8.2.

8.5.2 Segment selection.

The segments are chosen by selecting a starting node at random, and then randomly selecting a segment length, based on a skewed normal distribution. This was chosen as some reduction in the probability of selecting large segments was desired. A normally distributed random number can be generated by the application of equation 8.14 [145] to two uniformly distributed random numbers u_1 and u_2 in the range $(0, 1)$. This is altered to equation 8.18 to produce a skew distribution with the integer random numbers provided by the C library rand( ) function call.

$$\text{random number} = \sigma\sqrt{-2\ln(u_1)}\cos(2\pi u_2) \tag{8.14}$$

Removing the cos() term to introduce a bias.

```
BEGIN
Initialise
Plot graphics
FOR each generation
    FOR each child
        Generate permutation
        Perform permutation
        Evaluate cost
        IF cost is reasonable
            Remember it
    NEXT child
    IF best cost is reasonable
        Make it new parent
        IF best cost is best
            Dispaly new graphic
            Print update
NEXT generation
Write result state to files
END
```

Figure 8.2 Structure of the genetic optimiser.

$$\text{random number} = \sigma\sqrt{-2\ln(u_1)} \tag{8.15}$$

Incorporating the $\sqrt{2}$ into σ, this simplifies to:

$$\text{random number} = \sigma\sqrt{-\ln(u_1)} \tag{8.16}$$

For 32 bit integer random numbers $(0, 2^{31} - 1)$ and

since $\ln(b) - \ln(a) \equiv -\ln(a/b)$

$$\text{length} = \sigma\sqrt{\ln(2^{31} - 1) - \ln(\text{random})} \tag{8.17}$$

$$\text{length} = \sigma\sqrt{\text{constant} - \ln(\text{random})} \tag{8.18}$$

If the segment length would cause the segment to overflow the data strand, then new lengths are chosen until one is generated which fits within the strand. In the case of data segment exchanges, the location of the second segment starting position is modified if the length would cause an overflow.

This selection scheme is not ideal because the probability of each node being chosen for modification is not uniform. Nodes with very low numbers would be altered less frequently on average than nodes with higher numbers, with the effect being restricted to nodes with numbers less than approximately twice the variance of the distribution. A node could be modified either by being selected as the start node, or by a previous node being selected as the start node, with a length that would include the node in the segment. The probability that a node would be altered is given by equation 8.19

$$P(\text{node}_n) = P(\text{start}_n) + \sum_{i=1}^{n-1}(P(\text{start}_i) \times P(\text{length} \geq (n-i))) \qquad (8.19)$$

$$P(\text{node}_n) = P(\text{each node}) \times (1 + \sum_{i=1}^{n-1}(P(\text{length} \geq (n-i))) \qquad (8.20)$$

The probability of each node being chosen as the start node, and the length distribution for each node are assumed to be uniform, but the number of nodes before the node in question will clearly vary. The skewness of the distribution means that the effect will be most evident for the first few nodes, up to about half the variance, but will be noticeable up to about twice the variance.

This could be turned to advantage with the 234 node network by starting the length at the chosen node and working backwards. This would tend to choose the *out of place* nodes less frequently, while choosing the remainder of the *in place* nodes more frequently. This was not implemented, as the results obtained without this change were satisfactory, and it would not be suitable for the general problem, which could be solved either by allowing segments to *wrap around* the end of the strand, or by periodically rotating the elements in the strand, so that each element takes its turn near the start of the strand.

8.5.3 Additional permutations.

Additional permutations to the basic three of segment reversal, exchange and random setting were added, where run–time study of the optimiser showed that these would be advantageous. Permutations were added to assign every node in a segment to the same, random new area, and to increment or decrement the areas to which nodes are assigned, thereby moving a segment of nodes between areas. A special set of permutations were added to improve performance when the restriction on local connectivity was applied, which manipulated the contents of the whole data strand based on the assignment of

nodes to areas, instead of node numbers. Permutations could globally rotate the area numbers by one in either direction, or swap the contents two areas. These changes would only rarely be useful, and hence only rarely would they be successful, but could rescue the optimiser from a poor initial clustering. The permutations used are described in table 8.3.

8.5.4 Cost Function.

The cost function remains the same as that used for simulated annealing, but due to the more complex permutations of genetic optimisation, no efficient simplifications of the calculation are possible, so the complete cost must be re–evaluated each time. The cost function from equation 8.4 is repeated in equation 8.21 for convenience.

$$\text{Cost} = \sum_{i=1}^{areas} \alpha \times \text{nodes}^{\alpha_p} + \beta \times (\text{tie lines} + \gamma \times \text{far tie lines})^{\beta_p} \qquad (8.21)$$

Initial trials were made to determine the best settings for the configureable aspects of the optimiser, and the best performance was obtained with one parent, many children per generation, each with one permutation from the parent. Multiple parent populations tended to converge to just small variations on one theme, and the extra effort and space required to maintain multiple parents did not appear worthwhile. A more sophisticated replacement algorithm might improve performance with multiple parents, but no such method was tried. The large changes which can result from one permutation usually made one permutation sufficient, and many children gave the simple replacement strategy in the decision phase more choice.

8.5.5 Decision Phase.

The decision phase was simple, selecting the best child to become the new parent if its cost was lower than that of its parent. The best child would also be accepted if its cost was less than a given percentage above the best result found so far in the run. A value of about 20% was found to be suitable to limit the degredation, and this was combined with a watchdog which reset the solution to the best found if no improvement occurred in a fixed number of further generations.

Permutation types	
Permutation	Function
Exchange	Select two segments of similar length and swap their positions. This will only change the distribution of elements, unless more than one parent is used.
Reversal	Select a segment and reverse the order of the elements within, keeping the segment in the same place in the strand. This change is purely distributional.
Random	Select a segment and set each of the elements within to random new values, with a separate random new value for each element.
Increment	Select a segment and perform a modulo increment on each element within. A modulo increment adds one to each element, except for an element with the maximum value, which is set to the smallest value.
Decrement	Select a segment and perform a modulo decrement on each element within. This is the opposite to modulo increment.
Set	Select a segment and set each element within to a random new value, with each element using the same new value.
Negate	Select a segment and perform a modulo subtraction the maximum permissable value value for each element. The elements with the maximum value receive the minimum value, those with a value in the middle change relatively little.
Blocks	The next two permutations will only have an effect when far–tie–lines are penalised, because they operate on the whole strand, and it does not matter which areas are adjacent if the constraint is not applied.
Block–1	This permutation rotates the block allocations in either direction, performing a modulo increment or decrement on the whole strand.
Block–2	Selects two areas and swaps their elements in the whole strand.
Block–3	Chooses an element at random, and builds a segment of data of random length, or less, of elements allocated the same value, which are reachable, progressing via electrical connections, from the base element, without passing through an element which is allocated a different value.

Table 8.3 Premutations for the Genetic Optimiser.

8.6 Results for Genetic Optimisation.

This optimiser performed well, generally producing good splits. For this problem, a single parent, producing multiple children, each with a single modification was found to be most suitable. Multiple parents tended to converge to just small variations on a single theme, and the extra effort in maintaining them did not appear useful for this problem. A single modification per child gives a greater chance of isolating beneficial modifications, and many children gives a better chance of moving quickly towards a good solution with few generations.

Run from a random start, the optimiser quickly identified clusters, and combined these together to form groups of nodes suitable for allocation to individual processors. The clusters initially formed from mainly consecutively numbered nodes, but the final solution did not appear to follow the numbering scheme of the network. The numbering of the 234 node system was fairly logical (figure 8.3), but contained several out of place nodes at the end of the strand which resulted from oversights in the original numbering. These nodes proved difficult for the optimiser, as it is based on modifying small groups of nodes with adjacent numbers, but clusters formed around them. Their main contribution was to distort the cost function, and a simple routine was added to move these nodes to the area most common among their neighbours, thereby minimising the number of tie–lines. This proved satisfactory. The numbering scheme used for the I.E.E.E. 118 node test network shown in figure 8.4 is not so logical, but the optimiser coped with this network without any problems. It seems that only widely scattered, out of place nodes are problematical.

The optimiser was immune to the problems encountered with the constrained cost function with the simulated annealing optimiser, and although when end areas were occasionally emptied, they were usually re–established by a small cluster, which then grew to be of similar size to the remainder of the clusters. The exception to this was the search for partitions of the 234 node network using between five and seven areas, which almost always resulted in four connected areas. Eight areas seemed stable, while above eight involved some thrashing, but areas were only rarely deleted. The behaviour for five to seven areas indicates that no good solutions exist for this network and cost function, with the extra cost of the necessary tie lines being greater than the cost penalty of having fewer larger areas.

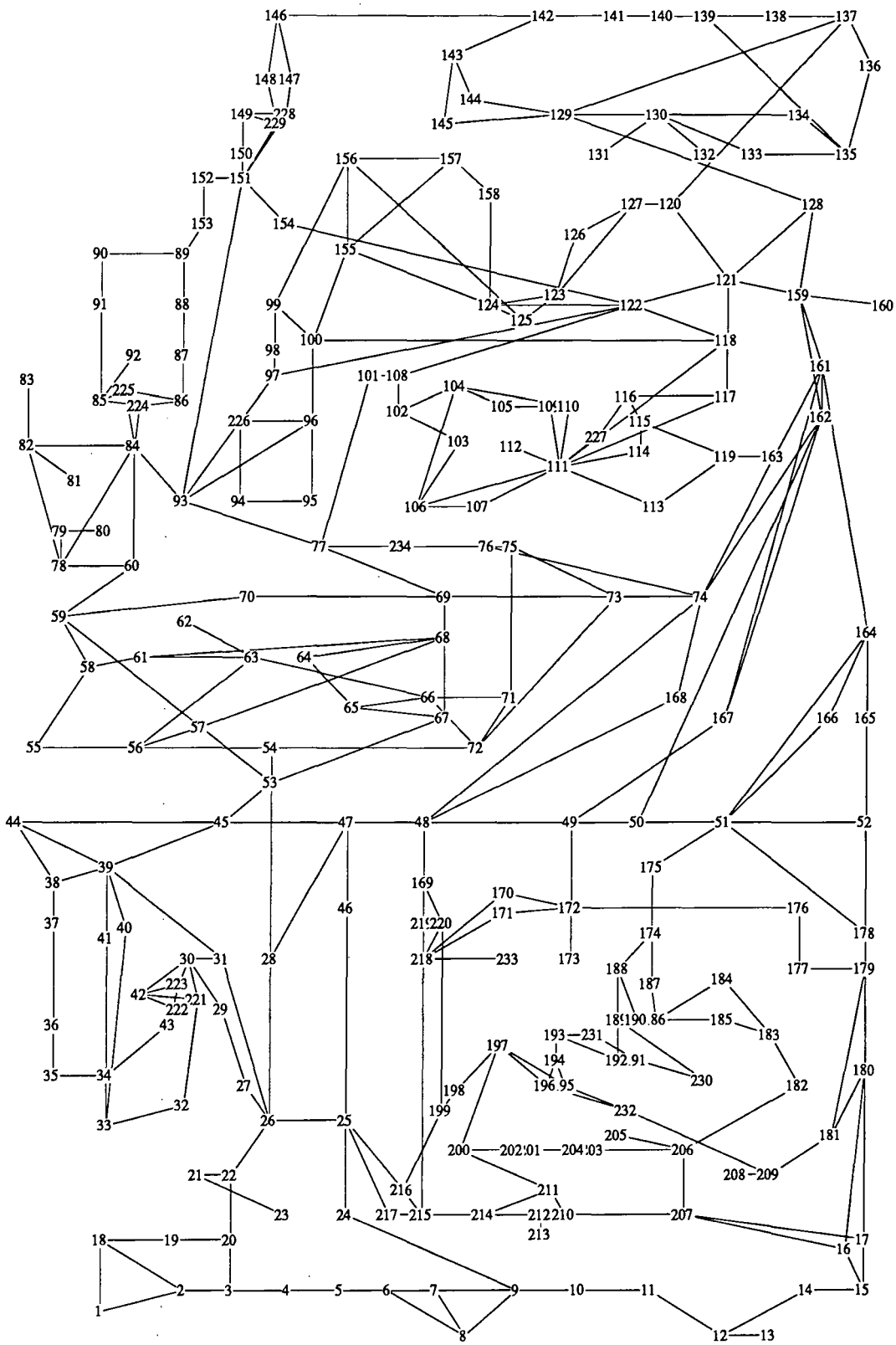


Figure 8.3 Numbered 234 node network diagram.

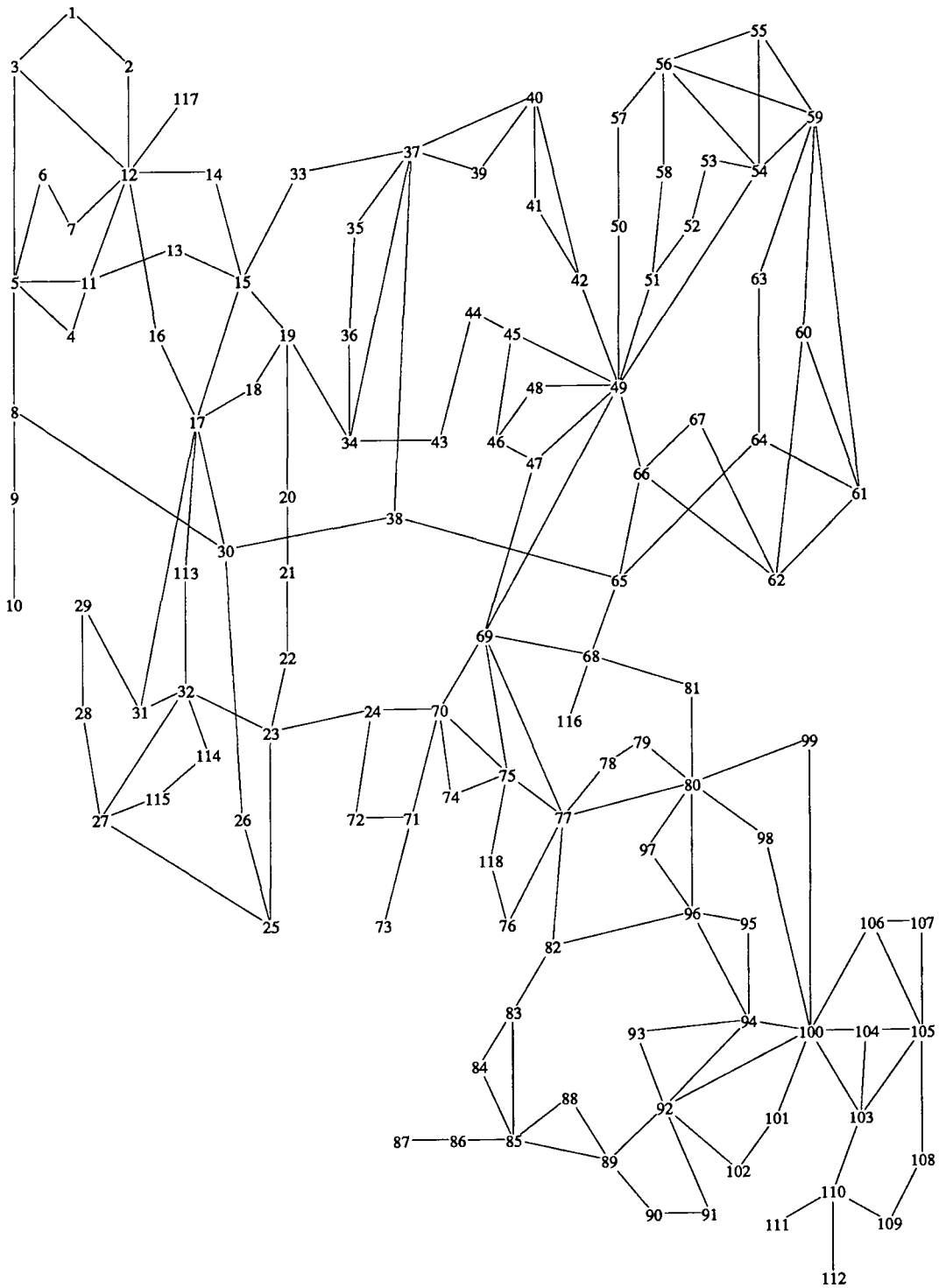


Figure 8.4 Numbered 118 node network diagram.

The I.E.E.E. 118 node system was handled well, with no such problems with any split up to 10 areas, although some areas were reduced to very few nodes, and were clearly senseless for a real simulator.

Diagrams for the 234 node network are presented in appendix C, while those for the 118 node network are presented in appendix D. Each split is represented by a diagram of the matrix, and a plot showing the allocation of the nodes to each area. Beneath each matrix is a summary of the performance of the split, which shows the total number of operations, and a break–down of the number of operations in each area. The counts under the 'current' heading show the operations which are required for each area, and those under 'previous' show the contribution that each area makes to the processing of the coordination matrix. The calculation of the speed of the parallel processing follows the scheme described in chapter 7. The summary line gives the number of operations in the critical path, compared to the number required for the new MDLRU ordering on a single processor. This total is compared to the critical path operations, and the speed–up is given in the final figure on the summary line. A good split should have matched totals under the current column, and a small total in the previous column, and will have a large (two plus) speed–up.

N–P	Operations		Time	Speed
	Total	Critical	%	%
1	4406	4406	100	100
2	4514	2394	54	184
3	5148	2185	49	202
3	5284	2323	52	190
4	4458	1559	35	283
4	4476	1574	36	280
4	4756	1678	38	263
4	4614	1595	36	276
4	4674	1739	39	253
10	7017	4180	94	105

Table 8.4 Genetic Optimiser 234 node results.

Results for the 234 node network are summarised in table 8.4, some of which are drawn from appendix C, while others are from my paper at UPEC 90 [143]. The speed–up and time figures are relative to the MDLRU ordering for a single processor, and indicate the predicted improvement for a multi–processor. The speed–up is best for a four area split, which gives a 283% speed–up. This figure takes no account of communication times, and is for the matrix solution only, so the matrix speed–up would be less than this figure, but the overall speed–up would probably be greater, due to the increased parallelism between the parts of the remainder of the simulator. Few splits were stable for between four and eight areas, and the poor performance for the ten areas shows that a smaller number of splits is better for this network.

N–P	Operations		Time	Speed
	Total	Critical	%	%
1	1570	1570	100	100
3	1772	693	44	227
3	1852	796	50	197
3	1614	568	36	276
4	1942	814	51	193
4	1752	622	39	252
4	1630	562	36	279
4	1786	688	43	228
10	1782	651	41	241

Table 8.5 Genetic Optimiser 118 node results.

Similar results are presented for the 118 node network in table 8.5, and here a similar maximum speed–up is achieved. The performance for ten areas holds up well, but there is no improvement in speed from the four–area splits. Four area processors again appear to be optimal for this network. The split for the 118 node network from Sasaki, Aoki and Yokoyama [126] was also input, and was found to be almost identical to several of the path based splits. It gave a speed–up of 2.8, which was very slightly better than any of the splits found by the optimiser, which produced a best split of 2.79. This was a hand–optimised split, so presumably many splits were tried before this was found. The optimiser has almost found the best split for this network.

8.7 Application of results.

Because of the limits of the cost functions which were used, the actual results from the parallel Zollenkopf factorisation routine were expected to be more variable than the final cost of the solution would suggest. This proved to be the case, but the cost function did enable the optimiser to find some very good solutions, particularly for the more difficult 234 node network.

The best split found for the 234 node network, gives a speed increase of 283% over the single processor technique when five processors are used, one of which processes the coordination matrix. This produces a saving of 65% in the time taken to solve the matrix equation of one iteration of the simulator. The good performance is due to the generation of very few fill–in elements during the factorisation, in addition to those produced by the uniprocessor case, which indicates that the split interferes little with the ordering routine. The optimiser has therefore found a good, natural partition for the network. The corresponding matrix highlights the sparseness of the coordination matrix, and the general lack of fill–in elements in the areas themselves. The network diagram shows the geographical areas which result from this split, the boundaries of which follow clear breaks in the system. A more detailed discussion of the results was presented in chapter 7.

8.8 Graphical Interface and User Interaction.

Experience with Simulated Annealing showed a need to be able to present the final state of the optimised network partition graphically, so that the partition produced could be rapidly assessed. From this, the obvious next step is to present the current state of the network graphically while the optimiser is running, and the next logical step is to provide the user with the ability to alter the current state. The provision of the X-11 network graphics system greatly eased the user interaction, as it provided mouse support, reasonably fast graphics and a limited toolkit of user interaction objects.

The first requirement was to display the nodes at fixed, known positions and alter their colours depending on the areas to which they had been assigned. For this, the networks had to be digitised, which was simple for the 118 and 30 node networks, but impossible for the 734 node system because no geographical data was present. A 234 node system

was used instead, simplified from a C.E.G.B. network diagram. This provided a much better network for both optimisers, as much of the chaff was removed from the 734 node network, while still retaining much of its overall connectivity. Lines connecting the nodes were drawn directly between node centres, and the node positions were slightly adjusted manually to avoid poor line angles or unnecessary crossings, while retaining the overall geographical layout. The nodes were displayed as coloured squares, with colour representing the different areas. The colours were user definable. Representing the split in this way removes the requirement of clearing and redrawing the whole image between displays, because only the nodes change colour, and overwriting a node with a new colour completely obliterates the previous colour. If different shapes are used, then this is not the case.

Once the display facility was provided, the X–toolkit routines were used to provide a mouse–driven interface to the optimiser, which allowed the optimiser to be paused, and the system state modified in several ways. The user could select individual nodes and change their area, he could select all nodes assigned to an area or all connected nodes assigned to an area and change them to a new area. An extra area in addition to those used by the main optimiser was provided to permit swaps. The state could also be saved during an edit, which could be used in conjunction with a 'restore previous state' command to undo actions with unforseen consequences. Information about the currently selected nodes is displayed, as is the current system cost.

This viewport on the optimiser proved invaluable for showing how the optimiser progressed between states, and what extra permutations would be beneficial. It also provided the opportunity to intervene in the optimisation to correct blatent non–optimalities, and to perform *'what if'* trials on the optimiser's results or on human generated splits. Because the optimiser produced output files which were directly readable by the parallel ordering routines, this graphical facility was also used as a convenient method of entering the splits obtained from analysis of factorisation paths, and provided a valuable visualisation of these otherwise abstract paths. The figures showing the geographical splits of various partitions were produced by a companion program which could read the optimiser output files and which output the diagrams in forms suitable either for direct printing or incorporation into a presentation graphics package. Shapes were used to represent the node allocation, with or without colour, so that monochrome reproduction was possible, and because fast redrawing was not required.

The similarity of the optimisers allowed the interface to be added to the simulated annealing optimiser with little work, and the contrast in operation of the optimisers was fascinating. The graphics clearly showed the crystallisation of the solution, and gave a good indication of the nodes which are not strongly fixed in any of the areas, as these are the only nodes whose entropy changes are small enough to permit them to change area at low temperatures.

8.9 Parallel Optimisation.

Examination of the genetic optimiser reveals that it is an almost ideal candidate for parallel execution, with no interdependence between the children of one generation apart from their parents. Most of the optimiser's time is spent creating new children and assessing their cost, so a near linear speed increase would be expected, until the delays in transmitting the cost and information required to generate the child round the system become significant for larger processor arrays. The most suitable arrangement for the processors would be a grid, to facilitate fast sorting of the best children.

The program structure for each generartion is shown in figure 8.5, where **CP—** refers to a control processor action.

Each processor produces a group of children and evaluates their costs, the best of which is transmitted to the processor's nearest neighbours. Each processor then compares its best result (initially that of its best child), with the results which it has received from its neighbours, and selects the best. This process repeats until the network has settled, which should be in half the maximum dimension of the grid. The processor which still retains its own child as the best, can now transmit more information about that child, to enable the remaining processors to reproduce it and use it as a new parent. Table 8.4 lists what information must be transmitted for each permutation type.

Due to the random nature of the permutations, it is not possible to transmit just the permutation type and segment(s) involved, because each processor must have an independent random number generator. This only affects the assignment of nodes to random new areas, as all other permutations involve either data movement or fixed values. The data requirements are shown in table 8.4. If only one parent is used, then no information about the parent needs to be transmitted, but if multiple parents are used, then both the parents used, and the parent to be replaced must be passed.

FOR each generation

CP—Initialise

PARALLEL

Generate new generation

Evaluate costs of own children

Transmit own best cost

Receive and transmit best costs

SEQUENTIAL

CP—Identify best cost

CP—Transmit verdict

Transmit or receive information to form child

PARALLEL

Form best child

Make it new parent

CP—Update files

CP—Update graphics

NEXT generation

Figure 8.5 Structure of parallel version of genetic optimisation.

Permutation Information Transmitted by Parallel Optimiser to Form New Child						
Exchange	Parent	Start	Length	Parent 2	Start 2	
Reverse	Parent	Start	Length			
Set	Parent	Start	Length			
Invert	Parent	Start	Length			
Increment	Parent	Start	Length			
Decrement	Parent	Start	Length			
Random	Parent	Start	Length	$Area_a$	... ...	$Area_b$
Block-rotate	Parent	Direction				
Block-swap	Parent	Area 1	Area 2			

Table 8.6 Information which must be transmitted to form new child.

To improve performance still further, the results could be passed to a seperate processor which could perform the file output and display the graphics. This would free the remaining processors to continue the search for an improved solution.

8.10 Conclusion.

Two optimisers have been developed for the purpose of splitting a transmission network into several parts for parallel processing. The optimiser which was based on Simulated Annealing produced increasingly erratic results for larger systems, and failed completely when a constrained cost function was used. The optimiser which was based on Genetic Optimisation performed much better, and was almost always stable. It handled the difficult, constrained cost function, and produced many usable splits, with a maximum predicted speed–up of 2.83 over a uni–processor solution.

Both optimisers simulate a natural optimisation process, with the slow solidification or crystalisation being simulated by the Simulated Annealing Optimiser, and the alteration of genetic information between generations due to the breaking and rejoining of DNA strands, by the Genetic Optimiser. The optimisers were, however, very different, with the Simulated Annealing Optimiser relying on evaluating many small changes, and using a random exponential acceptance gate to accept or reject cost increases.

The Genetic Optimiser altered the allocation of groups of nodes at one time, and also provided purely distributional changes, which left the area totals unaffected. Both of these provide direct routes between states that an optimiser forced to pass through the individual states en–route, would not find, due to high cost in one of the intermediate states. This optimiser was considered to be much more suitable for network tearing than the Simulated Annealing Optimiser.

A graphical interface was developed for both optimisers, and this proved invaluable in monitoring the progress of individual optimisations, and for human intervention. It gave an insigth into the intermediate states of the optimisation, and helped to develop a better set of permutations for the genetic optimiser. It also gave an immediate idea of the split.

Chapter 9.

Topology Determination.

9.1 Introdction.

A power system consists of many constituent parts, many of which should be modelled by a simulator. The degree of modelling required is different for each component type. Components such as transmission lines require detailed numerical models, while circuit breakers can be treated as logic elements. If these were modelled numerically, then the large difference between values for these and the circuit elements could cause numerical problems due to the limited precision which is used for the calculations. The purpose of topology determination is to reduce the complexity of the current system configuration, and to create a description of this in a form which is useable by the remainder of the simulator. Much of the work in topology determination has been concerned with the identification of the actual network structure [48 80 127 139], for the use of the control package. This involves slightly differentinput data than what is available in the simulator, but the overall aim is the same.

A power system can be conveniently split into elements which are internal to a substation, and those which are not. A substation will generally consist of busbars, circuit breakers, isolators, protection devices, possibly transformers and connections to the outside world. These connections could be transmission lines, transformers, load points, compensators or generators, and would almost certainly be attached to a busbar via a switch, (i.e., a circuit breaker and pair of isolators). For topology purposes, the elements wholly internal to a substation can be split into three types; busbars, switches and transformers. Those external can be split into lines, generators, switches and others, which do not influence either connectivity or activity. Lines connect between switches

on nodes, which are formed from connected sections of substations, and generators provide electrical energy to activate a connected group.

Connected groups of nodes are termed 'islands', which are said to be either active or passive depending on whether an active generator is connected to one of the constituent nodes. Because an island without generation (or other power feed) is virtually useless, it is usual to drop the active when referring to an active island, but to stress passive when referring to a passive island.

Due to the differences in scale between internal and external elements, it is usual to consider all connected internal elements to be at the same potential, or simply coalesced into a single point, which is viewed externally as a node. Routines which process elements individually, such as protection on an individual element, still differentiate between elements, but other routines just need the node data. The topology routines must therefore determine a minimal switch network for each substation, and a mapping from real breakers and isolators to these switches. Given these switches, the connectivity of each substation must be determined. Any resulting collection of busbars with an external connection to either a line or generator, must be kept as a possible node, while any collection without an external connection can be ignored.

Since external elements connect to these nodes, some form of node numbering scheme must be adopted. Simple schemes based on creation time or substation order are in use, but can involve re–numbering on topology changes, which ideally should be avoided. These schemes are required for matrix routines which require a full leading diagonal, which is provided by sequentially numbered nodes. A better approach, where subsequent routines permit, is to number the nodes according to the first busbar in the relevant substation. This assures that every node can be numbered (by definition, each node must contain at least one busbar), and avoids renumbering unmodified nodes. This method uses more memory for node tables than simpler methods, but provides more implied information.

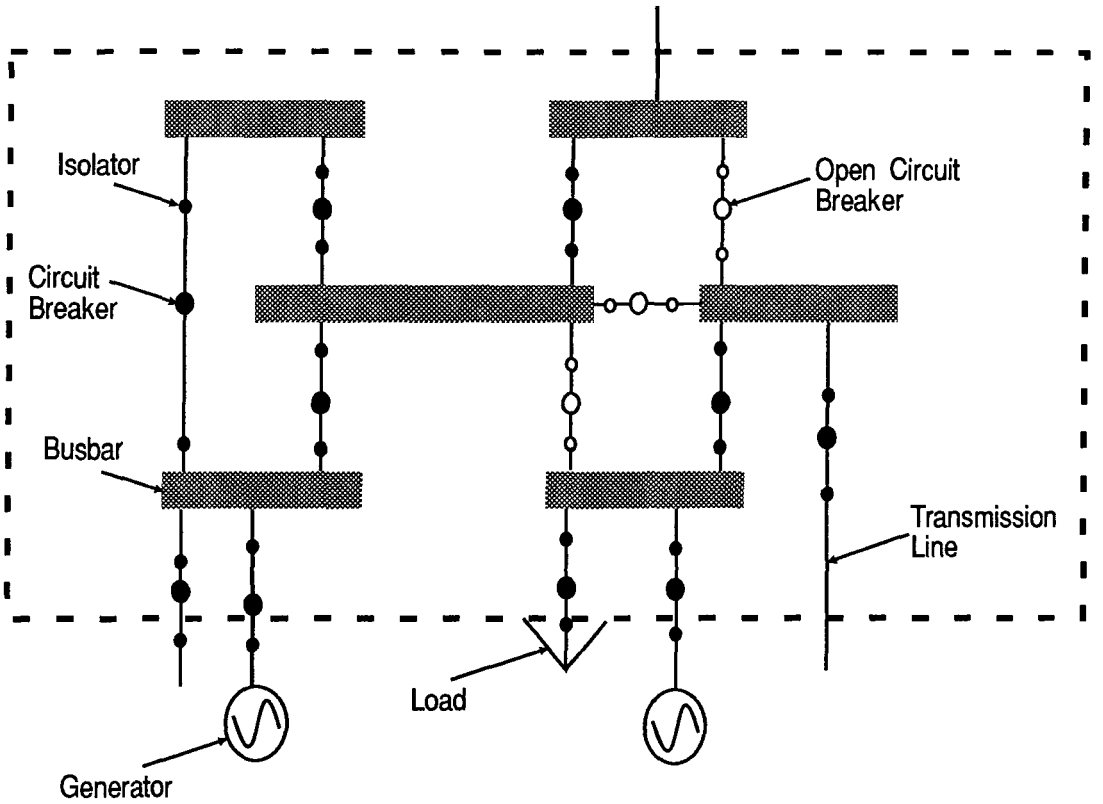
As external elements connect to these nodes, these must be updated when a substation has changed configuration. Lists must therefore be kept for each busbar, listing what connects to it. Preventing unnecessary node renumbering reduces these adjustments.

9.2 Internal Configuration in a Substation.

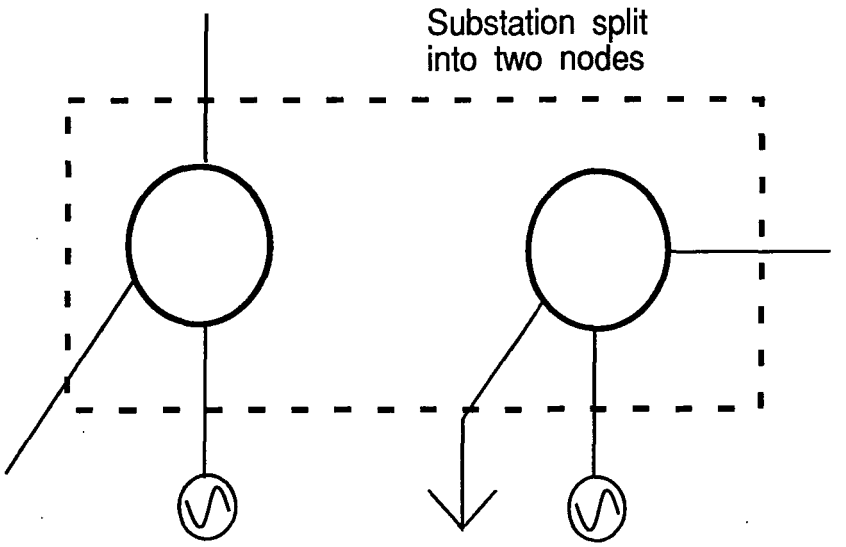
Substations consist of relatively few busbars, with a dense network of connections between them, to provide flexibility and redundancy. A ficticious substation is shown in figure 9.1, which shows circuit breaker and isolator layout, and several forms of external connections. The most common method for determining the topology of a network is to proceed as follows [139]. Each prospective node (busbar) is initially given a unique node number. This number only has to be unique in the current set, so the most usual schemes are absolute busbar numbers, or numbers relative to the first busbar in the substation. The latter scheme is to be preferred to make indexing into temporary arrays more effcient. Each busbar then has its node number compared with that of each of its neighbours, to which it is connected by a conducting link. It takes the minimum node number found as its new number, or keeps its own number if this is the smallest. This process is repeated until no further changes occur, a process analagous to a bubble sort, which requires an execution time of order $O(n^2)$ where n is the number of busbars. As n will typically be about 10, this is acceptable due to the limited set–up which is required.

The nodes will now be numbered according to the lowest numbered busbar which they contain. It is possible to renumber the nodes starting from the first busbar in the substation, and this is beneficial for some of the lists, but removes a convenient link back from node to busbar, and can cause confusion, if busbar 'x' is in node 'y', while busbar 'y' is not, yet all are in the same substation. This renumbering could also be combined with the removal of nodes without any external connections, as these can not participate in energy transfer until the substation is reconfigured.

Initially, this process must be performed for every substation in the network, but from then on, only substations in which a breaker has changed state need to be processed. Due to the size of each substation, the detection and special processing of special cases such as breakers closing is not worthwhile. It is simpler to just process the substation data from scratch. The only special case which should be detected, is the case when no externally visible change has occurred. This can only be detected after the nodes have been redetermined, so little saving is possible in the substation analysis routine. The benefits of detecting that no change has occurred are large when global topology is



Full Representation



Condensed Representation

Substation Topology

Figure 9.1 Example substation topology.

considered, as this substation does not cause any change in the overall nodal connectivity, and thereby require the update of several complex lists.

9.3 External Connections between substations.

Processing external connections is more difficult than internal connections, because while the connections are physically made through switches to busbars, most routines view the terminators as nodes, which change number as the simulation proceeds. After any substation has changed configuration, any reference to any node in that substation must be updated, before an attempt can be made to determine global topology. A compromise must be reached over the number of lists which should be used to keep commonly used information in a convenient place, as these lists will not only save time during topology processing, but will cost time for their maintenance.

If no substation changes configuration, then all the node mappings are correct, and the problem reduces to determining which nodes are connected to which others, a process called 'islanding'. The inputs are a set of possibly active nodes, a set of possibly conducting connections between the nodes, and a set of generators or other power supply points. The nodes must first be placed into islands, which are groups of nodes which are electrically connected by lines which are able to conduct. An attempt is then made to assign a generator or other power source to each of these islands. If one is found, then the island is considered to be active. The output is a set of electrically active islands, which is the model used by the simulator until the configuration next changes.

The previously outlined method for substations could be used for the external problem, but the larger n, combined with slow convergence resulting from sparser connections, dramatically slows the method. A method based on a direct search and assignment is better suited to this problem. The connectivity resembles the tree data structure familiar in computer science, and a search method which takes advantage of this should be used.

Initially all nodes are allocated to a dummy island. Any node is selected as the first base node (usually the first one for simplicity), and a new island is assigned to it. Any node which is attached to this node by conducting connections, and not already part of the island is assigned to the island, and saved to be used as a future base node. The nodes are saved by pushing then onto a stack, and recalled by popping off the stack. By

assigning newly discovered nodes to the island before saving them, the method ensures that they are only saved once, as by definition, a node is not newly discovered if it has already been found. This makes efficient use of processor time and limits the memory consumed by the stack.

Once a node has been used as a base node, all its neighbours have been saved for future use, or have already been found. A new base node is selected from among those saved, from which a search is made for undiscovered nodes. This proceeds until the saved set is empty, and the connections from the current base node are exhausted. All nodes in this island have been visited and assigned to the island, so the next island can be based on the next node which is found, by a linear search of the nodes, that has not yet been assigned to an island. If no such nodes are found, then all nodes have been processed, and this stage has been completed.

A search is then made through the generators, in an attempt to find an active generator for each island. If the generator is active, then the reference field of the island of the node to which the generator's busbar has been assigned is checked, and if void, then no generator was previously found for that island. The number of the generator is placed in this field, and the search continued with the next generator. After all generators have been processed, and island with a void reference field is inactive, and can be deleted. This will probably delete most of the single node islands found in the islanding search, but it is worth noting that a single node can be an island if it has both generation and load attached, while a large group of connected nodes might not be if they are without generation.

9.3.1 Using generators as seeds for islands.

A more efficient variation on the method outlined is to start searching at nodes which have generation or power in–feeds attached. This removes the processing of islands which will later be discarded, and results in fewer nodes being visited, and dispenses with the linear search through the node tables to find a node which has not been visited. There are likely to be between ten and five times more nodes than generators in a power system, so this saves time. The search through the active generators is also removed, resulting in a time saving.

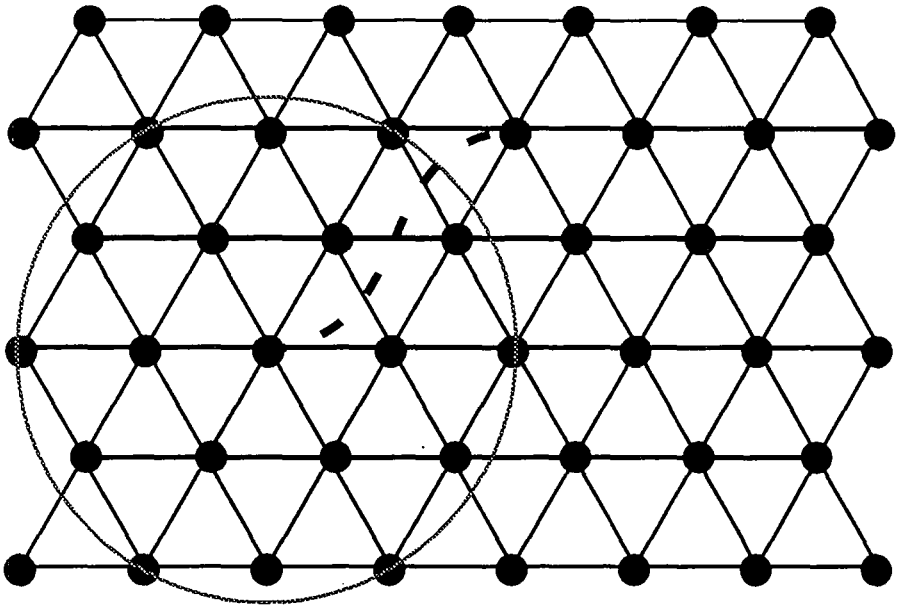
9.4 Variations on Search Method.

The method outlined earlier is a depth first search. As new nodes are discovered, they are placed on a push down stack until they are needed. The last new node connected to the current node will be used immediately, but other new nodes will not be processed until all new nodes resulting from that node have been processed. The method will therefore tend to search far away from the initial node, only returning to neighbouring nodes when all nodes further down the tree have been found and processed. The search order is unimportant for the initial islanding, as every node will be processed once, but for subsequent searches, this order could be inefficient.

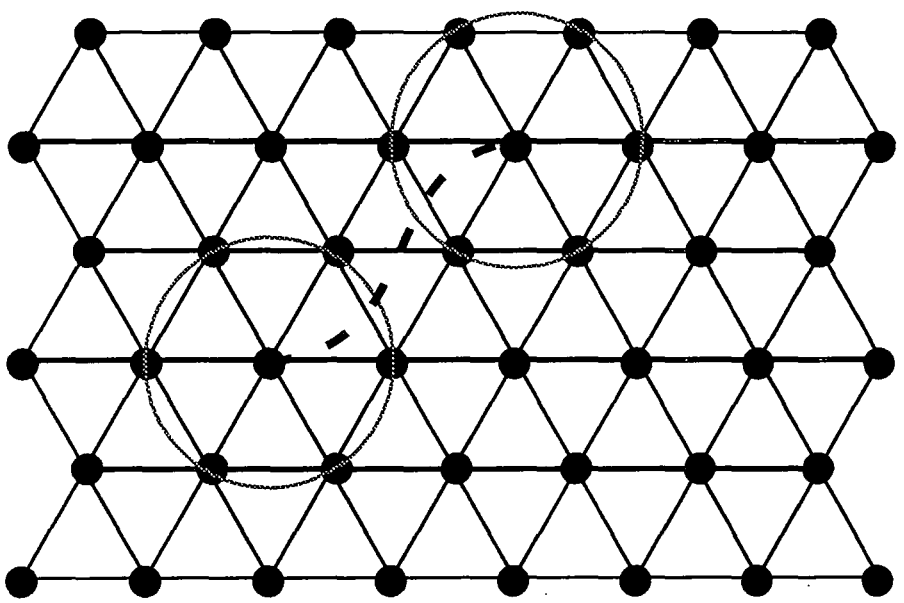
In an electrical transmission network, there is a high degree of local meshing, in order to improve reliability. Where a direct line exists between two nodes, there is usually at least one possible alternate path involving relatively few intermediate nodes. If a transmission line connected between two nodes is energised, a simple check of their original island numbers suffices to determine whether two islands have merged because of the change, if no other changes occur. If the nodes were originally in the same island, then nothing has changed, otherwise the two islands must be merged.

If a transmission line is denergised, it is possible that the original island could have been split into two islands, (passive or active). Two islands result if there is no longer a path between the two end nodes of the line. In this situation, the depth first search might search in completely the wrong direction, and process many nodes before reaching the target node. A breadth first search would be able to take advantage of the probable locality of any connection, and so would be expected to find the node at the other ened of the line more quickly. If no connection exists, every node connected to the first end must still be processed, in order to prove that no connection exists.

The depth first search can easily be modified to become breadth first, by changing the stack into a queue. Here, nodes are placed on the queue at one end when they are first encountered, and removed from the other end. All the nodes adjacent to the start node are processed before any node placed on the queue by one of these nodes, is accessed. The nodes searched will appear to radiate from the first node like a ripple emanating from a drop of water on a smooth lake, as is shown for a dense network in figure 9.2 a.



Single ripple, 18 nodes visited



Double ripple, 12 nodes visited

Connectivity after dashed connection removed

Figure 9.2 Ripple of search in a tightly meshed network.

The search can be made faster and more efficient by starting the breadth first search at both end nodes simultaneously. If they are both assigned new and different island numbers, and are placed into the queue as the initial nodes, then the search will alternate between them. All nodes attached to the first source node would be added, then those attached to the second source at the other end of the disconnected line would be added to the queue. Processing would then start on the nodes attached to the first node attached to the first source node. As each node is visited, its island number is compared against the island number of the other source, (i.e., the source number which is not the same as its discoverer). If this is found to be true, than a connection path has been proven. Otherwise, the island number of the new node is set to the same as its discoverer, and is added to the queue. If the two nodes are still connected, then this method will probably find the connection quickly, otherwise the two islands will be correctly labelled, as connectivity is still possible until all nodes have been processed.

The last statement is not strictly true. Connectivity is not possible if all nodes connected indirectly to one of the source nodes have been processed without finding a connection (by definition). This is of little benefit however, because if no connection path exists, every node in both islands must be processed to place them into new islands. A benefit might be obtained if one of the islands was very small, so that it would be more efficient to undo any changes made to nodes in the larger island, and just make one new island for the smaller number of nodes.

This modification could be achieved by maintaining two queues of nodes which have been found, but not yet processed. When one becomes empty, after few nodes have been processed, it would be possible to alter the nodes in the other area back to their original island. This is no guarantee of an increase in efficiency, unless a count is maintained of all nodes in each island, as the other island might amost be exhausted.

The dual search method could be visualised as the interesection of two ripples which are shown in figure 9.2 b, one originating from each of the two end nodes. If just one node initialises a ripple, then the area processed before the other node is reached is πr^2, while if both initiate ripples, then the area is halved at $2 \times \pi(r/2)^2$ where r is the distance between them. If nodes are assumed to occur with uniform densiy, then a dual search should be faster.

The memory required for the queue might at first seem to be a problem, as the queue will 'walk' through memory, but it is very similar to the worst case behaviour of the depth first search method. The depth first search method could put almost all nodes on the stack, if the initial nodes had exceptionally high connectivity. The queue will walk through memory, but this memory will be limited to the number of nodes in the system, as each node could only be queued once. To use less memory, a loop must be used, but the queue might then be too short, and some extra processing is required to perform modulo (clock–face) arithmetic on queue length and position.

9.5 Resilient Islanding.

The above method quickly and reliably handles single changes, but could loose islands and nodes when multiple changes occur during a single time–step. A more resilient method was therefore required. The chosen method is based on the initial depth first search, but allied with time–stamping. Each node (or all nodes in a substation where a change has occurred) at which a change has occurred is placed in a list, from which a list is made of original islands involved in change. To the node list are added any newly formed nodes. These nodes are then taken as seed nodes for the creation of new islands. If the time–stamp for the island containing a node is old, then a new island is formed from that node, otherwise, that node has already been processed in the current time step, and no further processing is required. Once a new set of islands has been formed, an attempt is made to assign generators from the previous islands to the new islands, to improve continuity of solution. A search is then made for generators for any island without generation, and finally inactive islands are emptied and deleted, and the old islands are simply deleted, as the previous processing must have emptied them.

Emptying an island involves walking round all its nodes, setting their island number to void, so that if the node is interrogated later, its inactivity is clear.

This method ensures that no node can be lost, as any node involved in change is explicitly processed. The relative efficiency is poor for single, local changes, but improves for multiple changes. Checks could be added to process simple events, such as a single line changing state, without degrading the security of the method, and indeed, the change of status of a generator is already handled seperately.

9.6 Connections to Busbars.

Because all connections are made to busbars (via switches), and the islanding routines work with nodes, a mapping between node and busbar numbers is required. This mapping should be made as infrequently as possible, and certainly not during an islanding search. For this reason, each entry for a busbar contains a value for the current node to which it is assigned. There is a less frequent need to find a constituent busbar given a node number, so this operation is performed by scanning the busbar entries of the relevant substation for the correct node number.

Another list is kept for each node, of all nodes to which a direct connection could be made by a transmission line. Nodes are included in the list regardless of the current conduction state of the line, which is handled by negating the target node number for non–conducting lines. The length of the list for each node is determined by the total number of external connections to each constituent busbar of the source node. Every time a substation changes configuration, the lists for each of its constituent nodes are re–formed, because the number of connections to each of the nodes could have changed. Since the possible number of external connections for each substation must remain constant, each substation occupies a fixed region of the array, so no data movement is required.

Updating these lists is a three–stage process. First, the number of connections to each node is calculated by summing the connections to the constituent busbars. These counts are then translated into pointers into the array, which are used to store the actual target node numbers into the array, as the busbars are again scanned as before.

When a substation changes configuration, the connection list of any substation possibly connected to it must also be modified, because the node numbers used as end points for the lines could be out of date. This task is simpler than adjusting the entries for the changed substation, because all the lines still originate from the same nodes in the remote substation, so no lines need to be moved between lists. Unless a special scheme for parallel lines is in operation, no data movement is required, and only target node numbers need to be changed. Parallel lines could be duplicated (this is the simplest case), so that search routines would process the connection twice, or special code could keep track of parallel lines, and maintain their status so that if both are conducting,

the islanding routine sees only one conducting, while either or neither conducting is handled appropriately.

Pointers are also maintained for each line, which indicate where the connection entries for that line are stored. One of these values requires updating when a substation changes configuration. This complicates the connection list modification slightly, as information must be maintained on which lines were moved where. These pointers permit the direct modification of entries in remote substations without searching, and more importantly, without regard to which node's list contains the entry for far end of the line, or which node that entry references. When several substations change configuration, each modified substation is processed in turn, so it is possible that the entry at the target end of the line does not match the actual source node, or that the entry is in the wrong list. If this occurrs, then the substation containing the target node must also have changed in the current time step, and remains to be processed, which will correct the data mismatch. Because no data is moved in the target nodes lists, this possible mismatch is unimportant, and the method is therefore resilient to multiple changes, whether inter–related or not.

9.7 Minimum alterations across topology changes.

If it is possible to keep information valid across topology changes, then this is advantageous because it helps the continuity of the solution, and would possibly lead to a reduction in the simulator workload. Three possibilities exist for benefits from tracking [112] the topology changes as they occur.

If the majority of the node numbers in the network remain fixed, then most of the network structure will not be altered from the previous state, and almost all the original connections will reference the same node numbers. This means that most of the unordered Zollenkopf arrays are still useable, and the connections which must be altered can be identified by moving down the column entries of the altered nodes, which also gives the columns containing the duplicate elements. These can then be deleted, before being the new connections to modified nodes are insterted. For modifications to a small number of nodes, this is very efficient.

The second advantage which is available can be used to reduce the number of iterations which are required after a topology change by trying to set the states of any modified nodes so that they would require the minimum correction for a consistent solution. This

can be achieved by maintaining the node numbers of adjacent nodes during a topology change, if these nodes are not reconfigured, and by saving the values of each of the constituent busbars and the previous busbar to node mapping of the altered node. This information can be used to decide on the best approximation for new values for the altered nodes. These valuse should provide the Newton–Raphson iteration with a good, warm start, which should require fewer iterations for convergance.

The third benefit can also reduce the number of iterations required for convergance, and will certainly improve the continuity of the simulation across connectivity changes. This involves trying to make the choice of generators which are chosen to be frequency references as consistent as possible across islanding and other connectivity changes. The fewer sudden frequency transients that are injected into the system, the fewer iterations are required to adjust the remainder of the system for the change in frequency. The algorithms described in this chapter all try to re–use an active reference if one exists.

9.8 Determination of global topology in parallel.

If the numerical solution of the network is split between processors, it is natural to split as much of the remainder of the simulator as possible between the same processors, so that speed increases will be maximised. If the remainder of the simulator could be split in the same way as the numerical matrix processing, then each processor would be more self contained, which must reduce inter–processor communication. The aims and dificulties of topology determination in parallel are similar to the numeric matrix solution, in that global adjustment is required between two parts which can be executed in parallel, and so this global part should be minimised by performing as much of the calculation as possible in parallel. The situation is eased slightly by the smaller amount of data which must be transferred between processors.

An examination of the uniprocessor algorithm reveals that the processing internal to a substation can be performed wholly within a single area, if no substations are split between processors. The islanding method requires modification, because each area can contain several islands, and each island can cover several areas. Each area can only form sections of islands which consist of nodes within the area. These must then be passed to the coordinator to determine which should be joined, and whether they are active or inactive.

The methods proposed for a single area on one processor can be modified to run in parallel by adding a new class of generator called a tie–line. This is treated identically to a generator, except that it has lower priority in being used to activate an island, and cannot be used to activate a global island. Each line connecting between two areas is assigned a unique tie–line number, so that the global routine can match both ends of these lines.

Each area performs the first section of islanding as in the uniprocessor case, and then scans through the islands trying to assign generators to these islands, again using the previous reference if it is still available. A new pass is then performed, assigning tie–lines to those islands without active generation. If an island has neither generation nor connections to an island in another area, which is signalled by the presence of a tie–line, then it cannot become a globally active island, and can therefore be deleted.

A scan is then made of the remaining tie–lines, to determine which local islands they are connected to. All that needs to be passed to the global routine is a list of these tie–lines and a list of local islands giving their generation status. This status could be one of: island contains previous global reference, island contains generation or island contains no active generation. Again, if previously active references are available, then a choice should be made from amongst these generators.

The area islanding code has effectively condensed all the nodes in each local island into one *super node*. Each super node in each area could have several generators attached, and also some tie–lines. The global problem now appears identical to the original islanding problem: a set of nodes, some with generation attached, which are connected together by a set of lines. The number of lines should be relatively small if the matrix split has been successful, since a large number of tie–lines would also slow the numerical calculations.

The global routine now knows about all local islands which could participate in islanding, and has all the information that is needed to build a set of interconnections between these local islands. The lists required are very similar to the process which has been performed in each of the areas to form the local islands. Once the connectivity data structures have been generated, a similar algorithm can be used to that of initial islanding in each area, whereby the local islands are used in turn as bases for depth first searches.

Once islands have been identified, a search is made to assign active generation to them from amongst the generators put forward as possibilities from the local areas themselves. Since the number of generators proposed would be small, a scan can be made of these, assigning them to an island if no generator of greater or similar priority had already been assigned, so that a current reference would take precedence over a prospective reference. Any island without generation at this stage must have been formed from local islands, all of which relied on tie–lines for generation.

A decision must be made about whether to pass local islands without tie–lines but with active generation to the global routines as normal islands. These islands clearly do not require further processing by the global islanding routine, but should be assigned global island status, so the global routine must have knowledge about them. Passing as normal islands would impare efficiency slightly, but would treat all islands as the same, reducing the possibility of errors or unexpected behaviour.

Information must be passed back to the local areas about which local islands have been accepted as parts of global islands, which local generators are global references, and which generators in other areas are being used as global references for locally–defined islands. Acting on this information, the local islanding can be completed. This involves emptying and deleting any passive islands which were put forward for global status on the strength of tie–line connections, and finalising frequency reference information for those islands which remain. The global island numbers must be kept separate from the local numbers if a tracking topology generator is implemented, because two local islands could form part of the same global island, and for future islanding trials, these must be differentiated between.

The method used for global topology determination is similar to those proposed for single processor operation, and the individual areas themselves. It does not, however, rely on any particular method being used in the local areas. All that is required is that it is presented with a connection list of what the tie–lines connect to, from each area processor, and that it also recieves graded nominations for generation in each local island.

9.9 Results.

This method has been tested on the I.E.E.E. 30 node test network, with the addition of generation at node 30 to provide more flexibility and variety in the network splits which were possible. The test network is shown in figure 9.3. All contingencies that were tried were handled correctly, and without any apparent time delay. Generation was shut down, restarted, moved between islands by reconnecting whole substations, and by moving individual busbars between islands. Islands became active and passive as generators changed status, and all results were consistent.

No timings were performed, because they would not be meaningful on the small 30 node test network. The complexity and size of the larger networks makes the manual checking of configuration very time–consuming and prone to error, so trials were restricted to the small network. No results are presented because even for this network, each statement of configuration would take several pages, and the differences are small between each of them.

9.10 Conclusion.

The aim of the work was achieved, since a topology processor was produced which could not only accomodate the type of network split proposed in chapter 7, but which could also perform a significant proportion of the topology determination and islanding in parallel, and which could be tailored to the specific requirements of the Zollenkopf algorithm.

The topology processor keeps as much information valid across topology changes as possible, and can even approximate new values for each node based on some function of the previous values of the constituent busbars at a changed substation, which should reduce the initial number of iterations required to achieve convergence. Hooks were also provided to enable efficient structural updates of the existing matrix for small connectivity changes.

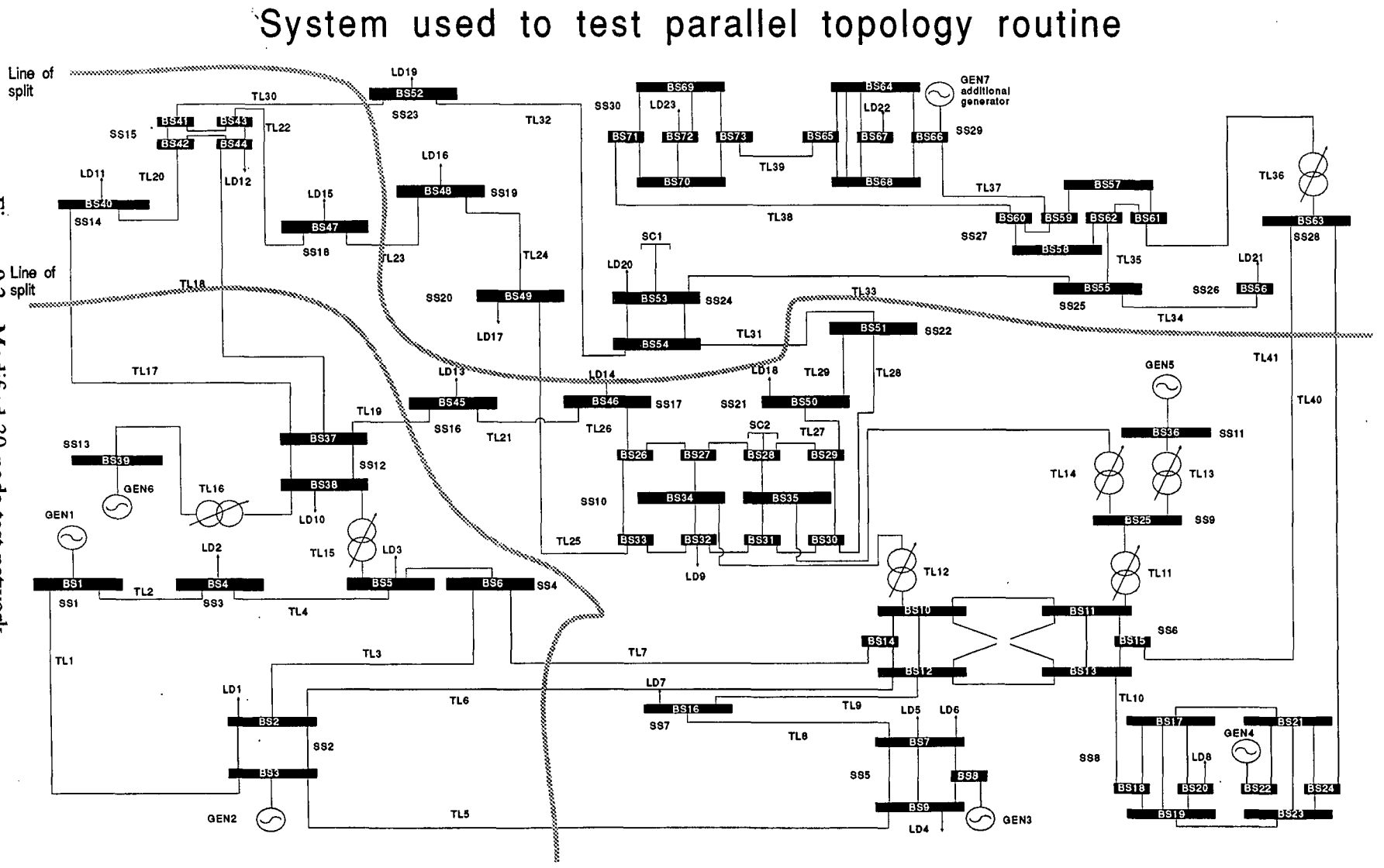


Figure 9.3 Modified 30 node test network.

Chapter 10.

Conclusion.

10.1 Recap of the aims.

The aims of the research presented in this thesis were to increase the speed of a power system simulator so that a more realistically sized network could be simulated to improve the environment for the development and testing of control software, and to investigate the increase in speed which could be obtained from the use of multiple processors executing in parallel. Specifically, a target was set of real–time simulation of the C.E.G.B. transmission system with its generation, at a time–step of one second. The characteristics of the current simulator used by the OCEPS group were to be retained, which defined the solution technique and the models which would be required. This simulator currently uses a 30 node American test network, but is believed to be capable of real–time simulation of the larger 118 node network, but with a possible reduction in the complexity of the models to achieve real–time operation. This simulator runs on an otherwise unloaded Perkin–Elmer minicomputer with an attached Floating Point Systems array processor.

10.1.1 The main problem areas.

The solution of a large, sparse matrix equation was identified as the most time–consuming part of the current simulator, and also the most difficult to split between multiple processors. The solution is not only computationally intensive, requiring double–precision data, but is extremely memory intensive, requiring many operand fetches and stores. These memory accesses are widely scattered, and the reuse of recent data is poor, which largely nullifies any general–purpose cache management scheme. Fast, low latency access to memory is therefore required. The sparsity structure of

the matrix is defined by the network topology, and therefore lacks much of the usable structure found in other sparse systems. The structure also changes during the simulation, as the network changes configuration, so the solution algorithms must be able to reformulate the equations in real–time.

10.2 Proposals from the research.

The research effort was divided into five main areas in the search for a faster solution method for the sparse matrix equation in particular, and the simulator in general. The areas are, of course, closely linked, with a development in one area needing to be incorporated into the remainder of the simulator, but the order in which the main points were presented was felt to minimise the 'forward references' to areas as yet undiscussed.

10.3 Alterations to the storage of the matrix.

The matrix solution algorithm used by the current simulator is believed to be optimal for problems involving network sparsity, but the method can be implemented in many different ways, and some alterations to the current method were investigated. The Zollenkopf program separates numeric computation from the integer calculations used to order the matrix elimination. This improves efficiency and also gives more flexibility to modify the implementation. The numeric calculations are performed by interpreting the linked list structure left by the ordering code, which for speed during the ordering phase, are left in a very scattered state.

A routine was inserted between the ordering and calculation stages to re–form the lists. The list entries were scanned in elimination order and the elements were placed into a different list structure in this order, which moves all entries for the same column into contiguous locations. The individual row entries were also sorted into elimination order, instead of column number order, which removes the need for an array of links, and makes the matrix structure much clearer by providing more implied information.

This new arrangement provides more variety in the order in which the numeric calculations can be performed, and this was used to reduce the possible data dependency problems when using a pipelined processor, and also to make the real–time generation of code, dedicated to solving the current matrix structure, much more efficient. The

solution process can be broken down into the elimination of a series of 'full', square, submatrices, and these form natural complex instructions, if an list of addresses of all the operands can be provided in a compact form. The presence of implied information in the new, re–ordered, lists, greatly reduced the actual addresses which were required to uniquely define the numerical calculations, and made the generation and interpretation / execution of the code efficient processes. An overall increase in speed was achieved on the VAX by generating the code and interpreting it, instead of simply interpreting the list structures directly. The speed–up would be much more dramatic on a processor designed specifically to execute the code. The implied information also moves all data dependencies to predictable places, and it was found that only one type presented a problem with the pipeline depths encountered, and the work of the next section reduced the number of occurrencies of this special case.

10.4 Alteration to the elimination ordering.

The order in which the columns of a power system sparse matrix are processed during the matrix solution has a massive impact on the number of calculations which are required to solve the matrix, and how many new matrix terms are generated. All code uses a variation of the second ordering proposed by Tinney and Walker, which monitors the number of non–zero terms (degree) in each column as the elimination proceeds, and selects the next column from amongst those with minimum degree. If more than one column has the minimum degree, the choice is arbitrary, and many sub–criteria have been proposed to enhance performance, but all use the same linear list searching technique to find the best column. This was found to be very slow for large matrices, because the number of comparisons grows with the square of the matrix size.

A data structure was devised which eliminated the searches, and replaced them with list management operations. The lists were optimised to reduce bookkeeping, and imposed an implied sub–criterion on the Tinney ordering, depending on whether modified columns were added to the front or rear of the queues for each degree. This would give preference to either the least or most recently modified column with minimum degree. The former was chosen, because this reduced the number of the most problematical pipeline dependencies.

The use of a data structure produced impressive speed increases for the larger networks, with a four–fold speed increase for the large C.E.G.B. network over the standard ordering. The ordering time is made up of time to select the next column, and time to alter the matrix structure for each eliminated column, and when this time was subtracted from the overall ordering time, a ten–fold speed increase was achieved for the new ordering. The decrease in time for ordering doubled the remaining time in the one second time–step for the numerical solution of the matrix, which makes the solution of the large matrix feasible using current microprocessor technology. Without this new ordering method, real–time simulation for these large networks would not be possible, unless some form of delayed processing was used, in which case, the simulator might fall out of real–time operation during a severe transient, due to the snow–balling effect of protection violations causing further trips. This is precisely when a simulator is most vital for operator training and the development of control algorithms.

Other ordering methods were compared to the new ordeing method, but all were slower than Tinney–2. Several of these orderings were aimed at reducing the factorisation path length, and it was found that the proposed new method produced shorter and more balanced path lengths than the other methods, in a small fraction of the time. This is a by–product of the least–recently–used selection sub–criterion. One of these methods was combined with the new ordering, and was this was found to improve the new ordering still further, at a small time penalty. The penalty was reduced by applying optimisations which were not possible with the linear list searches ordering.

10.5 Selection of hardware for bifactorisation.

The bifactorisation process is extremely memory and computationally intensive. The types of memory access required do not fit well with the most general purpose computer architectures, and in particular, render most caches ineffective. A survey was made of the available microprocessors, with the pseudo–code of chapter 3 in mind. This concluded that although RISC processors were much better suited to the problem, only two processors were capable of the sustained calculation rates required for a uni–processor solution in real time, and that even these would require a special memory design to transfer the operands at the required rate. The processor bus was capable in both cases, but the lack of pre–fetch cycles due to the double–word transfers, and to the inaccessibility of the address generation logic would cause processor stalls.

The availability of dedicated FPUs from microprogrammable chipsets provided an alternative solution, and this was investigated in chapter 6. The complexity of microprogrammable design was reduced by removing most of the general purpose facilities, and producing a design dedicated to one task, the solution of sparse matrices. The matrix would therefore have to be generated and post–processed by other processors. It was found that the pseudo–code was an ideal match for this type of processor, both due to the processor itself, and the flexibility of microcode design. The early availability of almost complete operand addresses, coupled with open address generation under the control of microcode would allow either prefetching of operands, or the early start of cycles which would utilise the memory cycle more efficienty than would be possible for single–cycle RISC processor initiated transfers. Although the design would be capable of very high sustained rates of calculation, it was not taken further. The physical design difficulties of electical noise from the many wide buses on the card could not be solved with the facilities available, and estimates showed that it would take longer to load data onto the card, than to actually perform the matrix solution.

The design excercise served to clarify the major problems for the fast processing of the bifactorisation algorithm, and proved that pseudo–code, coupled with a dedicated pseudo–code interpreter/ processor is the fastest method of implementing this algorithm. If the complexities of constructing a special–purpose processor are too great, then one of the better RISC processors can be used instead, with highly optimised routines. This is subject to the proviso that memory accesses will be critical, unless a special memory interface can be developed.

10.6 Parallel processing.

The increase of solution time caused by the data transfers required by the solution proposed in chapter 6 were taken into consideration in the investigation of one method of splitting the matrix solution itself between processors. A matrix block structure was taken from Tinney. This possessed the useful quality that no reformulation of the system equations was required, and whether the solution was performed in parallel or not, was invisible to the development of the system models. The tight meshing of the C.E.G.B. system makes it more difficult to split than the American systems, which tend to have natural splits between the networks of each individual private power company. A restriction on the splits was proposed which would effectively slice the network, by

allowing one area to connect to, at most, two others. It was hoped that this would reduce the number of elements in the coordination matrix, which must be solved serially during the otherwise parallel solution, or at least, restrict the non–zero terms to certain areas, which might permit non–sparse solution techniques to out–perform sparse methods.

10.6.1 Testing trial splits.

Many network splits were required to test the restriction, and three methods of generating these were tried. Two of the methods were optimisers which attempt to simulate the optimisation techniques which are found in nature, and the other involved the analysis of the data dependencies which result from the choice of elimination order. A cost function which weighted the number of nodes assigned to each area and the number of local and non–local tie–lines was used for both optimisers. An optimiser based on Simulated Annealing developed by Dr. Irving was initially tried, but this produced very erratic results and failed completely when the restriction on inter–area connectivity was applied. This optimiser changes the processor allocation of a single node at a time, and accepts the change if a cost decrease results, and uses an exponential acceptance probability if a cost increase results. The bias in the cost function against the end areas was tending to empty these, which in turn caused the next areas to empty, an so on, until only two occupied areas remained.

An optimiser based on Genetic Optimisation was then developed, and this was found to be greatly superior to Simulated Annealing. It models the changes which take place in DNA as it is passed between generations, and inherently modifies the alloctaion of several neighbouring nodes at one time. This was found to remove the problem of the end areas being emptied, and aided the rapid determination of viable clusters. The node numbering scheme affected the early clustering, but the optimiser was capable of escaping from it. Another major advantage was that some of the permutations could change the allocation of nodes between processors without altering the totals assigned to each processor, which removed cost obstacles which would otherwise occur in intermediate steps, and gave priority to the distribution of nodes between processors.

The origins of the network split indicated another method of splitting the network, based on the data dependencies which result from the elimination order. The blocks which Tinney identified as being largely independent can be identified from the tree of data

dependencies. Independent branches can be processed in parallel until they meet. The nodes which are low down the tree, and are in the path of two or more branches must be processed in parallel. Several path trees were used as the base for matrix partitions, and these were compared to those produced by genetic optimisation. The best paths were obtained from the new ordering methods proposed in chapter 4, because these tended to produce the flattest and best balanced elimination trees. These splits do not have the restricted inter–area connectivity proposed by the optimisers, and can therefore be used to assess this restriction.

The best of the splits were found to be very similar, with differences arising from the difference between the optimiser cost function, and the cost function of the elimination orderings. A speed–up over the uni–processor solution of between two– and three–fold was found for the matrix solution itself, which would be expected to decrease when communication times are included. An exceptional split was found, from the path diagram produced by a variant of the new ordering method, which gave a speed–up of 3.3. No other split approached this performance. The three–fold speed up was variously obtained by the use of between four and seven processors. The overall speed–up for each iteration would, however, be better than this, because the formation and post–processing of the solution for each area is almost completely independent of all the other areas, so a linear speed–up would be expected with relation to the number of processors in use. About a three–fold overall speed increase would be expected. The similarity of the solutions produced by path analysis and optimisation shows that there is little advantage in the proposed restriction on inter–area connectivity.

10.7 Topology determination in parallel.

The determination of system topology has much in common with the matrix solution, because the processing of both is affected by network connectivity. The topology determination in use in the current simulator is only suitable for small networks, so an alternative topology determination method was developed. The method uses the original topology algorithm internally within each substation, because it is efficient for the high connectivity and small network sizes encountered within substations. A depth first search was proposed for the inter–substation connectivity, and a resilient islanding routine was developed from this, which is capable of handling any number of topology changes in a single iteration. The method is based on time stamping each island as it

is created, and only processing changed nodes which belong to an island which was not created in the current time–step. This is very efficient for multiple changes, but inefficient for small, single changes. The method does not preclude the identification and special processing of these special cases.

The processor tries to keep as much of the pre–change state of the system valid across topology changes as possible, so that the iterative solution can start from the best initial state possible. The method is also capable of incremental updates to the un–ordered sparse matrix structure, so that the matrix does not have to be re–build following a topology change.

The method is also amenable to parallel processing itself, using a similar data split to the matrix solution, and correctly identifies global electrical islands which are made up from islands local to each area. Again, as much information as possible is maintained across invocations of the topology processer, particularly which generators are in use by each island as voltage and frequency referencies. No speed trials were made, because the test network was too small to make these meaningful, and no results are presented, because, although the network is small, the result files are not. The method did, however, work without noticable time–delay, and produced correct island splits, even for multiple topology changes.

10.8 Main conclusions.

The research presented here has shown that the real–time solution of the C.E.G.B. transmission network with its associated generation is possible on micro–computer based hardware without sacrificing any detail from the models used by the current OCEPS simulator. The required performance is just attainable by the fastest of the current RISC microprocessors for a uni–processor solution in the simulator core. As many calculations as possible should be removed from this processor, in particular, the communications, addition of noise to the results, topology processing and matrix ordering can be removed to one or more other processors. Any data transfers involved in this split would consist of block transfers, which are relatively simple to schedule. To achieve full speed from such a processor, the memory design would have to be specially optimised for the bifactorisation process, because the general purpose cache designs perform poorly for

these transfers, and memory speeds are too slow to prevent almost constant pipeline stalls if no cache is available.

The use of multiple processors working in parallel on the iterative solution can achieve up to a three–fold speed increase over a single processor working alone. In addition to solving larger systems, or more detailed models, this could either be used to provide more certainty of obtaining converged solutions in real–time, or to accept slower matrix solutions, which would dispense with the requirement for complex memory architectures necessary for a uni–processor solution.

The improvements in the ordering algorithm doubled the time remaining in the time–step after a topology change for the numerical solution to converge, which is equally valuable for a uni–processor or multi–processor solution. The new methods also provide additional advantages over existing methods, in the reduction of pipeline stalls and the elimination path length. The latter improvement was shown to be helpful in splitting the network for multiple processing.

The rearrangement of the sparse lists after elimination ordering was shown to repay the time taken by permitting a reduction in memory requirements, providing more implied information in the resulting address lists, and by giving greater flexibility in the order of calculations, so that they may be tailored to the hardware used for their solution. It also simplified the generation of efficient, dedicated code in real–time, for the solution of the current matrix structure.

A flexible and reliable optimisation technique for a variety of matrix splitting problems has also been developed, and made into a viable tool by the addition of an interactive, graphical interface.

These improvements to the existing simulation method can be used individually, although some are naturally linked, to improve the solution speed of the current OCEPS simulator.

References

The abbreviations used when referring to Instutute of Electrical and Electronic Engineers publications are:

IEEE Trans. PAS	IEEE Transactions on Power Apparatus and Systems.
IEEE Trans. PWRS	IEEE Transactions on Power Systems.
IEEE Trans. CAS	IEEE Transactions Circuits and Systems.
IEEE Trans. CT	IEEE Transactions on Circuit Theory.

1. ABUR, A., "A parallel scheme for the forward / backward substitutions in solving sparse linear equations." *IEEE Trans. PWRS*–3, No. 4, November, 1988, pp. 1471–1478.

2. ADVANCED MICRO DEVICES, "Am29000 streamlined instruction processor." *Data sheet,* 1987. Sunnyvale, California, U.S.A.

3. ADVANCED MICRO DEVICES, "Am29000. 32–Bit streamlined instruction processor." *User manual,* 1988. Sunnyvale, California, U.S.A.

4. ADVANCED MICRO DEVICES, "Am29027 arithmetic accelerator." *Data sheet,* 1987. Sunnyvale, California, U.S.A.

5. ADVANCED MICRO DEVICES, "Am29C300. 32–Bit microprogrammable products." *Data book,* 1988. Sunnyvale, California, U.S.A.

6. ADVANCED MICRO DEVICES, "Floating–point microprocessor implements high–speed math functions." *in* ADVANCED MICRO DEVICES, *"Am29C300. 32–Bit microprogrammable products. ", AMD,* 1988, pp. 6-116—122. Sunnyvale, California, U.S.A.

7. ADVANCED MICRO DEVICES, "Am29C325 CMOS 32 bit floating point processor." *Data sheet,* 1987. Sunnyvale, California, U.S.A.

8. ADVANCED MICRO DEVICES, "Am29C327 double–precision floating point processor." *Data sheet,* 1987. Sunnyvale, California, U.S.A.

9. ADVANCED MICRO DEVICES, "Am29C327 double–precision floating point processor." *User manual,* 1988. Sunnyvale, California, U.S.A.

10. ALVARADO, F.L., REITAN, D.K., BAHARI–KASHANI, M., "Sparsity in diakoptic algorithms." *IEEE Trans. PAS–96,* No. 5, September, 1977, pp. 1450–1459.

11. ALVARADO, F.L., YU, D.C., BETANCOURT, R., "Partitioned sparse A^{-1} Methods." *IEEE Trans. PWRS–5,* No. 2, May, 1990, pp. 452–459.

12. ANDRETICH, R.G., BROWN, H.E., et al., "Piecewise load flow solutions of very large size networks." *IEEE Power Group,* Summer Meeting, 1970. Los Angeles, California, U.S.A.

13. BETANCOURT, R., "Efficient parallel processing technique for inverting matrices with random sparsity." *Proc. IEE, Pt. E,* Vol. 133, No. 4, July, 1986, pp. 235–240.

14. BETANCOURT, R., "An efficient heuristic ordering algorithm for partial matrix refactorisation." *IEEE Trans. PWRS–3,* No. 3, August, 1988, pp. 1181–1187.

15. BETANCOURT, R., ALVARADO, F.L., "Parallel inversion of sparse matrices." *IEEE Trans. PWRS–1,* No. 1, February, 1986, pp. 74–81.

16. BIGLAN, H., CASHAR, E.E., et al., "A dispatcher training simulator design with multipurpose interfaces." *IEEE Trans. PAS–104,* No. 6, June, 1985, pp. 1276–1280.

17. BIRCH, A.P., "Adaptive load frequency control of electrical power systems." *Ph.D. Thesis,* November, 1988, School of Engineering and Applied Science, University of Durham, England.

18. BORRILL, P.L., "Microstandards special feature: a comparison of 32 bit buses." *IEEE Micro,* Vol. 5, No. 6, December, 1985, pp. 71–79.

19. BRASCH, F.M., Van NESS, J.E., KANG, S.C., "Simulation of a multiprocessor network for power system problems." *IEEE Trans. PAS–101,* No. 2, February, 1982, pp. 295–301.

20. BURDEN, R.L., FAIRES, J.D., "Numerical analysis." ISBN *0-87150-857-5,* 1985, Pringle, Weber and Schmidt, Boston, Massachusetts, U.S.A.

21. CAFARO, G., MARGARITA, E., "Method for optimal tearing of electrical networks." *Electrical Power and Energy Systems,* Vol. 4, No. 3, July, 1982, pp. 185–191.

22. CARRÉ, B.A., "Solution of load–flow problems by partitioning systems into trees." *IEEE Trans. PAS–87,* No. 11, November, 1968, pp. 1931–1938.

23. CHAN, S.M., BRANDWAJN, V., "Partial matrix refactorisation." *IEEE Trans. PWRS–1,* No. 1, February, 1986, pp. 193–200.

24. CHENG, D.T.Y., "Fast decoupled loadflow with transformer representations and branch outages." *Internal Report,* June, 1983, School of Engineering and Applied Science, University of Durham, England.

25. CRAWFORD, J.H., "The i486 CPU executing instructions in one clock cycle." *IEEE Micro,* Vol. 10, No. 1, February, 1990, pp. 27–36.

26. de BOOR, C., CONTE, S.D., "Elementary numerical analysis." ISBN. 0-07-012447-7, 1980, McGraw Hill Inc., Tokyo, Japan.

27. DEMBART, B., ERISMAN, A.M., "Hybrid sparse matrix methods," *IEEE Trans. CT–20,* No. 6, November, 1973, pp. 641–649.

28. DETIG, D.M., "Effects of special purpose hardware in scientific computation with emphasis on power system applications." *IEEE Trans. PAS–101,* No. 2, February, 1982, pp. 265–270.

29. DETTMER, R., "The affordable gigaflop." *IEE Review* March 1988, pp. 123–126.

30. DEW, P.M., BUCKLEY, T.F., BERZINS, M., "Application of VLSI devices to computational problems in the gas industry." *Report 163,* Department of Computer Studies, University of Leeds, England.

31. DODSON, D.S., "A study of sparse matrix algorithms applicable to electric power flow problems." *Proceedings of 'Array' Conference,* 1981, Floating Point Systems User Society, U.S.A.

32. DUGAN, R.C., DURHAM, I., TALUKDAR S.N., "An algorithm for power system simulation by parallel processing." *IEEE Power Engineering Society,* Summer Meeting, 1979. New York, U.S.A.

33. DUGAN, R.C., DURHAM, I., et al., "Power system simulation on a multiprocessor." *IEEE Power Engineering Society,* Summer Meeting, 1979. New York, U.S.A.

34. DUNCAN, R., "A survey of parallel computer architectures." *IEEE Computer,* Vol. 23, No. 2, February, 1990, pp. 5–16.

35. DYER, S.A., MORRIS, L.R., "A new era for DSP systems design." *IEEE Micro,* Vol. 8, No. 6, December, 1988, pp. 11.

36. DYER, S.A., MORRIS, L.R., "Floating point DSP chips. The end of the supercomputer era?" *IEEE Micro,* Vol. 8, No. 6, December, 1988, pp. 86.

37. EDENFIELD, R.W., GALLUP, M.G., et al., "The 68040 Processor", *IEEE Micro,* Vol. 10, No. 1, February, 1990, pp. 67–77.

38. EL–MARSATAWY, M., MENZIES, K.W., MATHUR, R.M., " A new, exact, diakoptic, fast–decoupled load–flow technique for very large power systems." *IEEE Power Engineering Society,* Summer Meeting, Vancouver, Canada, 1979. New York, U.S.A.

39. ELDER, L., METCALF, M.J., "An efficient method for real–time simulation of large power system disturmances." *IEEE Trans. PAS*–101, No. 2, February, 1982, pp. 334–339.

40. ENNS, M.K., TINNEY, W.F., ALVARADO, F.L., "Sparse matrix inverse factors." *IEEE Trans. PWRS*–5, No. 2, May, 1990, pp. 466–471.

41. FAIRNEY, W., "The integrated grid concept." and "Operational aspects of power systems."*Lecture notes,* Department of Engineering, University of Durham, England.

42. FISCHER, W., "IEEE P1014–a standard for the high performance VME bus." *IEEE Micro,* Vol. 5, No. 1, February, 1985, pp. 31–41.

43. FLAXMAN, J.W., "Multi–processor power system simulation." *Ph.D. Thesis,* 1987, School of Engineering and Applied Science, University of Durham, England.

44. FONG, J., POTTLE, C., "Parallel processing of power system analysis problems via simple parallel microcomputer structures." *IEEE Trans. PAS–97*, No. 5, September, 1978, pp. 1834–1841.

45. FUCCIO, M.L., GADENZ, R.N., et al., "The DSP32C, AT&T's second generation floating point digital signal processor." *IEEE Micro,* Vol. 8, No. 6, December, 1988, pp. 31–48.

46. FULLER, S.H., OUSTERHOUT, J.K., et al., "Multi–microprocessors. An overview and working example." *Proc. IEEE,* Vol. 66, No. 2, February, 1978, pp. 228–237.

47. GLEICK, J., "Chaos." ISBN. 0-7474-0413-5, 1987, pp. 217–220, and colour insert. Sphere Books, London, England.

48. GODERYA, F., METWALLY, A.A., MANSOUR, O., "Fast detection and identification of islands in power networks." *IEEE Trans. PAS–99*, No. 1, January, 1980, pp. 217–221.

49. GÓMEZ, A., FRANQUELO, L.G., "Node ordering algorithms for sparse vector method improvement." *IEEE Trans. PWRS–3*, No. 1, February, 1988, pp. 73–79.

50. GÓMEZ, A., FRANQUELO, L.G., "An efficient ordering algorithm to improve sparse vector methods." *IEEE Trans. PWRS–3*, No. 4, November, 1988, pp. 1538–1540.

51. GORE, A, CORMIE, D, "Designing with the IMS T414 and IMS T800 memory interface." *Technical note 9,* 1987, INMOS, Bristol, England.

52. GROSS, C.A., "Power system analysis." ISBN. 0-471-83732-6, 1979, John Wiley and Sons, New York, U.S.A.

53. GUSTAVSON, F.G., LINIGER, W., WILLOUGHBY, R.A., "Symbolic generation of an optimal Crout algorithm for sparse systems of equations." *J. ACM.,* Vol. 17, 1970, pp. 87–109.

54. HAPP, H.H., "The application of diakoptics to the solutions of power system problems." *in* ERISMAN, A.M., NEVES, K.W., DWARAKANANATH ,M.H., *"Electric Power Problems, The mathematical challenge", SIAM,* ISBN. *0898711738,* 1980, pp. 69–103. Philadelphia, U.S.A.

55. HAPP, H.H., "Parallel processing in power systems." *7th PSCC, Lausanne, Switzerland,* 1981, pp. 9–16.

56. HAPP, H.H., POTTLE, C., WIRGAU, K.A., "Future computer technology for large power system simulation." *IFAC Symposium on "Computer Applications in Large Scale power Systems". New Delhi, India,* 1979, pp. 621–628. Pergammon Press, U.K.

57. HART, J.F., "Computer Approximations." ISBN. 0-471-35630-X, 1968, John Wiley and Sons, New York, U.S.A.

58. HATCHER, W.L., BRASCH, F.M., Van NESS, J.E., "A feasibility study for the solution of transient stability problems by multiprocessor structures." *IEEE Trans. PAS–96,* No. 6, November, 1977, pp. 1789–1797.

59. HELLER, D., "A survey of parallel algorithms in numerical linear algebra." *SIAM Review,* Vol. 20, No. 4, October, 1978, pp. 740–777. Philadelphia, U.S.A.

60. HOCKNEY, R.W., JESSHOPE, C.R., "Parallel Computers 2." ISBN. 0-85274-812-4, 1988, Adam Hilger, Bristol, England.

61. HOMEWOOD, M., MAY, D., et al., "The IMS T800 Transputer." *IEEE Micro,* Vol. 7, No. 5, October, 1987, pp. 10–26.

62. HOUSOS, E.C., WING, O., "Parallel nonlinear minimisation methods with applications to power system problems." *in* ERISMAN, A.M., NEVES, K.W., DWARAKANANATH ,M.H., *"Electric Power Problems, The mathematical challenge", SIAM,* ISBN. *0898711738,* 1980, pp. 403–413. Philadelphia, U.S.A.

63. IEEE., "ANSI/IEEE Standard 754-1985 for Binary Floating Point Aritmetic." *IEEE Computer Society,* 1985. IEEE, Los Alamitos, California, U.S.A.

64. INTEGRATED DEVICE TECHNOLOGY Inc., "IDT7MB6040 dual ($16K \times 64$) data / instruction cache module for general CPUs." *Data sheet,* 1988. IDT, Santa Clara, California, U.S.A.

65. INMOS Ltd., "IMS T800 Transputer," *Data sheet,* 1987. Bristol, U.K.

66. INMOS Ltd., "The Transputer Instruction Set. A compiler writers guide." *Manual,* 1987. Bristol, U.K.

67. INMOS, "IMS C004 programmable link switch." *Data sheet,* 1987, Bristol, England.

68. INTEGRATED DEVICE TECHNOLOGY, Inc., "R3000 and R3010 RISComponents performance brief." 1988, IDT Inc., Santa Clara, California, U.S.A.

69. INTEGRATED DEVICE TECHNOLOGY, Inc., "R3000 family data sheets." *Data sheet,* 1988, IDT Inc., Santa Clara, California, U.S.A.

70. INTEL Corp., "iAPX 86,88 user's manual." 1981. Intel Corp., Santa Clara, California, U.S.A.

71. INTEL Corp., "iAPX 186 high integration 16–bit microprocessor.." *Data sheet,* 1982. Intel Corp., Santa Clara, California, U.S.A.

72. INTEL Corp., "i860 64–bit microprocessor." *Data sheet,* 1989. Intel Corp., Santa Clara, California, U.S.A.

73. IRVING, M.R., STERLING, M.J.H., "Optimal network tearing using simulated annealing." *Submitted, IEE.*

74. IRVING, M.R., STERLING, M.J.H., "Power system simulation pilot study," *Report,* CEGB contract HQ(SP)6341, April, 1988. School of Engineering and Applied Science, University of Durham, England.

75. IRVING, M.R., STERLING, M.J.H., "Efficient Newton–Raphson algorithm for load–flow calculation in transmission and distribution networks." *Proc. IEE, Pt. C,* Vol. 134, No. 5, September, 1987, pp. 325–328.

76. JOHNSON, R.B.I., CORY, B.J., SHORT, M.J., "A tunable integration method for the simulation of power system dynamics. "*IEEE Trans. PWRS*–3, No. 4, November, 1988, pp. 1530–1537.

77. JOETTEN, R., WESS, T., et al., "A new real time simulator for power system studies." *IEEE Trans. PAS*–104, No. 9, September, 1985, pp. 2604–2611.

78. JONES, A.K., CHANSLER, R.J., et al., "Programming issues raised by a multiprocessor." *Proc. IEEE,* Vol. 66, No. 2, February, 1978, pp. 229–237.

79. KANE, G., "MIPS RISC architecture." ISBN. 0-13-584749-4, 1988, Prentice–Hall Inc., Eaglewood Cliffs, U.S.A.

80. KATO, K., HAMMERLUND, A.E., "Real time network topology determination." *IEEE Power Engineering Society,* Summer Meeting, 1977, Mexico.

81. KEYANI, A., "Development of an interactive power system research simulator." *IEEE Trans. PAS*–103, No. 3, March, 1984, pp. 516–521.

82. KEYANI, A., "Study of fast decoupled load flow algorithms with substantially reduced memory requirements." *Electric Power Systems Research,* Vol. 9, 1985, pp. 1–9. Elsevier Sequoia, Lausanne, Switzerland.

83. KLOS, A., BARTCZAK, J., BIALEK, J., "Steady–state power system simulation using process of homeostasis." *25th Universities' Power Engineering Conference,* September, 1990, pp. 803–806 Robert Gordon's Institute of Technology, Aberdeen, Scotland.

84. KNOWLES, A., KANTCHEV, T., "Message passing in a Transputer system." *Microprocessors and Microsystems,* Vol. 13, No. 2, March, 1989, pp. 113–123. Butterworth and Co.

85. KNUTH, D.E., "The art of computer programming, Volume 1, Fundamental Algorithms." ISBN. *0-201-0321-8,* 1973, pp. 278–279. Addison Wesley, London, U.K.

86. KNUTH, D.E., "The art of computer programming, Volume 3, Sorting and sort systems." ISBN. *0-201-03803-X,* 1973, Addison Wesley, London, U.K.

87. KREYSZIG, E., "Advanced engineering mathematics." ISBN. 0-471-88941-5, 1983, John Wiley and Sons, New York, U.S.A.

88. Van LAARHOVEN, P.J.M., AARTS, E.H.L., "Simulated Annealing." ISBN *90-277-2513-6,* 1987, pp. 7–15. D Reidel Publishing Co., Holland.

89. Van LAARHOVEN, P.J.M., AARTS, E.H.L., "Simulated Annealing." ISBN *90-277-2513-6,* 1987, pp. 39–53. D Reidel Publishing Co., Holland.

90. LATIMER, J.R., MASIELLO, R.D., "Design of a dispatcher training system." *IEEE Power Engineering Society,* Industry Computer Application Conference, 24–25th May, 1977, pp. 87–92. Toronto, Canada.

91. LAWRIE, D.H., "Access and alignment of data in an array processor." *IEEE Trans. Computers,* Vol. 24, No. 12, December, 1975, pp. 173–183.

92. LAZZERINI, B., "The RISC approach." *IEEE Micro,* Vol. 9, No. 1, February, 1989, pp. 57–64.

93. LEWIS, J.G., POOLE, W.G., "Ordering algorithms applied to sparse matrices in electrical power problems." *in* ERISMAN, A.M., NEVES, K.W., DWARAKANANATH ,M.H., *"Electric Power Problems, The mathematical challenge", SIAM,* ISBN. *0898711738,* 1980, pp. 115–124. Philadelphia, U.S.A.

94. MAGEE, D., FLYNN, F., et al., "A large two computer dispatcher training simulator." *IEEE Trans. PAS*–104, No. 6, June, 1985, pp. 1433–1438.

95. MOTOROLA INC., "MC68020." *Data sheet,* East Kilbride, Glasgow, Scotland.

96. MOTOROLA INC., "MC68030 Technical summary." *Data sheet,* East Kilbride, Glasgow, Scotland.

97. MOTOROLA INC., "MC68881." *Data sheet,* East Kilbride, Glasgow, Scotland.

98. MOTOROLA INC., "MC68882 Technical summary." *Data sheet,* East Kilbride, Glasgow, Scotland.

99. MUKAI, H., "Parallel algorithms for solving systems of nonlinear equations." *Computers and Mathematics with Applications,* Vol. 7, 1981, pp. 235–250.

100. NICOUD, J.D., "Video RAMS." *IEEE Micro,* Vol. 8, No. 1, February, 1988, pp. 8–27.

101. NICOUD, J.D., TYRRELL, A.M., "The Transputer T414 instruction set." *IEEE Micro,* Vol. 9, No. 3, June, 1989, pp. 61–75.

102. OGBUOBIRI, E.C., TINNEY, W.F., WALKER, J.W., "Sparsity directed decomposition for Gaussian Elimination in matrices." *IEEE Trans. PAS*–89, No. 1, January, 1970, pp. 141–150.

103. O'GRADY, E.P., "A performance study of mutual exclusion / synchronization mechanisms in an IEEE 796 bus multiprocessor." *IEEE Micro,* Vol. 5, No. 4, August, 1985, pp. 32–47.

104. OTT, G.E., WALKER, L.N., WONG, D.T.Y, "Hybrid simulation for long term dynamics." *IEEE trans. PAS*–96, No. 3, May, 1977, pp. 907–915.

105. PAPAMICHALIS, P., SIMAR, R., "The TMS 320C30 floating point digital signal processor." *IEEE Micro,* Vol. 8, No. 6, December, 1988, pp. 13–29.

106. PATTERSON, D.A., SEQUIN, C.H., "RISC–1: a reduced instruction set VLSI computer." *proc. Eighth International Symposium on Computer Architecture,* IEEE, May, 1981, pp. 443–457. IEEE CS Press, los Alamitos, califomia, U.S.A.

107. PEITGEN, H.–O., RICHTER, P.H., "The beauty of fractals." ISBN. 3-540-15851-0, 1986. Springer Verlag, Berlin, West Germany.

108. PENELLO,T.J., "Compiler challenges with RISC." *IEEE Micro,* Vol. 10, No. 1, February, 1990, pp. 37–43.

109. PODMORE, R., GIRI, J.C., et al., "An advanced dispatcher training simulator." *IEEE trans. PAS*–101, No. 1, January, 1982, pp. 17–25.

110. POTTLE, C., NORIN, R., SMITH, B., "Proceedings of a computer hardware workshop." *in* ERISMAN, A.M., NEVES, K.W., DWARAKANANATH ,M.H., *"Electric Power Problems, The mathematical challenge", SIAM,* ISBN. *0898711738,* 1980, pp. 499–512. Philadelphia, U.S.A.

111. POUNTAIN, D., MAY, D., "A tutorial introduction to OCCAM programming." 1987, INMOS, Bristol, England.

112. PRAIS, M., BOSE, A., "Topology processor that tracks network modifications over time." *IEEE Trans. PWRS*–3, No. 3, August, 1988, pp. 992–998.

113. PRAIS, M., ZHANG, G., et al., "Operator training simulator: algorithms and test results." *IEEE Trans. PWRS*–4, No. 3, August, 1989, pp. 1154–1166.

114. RAFIAN, M., STERLING, M.J.H., IRVING, M.R., "Real–time power system simulation." *Proc. IEE, Pt. C,* Vol. 134, No. 3, May, 1987, pp. 206–223.

115. RAFIAN, M., STERLING, M.J.H., IRVING, M.R., "Decomposed load flow." *Internal Report,* December, 1984, School of Engineering and Applied Science, University of Durham, England.

116. RAFIAN,M., STERLING, M.J.H., IRVING, M.R., "Parallel processor algorithm for power system simulation." *Proc. IEE, Pt. C,* Vol. 135, No. 4, July, 1988, pp. 285–290.

117. REID, J.K., "A survey of sparse matrix computation." *in* ERISMAN, A.M., NEVES, K.W., DWARAKANANATH ,M.H., *"Electric Power Problems, The mathematical challenge", SIAM,* ISBN. *0898711738,* 1980, pp. 41–68. Philadelphia, U.S.A.

118. RICHARDSON, S., GANAPATHI, M., "Code optimisation across procedures." *IEEE Computer,* Vol. 22, No. 2, February, 1989, pp. 42–50.

119. RISTANOVIĆ, P., BJELOGRLIĆ, M., BABIĆ, B.S., "Improvements in sparse matrix / vector technique applications for on–line load flow calculation." *IEEE Trans. PWRS*–4, No. 1, February, 1989, pp. 190–196.

120. ROWEN, C., JOHNSON, M., RIES, P., "The MIPS R3010 floating point processor." *IEEE Micro,* Vol. 8, No. 3, June, 1988, pp. 53–62.

121. RYGOL, M., WATSON, T., "Connecting INMOS links." *Technical note 18,* 1987, INMOS, Bristol, England.

122. SAGE, A.P., SMITH, S.L., "Real–time digital simulation for systems control." *Proc. IEEE,* Vol. 54, No. 12, December, 1966, pp. 1802–1812.

123. SAIKAWA, K., GOTO, M., et al., "Real time simulation of large scale power system dynamics for a dispatcher training simulator." *IEEE trans. PAS*–103, No. 12, December, 1984, pp. 3496–3501.

124. SALEH, A.O.M., LAUGHTON, M.A., "Cluster analysis of power system networks for array processing solutions." *Proc. IEE, Pt. C,* Vol. 132, No. 4, July, 1985, pp. 172–178.

125. SANGIOVANNI–VINCENTELLI, A., CHEN, L.–K., CHUA, L.O., "An efficient heuristic cluster algorithm for tearing large–scale networks." *IEEE Trans. CAS*–24, No. 12, December, 1977, pp. 709–717.

126. SASAKI, H., AOKI, K., YOKOYAMA, R., "A parallel computation algorithm for static state estimation by means of matrix inversion lemma." *IEEE Trans. PWRS*–2, No. 3, August, 1987, pp. 624–632.

127. SASSON, A.M., EHRMANN, S.T., et al., "Automatic power system network topology determination." *IEEE Trans. PAS*–92, 1973, pp. 610–618.

128. SATO, K., YAMAZAKI, Z., et al., "Dynamic simulation of a power system network for dispatcher training." *IEEE Trans. PAS*–101, No. 10, October, 1982, pp. 3742–3750.

129. SCHNEIDER, K., "Evolving the best solution." *Electrical Revue,* Vol. 222, No. 19, 4th October, 1989, pp. 27–28.

130. SHIPLEY, R.B., AYOUB, A.K., "Bus admittance tearing and the pertial inverse." *IEEE Power Engineering Society,* Winter Meeting, 1972. New York, U.S.A.

131. SIEWIOREK, D.P., BELL, C.G., NEWELL, A., "Computer structures. Principles and examples." ISBN. 0-07-066567-2, 1982, pp. 508–532. McGraw Hill, Tokyo, Japan.

132. SOHIE, G.R.L., KLOKER K.L., "A digital signal processor with IEEE floating point arithmetic." *IEEE Micro,* Vol. 8, No. 6, December, 1988, pp. 49–67.

133. STOTT, B., "Decoupled Newton Loadflow." *IEEE Power Engineering Society,* Winter Meeting, 1972. New York, U.S.A.

134. STOTT, B., "Effective starting process for Newton–Raphson load flows." *Proc. IEE,* Vol. 118, No. 8, August, 1971, pp. 983–987.

135. STOTT, B., ALSAÇ, O., "Fast decoupled load flow." *IEEE Trans PAS*–93, No. 3, May, 1974, pp. 859–869.

136. STOTT, B., HOBSON, E., "Solution of large power system networks by ordered elimination: a comparison of ordering schemes." *Proc. IEE,* Vol. 118, No. 1, January, 1971, pp. 125–134.

137. STROUSTRUP, B., "The C++ programming language." ISBN. 0-201-12078-X, 1986, Addison–Wesley, Reading, Massachusetts, U.S.A.

138. SUAUMAGO, I., SUZUKI, M., MIYAMA, K., "Development of a large scale dispatcher training simulator and training results." *IEEE trans. PWRS*–1, No. 2, May, 1986, pp. 67–75.

139. SULLIVAN, A.C., REICHERT, K., SALY, S., "An on–line technique for network topology determination for use in security analysis studies." *IEE Conference on "on–line operation and optimisation on transmission and distribution systems", London,* 1976, pp. 61–66.

140. TAKATOO, M., ABE, S., et al., "Floating point vector processor for power system simulation." *IEEE Trans. PAS*–104, No. 12, December, 1985, pp. 3361–3366.

141. TAOKA, H., ABE, S., "Fast solution of sparse network equations with an array processor." *IEEE Trans. PWRS*–1, No. 4, November, 1988, pp. 215–221.

142. TAUB, D.M., "Improved control acquisition scheme for the IEEE 896 Futurebus." *IEEE Micro,* Vol. 7, No. 3, June, 1987, pp. 52–62.

143. TAYLOR, A.J.E., IRVING, M.R., STERLING, M.J.H., "Power system partitioning using genetic optimisation." *25th Universities' Power Engineering Conference,* September, 1990, pp. 607–610. Robert Gordon's Institute of Technology, Aberdeen, Scotland.

144. TEWARSON, R.P., "Sparse matrices." ISBN. 0-12-685650-8, 1973, Academic Press, New York, U.S.A.

145. TEXAS INSTRUMENTS INCORPORATED, "Math / Utilities for Ti 58 and Ti 59 calculators." *Solid State Software Module Handbook,* Normal Distribution, pp. 48–50. Texas Instruments, Dallas, Texas, U.S.A.

146. TINNEY. W.F., BRANDWAJN, V.CHAN, S.M., "Sparse vector methods." *IEEE Trans. PAS*–104, No. 2, February, 1985, pp. 295–301.

147. TINNEY, W.F., WALKER, J.W., "Direct solution of sparse matrix equations by optimally ordered triangular factorisation." *Proc. IEEE,* Vol. 55, No. 11, November, 1967, pp. 1801–1809.

148. UNDRILL, J.M., HAPP, H.H., "Automatic sectionalization of power system networks for network solutions." *IEEE Trans. PAS*–90, No. 1, January, 1971, pp. 46–53.

149. WATSON, W., "Forth Processor hardware." *Electronics And Wireless World,*

150. WEITEK Corp., "WTL 1032, WTL 1033 single precision multiplier / ALU." *Data sheet,* Sunnyvale, California, U.S.A.

151. WEITEK Corp., "WTL 1064, WTL 1065 double precision multiplier / ALU." *Data sheet,* Sunnyvale, California, U.S.A.

152. WEITEK Corp., "WTL 1066, 32×32, six port register file." *Data sheet,* Sunnyvale, California, U.S.A.

153. WEITEK Corp., "WTL 1164, WTL 1165 low latency, double precision multiplier / ALU." *Data sheet,* Sunnyvale, California, U.S.A.

154. WEITEK Corp., "WTL 2264, WTL 2265 vector floating point processor." *Data sheet,* Sunnyvale, California, U.S.A.

155. WEITEK Corp., "XL–8136 Program sequencing unit." *Data sheet,* Sunnyvale, California, U.S.A.

156. WEITEK Corp., "XL–8137 Integer processing unit." *Data sheet,* Sunnyvale, California, U.S.A.

157. WEITEK Corp., "WTL/XL–3164, WTL/XL–3364 64–bit floating point data path units." *Data sheet,* Sunnyvale, California, U.S.A.

158. WEITEK Corp., "XL series overview." January, 1988, Sunnyvale, California, U.S.A.

159. WEXLER, J., PRIOR, D., "Solving problems with Transputers: background and experience." *Microprocessors and microsystems,* Vol. 13, No. 2, March, 1989, pp. 67–78.

160. YU, D.C., "A new parallel LU decomposition method." *IEEE Trans. PWRS–5,* No. 1, February, 1990, pp. 303–309.

161. YU, D.C., WANG, H., "A new approach for the forward and backward substitutions of parallel solution of sparse linear equations based on dataflow architecture." *IEEE Trans. PWRS–5,* No. 2, May, 1990, pp. 621–627.

162. ZOLLENKOPF, K., "Bi–Factorisation—basic computational algorithm and programming techniques." *in* REID, J.K., *"Large Sparse Sets of Linear Equations.",* 1969, pp. 75–96. Academic Press, New York, U.S.A.

Bibliography

ABOU-RABIA, O., BIRTA, L.G., "The design of a multimicrocomputer system for continuous system simulation." *J. Microcomputer Applications,* Vol. 10, 1987, pp. 137–151. Academic Press, New York, U.S.A.

ALVARADO, F.L., LIU, Y., "General purpose symbolic simulation tools for electric networks." *IEEE Trans. PWRS*–3, No. 2, May, 1988, pp. 689–697.

AZZAM, M., MARSH, J.F., "Subsystem bad data identification for power systems." *Report, Brunel University,* Uxbridge, England.

BHUYAN, L.N., YANG, Q., AGRAWAL, D.P., "Performance of multiprocessor interconnection networks." *IEEE Computer,* Vol. 22, No. 2, February, 1989, pp. 25–37.

BIRMAN, M., SAMUELS, A., et al., "Developing the WTL3170 / WTL3171 SPARC floating point coprocessors." *IEEE Micro,* Vol. 5, No. 1, February, 1985, pp. 55–63.

BOMING, Z., NIADE, X., SHIYANG, W., "Unified piecewise solution of power system networks combining both branch cutting and node tearing." *Electical Power and Energy Systems,* Vol. 11, No. 4, October, 1989, pp. 283–288. Butterworth and Co.

BRITTON, J.P., "Improved area interchange control for Newton's Method load flows." *IEEE Trans. PAS*–88, No. 10, October, 1969, pp. 1577–1581.

DUSONCHET, Y.P., TALUKDAR, S.N., et al., "Loadflows using a combination of Point Jacobi and Newton's Method." *IEEE Power Group,* Summer Meeting, 1970. Los Angeles, California, U.S.A.

GEORGE, A., LIU, J.W., "Computer solution of large sparse positive definite systems." ISBN. 0-13-165274-5, 1981, Prentice–Hall, Eaglewood Cliffs, U.S.A.

GUTTAG, K.M., ALBERS, T.M., et al., "The TMS34010. An embedded microprocessor." *IEEE Micro,* Vol. 8, No. 3, June, 1988, pp. 39–52.

HACHTEL, G.D., SANGIOVANNI-VINCENTELLI, A.L., "A survey of third generation simulation techniques." *Proc. IEEE,* Vol. 69, No. 10, October, 1981, pp. 1264–1280.

HONG, H.W., YOUNG, C.C., GALPERIN, R., "Implementation experience with super–microcomputer technology." *IEEE Trans. PWRS*–3, No. 4, November, 1988, pp. 1713–1716.

HUMPAGE, W.D., WONG, K.P., NGUYEN, T.T., "PROLOG Network graph generation in system surveillance." *Electric Power Systems Research,* Vol. 9, pp. 37–48. Elsevier Sequoia, Lausanne, Switzerland.

IACOBOVICI, S., "A pipelined interface for high floating point performance with precise exceptions." *IEEE Micro,* Vol. 8, No. 3, June, 1988, pp. 77–87.

KAWASAKI, S., WATABE, M., MORINAGA, S., "A floating point VLSI chip for the TRON architecture." *IEEE Micro,* Vol. 9, No. 3, June, 1989, pp. 27–44.

KERNIGHAN, B.W., RITCHIE, D.M., "The C programming language." ISBN. 0-13-110163-3, 1978, Prentice–Hall Inc., Eaglewood Cliffs, N.J., U.S.A.

KESAVAN, H.K., PAI, M.A., BHAT, M.V., "Piecewise solution of the loadflow problem." *IEEE Power Engineering Society,* Summer Meeting, 1971. Portland, U.S.A.

KOMORI, S., MIYATA, S., TERADA, H., "The data driven microprocessor." *IEEE Micro,* Vol. 9, No. 3, June, 1989, pp. 45–49.

McMAHON, F.M., "The Livermore FORTRAN kernels: a computer test of the numerical performance range." 1986, Lawrence Livermore national Laboratory, California, U.S.A.

MAMANDUR, K.R.C., BERG, G.J., "Automatic adjustment of generator voltages in Newton–Raphson method of power flow solutions." *IEEE Power Engineering Society,* Summer Meeting, 1981. Portland, Oregon, U.S.A.

MARSH, J.F., AZZAM, M., "MCHSE: a versatile framework for the design of two–level power system state estimators." *Proc. IEE, Pt. C,* Vol. 135, No. 4, July, 1988, pp. 291–298.

MARSH, J.F., "Decomposition of power systems for state estimation and bad data analysis." *Submitted IEEE.*

NAGDY, N.M., RUMPEL, D., "Methods of potential network topology description." *Electrical Power and Energy Systems,* Vol. 5, No. 1, January, 1983, pp. 55–60. Butterworth and Co.

ROY, L., "Piecewise solution of large electrical systems by nodal admittance matrix." *IEEE Power Engineering Society,* Summer Meeting, 1971. Portland, Oregon, U.S.A.

STERLING, M.J.H., "Power system control." *IEE,* ISBN. 0-86341-085-5, Peter Peregrinus Ltd., London, 1978.

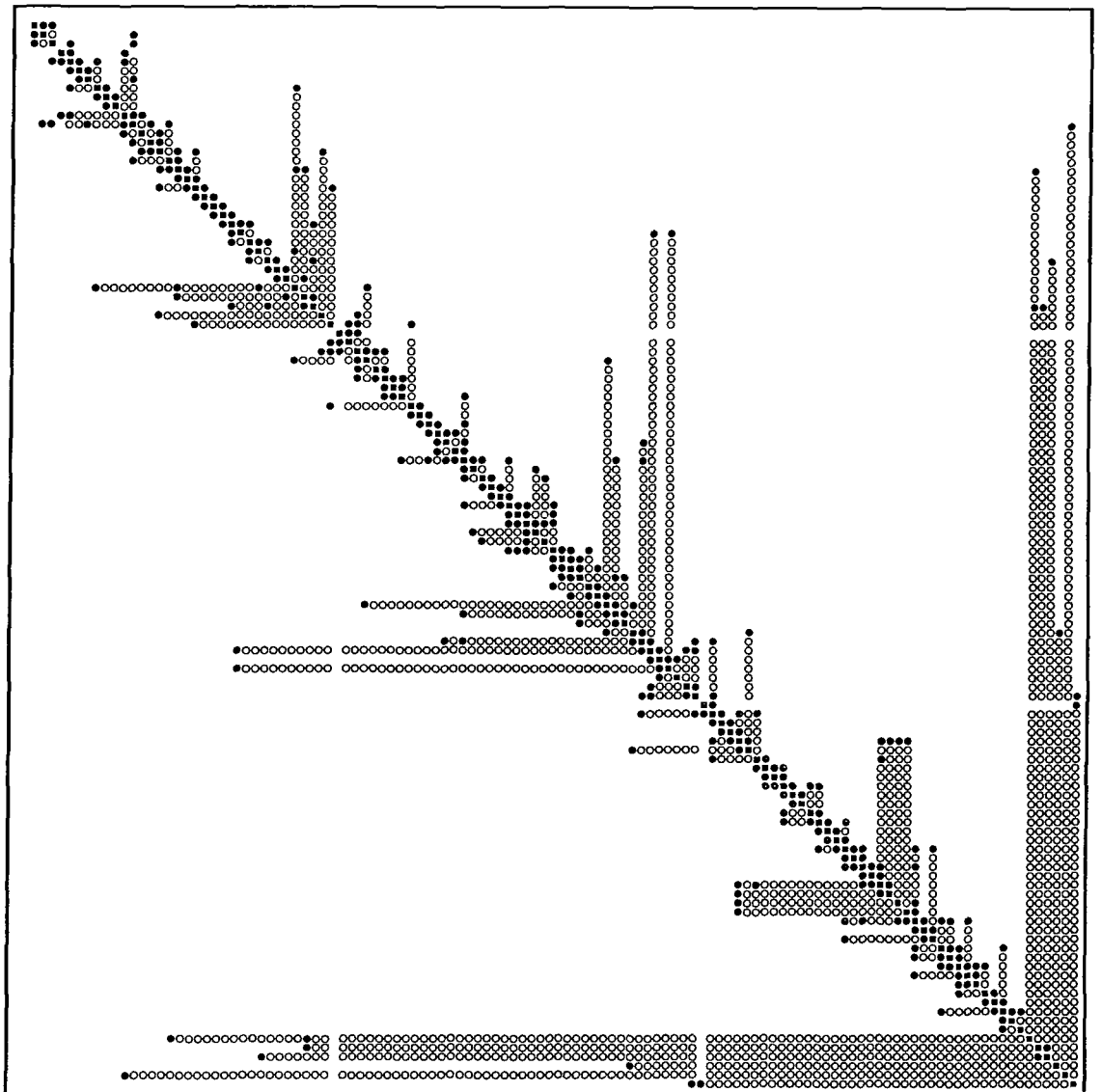
TALUKDAR, S.N., THOMAS, D., "Modular algorithm and multiprocessors for simulating power systems." *EPRI report, EL-566-SR, "Exploring applications of Parallel Processing to Power Systems".*

TINNEY, W.F., BRIGHT, J.M., "Adaptive reductions for power system equivalents." *IEEE Trans PRWS–2,* No. 2, May, 1987, pp. 351–360.

WEITEK Corp., "WEITEK Corporation product summary." March, 1988, Sunnyvale, California, U.S.A.

Appendix A.

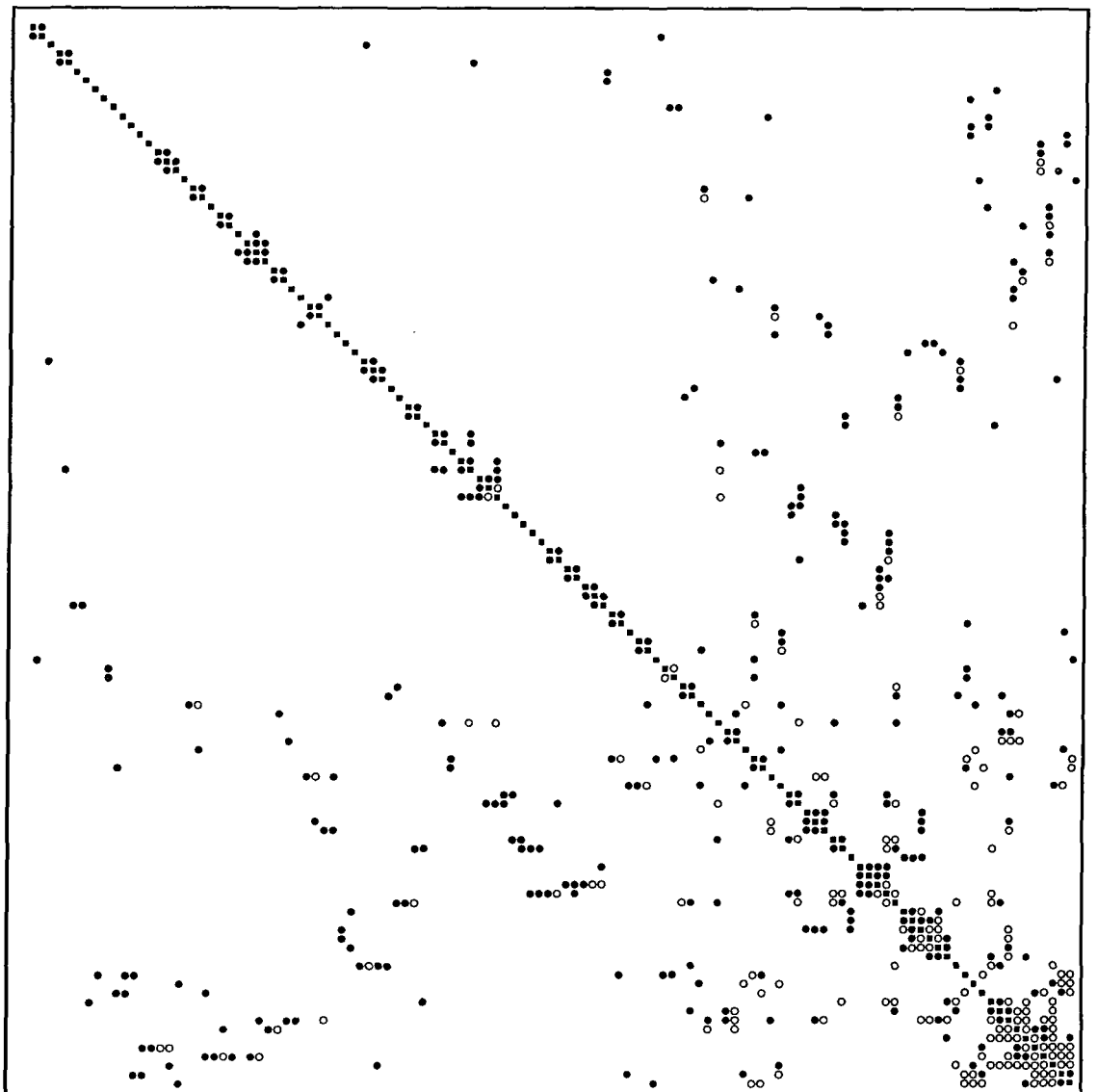
Matrix Plots



Figure—1 Tinney-1 ordering of the 118 node network.

```
Ordering took                  100 ms
Pointer alteration took         30 ms
Reduction took                  60 ms
Solution took                   10 ms
Calculation took                00 ms
Ordering, reduction and solution took   200 ms plus calculation   200 ms
Elements:  before | after        476   1143    846  472.6%
Multiplications generated      11190   2168
Additions  generated           10165   2050
Divisions  generated             118      0
Multiply-additions generated   11190   2168
Stalls generated                 114      1
```

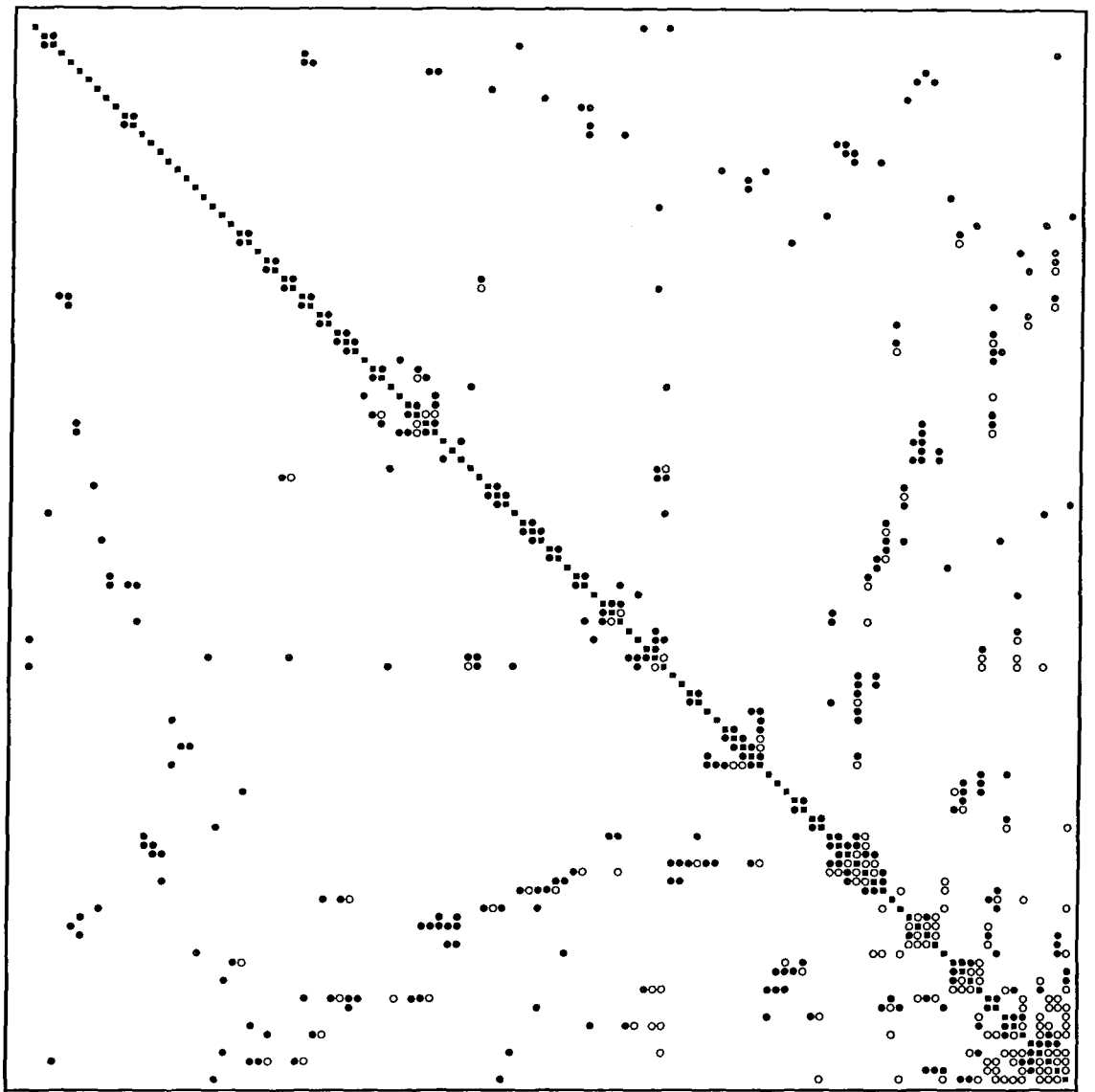
Figure A.1 Matrix of 118 node network, Tinney–1 ordering.



Figure—2 Tinney-2 ordering of the 118 node network.

```
Ordering took                  50 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took    80 ms plus calculation    80 ms
Elements:  before | after       476    384     87   48.6%
Multiplications generated       930    650
Additions  generated            664    532
Divisions  generated            118      0
Multiply-additions generated    930    650
Stalls generated                 49      4
```

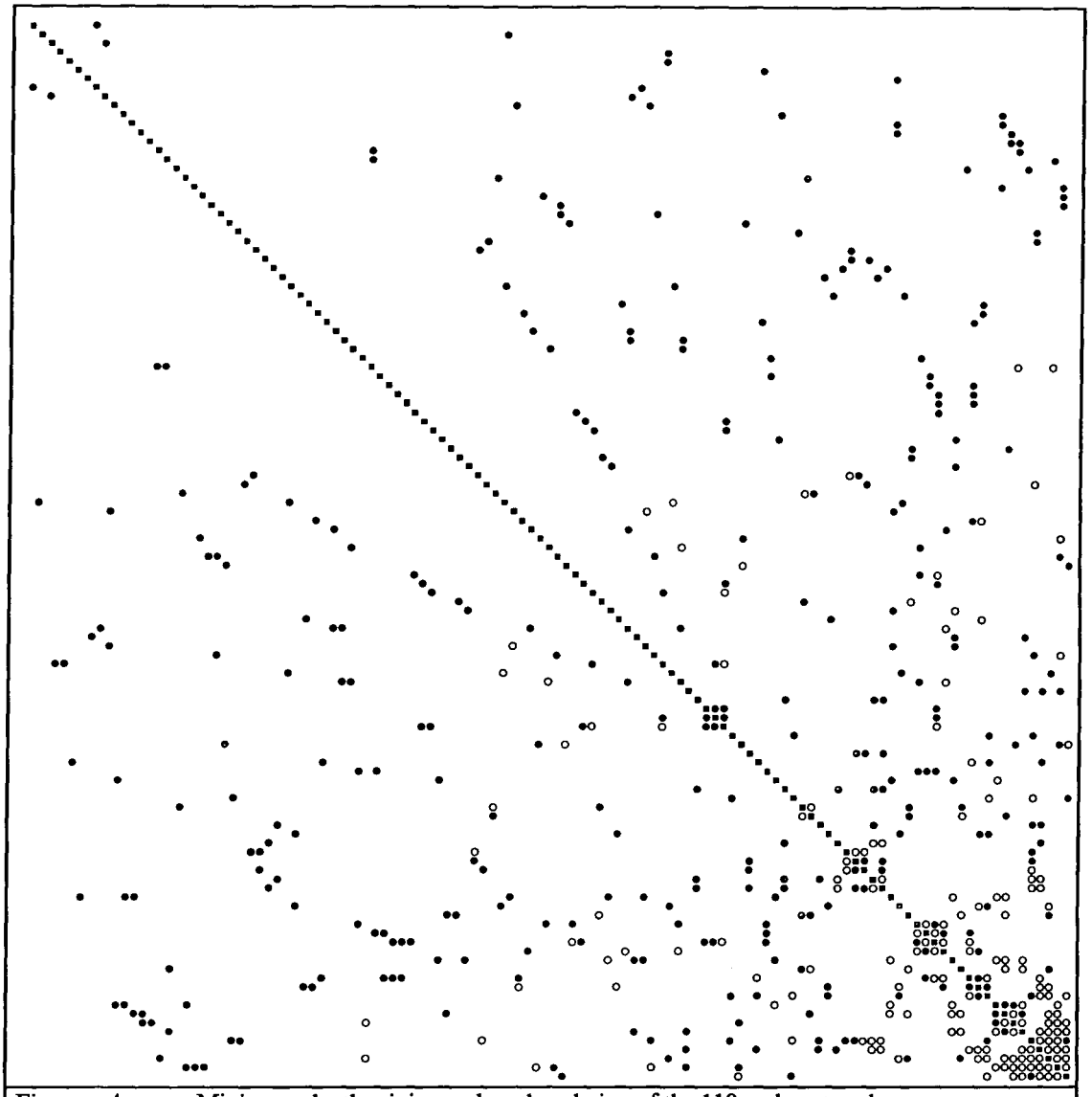
Figure A.2 Matrix of 118 node network, Tinney–2 ordering.



Figure—3 Tinney-3 ordering of the 118 node network.

```
Ordering took                  420 ms
Pointer alteration took         10 ms
Reduction took                  00 ms
Solution took                   10 ms
Calculation took                00 ms
Ordering, reduction and solution took   440 ms plus calculation   440 ms
Elements:  before | after       476    381     84   46.9%
Multiplications generated       910    644
Additions  generated            647    526
Divisions  generated            118      0
Multiply-additions generated    910    644
Stalls generated                 49      3
```

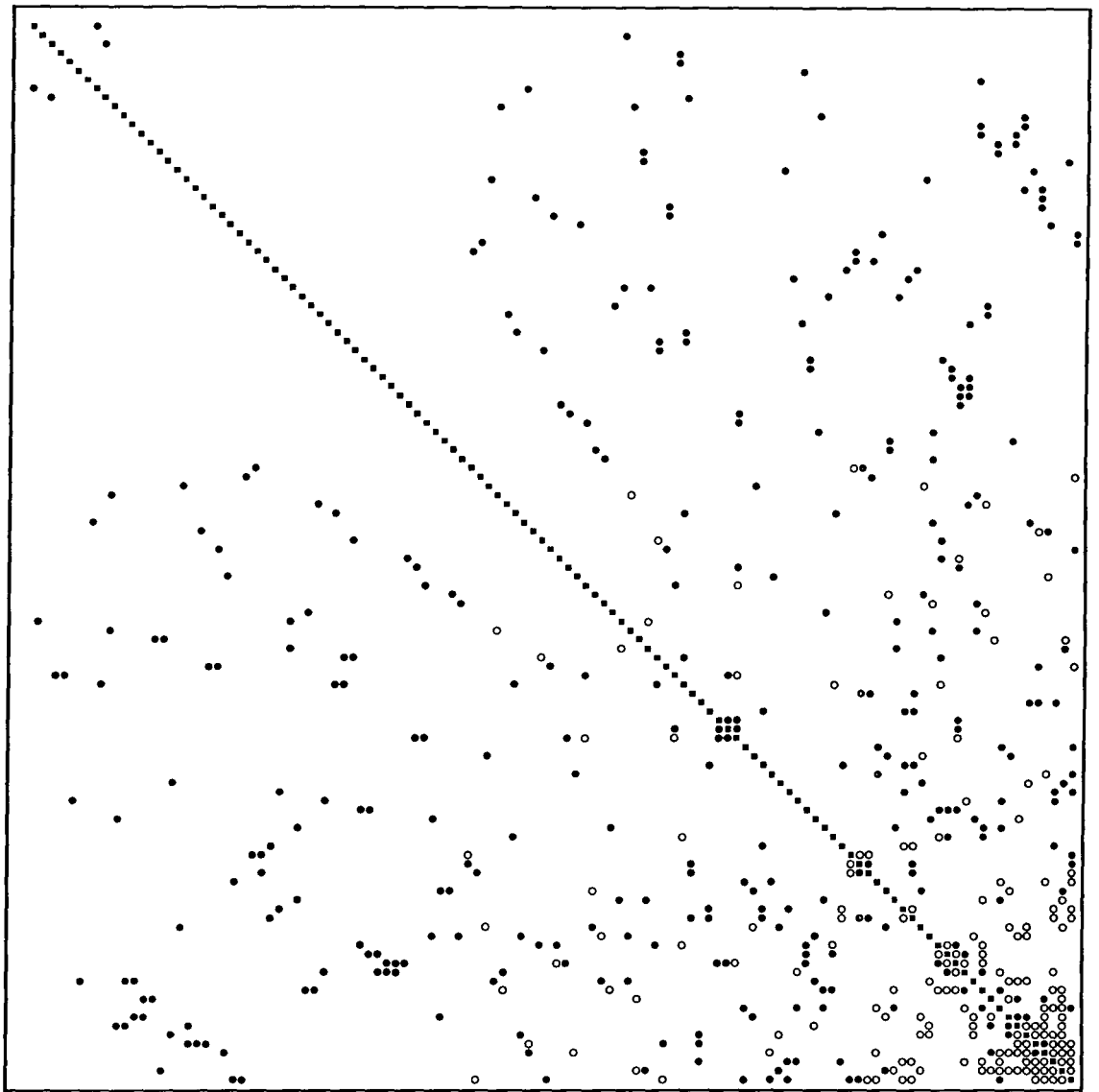
Figure A.3 Matrix of 118 node network, Tinney–3 ordering.



Figure—4 Minimum-depth minimum-length ordering of the 118 node network.

```
Ordering took                  60 ms
Pointer alteration took        10 ms
Reduction took                 00 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took    80 ms plus calculation    80 ms
Elements:  before | after        476    382     85   47.5%
Multiplications generated        916    646
Additions  generated             652    528
Divisions  generated             118      0
Multiply-additions generated     916    646
Stalls generated                  18      1
```

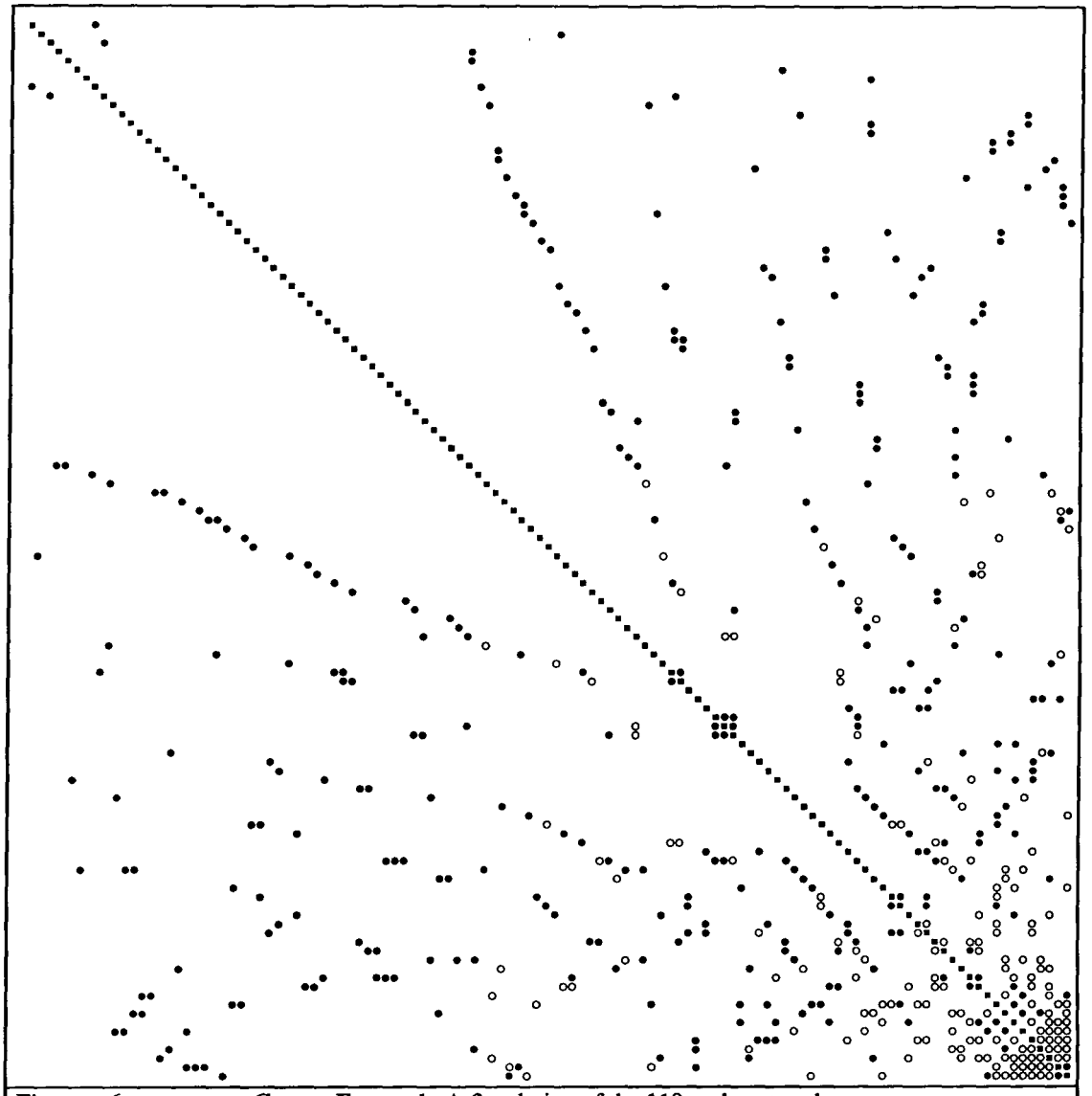
Figure A.4 Matrix of 118 node network, MDML ordering.



Figure—5 Gomez-Franquelo A-1 ordering of the 118 node network.

```
Ordering took                  60 ms
Pointer alteration took        10 ms
Reduction took                 00 ms
Solution took                  20 ms
Calculation took               00 ms
Ordering, reduction and solution took    90 ms plus calculation    90 ms
Elements:  before | after       476    383     86   48.0%
Multiplications generated       922    648
Additions  generated            657    530
Divisions  generated            118      0
Multiply-additions generated    922    648
Stalls generated                 15      1
```

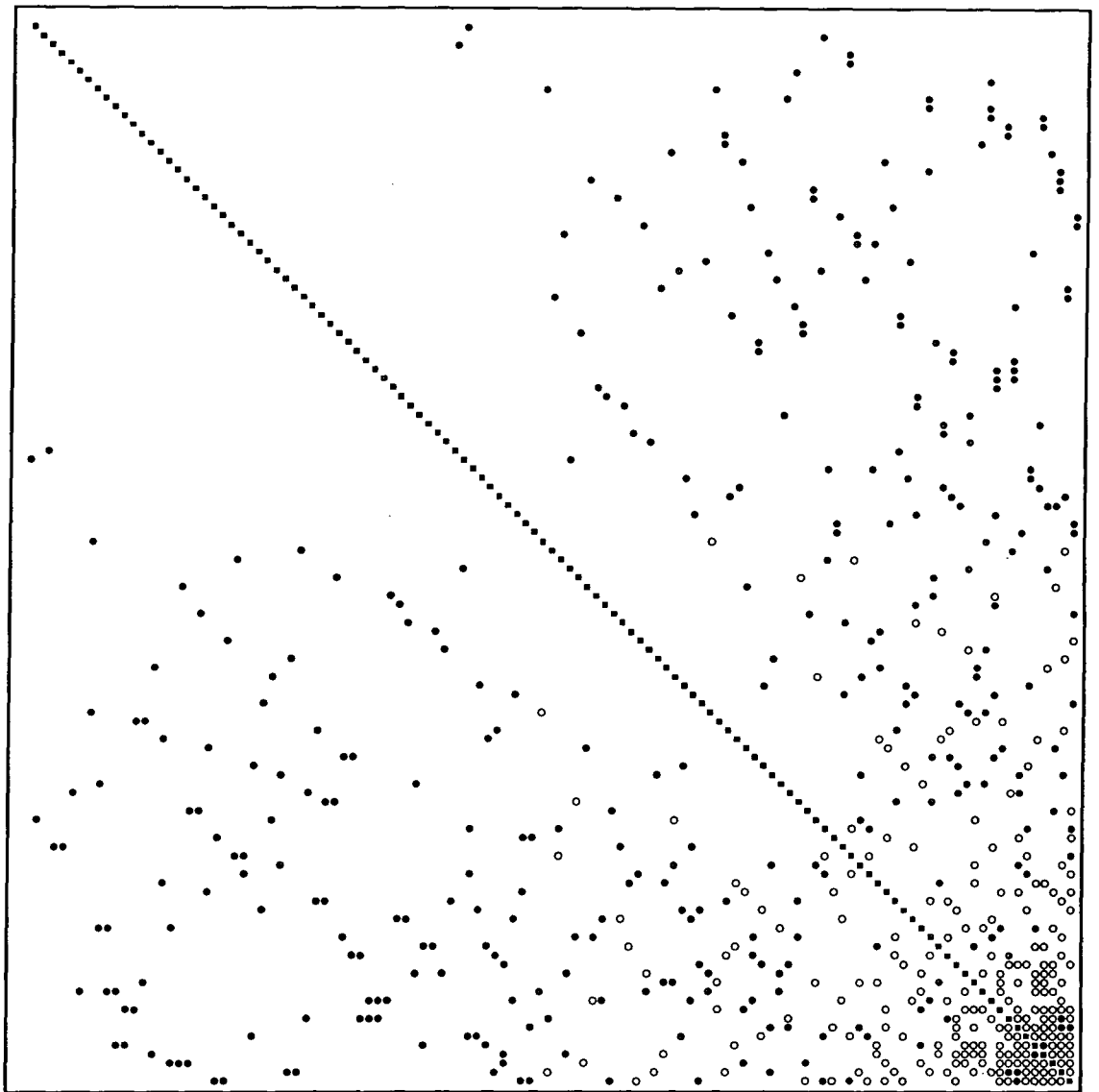
Figure A.5 Matrix of 118 node network, GF–1 ordering.



Figure—6 Gomez-Franquelo A-2 ordering of the 118 node network.

```
Ordering took                  60 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took    90 ms plus calculation    90 ms
Elements:  before | after       476    383     86   48.0%
Multiplications generated       922    648
Additions  generated            657    530
Divisions  generated            118      0
Multiply-additions generated    922    648
Stalls generated                 11      1
```

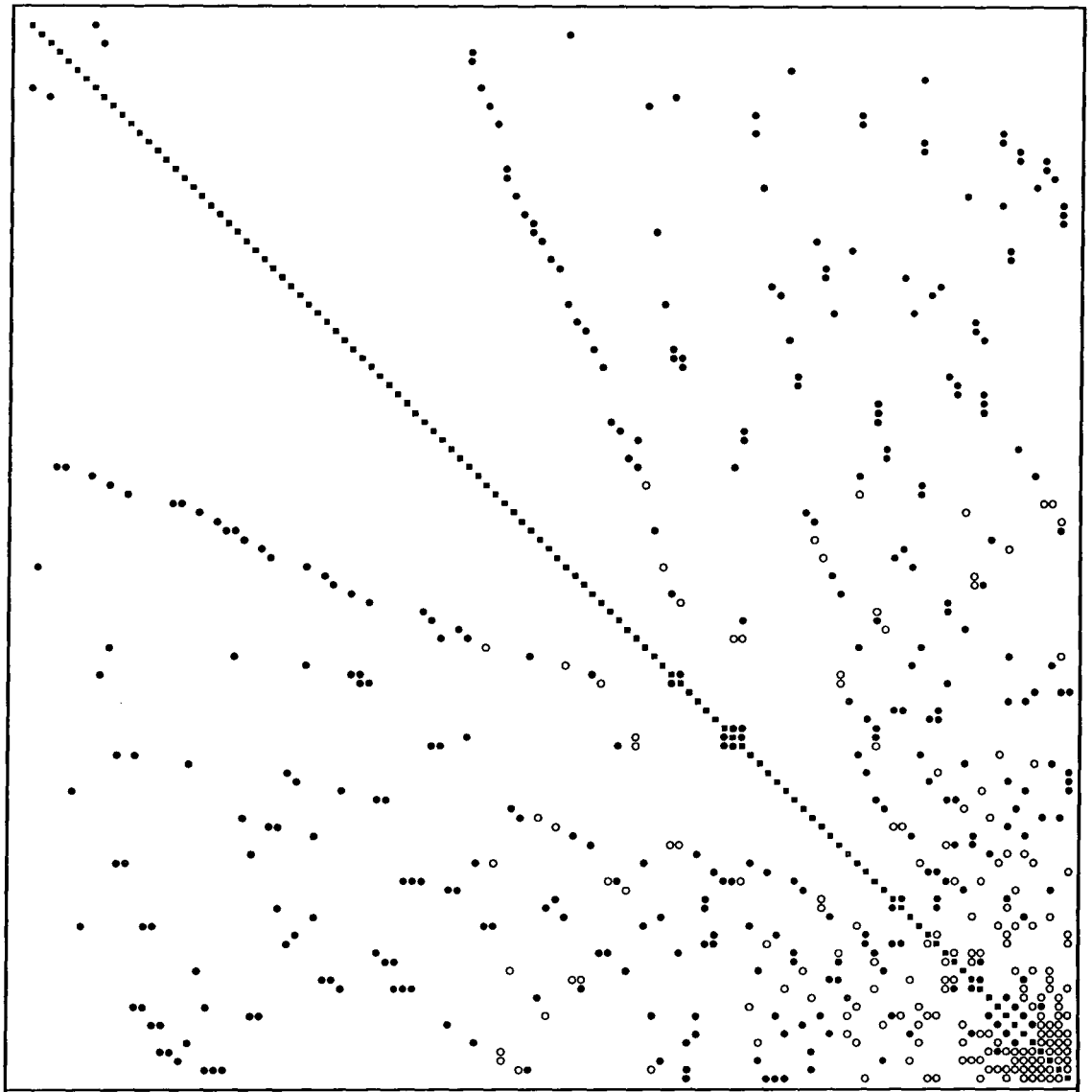
Figure A.6 Matrix of 118 node network, GF–2 ordering.



Figure—7 Gomez-Franquelo A-3 ordering of the 118 node network.

```
Ordering took                  60 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  00 ms
Calculation took               00 ms
Ordering, reduction and solution took    80 ms plus calculation    80 ms
Elements:  before | after       476     409    112   62.6%
Multiplications generated      1148     700
Additions  generated            857     582
Divisions  generated            118       0
Multiply-additions generated   1148     700
Stalls generated                  7       1
```

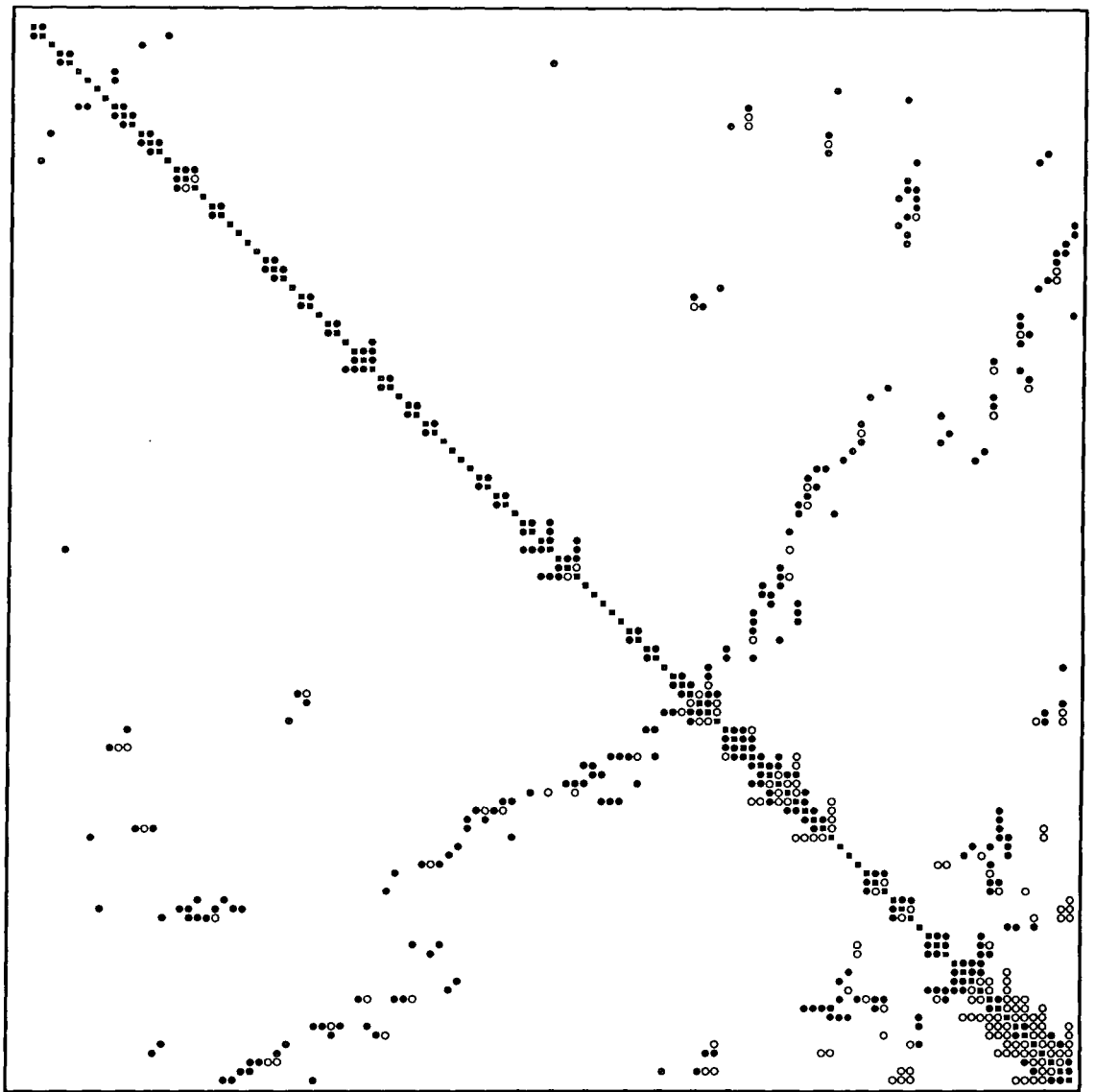
Figure A.7 Matrix of 118 node network, GF–3 ordering.



Figure—8 MD - Least Recently Used ordering of the 118 node network.

```
Ordering took                  60 ms
Pointer alteration took        00 ms
Reduction took                 10 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took    80 ms plus calculation    80 ms
Elements:  before | after        476    383     86   48.0%
Multiplications generated        922    648
Additions  generated             657    530
Divisions  generated             118      0
Multiply-additions generated     922    648
Stalls generated                  11      1
```

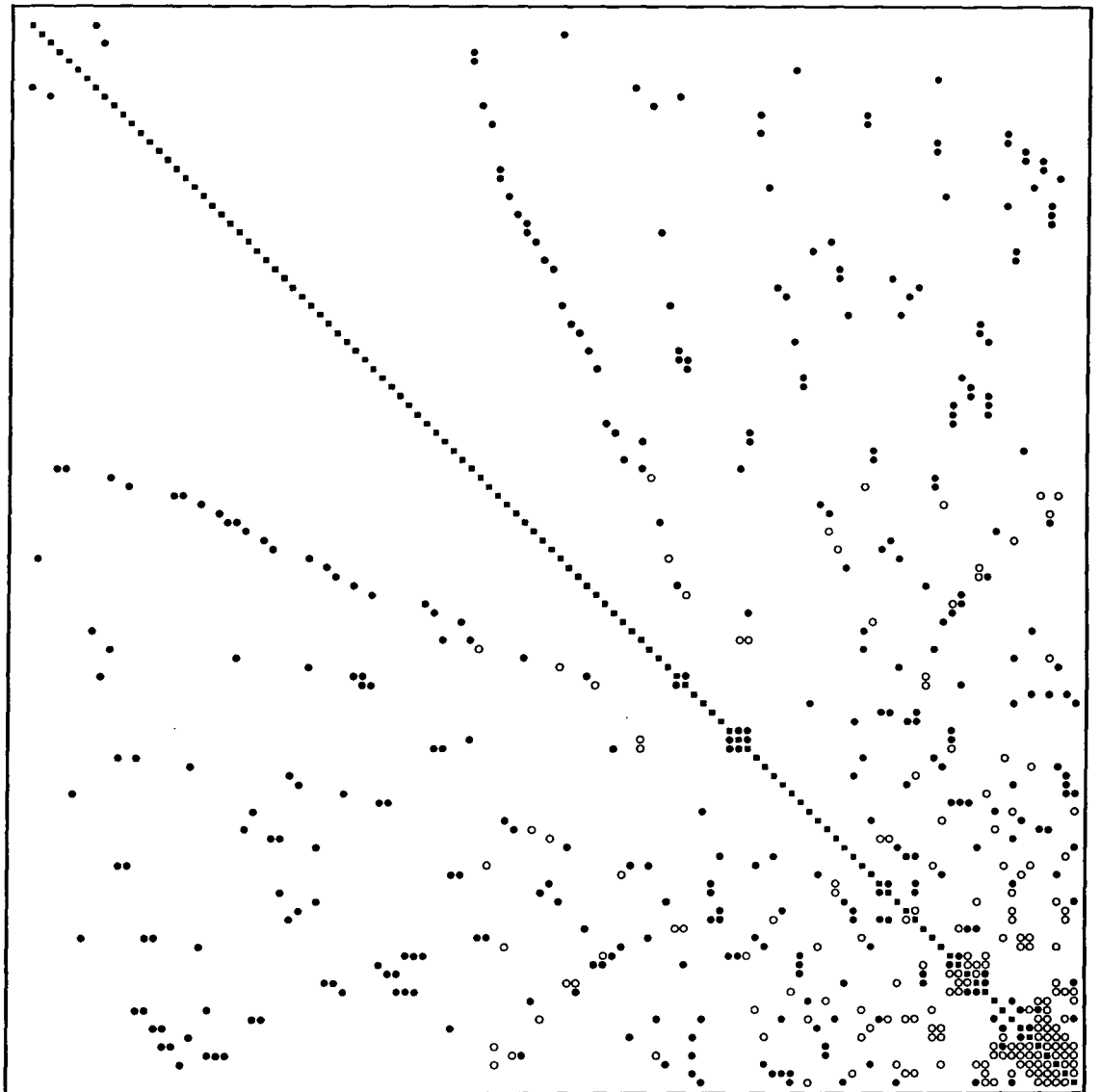
Figure A.8 Matrix of 118 node network, MDLRU ordering.



Figure—9 MD - Reversed LRU ordering of the 118 node network.

```
Ordering took                  50 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  00 ms
Calculation took               00 ms
Ordering, reduction and solution took    70 ms plus calculation    70 ms
Elements:  before | after        476    381     84   46.9%
Multiplications generated        910    644
Additions  generated             647    526
Divisions  generated             118      0
Multiply-additions generated     910    644
Stalls generated                  62      4
```

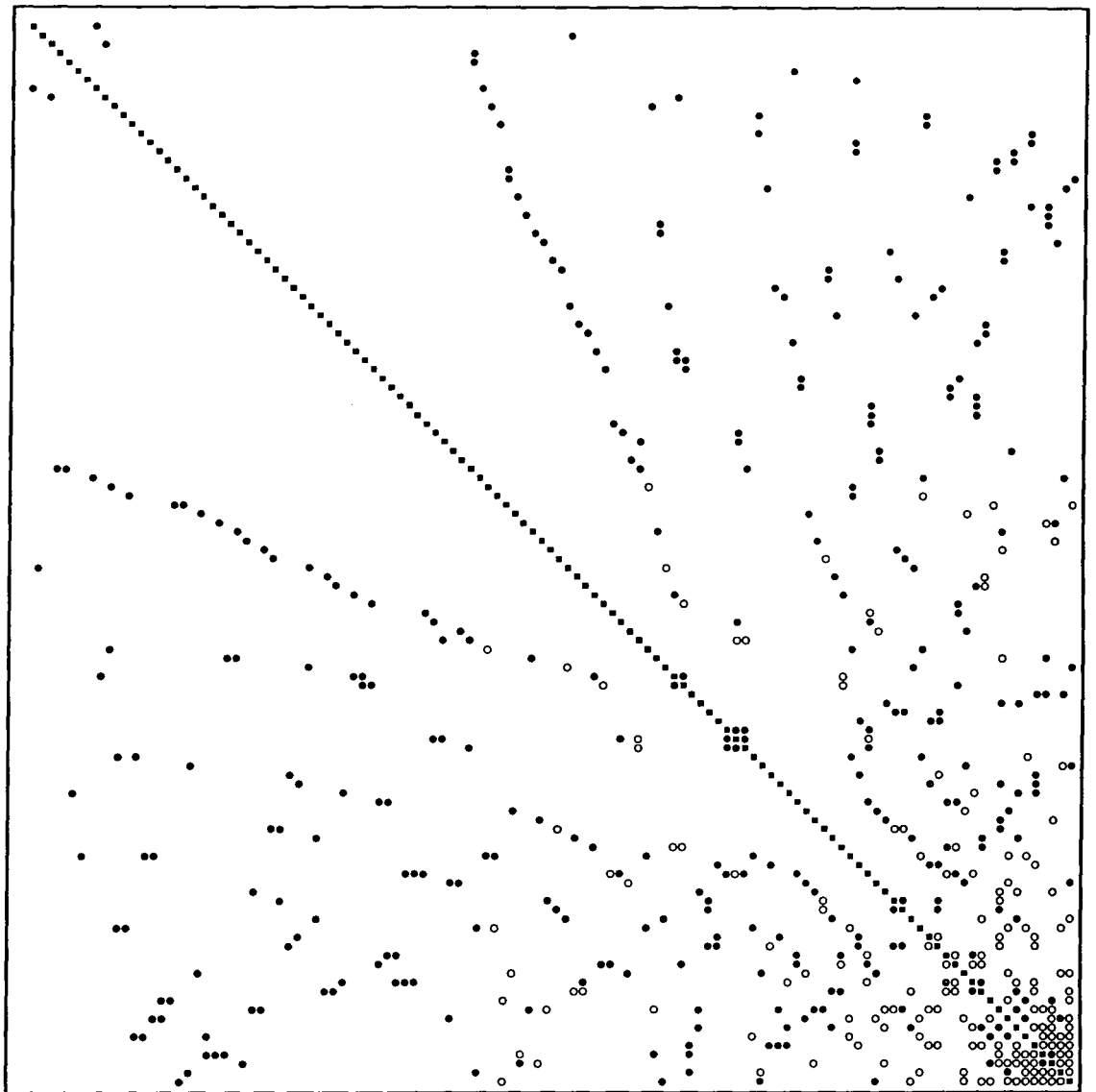
Figure A.9 Matrix of 118 node network, MDLRUR ordering.



Figure—10 Least Recent + Min Length ordering of the 118 node network.

```
Ordering took                  60 ms
Pointer alteration took        10 ms
Reduction took                 00 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took    80 ms plus calculation    80 ms
Elements:  before | after        476    383     86   48.0%
Multiplications generated        922    648
Additions  generated             657    530
Divisions  generated             118      0
Multiply-additions generated     922    648
Stalls generated                  15      1
```

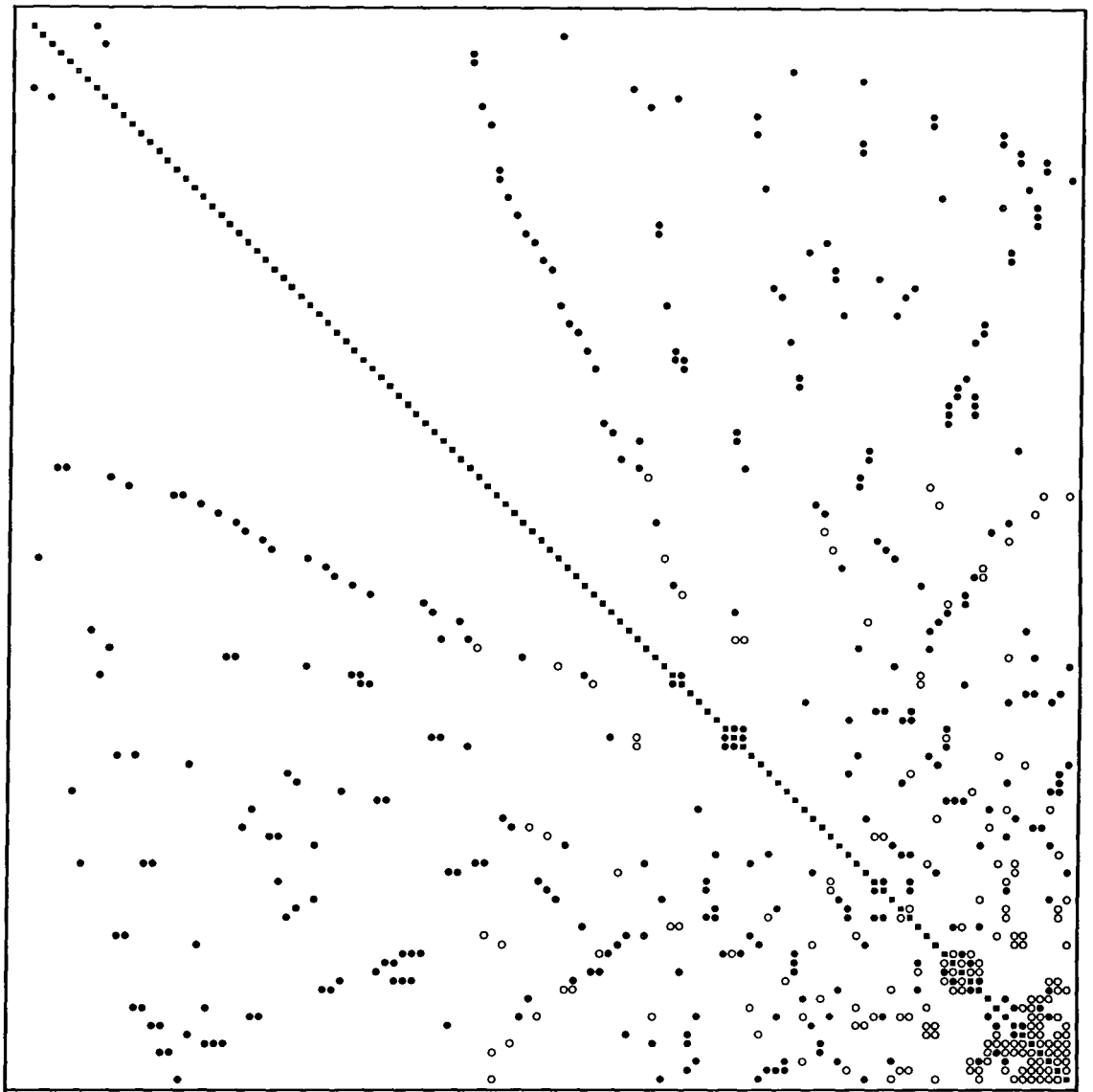
Figure A.10 Matrix of 118 node network, MDLRUML ordering.



Figure—11 MD - Least Recently Used RA ordering of the 118 node network.

```
Ordering took                50 ms
Pointer alteration took      00 ms
Reduction took               10 ms
Solution took                10 ms
Calculation took             00 ms
Ordering, reduction and solution took    80 ms plus calculation    80 ms
Elements:  before | after       476    383     86   48.0%
Multiplications generated       922    648
Additions  generated            657    530
Divisions  generated            118      0
Multiply-additions generated    922    648
Stalls generated                 11      1
```

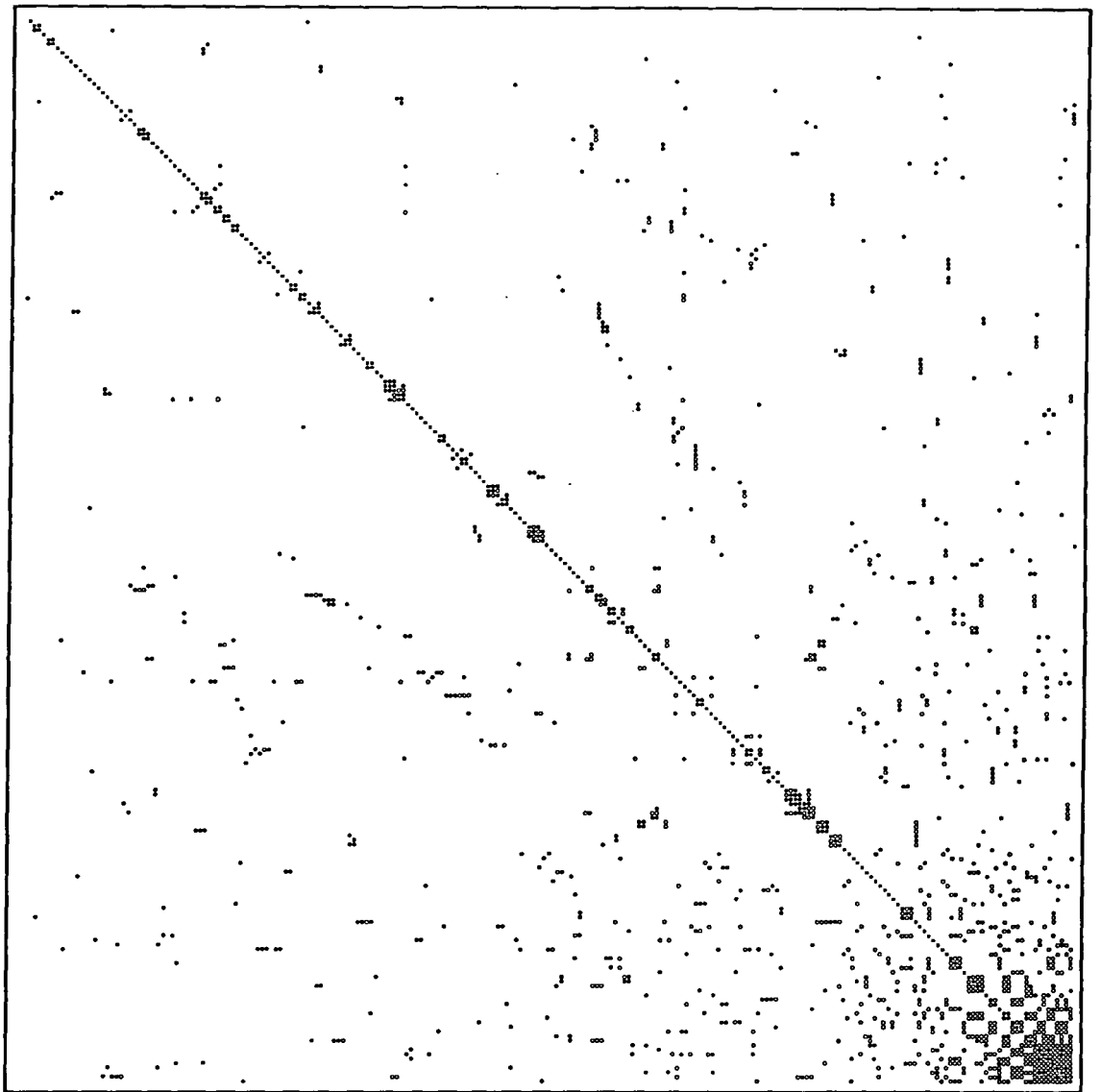
Figure A.11 Matrix of 118 node network, MDLRURA ordering.



Figure—12 Least Recent A + Min Length ordering of the 118 node network.

```
Ordering took                  60 ms
Pointer alteration took        10 ms
Reduction took                 00 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took    80 ms plus calculation    80 ms
Elements:  before | after       476    383     86   48.0%
Multiplications generated       922    648
Additions  generated            657    530
Divisions  generated            118      0
Multiply-additions generated    922    648
Stalls generated                 15      1
```

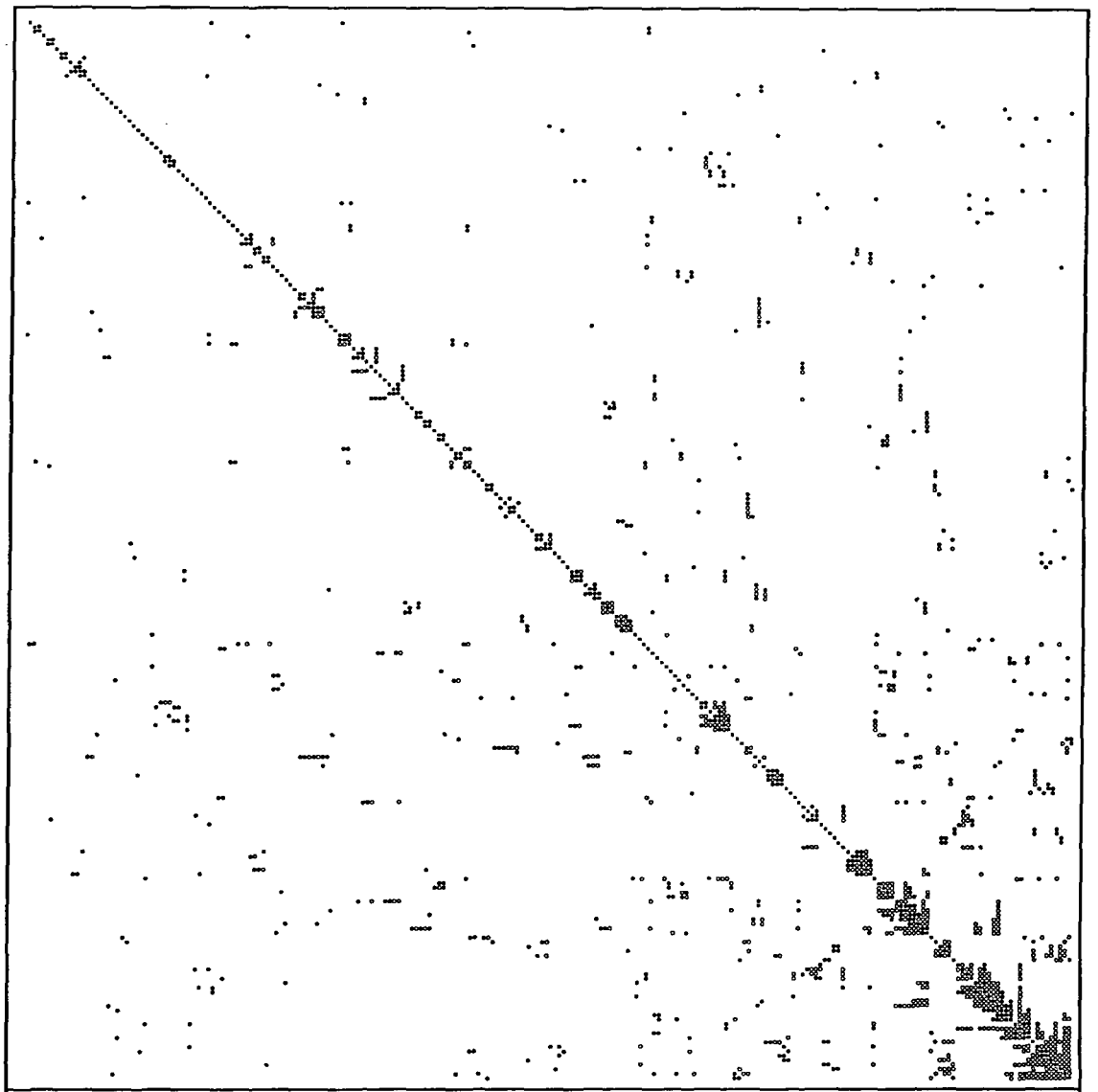
Figure A.12 Matrix of 118 node network, MDLRUMLA ordering.



Tinney-2 ordering of the 234 node network.

```
Ordering took                110 ms
Pointer alteration took       20 ms
Reduction took                20 ms
Solution took                 20 ms
Calculation took              00 ms
Ordering, reduction and solution took   170 ms plus calculation   170 ms
Elements:  before | after       928    895    314   90.5%
Multiplications generated      2996   1556
Additions  generated           2335   1322
Divisions  generated            234      0
Multiply-additions generated   2996   1556
Stalls generated                 66      3
```

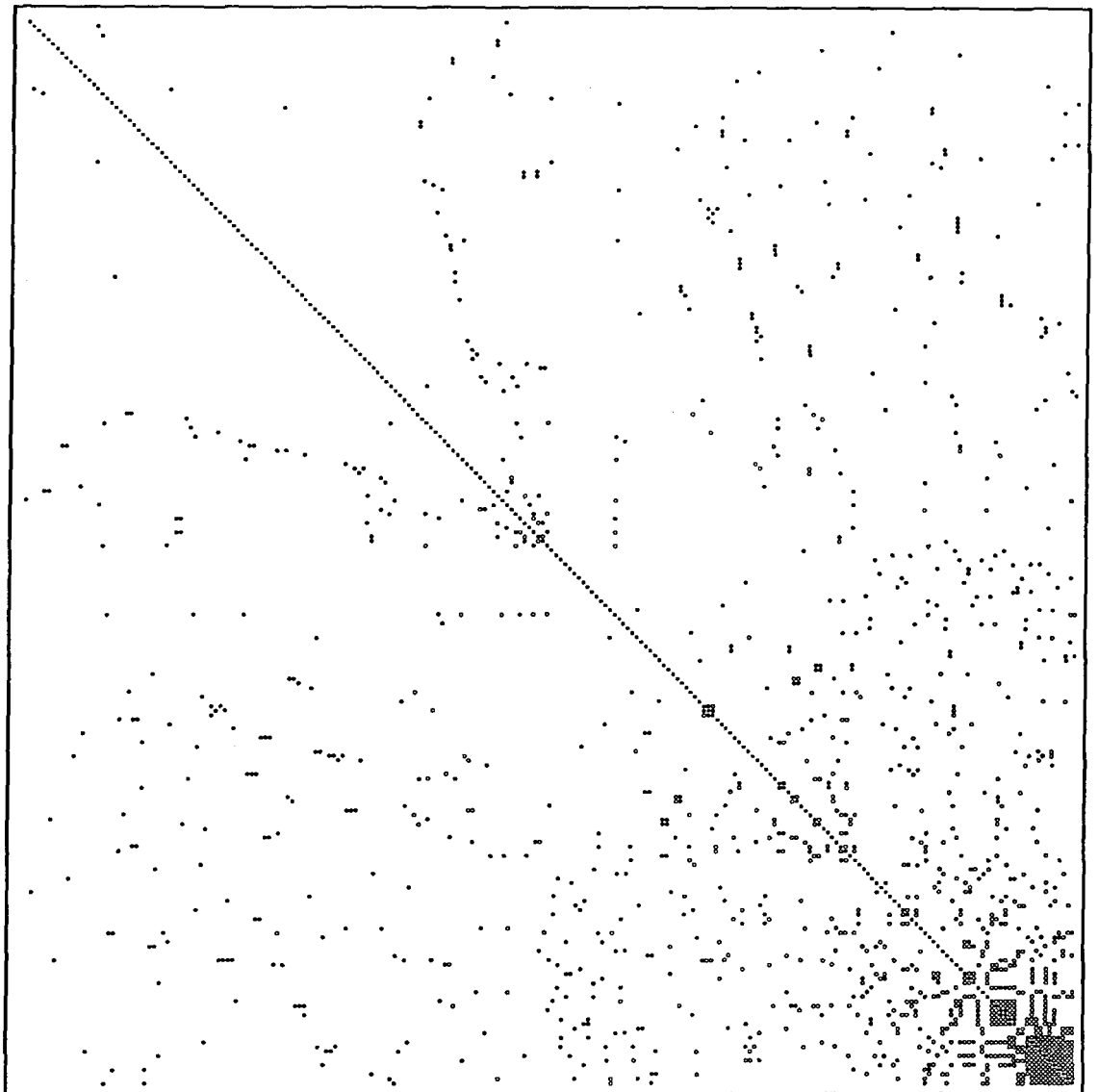
Figure A.13 Matrix of 234 node network, Tinney–2 ordering.



Tinney-3 ordering of the 234 node network.

```
Ordering took                1760 ms
Pointer alteration took        20 ms
Reduction took                 10 ms
Solution took                  20 ms
Calculation took               00 ms
Ordering, reduction and solution took  1810 ms plus calculation  1810 ms
Elements:  before | after       928     860    279   80.4%
Multiplications generated      2602    1486
Additions  generated           1976    1252
Divisions  generated            234       0
Multiply-additions generated   2602    1486
Stalls generated                 82       4
```

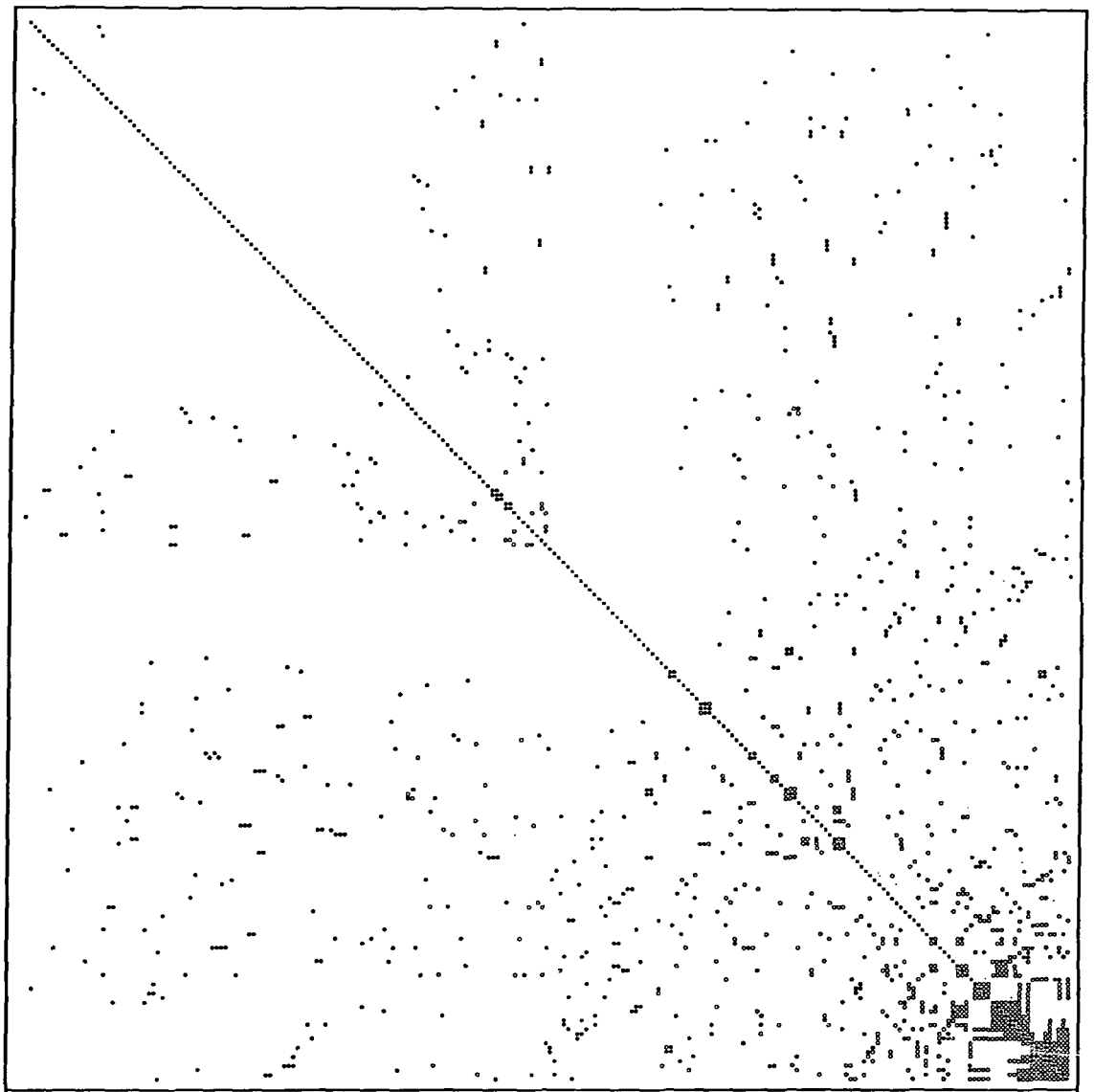
Figure A.14 Matrix of 234 node network, Tinney–3 ordering.



Minimum-depth minimum-length ordering of the 234 node network.

```
Ordering took                  120 ms
Pointer alteration took         20 ms
Reduction took                  20 ms
Solution took                   20 ms
Calculation took                00 ms
Ordering, reduction and solution took   180 ms plus calculation   180 ms
Elements:  before | after        928    890    309   89.0%
Multiplications generated       2964   1546
Additions  generated            2308   1312
Divisions  generated             234      0
Multiply-additions generated    2964   1546
Stalls generated                  26      1
```

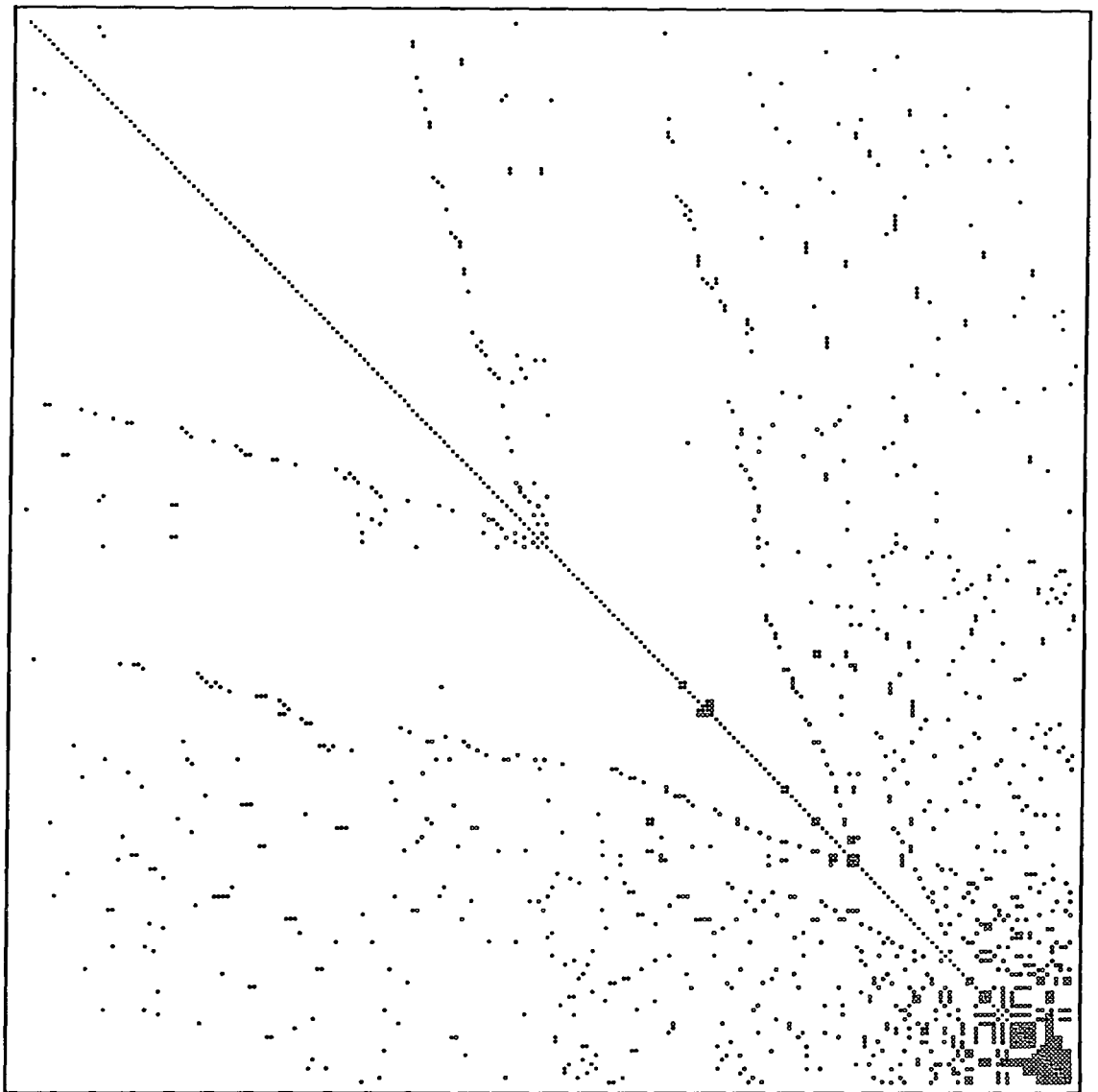
Figure A.15 Matrix of 234 node network, MDML ordering.



Gomez-Franquelo A-1 ordering of the 234 node network.

```
Ordering took                  120 ms
Pointer alteration took         20 ms
Reduction took                  20 ms
Solution took                   10 ms
Calculation took                00 ms
Ordering, reduction and solution took   170 ms plus calculation   170 ms
Elements:  before | after       928    892    311   89.6%
Multiplications generated      2964   1550
Additions  generated           2306   1316
Divisions  generated            234      0
Multiply-additions generated   2964   1550
Stalls generated                 35      1
```

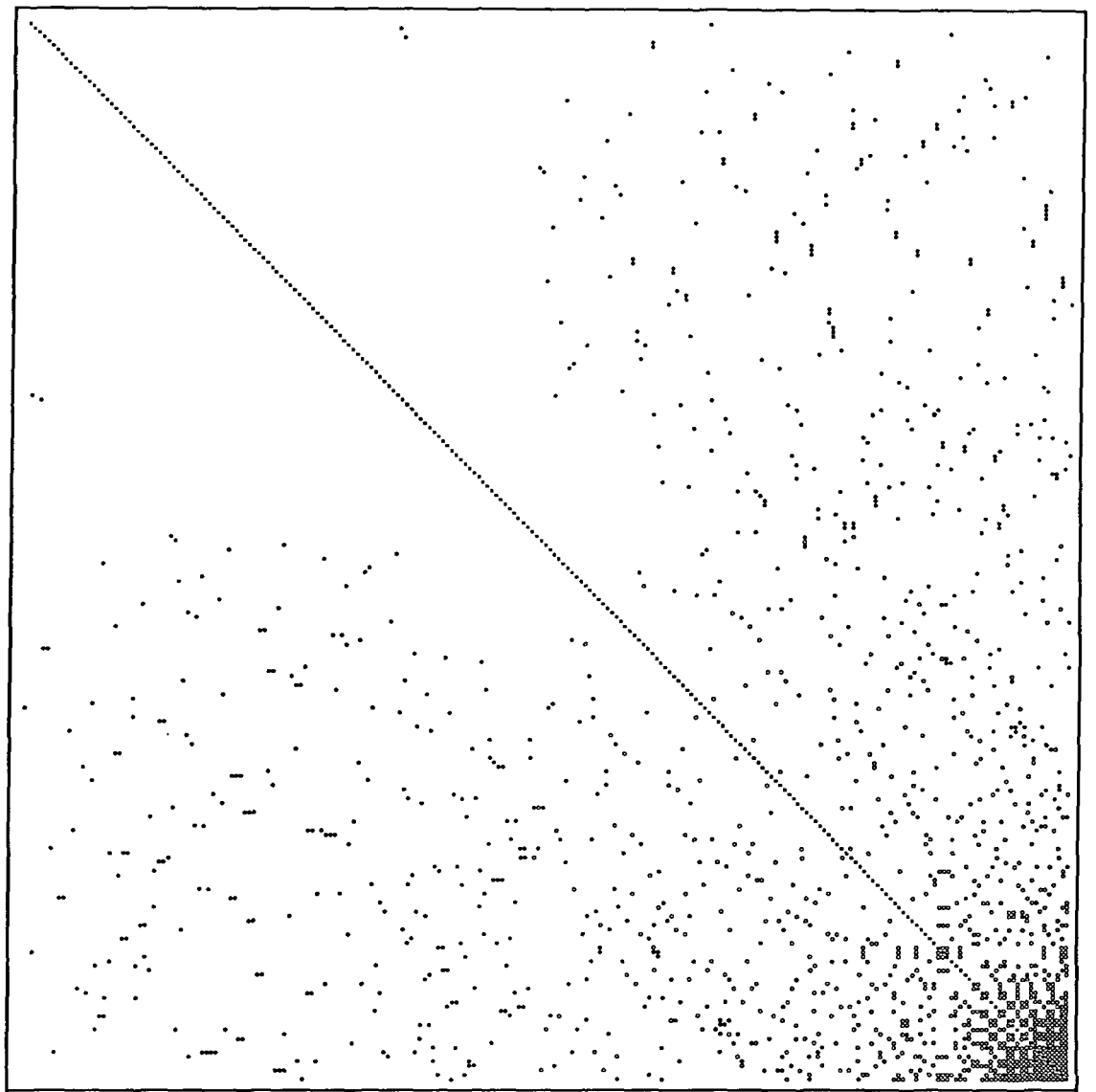
Figure A.16 Matrix of 234 node network, GF–1 ordering.



Gomez-Franquelo A-2 ordering of the 234 node network.

```
Ordering took                110 ms
Pointer alteration took       10 ms
Reduction took                20 ms
Solution took                 20 ms
Calculation took              00 ms
Ordering, reduction and solution took   170 ms plus calculation   170 ms
Elements:  before | after       928    887    306   88.2%
Multiplications generated      2902   1540
Additions  generated           2249   1306
Divisions  generated            234      0
Multiply-additions generated   2902   1540
Stalls generated                 22      1
```

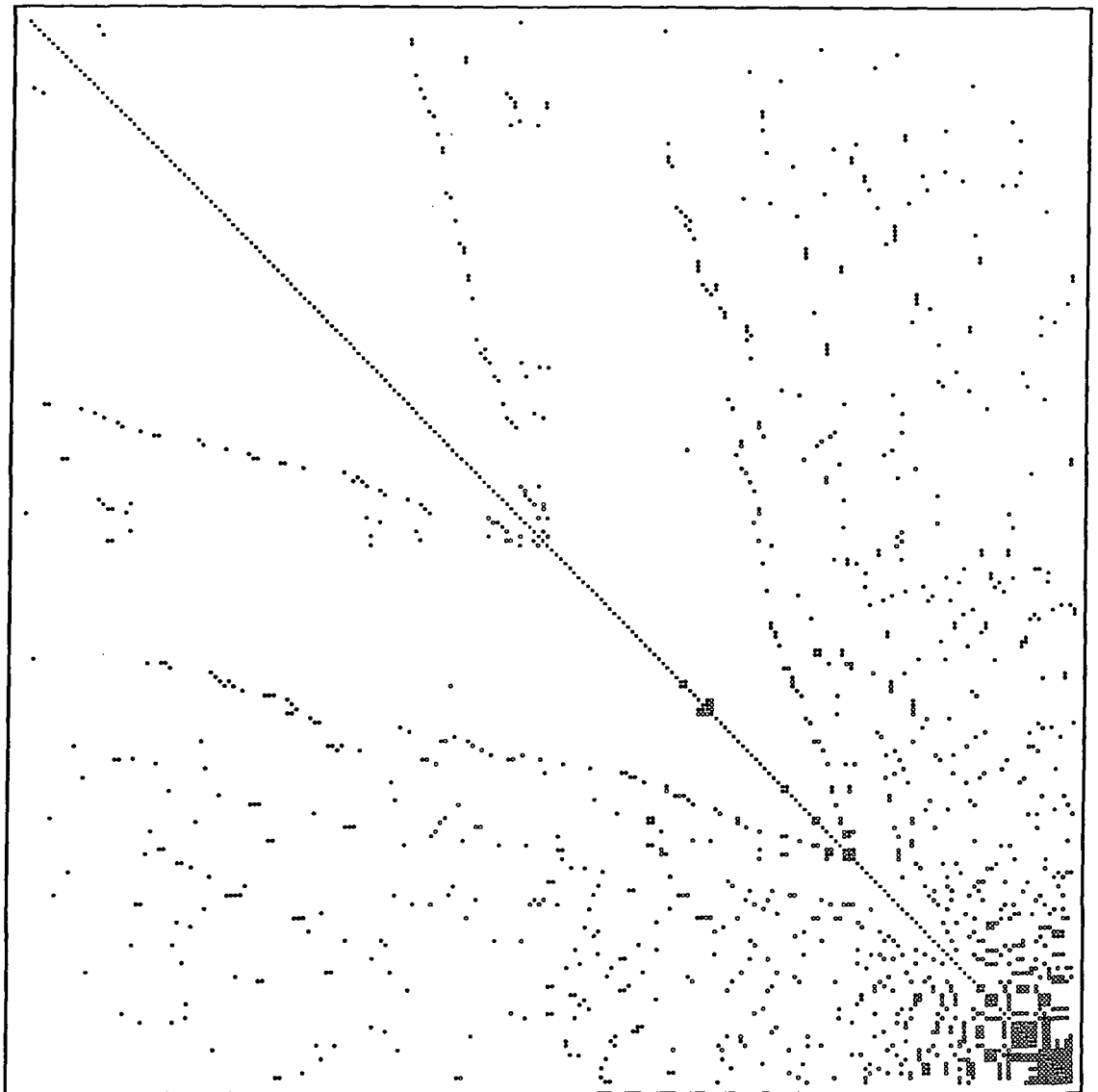
Figure A.17 Matrix of 234 node network, GF–2 ordering.



Gomez-Franquelo A-3 ordering of the 234 node network.

```
Ordering took                  130 ms
Pointer alteration took         20 ms
Reduction took                  20 ms
Solution took                   20 ms
Calculation took                00 ms
Ordering, reduction and solution took   190 ms plus calculation   190 ms
Elements:  before | after        928     956    375  108.1%
Multiplications generated       3714    1678
Additions  generated            2992    1444
Divisions  generated             234       0
Multiply-additions generated    3714    1678
Stalls generated                  15       1
```

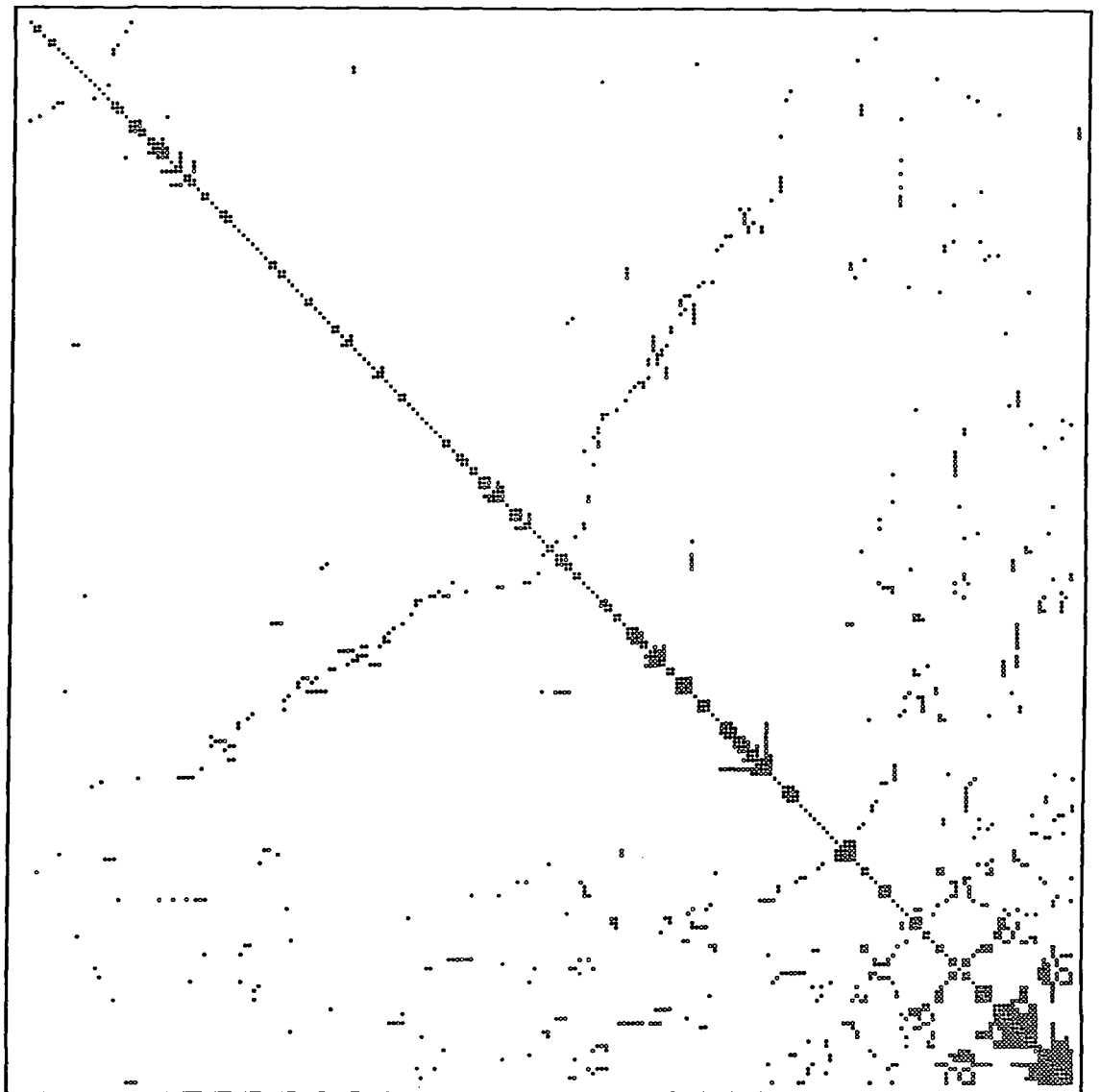
Figure A.18 Matrix of 234 node network, GF–3 ordering.



MD--Least Recently Used ordering of the 234 node network.

```
Ordering took                  70 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took   120 ms plus calculation   120 ms
Elements:  before | after       928    885    304   87.6%
Multiplications generated      2870   1536
Additions  generated           2219   1302
Divisions  generated            234      0
Multiply-additions generated   2870   1536
Stalls generated                 22      1
```

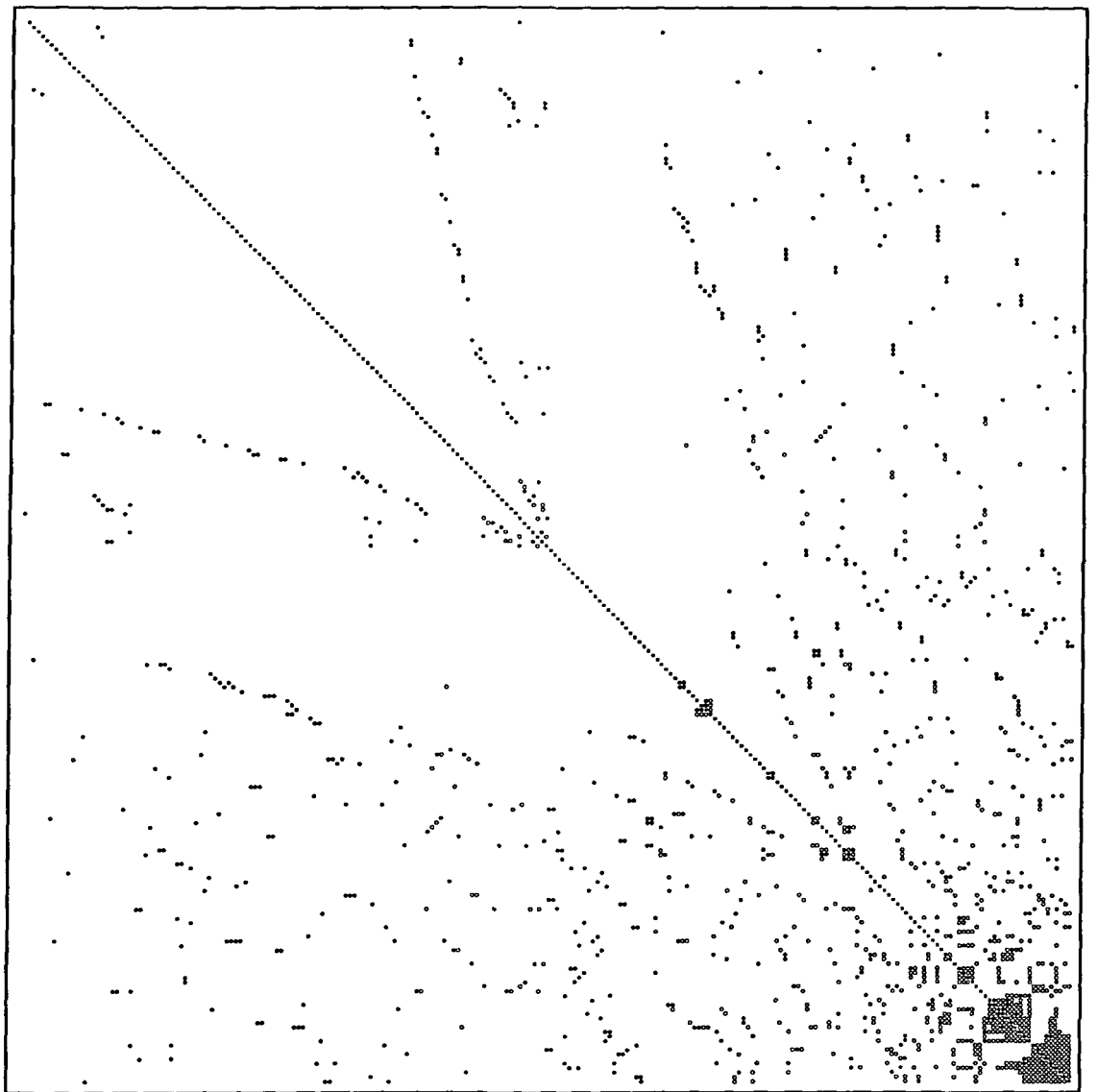
Figure A.19 Matrix of 234 node network, MDLRU ordering.



MD--Reversed LRU ordering of the 234 node network.

```
Ordering took                  70 ms
Pointer alteration took        30 ms
Reduction took                 20 ms
Solution took                  10 ms
Calculation took               00 ms
Ordering, reduction and solution took   130 ms plus calculation   130 ms
Elements:  before | after        928    871    290   83.6%
Multiplications generated       2724   1508
Additions  generated            2087   1274
Divisions  generated             234      0
Multiply-additions generated    2724   1508
Stalls generated                 102      3
```

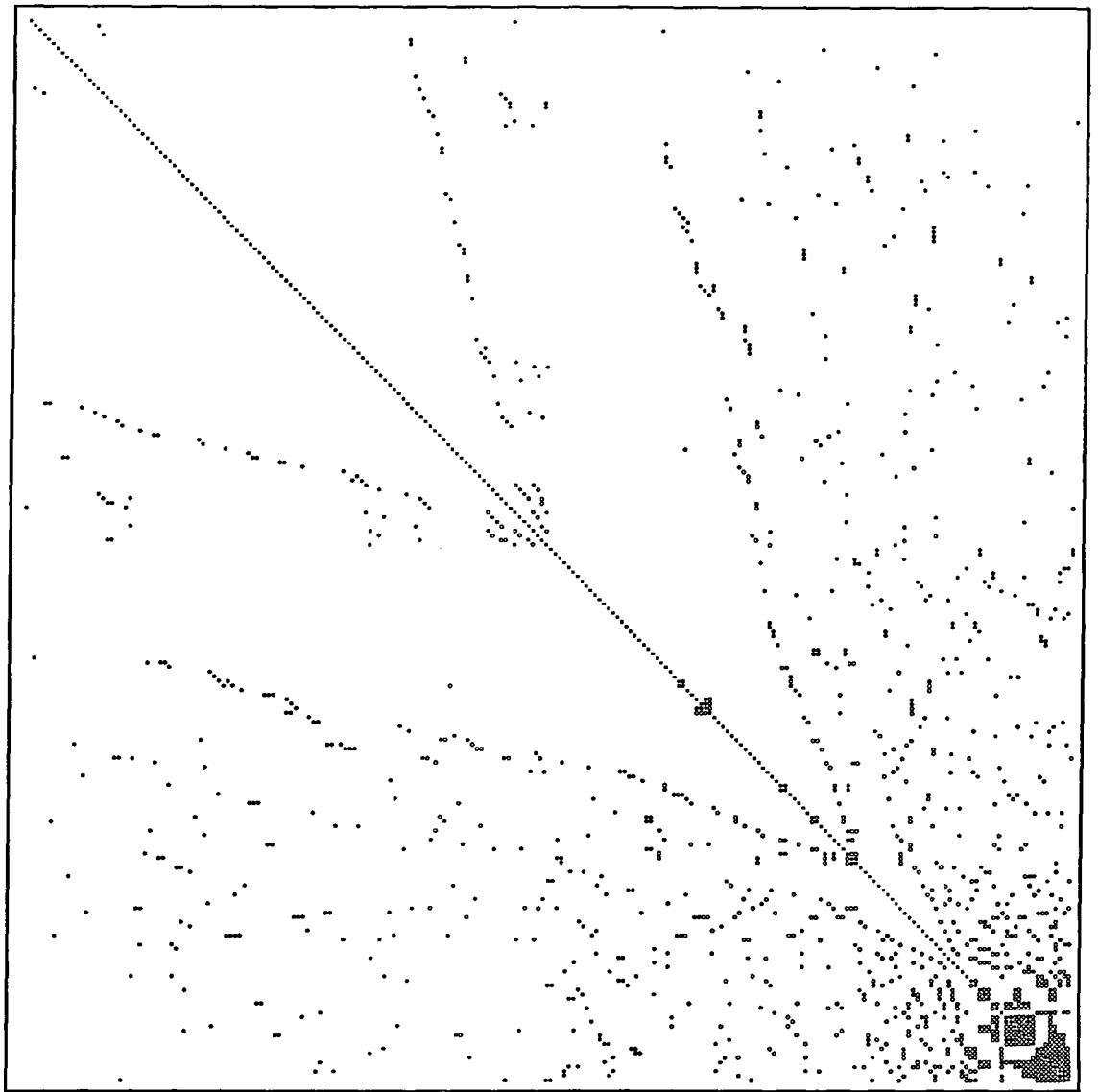
Figure A.20 Matrix of 234 node network, MDLRUR ordering.



Least Recent + Min Length ordering of the 234 node network.

```
Ordering took                  80 ms
Pointer alteration took        10 ms
Reduction took                 20 ms
Solution took                  20 ms
Calculation took               00 ms
Ordering, reduction and solution took   130 ms plus calculation   130 ms
Elements:  before | after       928    892    311   89.6%
Multiplications generated      2968   1550
Additions  generated           2310   1316
Divisions  generated            234      0
Multiply-additions generated   2968   1550
Stalls generated                 28      1
```

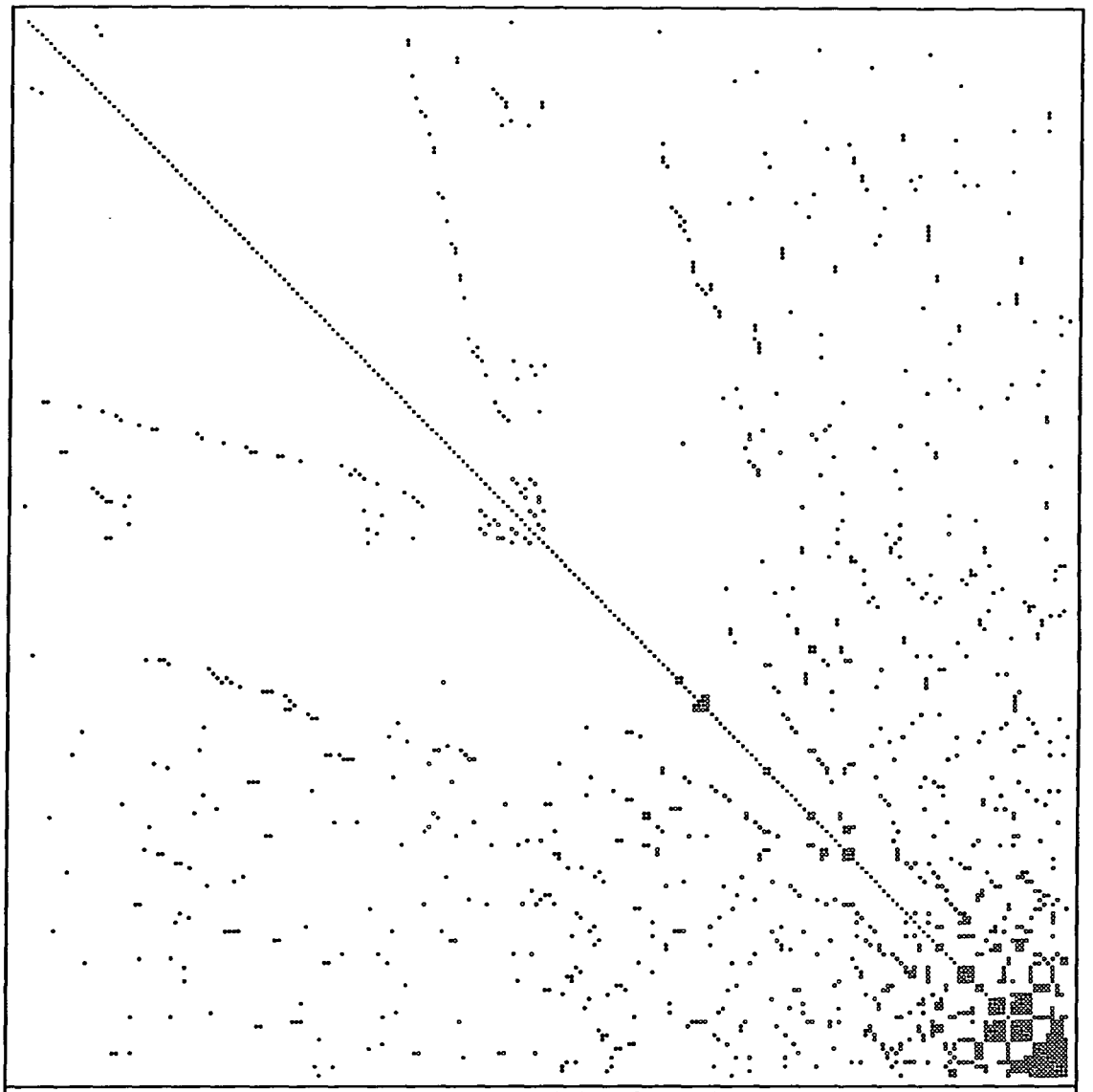
Figure A.21 Matrix of 234 node network, MDLRUML ordering.



MD--Least Recently Used RA ordering of the 234 node network.

```
Ordering took                  70 ms
Pointer alteration took        10 ms
Reduction took                 20 ms
Solution took                  20 ms
Calculation took               00 ms
Ordering, reduction and solution took   130 ms plus calculation   130 ms
Elements:  before | after        928    883    302   87.0%
Multiplications generated       2850   1532
Additions  generated            2201   1298
Divisions  generated             234      0
Multiply-additions generated    2850   1532
Stalls generated                  24      1
```

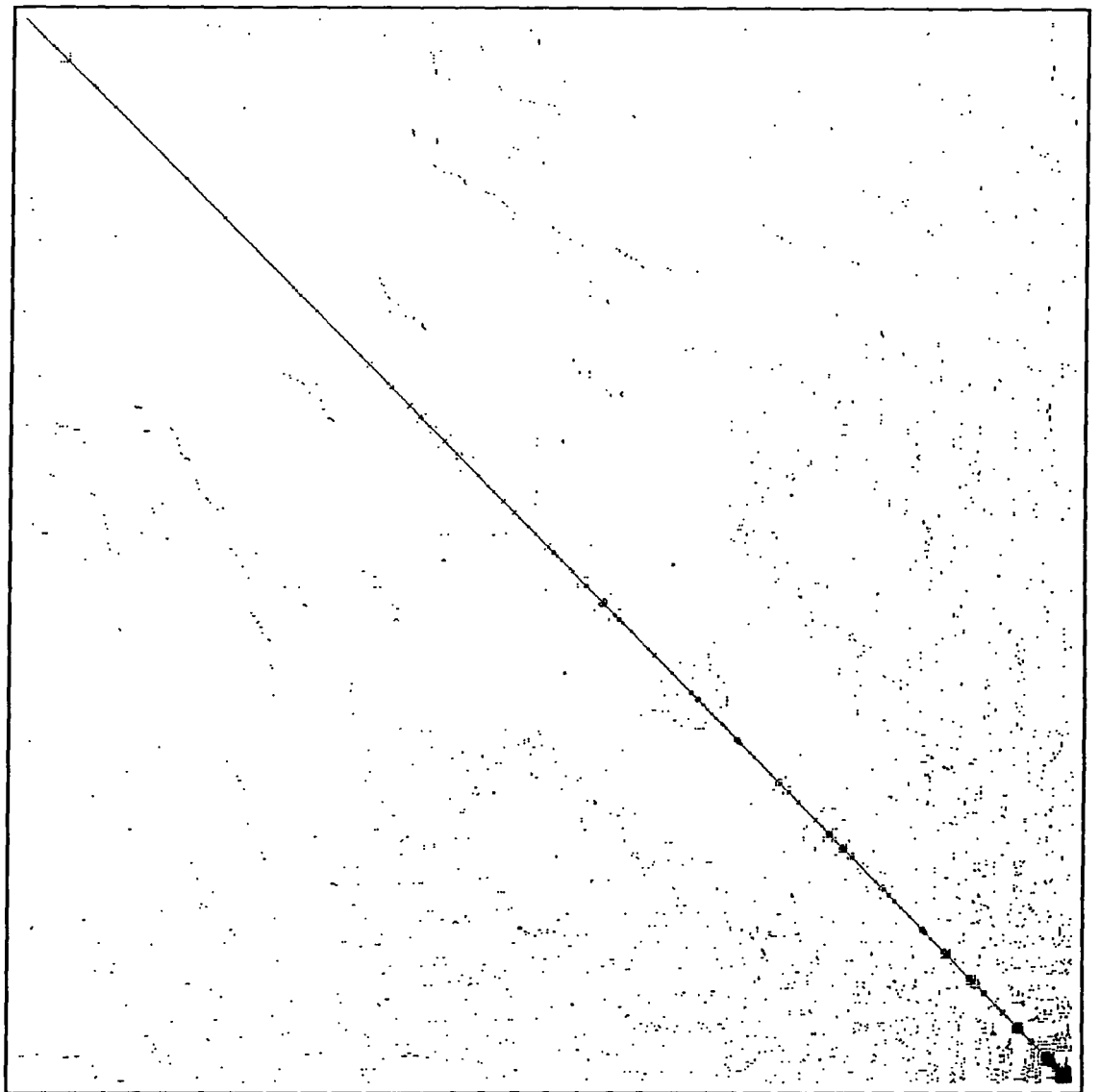
Figure A.22 Matrix of 234 node network, MDLRURA ordering.



Least Recent A + Min Length ordering of the 234 node network.

```
Ordering took                  80 ms
Pointer alteration took        20 ms
Reduction took                 20 ms
Solution took                  20 ms
Calculation took               00 ms
Ordering, reduction and solution took   140 ms plus calculation   140 ms
Elements:  before | after       928    887    306   88.2%
Multiplications generated      2906   1540
Additions  generated           2253   1306
Divisions  generated            234      0
Multiply-additions generated   2906   1540
Stalls generated                 26      1
```

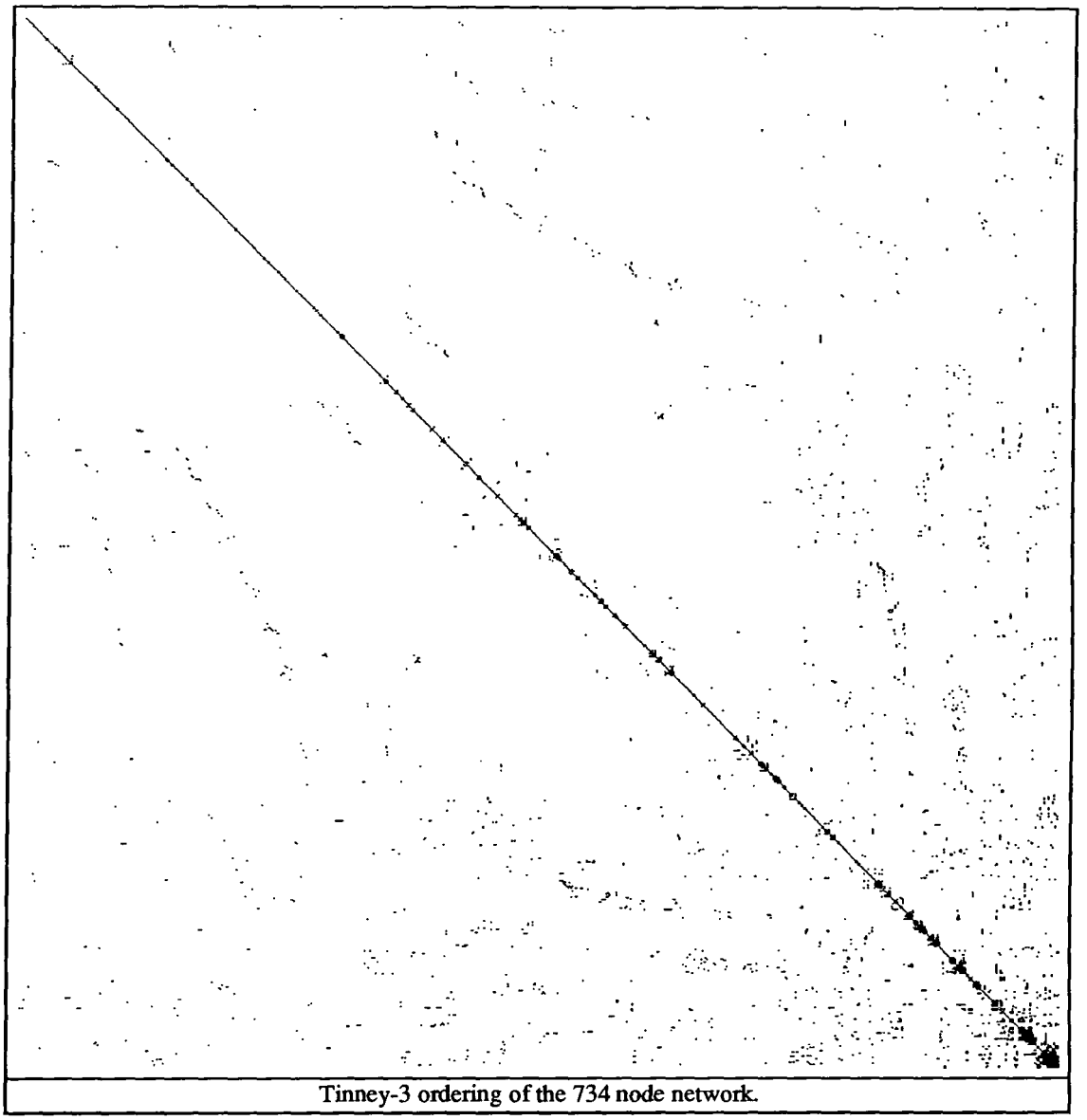
Figure A.23 Matrix of 234 node network, MDLRUMLA ordering.



Tinney-2 ordering of the 734 node network.

```
Ordering took                530 ms
Pointer alteration took       50 ms
Reduction took                50 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   680 ms plus calculation   680 ms
Elements:  before | after      2696   2363    648   66.1%
Multiplications generated      6614   3992
Additions  generated           4985   3258
Divisions  generated            734      0
Multiply-additions generated   6614   3992
Stalls generated                148     18
```

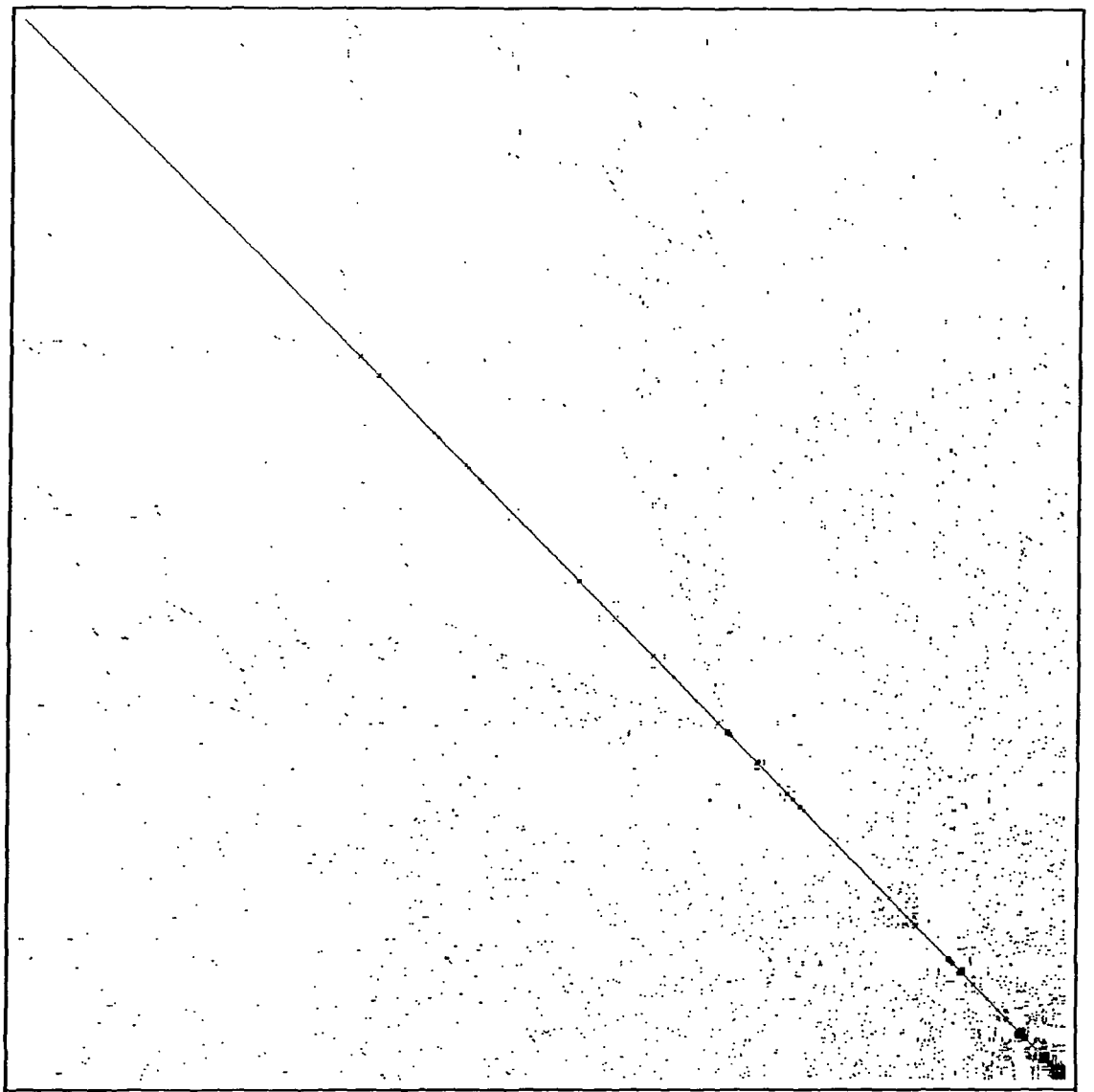
Figure A.24 Matrix of 734 node network, Tinney–2 ordering.



Tinney-3 ordering of the 734 node network.

```
Ordering took              13510 ms
Pointer alteration took       50 ms
Reduction took                50 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took 13660 ms plus calculation 13660 ms
Elements:  before | after      2696   2318    603   61.5%
Multiplications generated      6136   3902
Additions  generated           4552   3168
Divisions  generated            734      0
Multiply-additions generated   6136   3902
Stalls generated                181     19
```

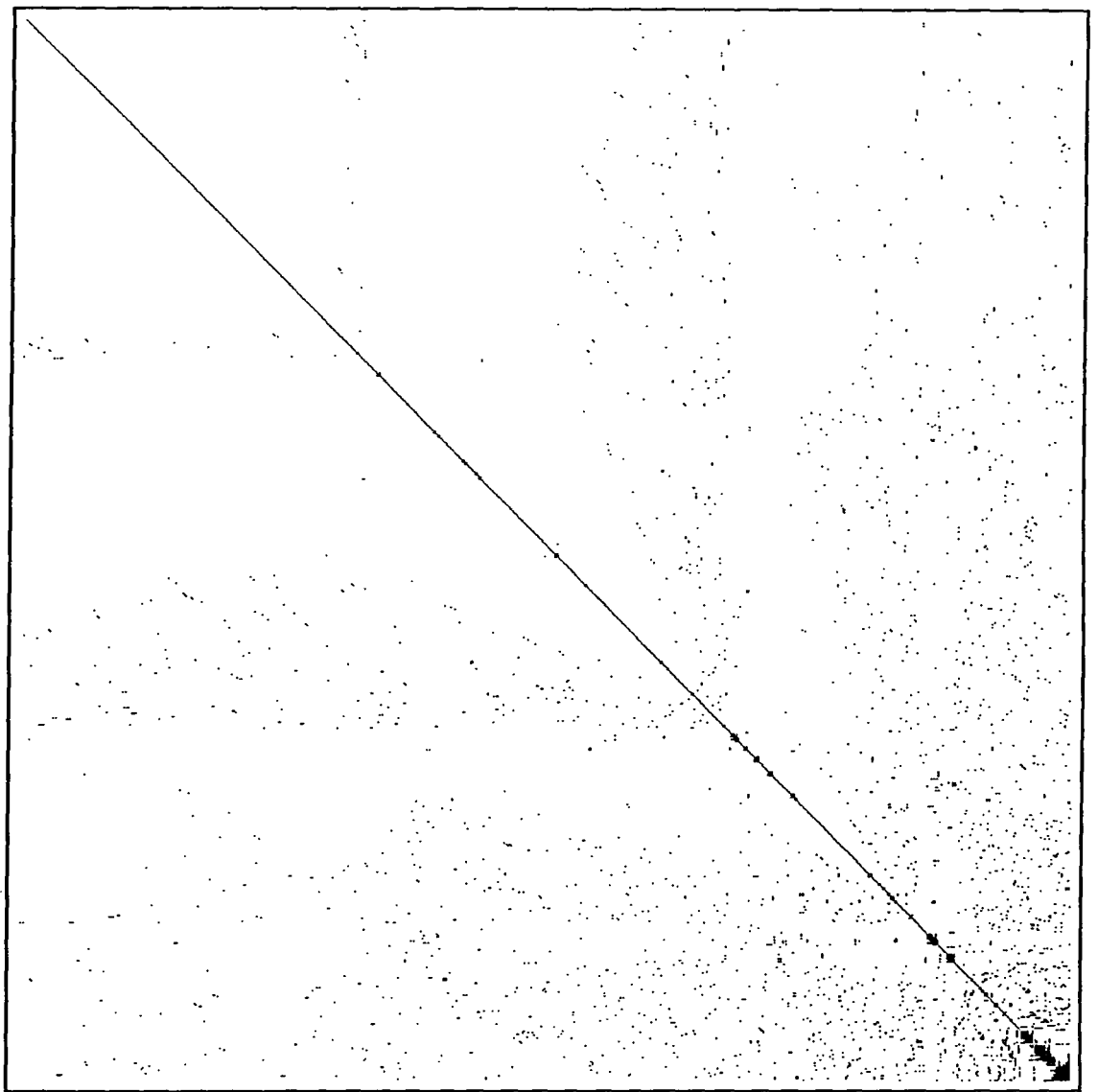
Figure A.25 Matrix of 734 node network, Tinney–3 ordering.



Minimum-depth minimum-length ordering of the 734 node network.

```
Ordering took                  700 ms
Pointer alteration took         60 ms
Reduction took                  40 ms
Solution took                   40 ms
Calculation took                00 ms
Ordering, reduction and solution took   840 ms plus calculation   840 ms
Elements:  before | after      2696   2343    628   64.0%
Multiplications generated      6358   3952
Additions  generated           4749   3218
Divisions  generated            734      0
Multiply-additions generated   6358   3952
Stalls generated                 71      2
```

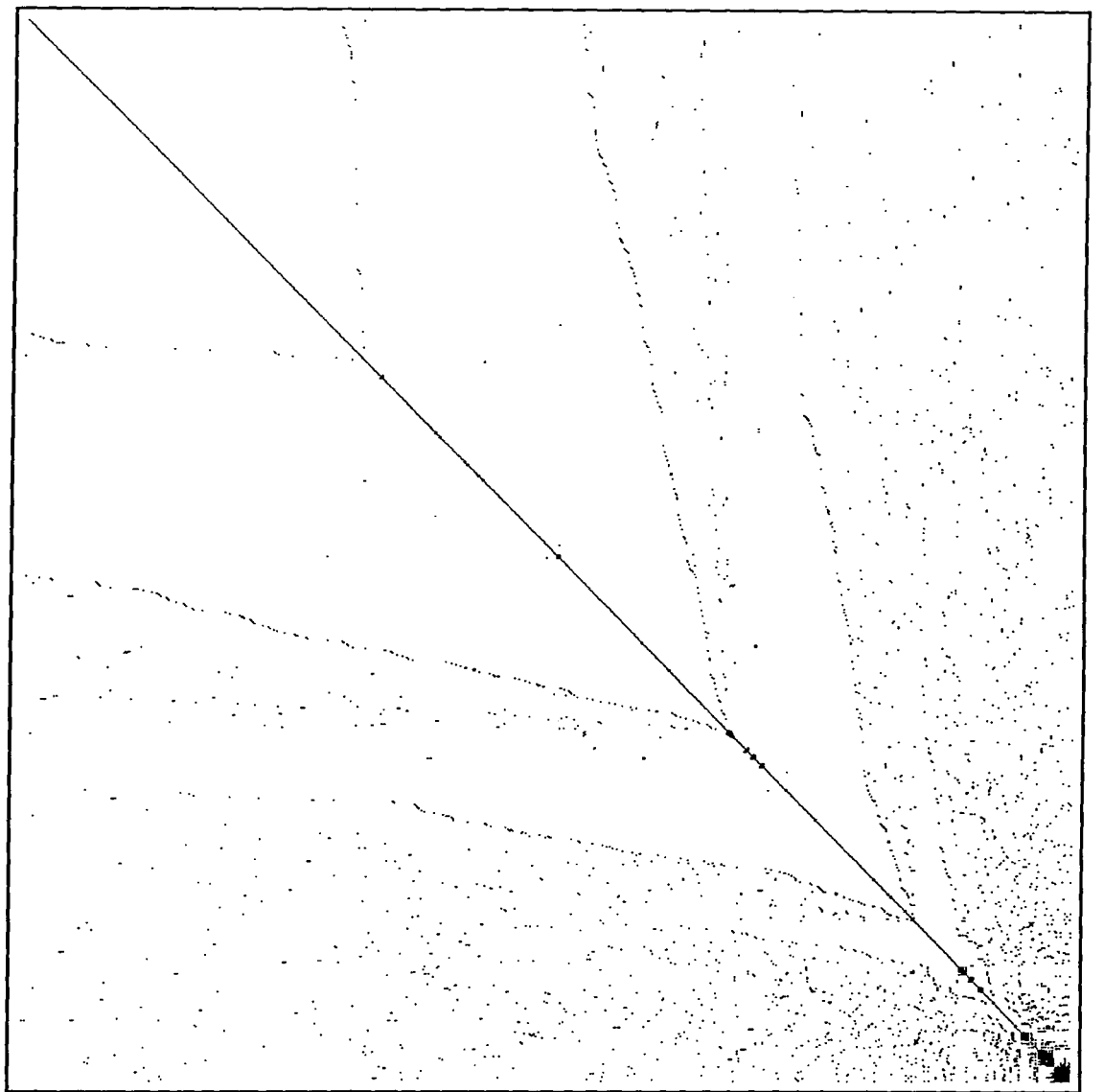
Figure A.26 Matrix of 734 node network, MDML ordering.



Gomez-Franquelo A-1 ordering of the 734 node network.

```
Ordering took                650 ms
Pointer alteration took       60 ms
Reduction took                40 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   790 ms plus calculation   790 ms
Elements:  before | after      2696   2334    619   63.1%
Multiplications generated      6268   3934
Additions  generated           4668   3200
Divisions  generated            734      0
Multiply-additions generated   6268   3934
Stalls generated                 87      3
```

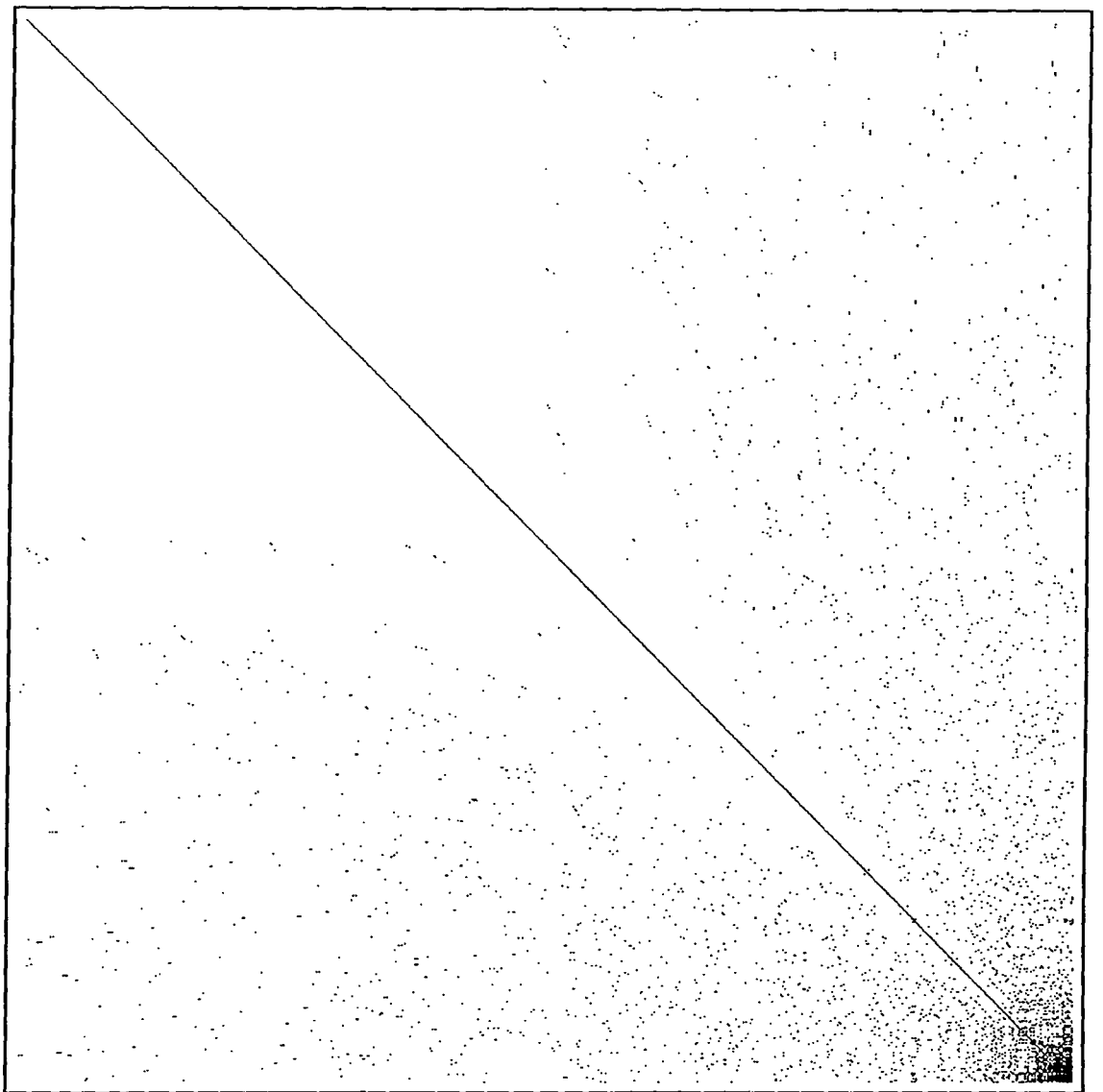
Figure A.27 Matrix of 734 node network, GF–1 ordering.



Gomez-Franquelo A-2 ordering of the 734 node network.

```
Ordering took                650 ms
Pointer alteration took       60 ms
Reduction took                40 ms
Solution took                 50 ms
Calculation took              00 ms
Ordering, reduction and solution took   800 ms plus calculation   800 ms
Elements:  before | after      2696   2358    643   65.5%
Multiplications generated      6550   3982
Additions  generated           4926   3248
Divisions  generated            734      0
Multiply-additions generated   6550   3982
Stalls generated                 68      2
```

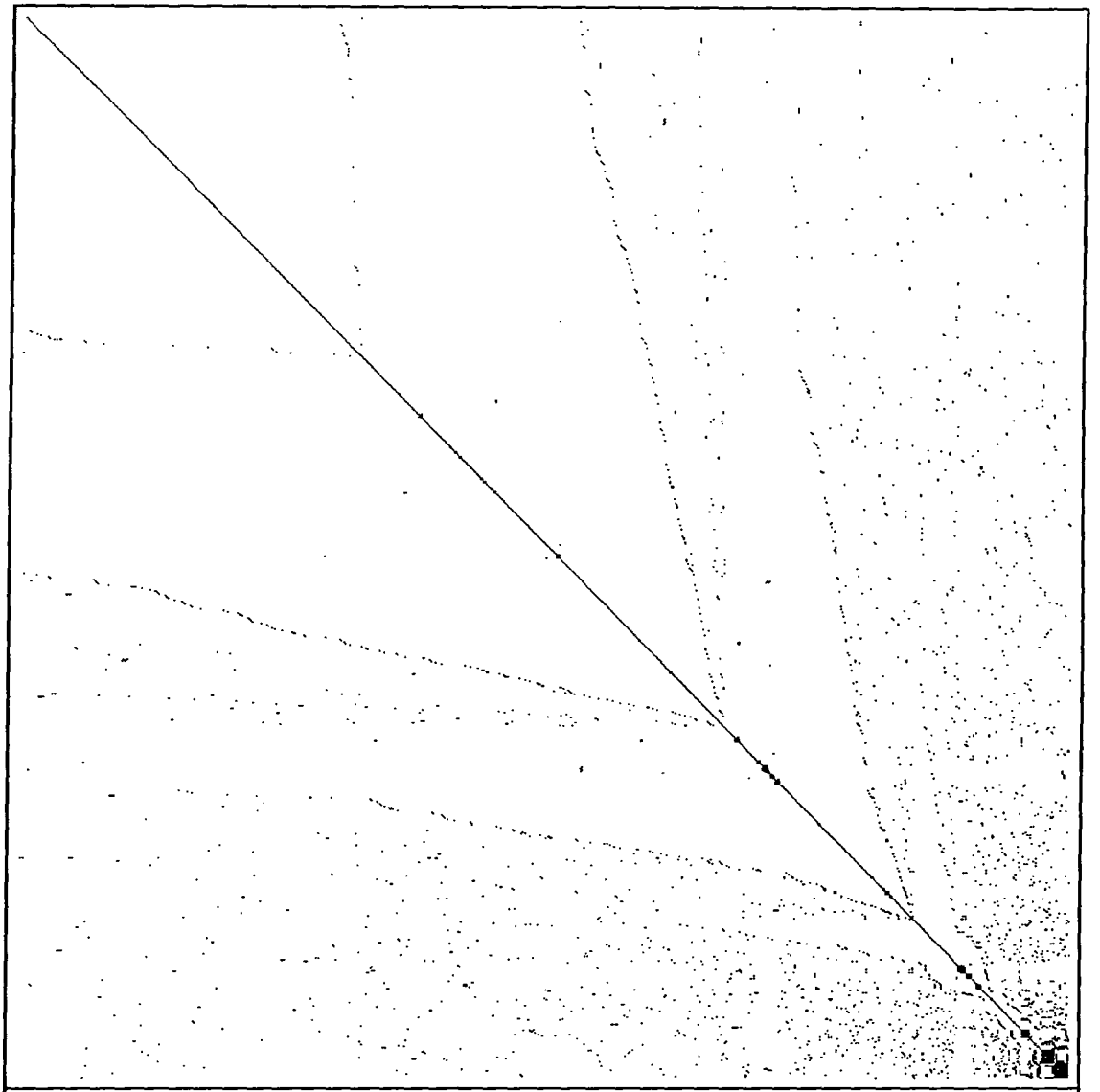
Figure A.28 Matrix of 734 node network, GF–2 ordering.



Gomez-Franquelo A-3 ordering of the 734 node network.

```
Ordering took                  700 ms
Pointer alteration took         60 ms
Reduction took                  60 ms
Solution took                   50 ms
Calculation took                00 ms
Ordering, reduction and solution took   870 ms plus calculation   870 ms
Elements:  before | after      2696   2578    863   88.0%
Multiplications generated      9428   4422
Additions  generated           7584   3688
Divisions  generated            734      0
Multiply-additions generated   9428   4422
Stalls generated                 16      1
```

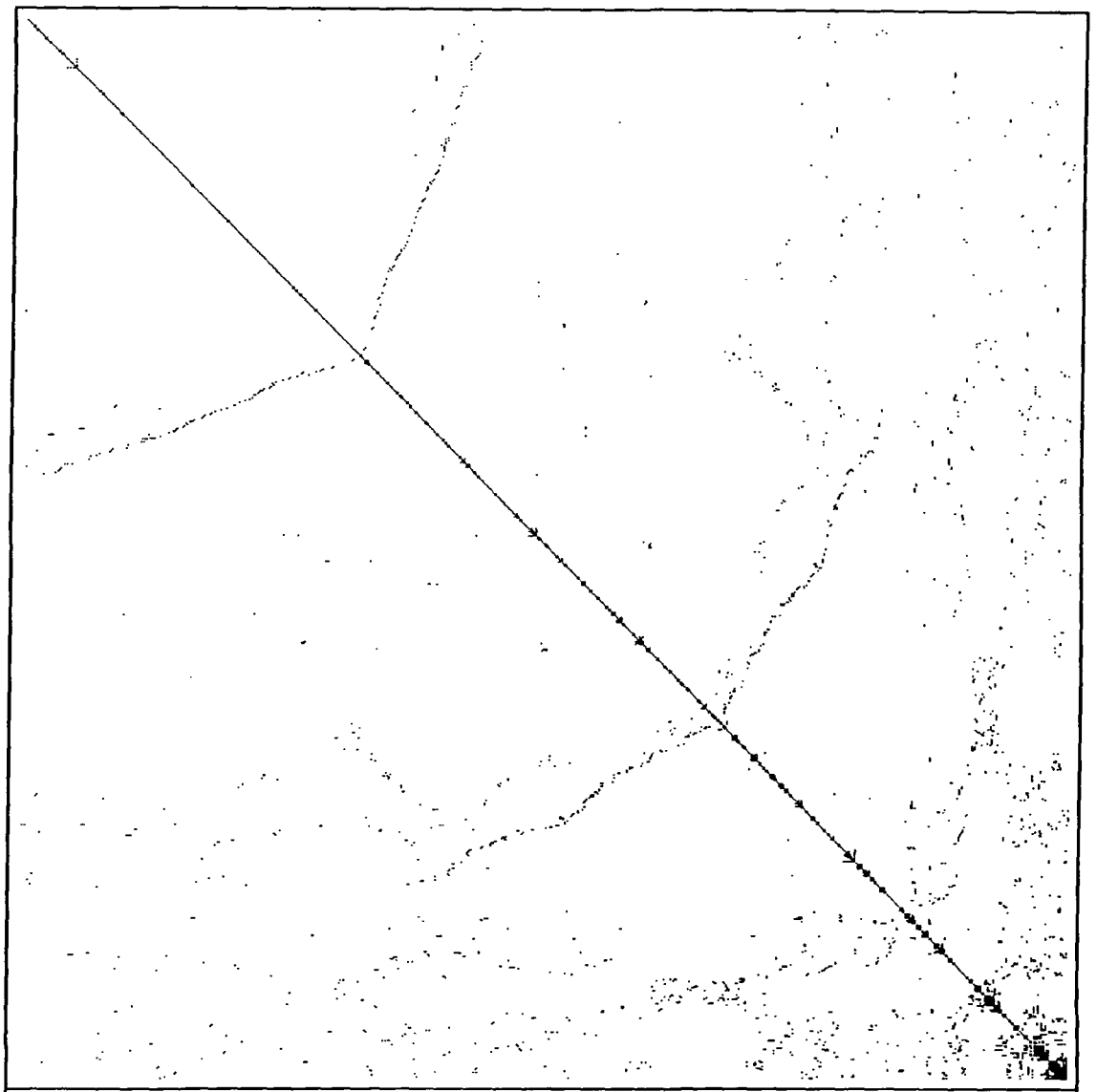
Figure A.29 Matrix of 734 node network, GF–3 ordering.



MD - Least Recently Used ordering of the 734 node network.

```
Ordering took                140 ms
Pointer alteration took       60 ms
Reduction took                40 ms
Solution took                 50 ms
Calculation took              00 ms
Ordering, reduction and solution took   300 ms plus calculation   300 ms
Elements:  before | after      2696   2369    654   66.7%
Multiplications generated      6718   4004
Additions  generated           5083   3270
Divisions  generated            734      0
Multiply-additions generated   6718   4004
Stalls generated                 67      2
```

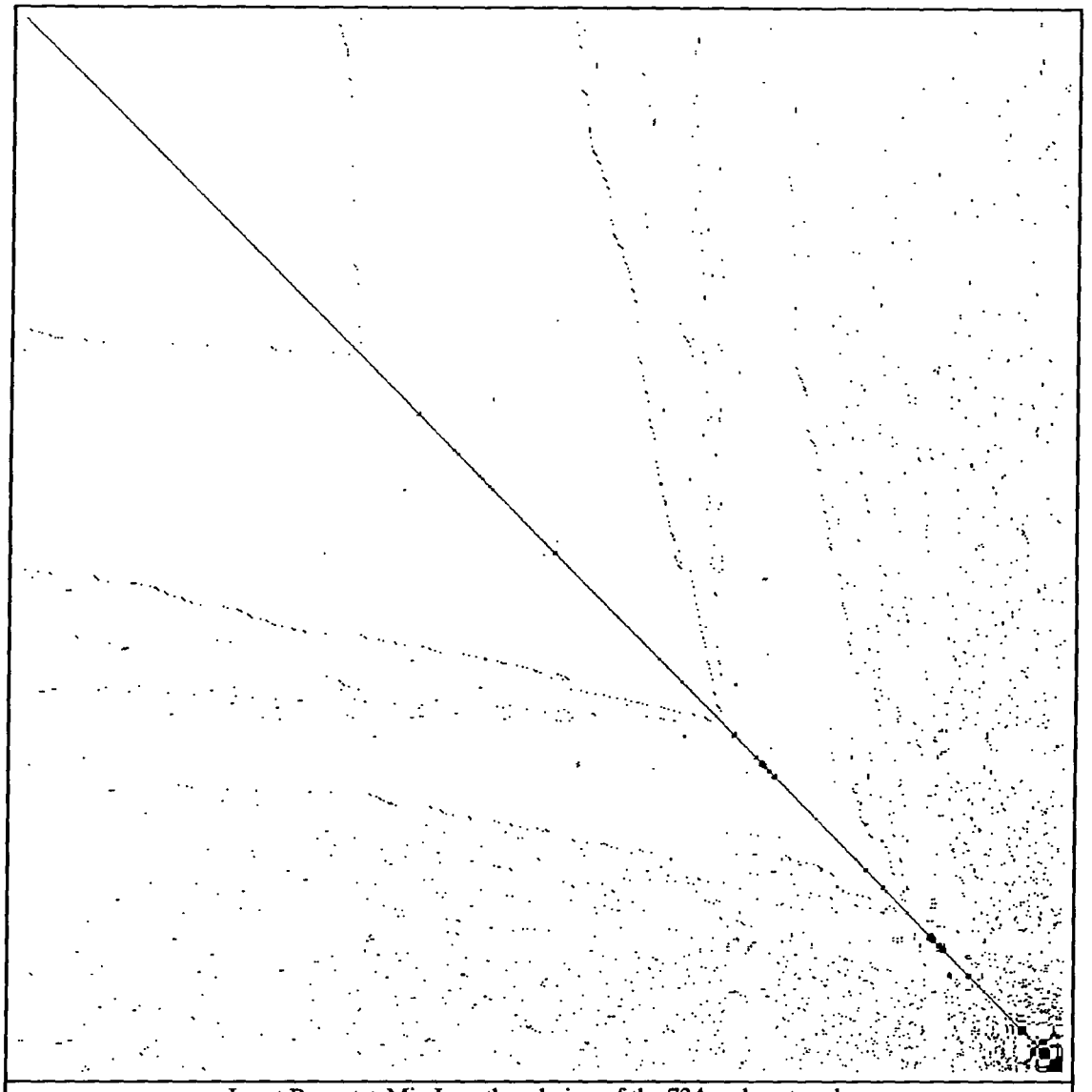
Figure A.30 Matrix of 734 node network, MDLRU ordering.



MD - Reversed LRU ordering of the 734 node network.

```
Ordering took                110 ms
Pointer alteration took       50 ms
Reduction took                40 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   240 ms plus calculation   240 ms
Elements:  before | after      2696   2336    621   63.3%
Multiplications generated      6280   3938
Additions  generated           4678   3204
Divisions  generated            734      0
Multiply-additions generated   6280   3938
Stalls generated                232     23
```

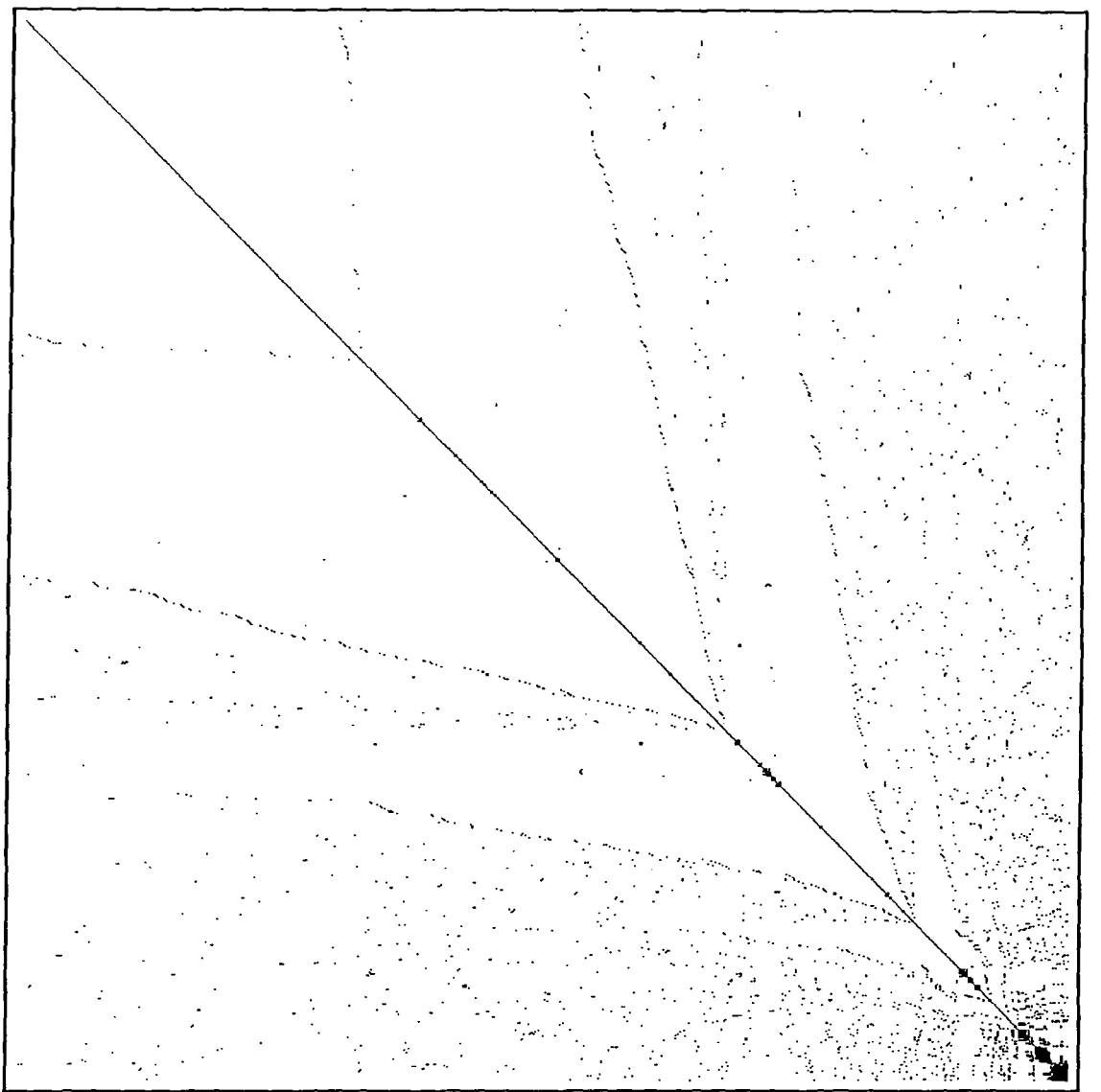
Figure A.31 Matrix of 734 node network, MDLRUR ordering.



Least Recent + Min Length ordering of the 734 node network.

```
Ordering took                150 ms
Pointer alteration took       50 ms
Reduction took                40 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   280 ms plus calculation   290 ms
Elements:  before | after      2696   2349    634   64.6%
Multiplications generated      6460   3964
Additions  generated           4845   3230
Divisions  generated            734      0
Multiply-additions generated   6460   3964
Stalls generated                 74      2
```

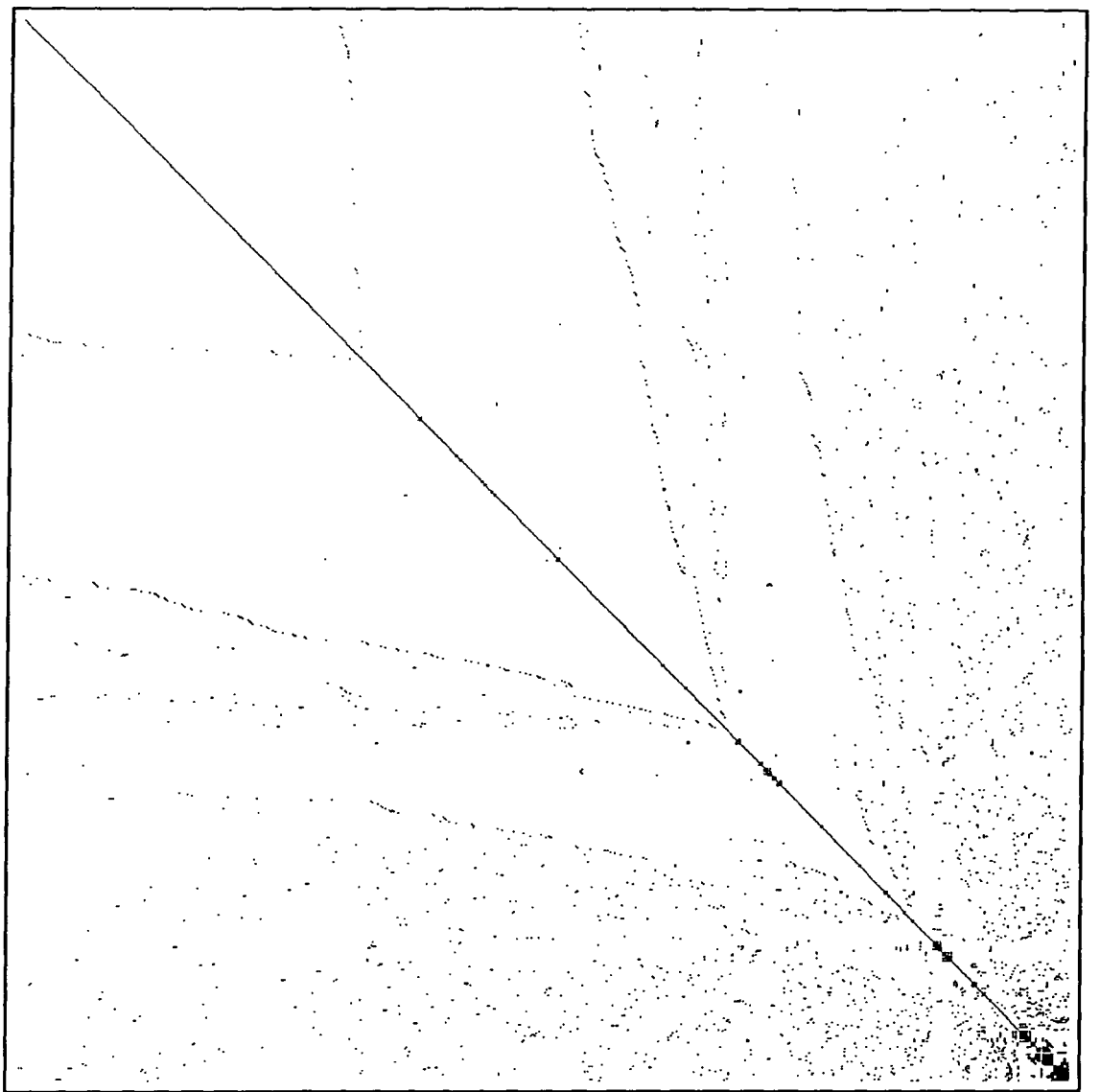
Figure A.32 Matrix of 734 node network, MDLRUML ordering.



MD - Least Recently Used RA ordering of the 734 node network.

```
Ordering took                150 ms
Pointer alteration took       60 ms
Reduction took                40 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   290 ms plus calculation   290 ms
Elements:  before | after      2696   2351    636   64.8%
Multiplications generated      6464   3968
Additions  generated           4847   3234
Divisions  generated            734      0
Multiply-additions generated   6464   3968
Stalls generated                 70      2
```

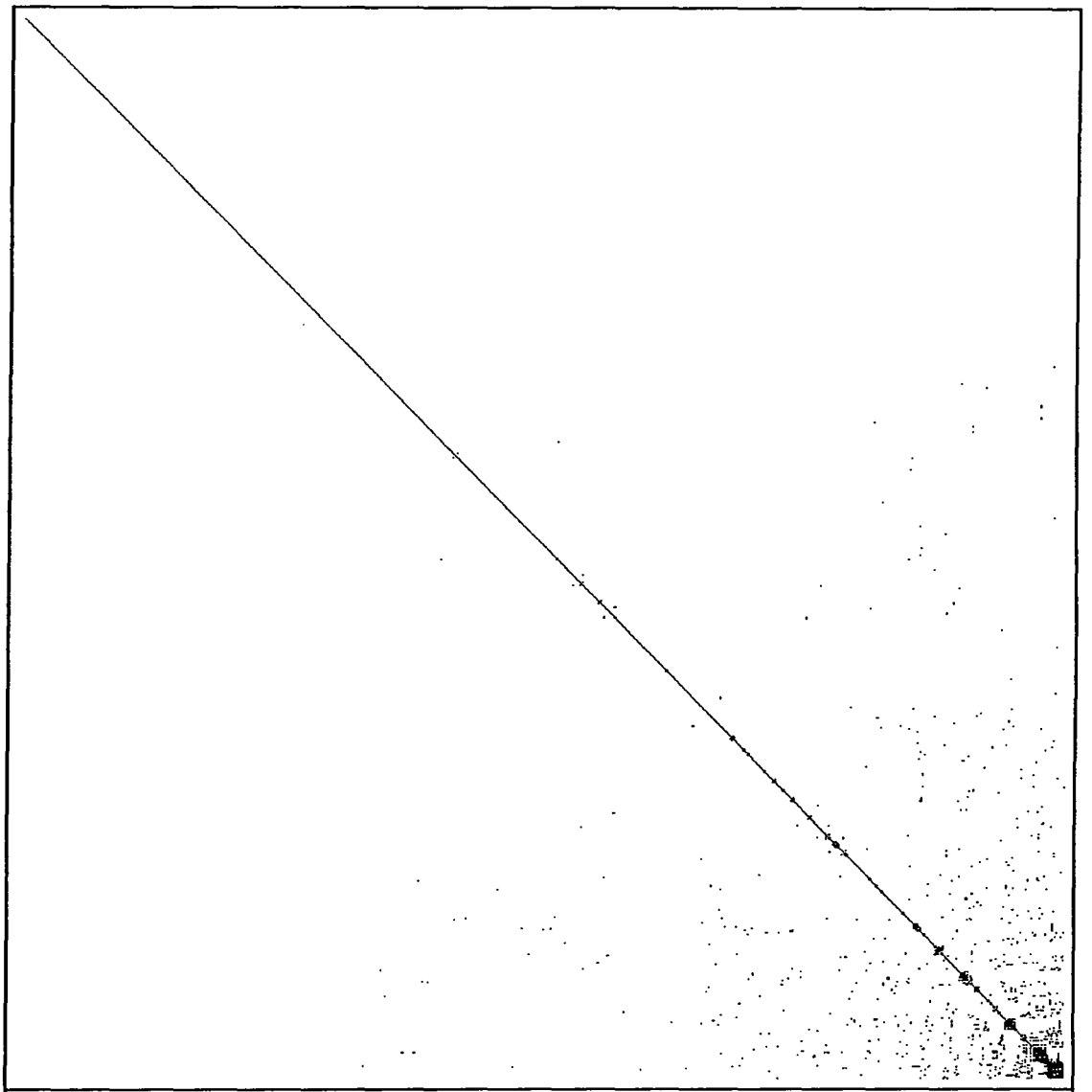
Figure A.33 Matrix of 734 node network, MDLRURA ordering.



Least Recent A + Min Length ordering of the 734 node network.

```
Ordering took                150 ms
Pointer alteration took       50 ms
Reduction took                50 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   290 ms plus calculation   290 ms
Elements:  before | after      2696   2340    625   63.7%
Multiplications generated      6328   3946
Additions  generated           4722   3212
Divisions  generated            734      0
Multiply-additions generated   6328   3946
Stalls generated                 74      2
```

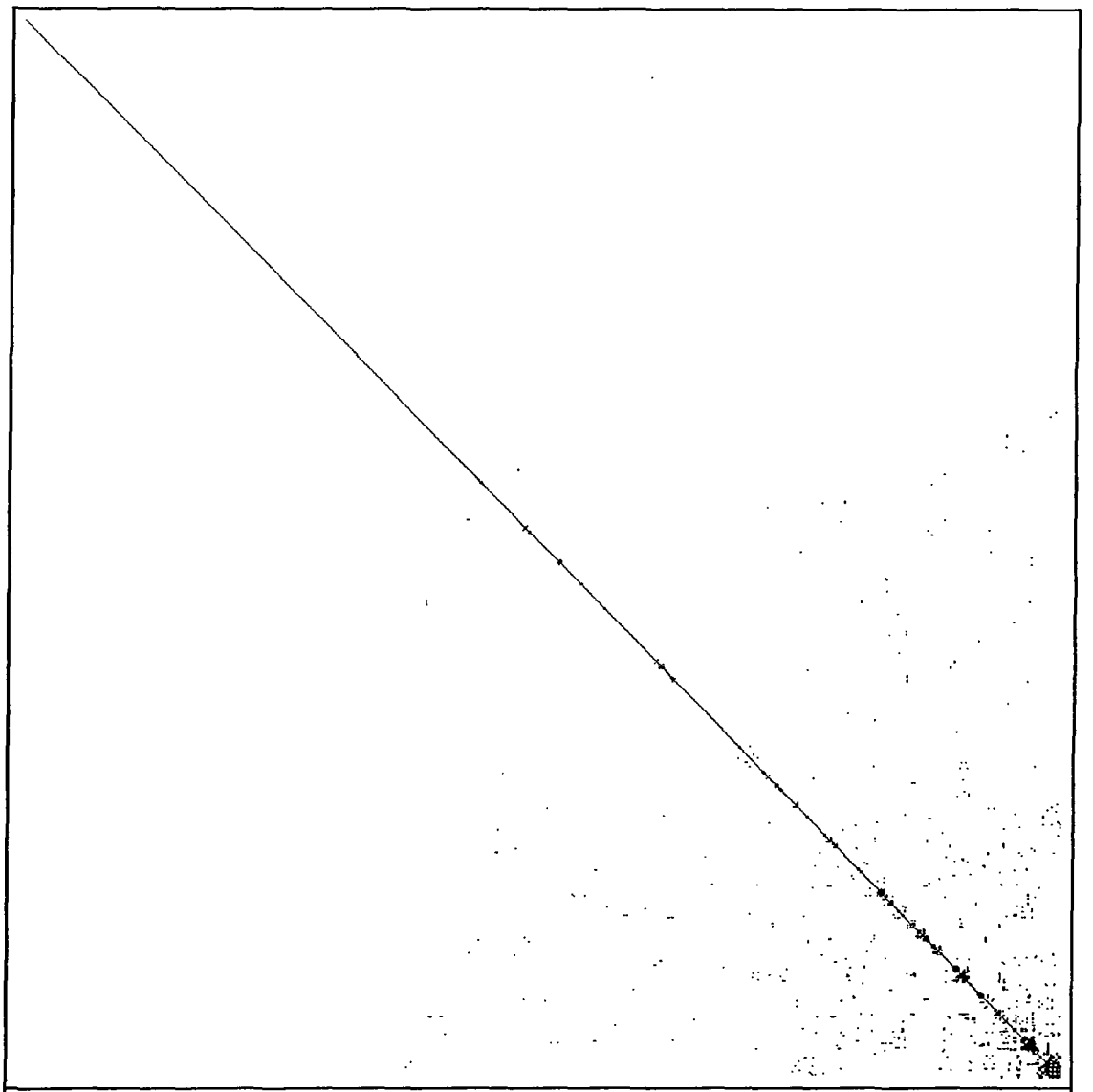
Figure A.34 Matrix of 734 node network, MDLRUMLA ordering.



Tinney-2 ordering. Fill for 734 node network.

```
Ordering took                550 ms
Pointer alteration took       60 ms
Reduction took                30 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   680 ms plus calculation   680 ms
Elements:  before | after      2696   2363    648   66.1%
Multiplications generated      6614   3992
Additions  generated           4985   3258
Divisions  generated            734      0
Multiply-additions generated   6614   3992
Stalls generated                148     18
```

Figure A.35 Fill for 734 node network, Tinney–2 ordering.



Tinney-3 ordering. Fill for 734 node network.

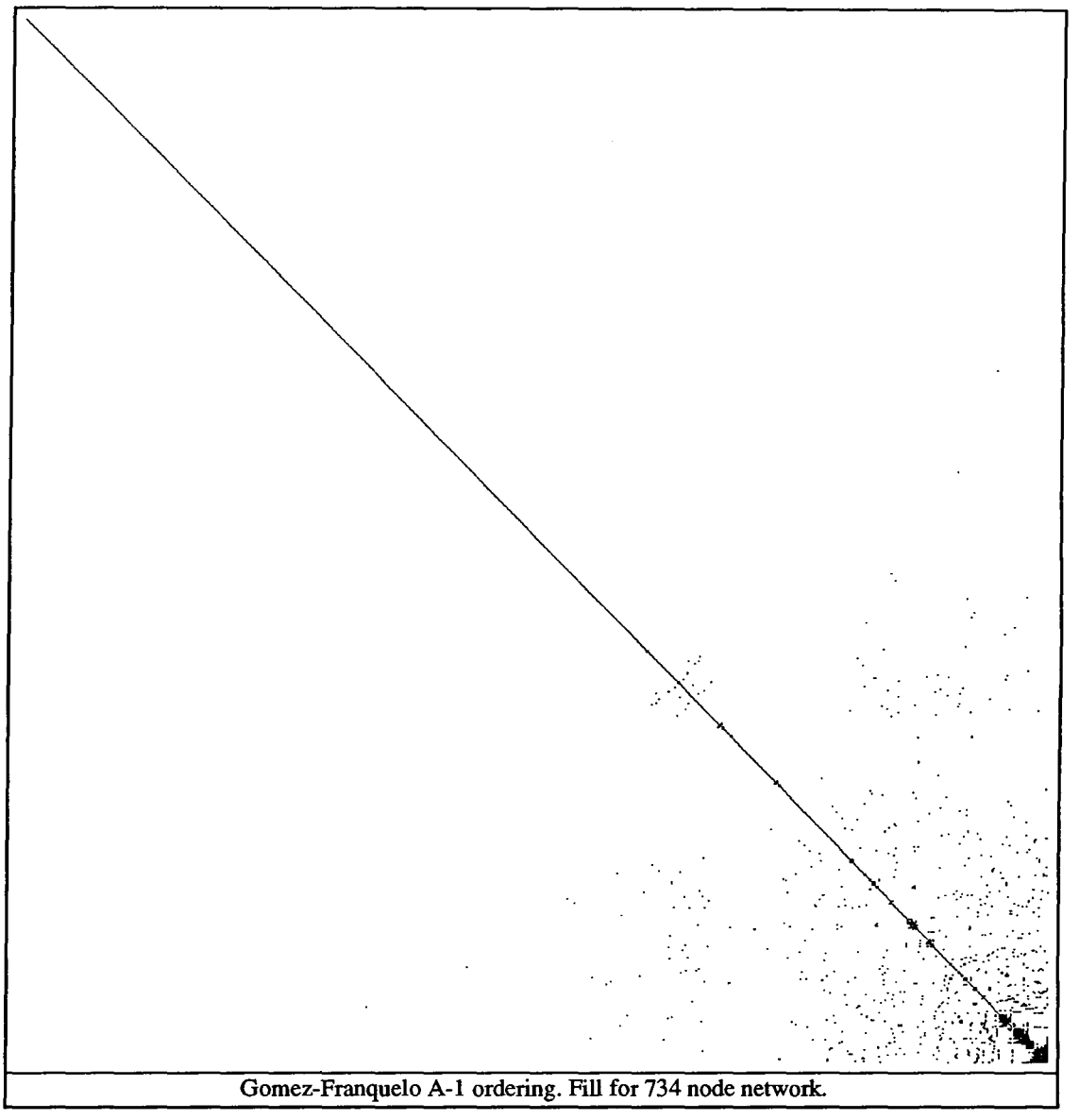
```
Ordering took            14090 ms
Pointer alteration took     60 ms
Reduction took              40 ms
Solution took               40 ms
Calculation took            00 ms
Ordering, reduction and solution took 14240 ms plus calculation 14240 ms
Elements:  before | after      2696   2318    603   61.5%
Multiplications generated      6136   3902
Additions  generated           4552   3168
Divisions  generated            734      0
Multiply-additions generated   6136   3902
Stalls generated                181     19
```

Figure A.36 Fill for 734 node network, Tinney–3 ordering.

Minimum-depth minimum-length ordering. Fill for 734 node network.

```
Ordering took                710 ms
Pointer alteration took       60 ms
Reduction took                40 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   850 ms plus calculation   850 ms
Elements:  before | after      2696   2343    628   64.0%
Multiplications generated      6358   3952
Additions  generated           4749   3218
Divisions  generated            734      0
Multiply-additions generated   6358   3952
Stalls generated                 71      2
```

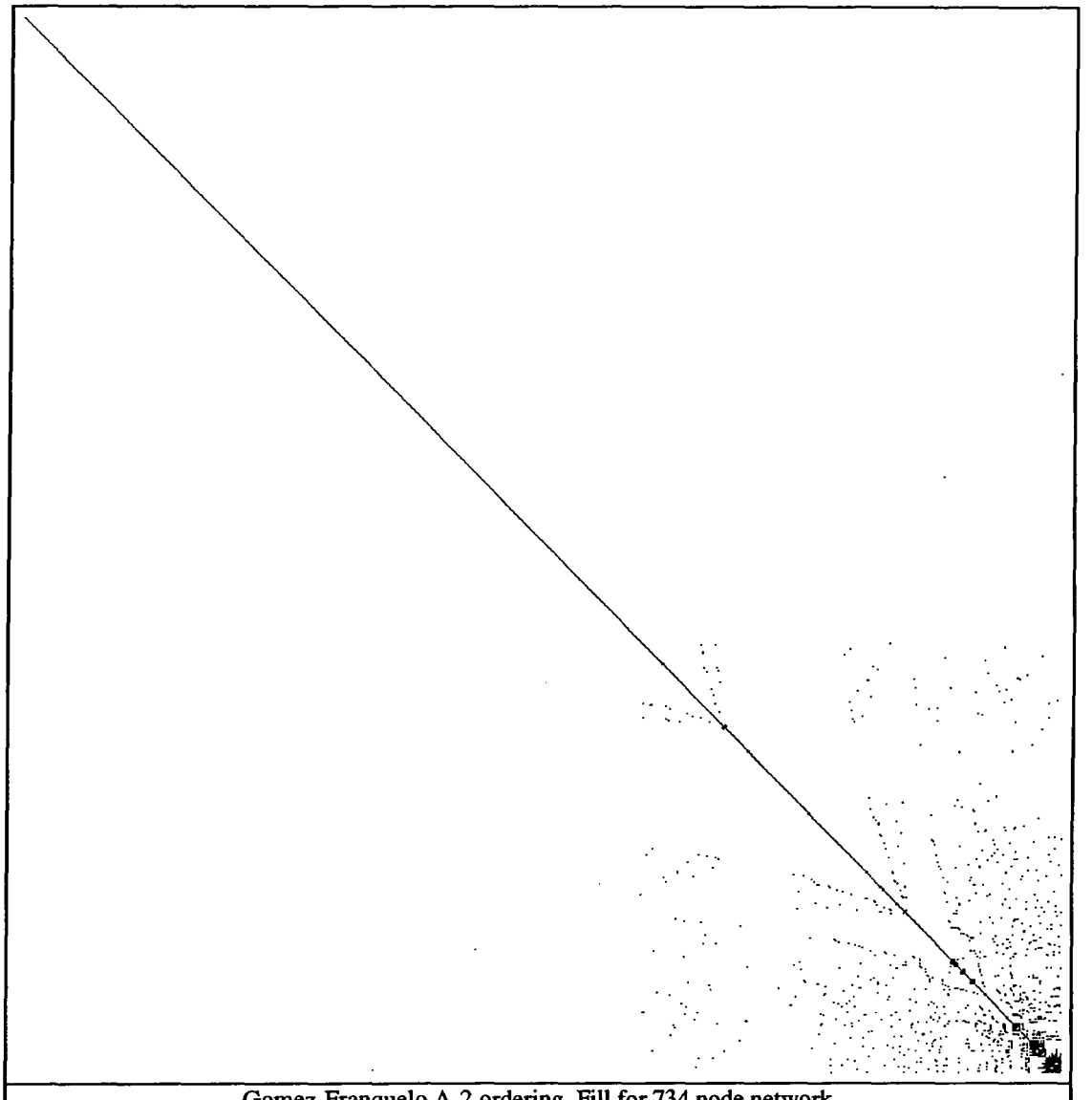
Figure A.37 Fill for 734 node network, MDML ordering.



Gomez-Franquelo A-1 ordering. Fill for 734 node network.

```
Ordering took                650 ms
Pointer alteration took       50 ms
Reduction took                50 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   800 ms plus calculation   800 ms
Elements:  before | after      2696   2334    619   63.1%
Multiplications generated      6268   3934
Additions  generated           4668   3200
Divisions  generated            734      0
Multiply-additions generated   6268   3934
Stalls generated                 87      3
```

Figure A.38 Fill for 734 node network, GF–1 ordering.



Gomez-Franquelo A-2 ordering. Fill for 734 node network.

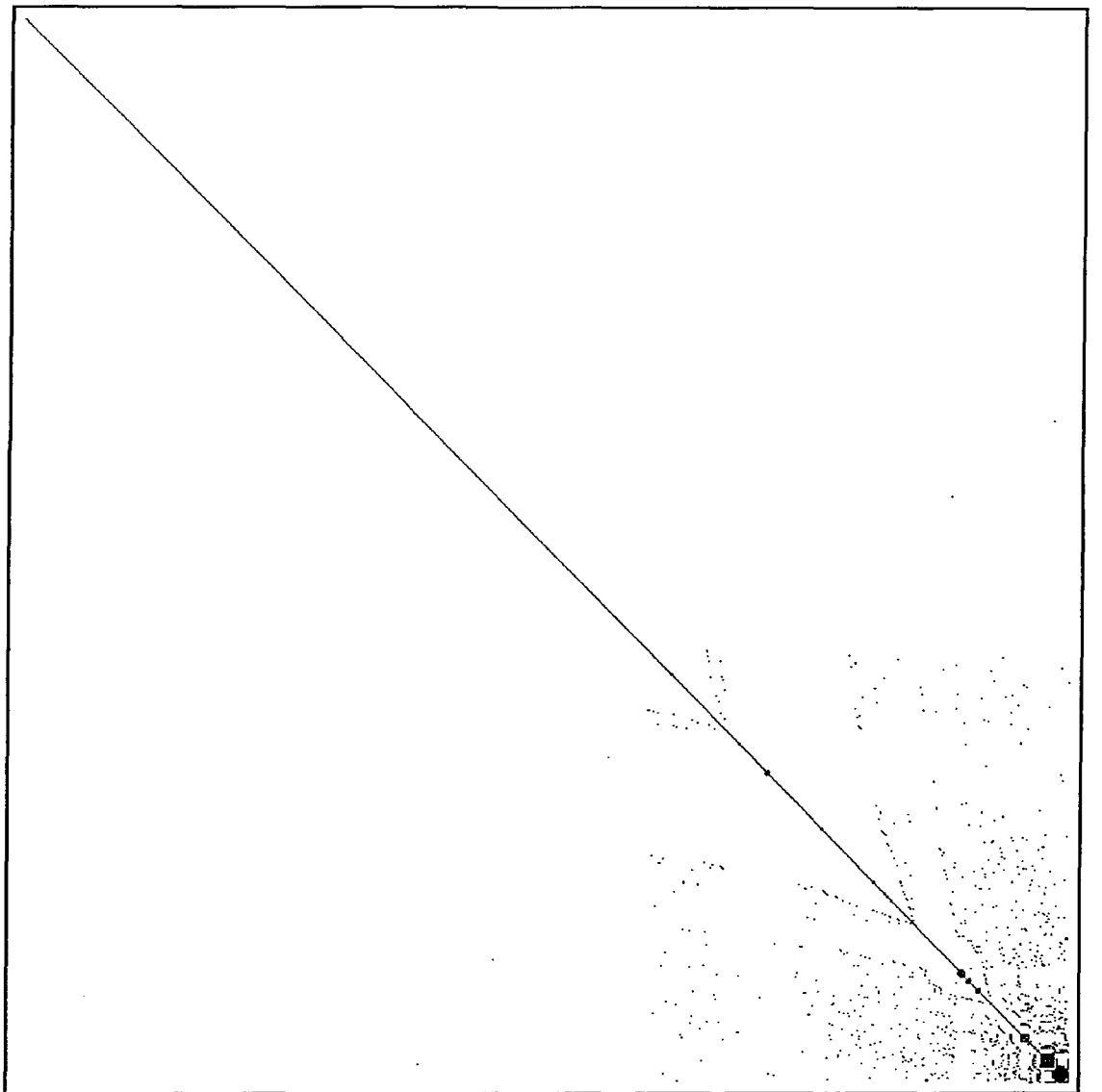
```
Ordering took                  670 ms
Pointer alteration took         60 ms
Reduction took                  40 ms
Solution took                   50 ms
Calculation took                00 ms
Ordering, reduction and solution took   840 ms plus calculation   840 ms
Elements:  before | after      2696   2358    643   65.5%
Multiplications generated      6550   3982
Additions  generated           4926   3248
Divisions  generated            734      0
Multiply-additions generated   6550   3982
Stalls generated                 68      2
```

Figure A.39 Fill for 734 node network, GF–2 ordering.

Gomez-Franquelo A-3 ordering. Fill for 734 node network.

```
Ordering took                  690 ms
Pointer alteration took         60 ms
Reduction took                  60 ms
Solution took                   40 ms
Calculation took                00 ms
Ordering, reduction and solution took   860 ms plus calculation   860 ms
Elements:  before | after      2696   2578    863   88.0%
Multiplications generated      9428   4422
Additions  generated           7584   3688
Divisions  generated            734      0
Multiply-additions generated   9428   4422
Stalls generated                 16      1
```

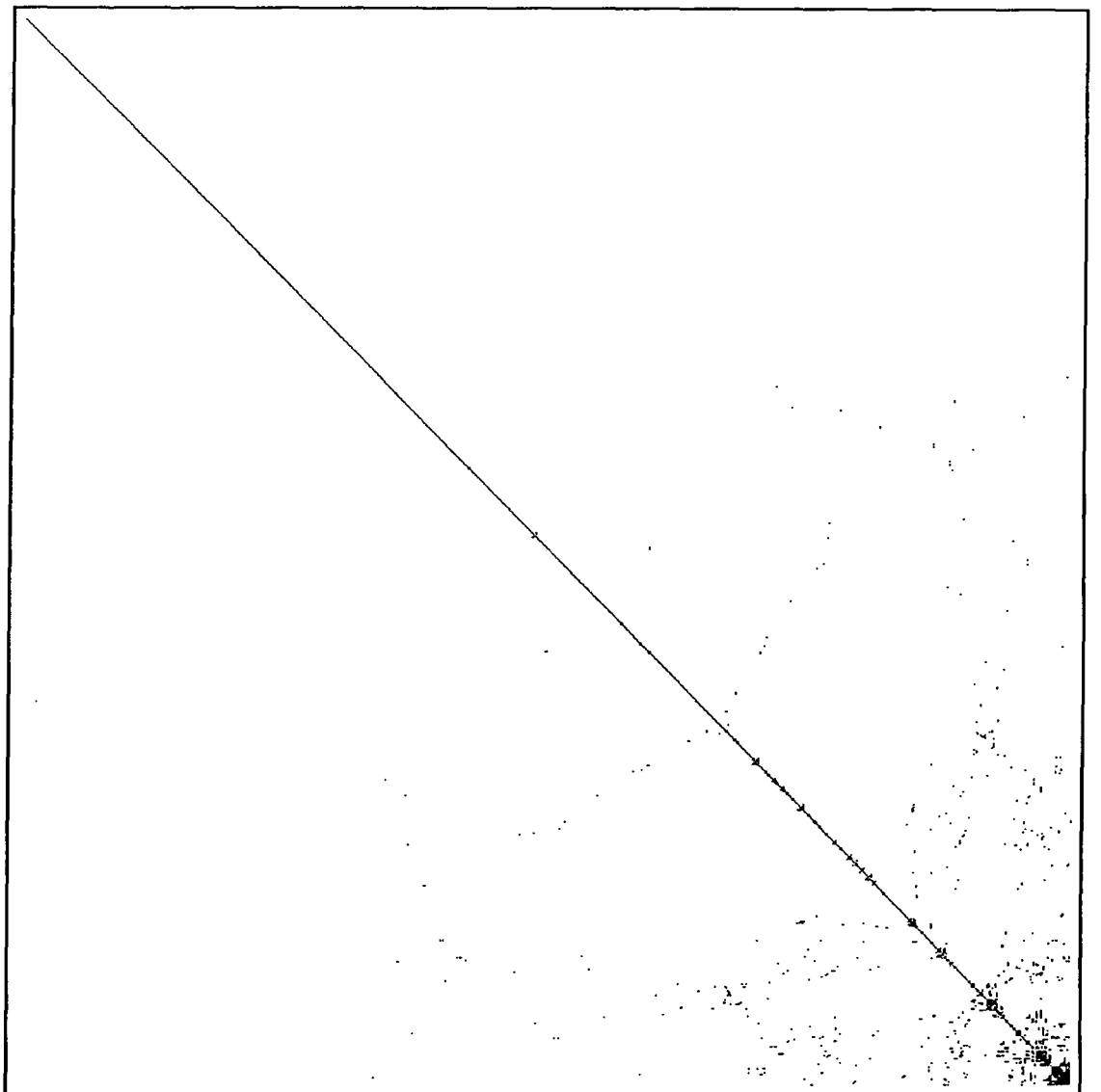
Figure A.40 Fill for 734 node network, GF–3 ordering.



MD--Least Recently Used ordering. Fill for 734 node network.

```
Ordering took                150 ms
Pointer alteration took       60 ms
Reduction took                50 ms
Solution took                 50 ms
Calculation took              00 ms
Ordering, reduction and solution took   310 ms plus calculation   310 ms
Elements:  before | after      2696   2369    654   66.7%
Multiplications generated      6718   4004
Additions  generated           5083   3270
Divisions  generated            734      0
Multiply-additions generated   6718   4004
Stalls generated                 67      2
```

Figure A.41 Fill for 734 node network, MDLRU ordering.



MD--Reversed LRU ordering. Fill for 734 node network.

```
Ordering took                140 ms
Pointer alteration took       60 ms
Reduction took                40 ms
Solution took                 40 ms
Calculation took              00 ms
Ordering, reduction and solution took   280 ms plus calculation   280 ms
Elements:  before | after      2696   2336    621   63.3%
Multiplications generated      6280   3938
Additions  generated           4678   3204
Divisions  generated            734      0
Multiply-additions generated   6280   3938
Stalls generated                232     23
```

Figure A.42 Fill for 734 node network, MDLRUR ordering.

Appendix B.

Path Diagrams

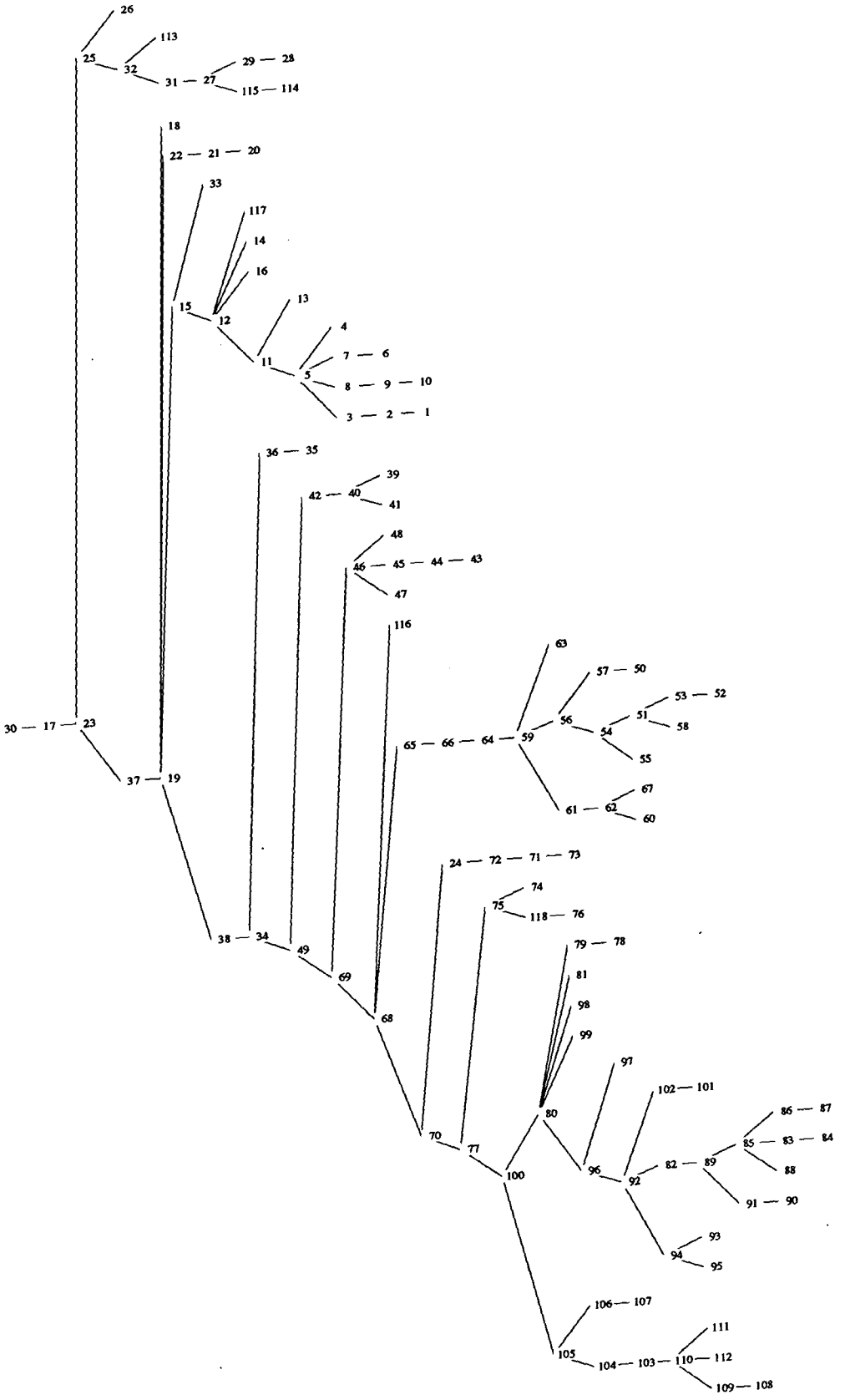


Figure B.1 Path for 118 node network, Tinney–2 ordering.

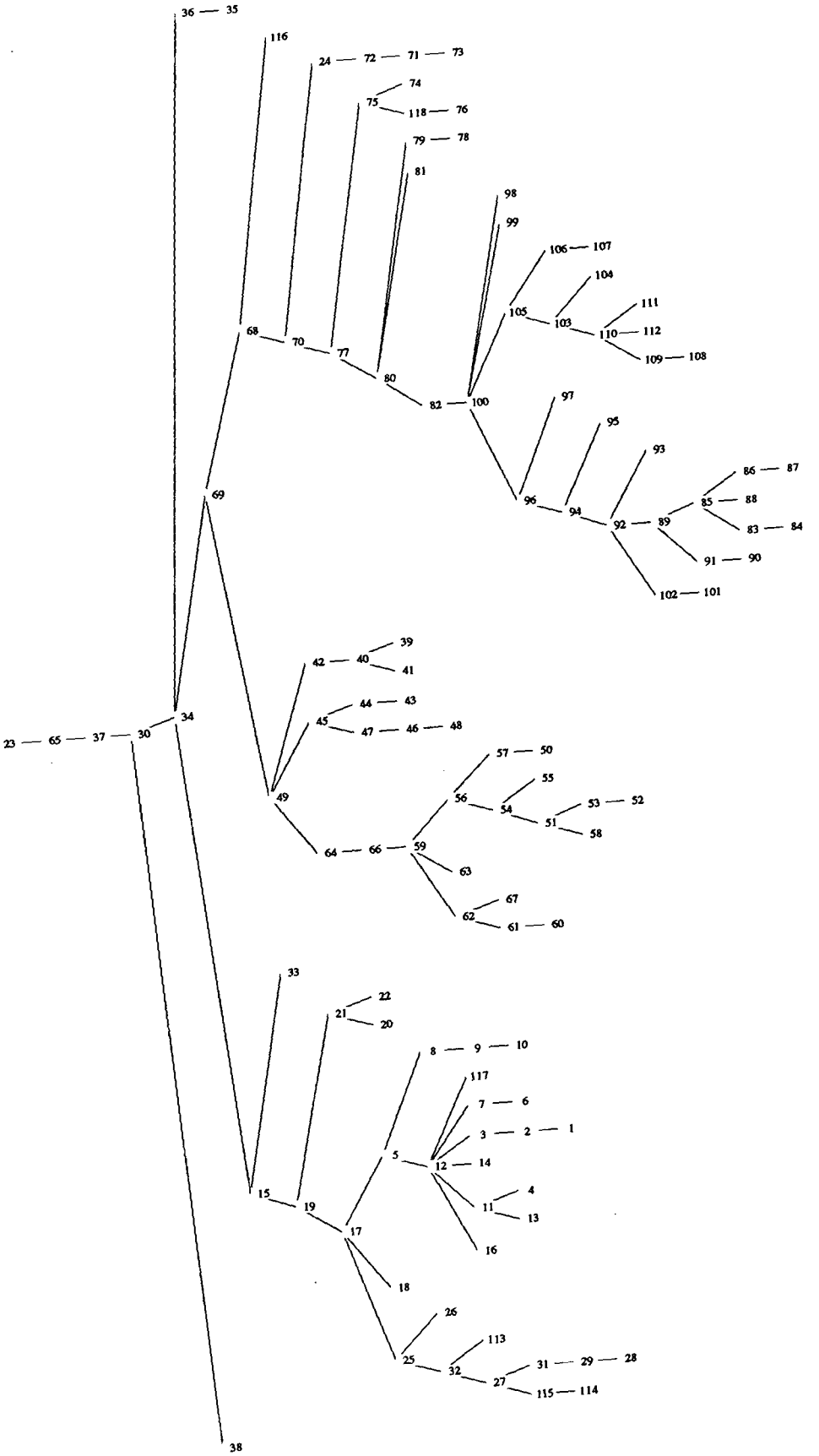


Figure B.2 Path for 118 node network, Tinney–3 ordering.

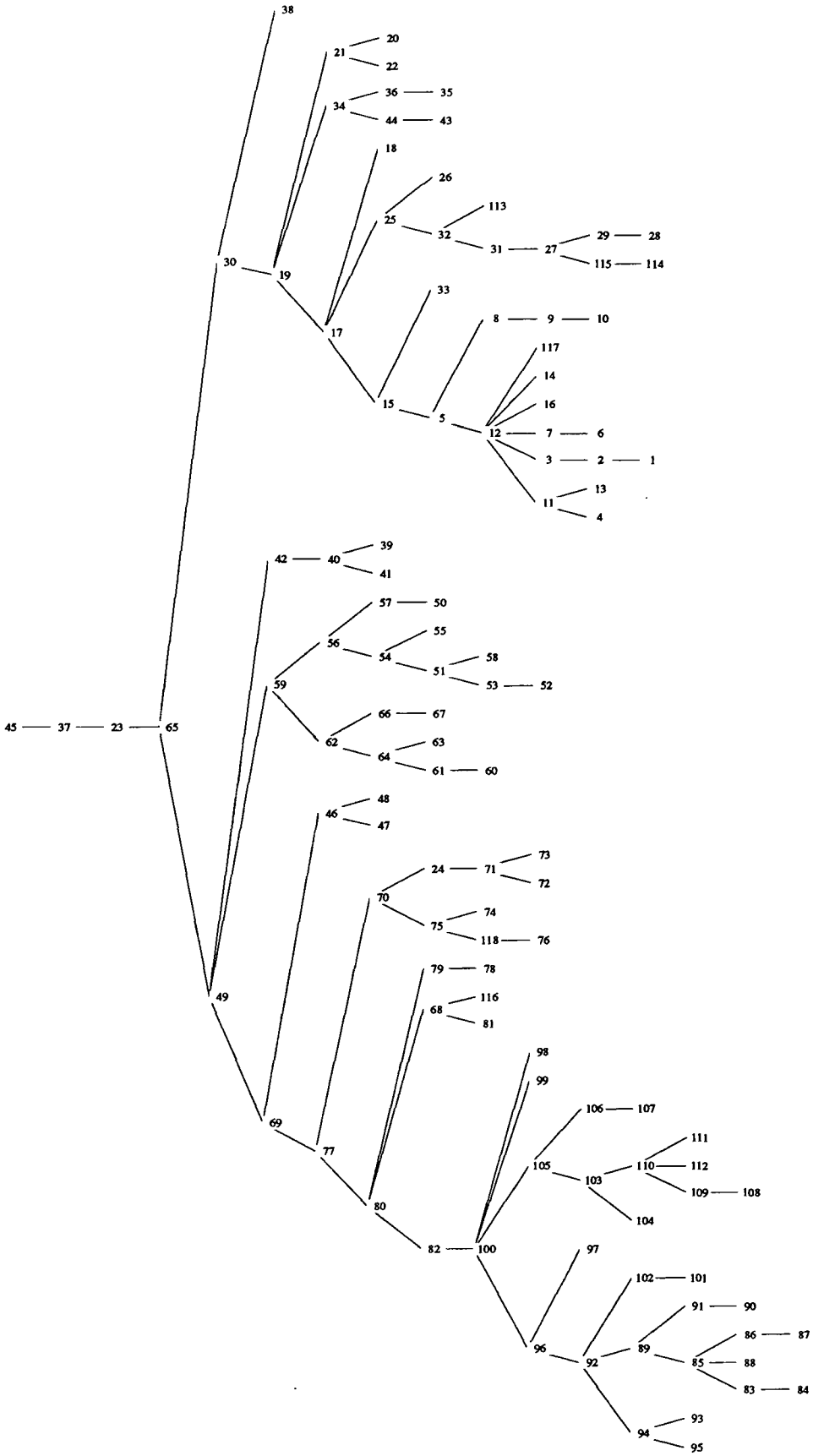


Figure B.3 Path for 118 node network, MDML ordering.

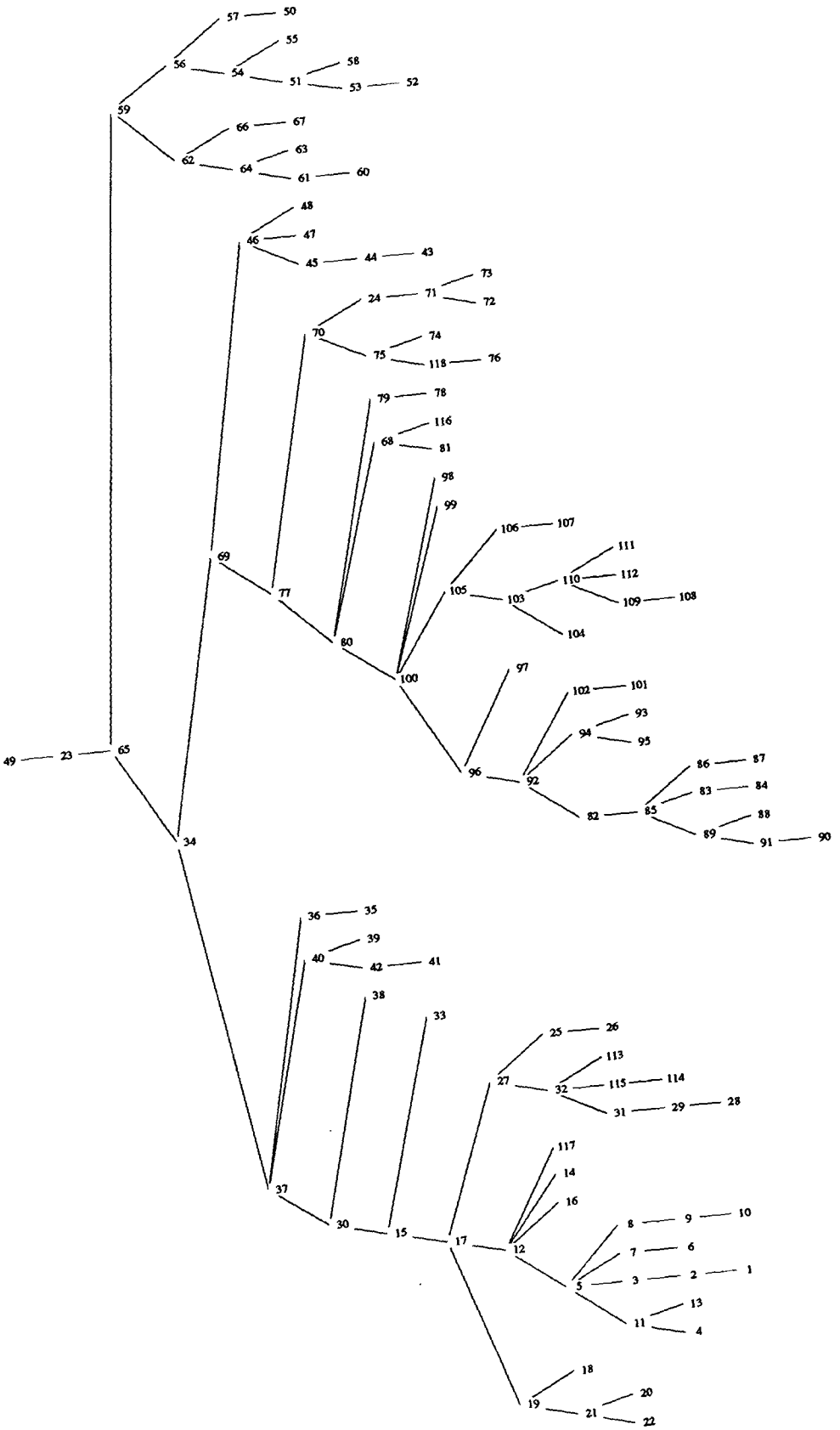


Figure B.4 Path for 118 node network, GF–1 ordering.

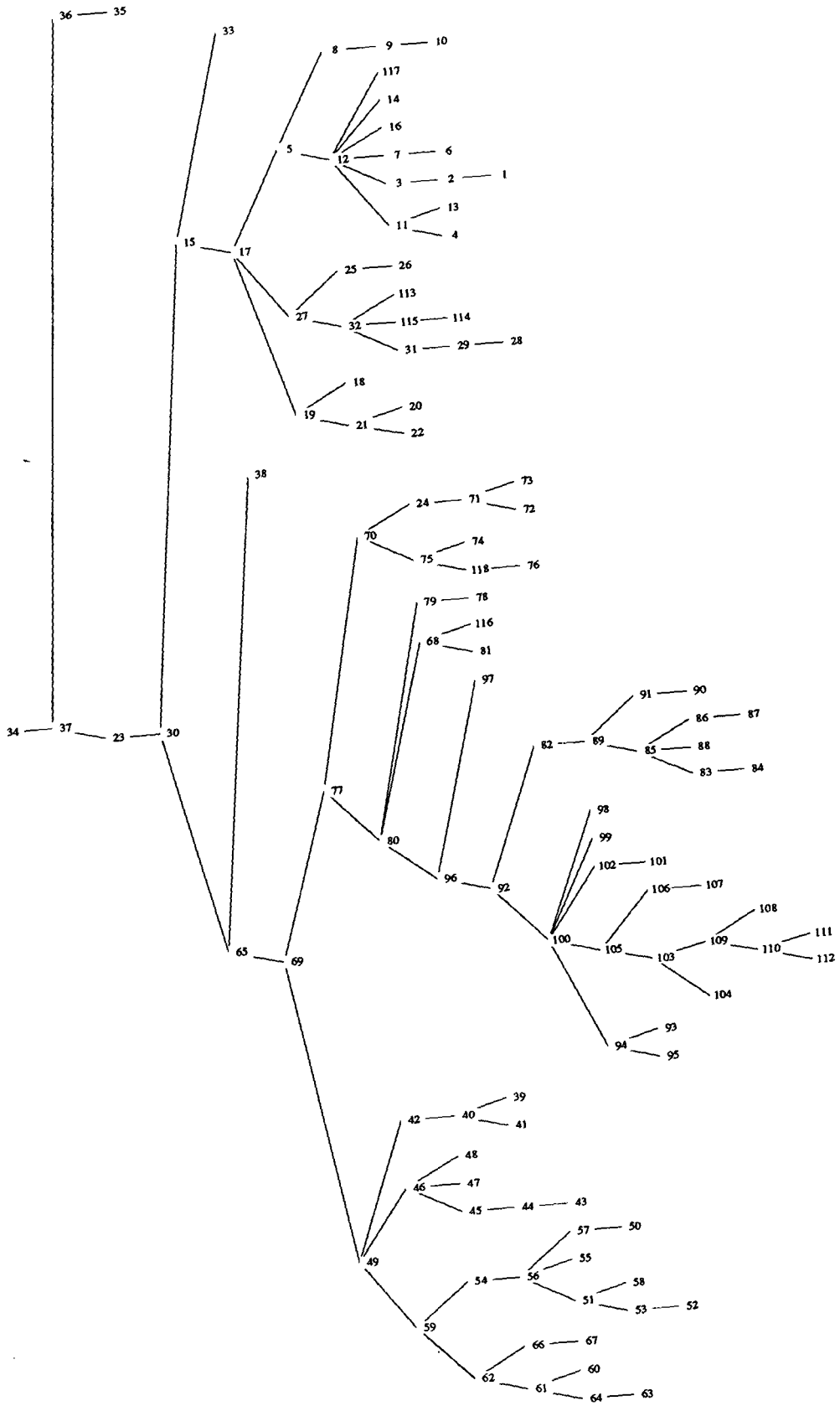


Figure B.5 Path for 118 node network, GF–2 ordering.

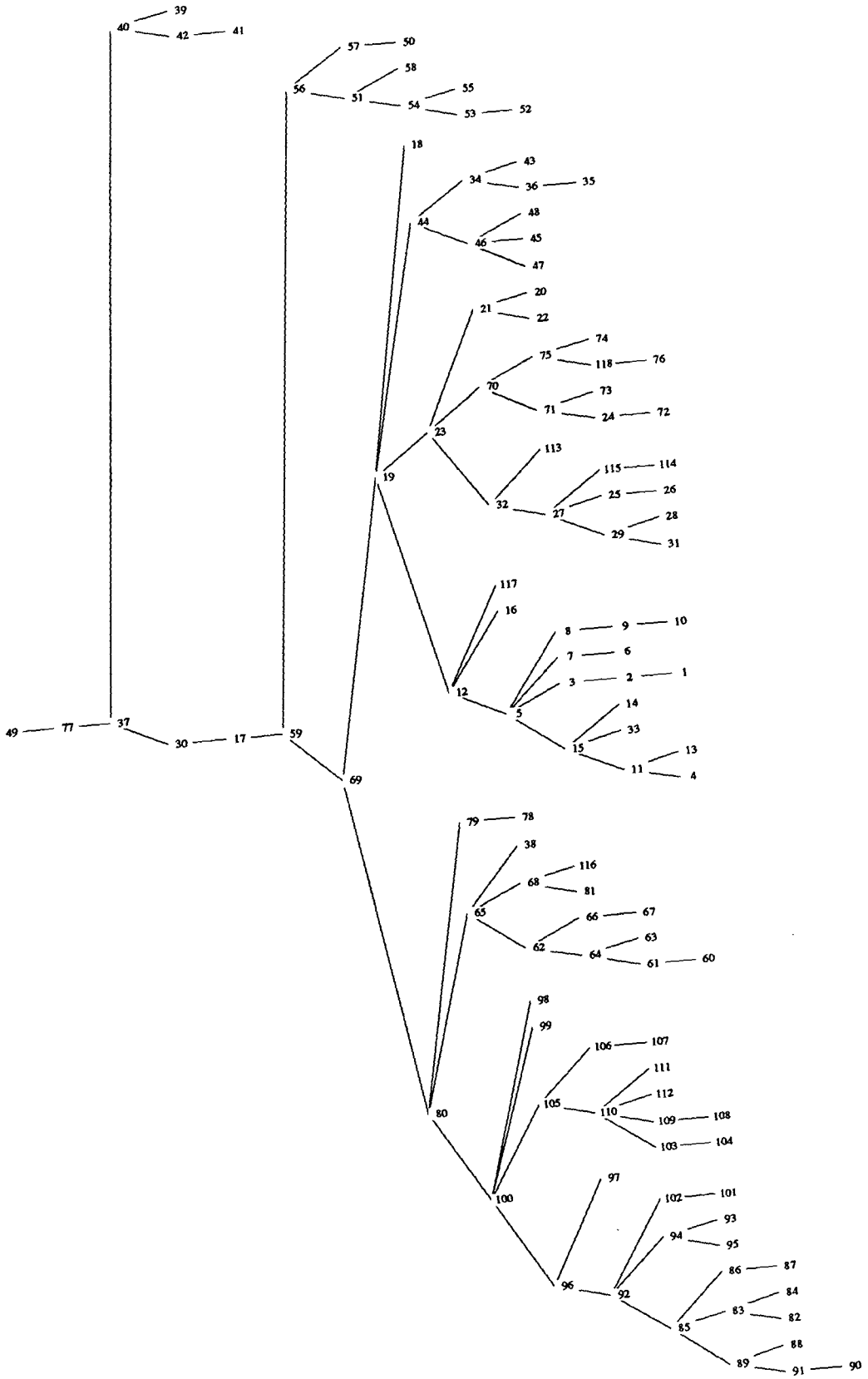


Figure B.6 Path for 118 node network, GF–3 ordering.

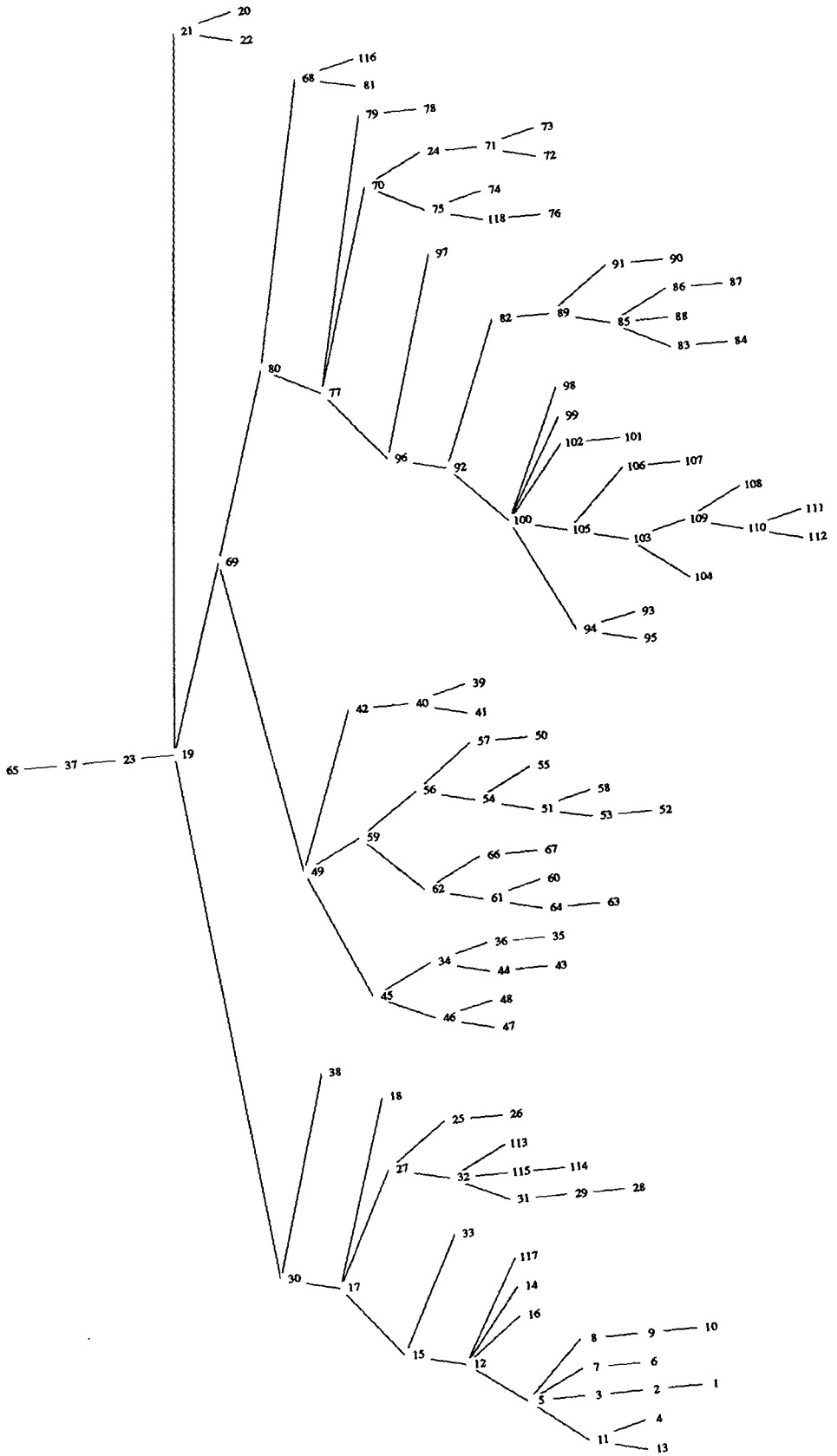


Figure B.7 Path for 118 node network, MDLRU ordering.

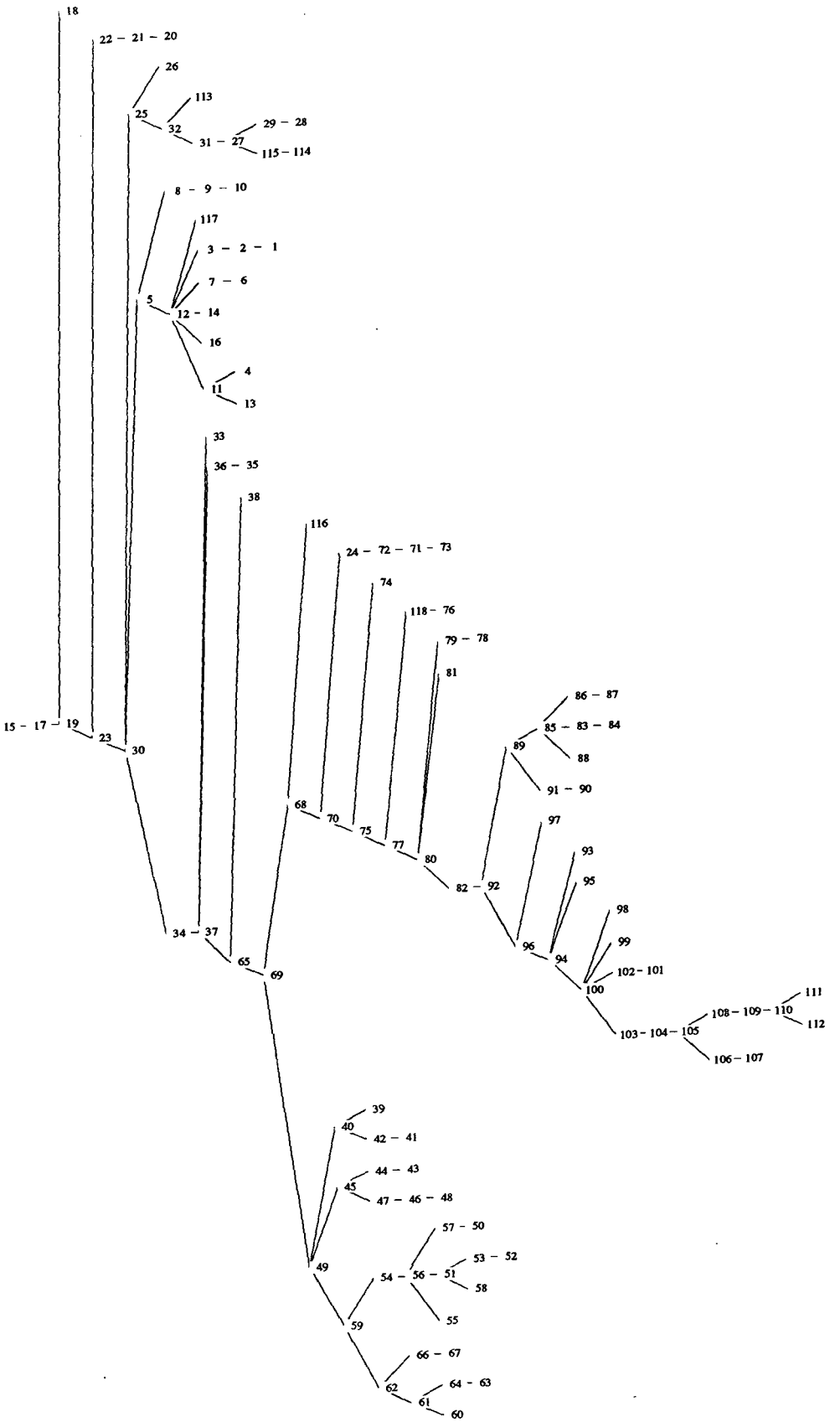


Figure B.8 Path for 118 node network, MDLRUR ordering.

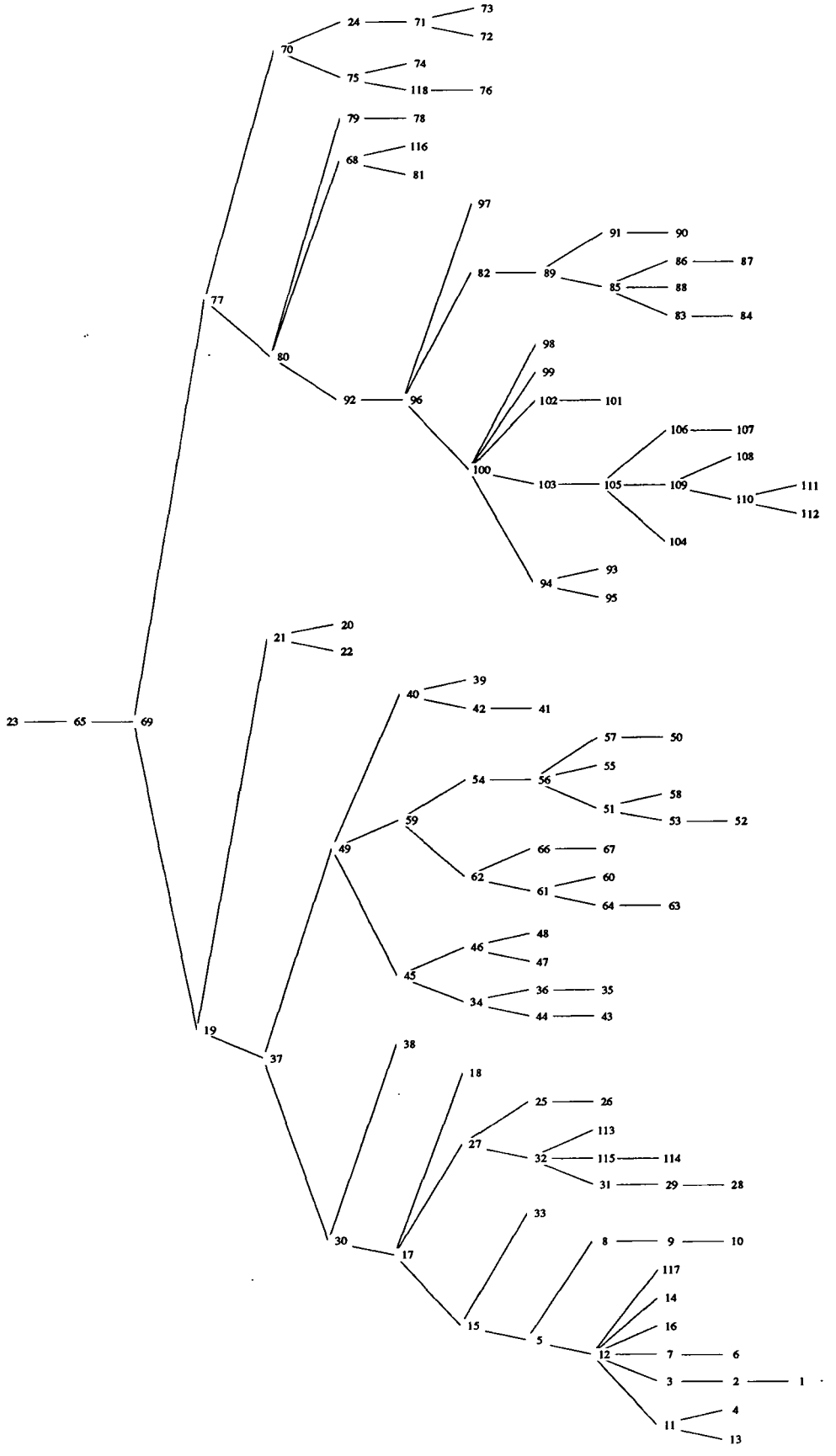


Figure B.9 Path for 118 node network, MDLRUML ordering.

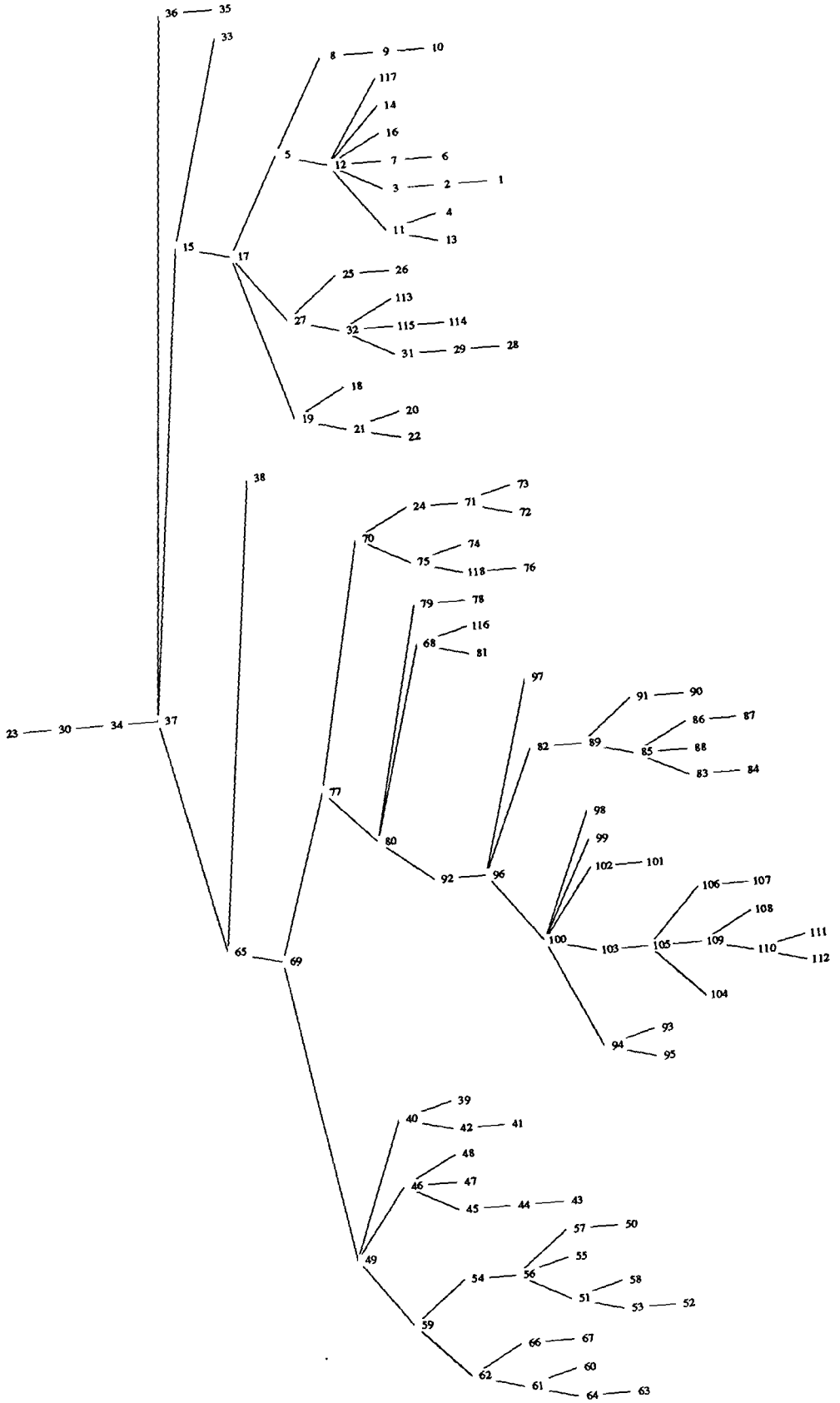


Figure B.10 Path for 118 node network, MDLRURA ordering.

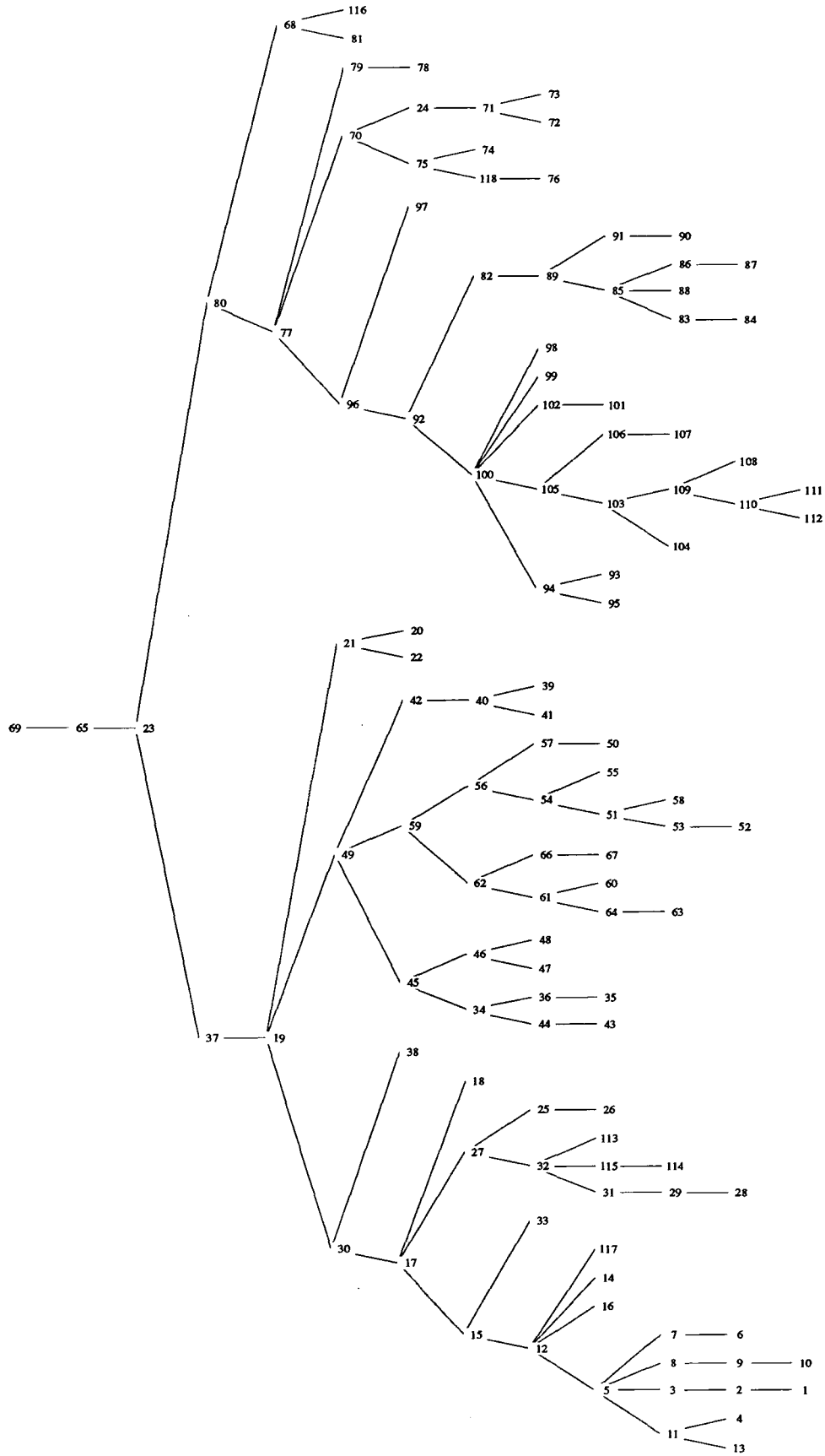


Figure B.11 Path for 118 node network, MDLRUMLA ordering.

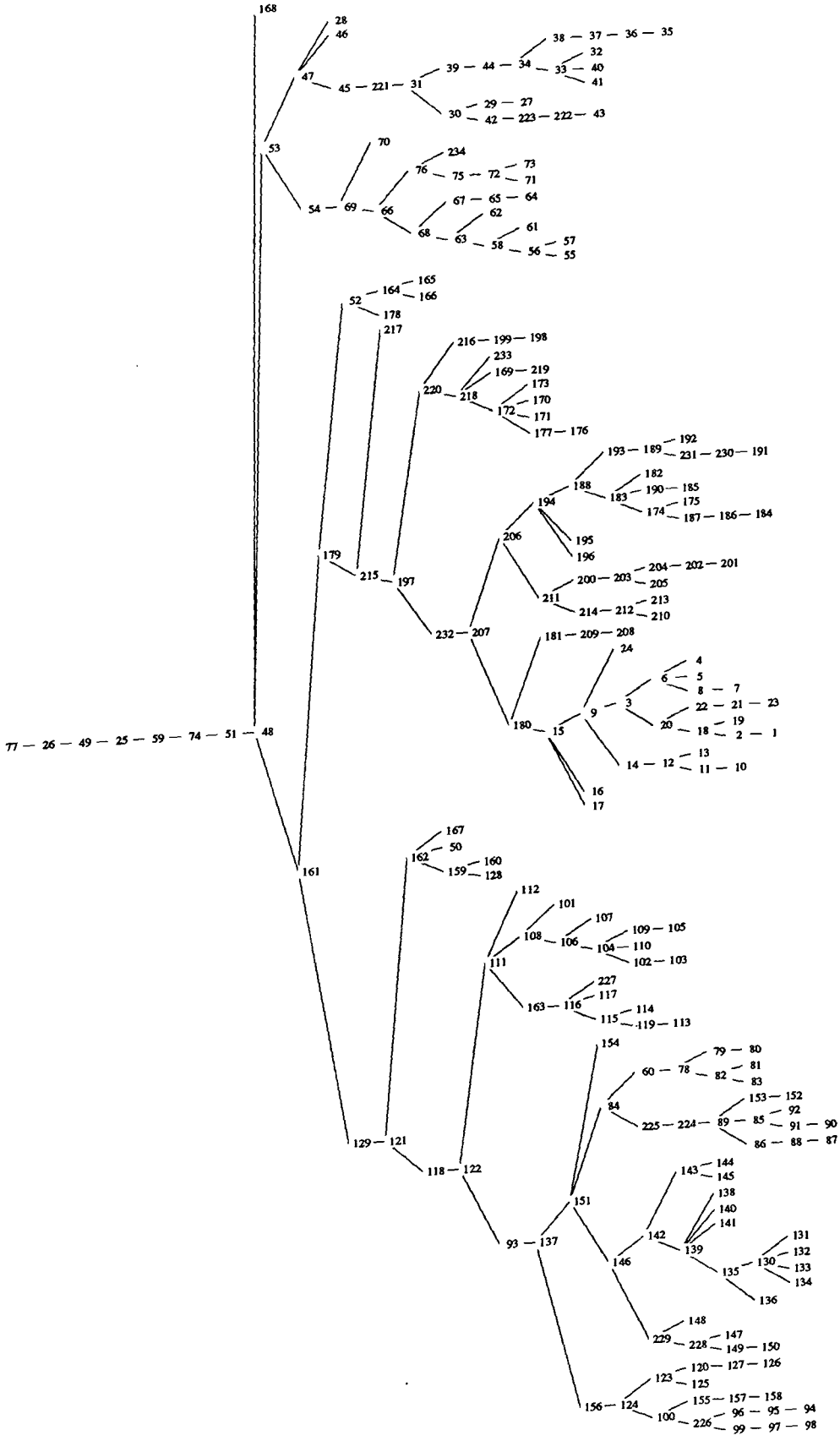


Figure B.12 Path for 234 node network, Tinney–2 ordering.

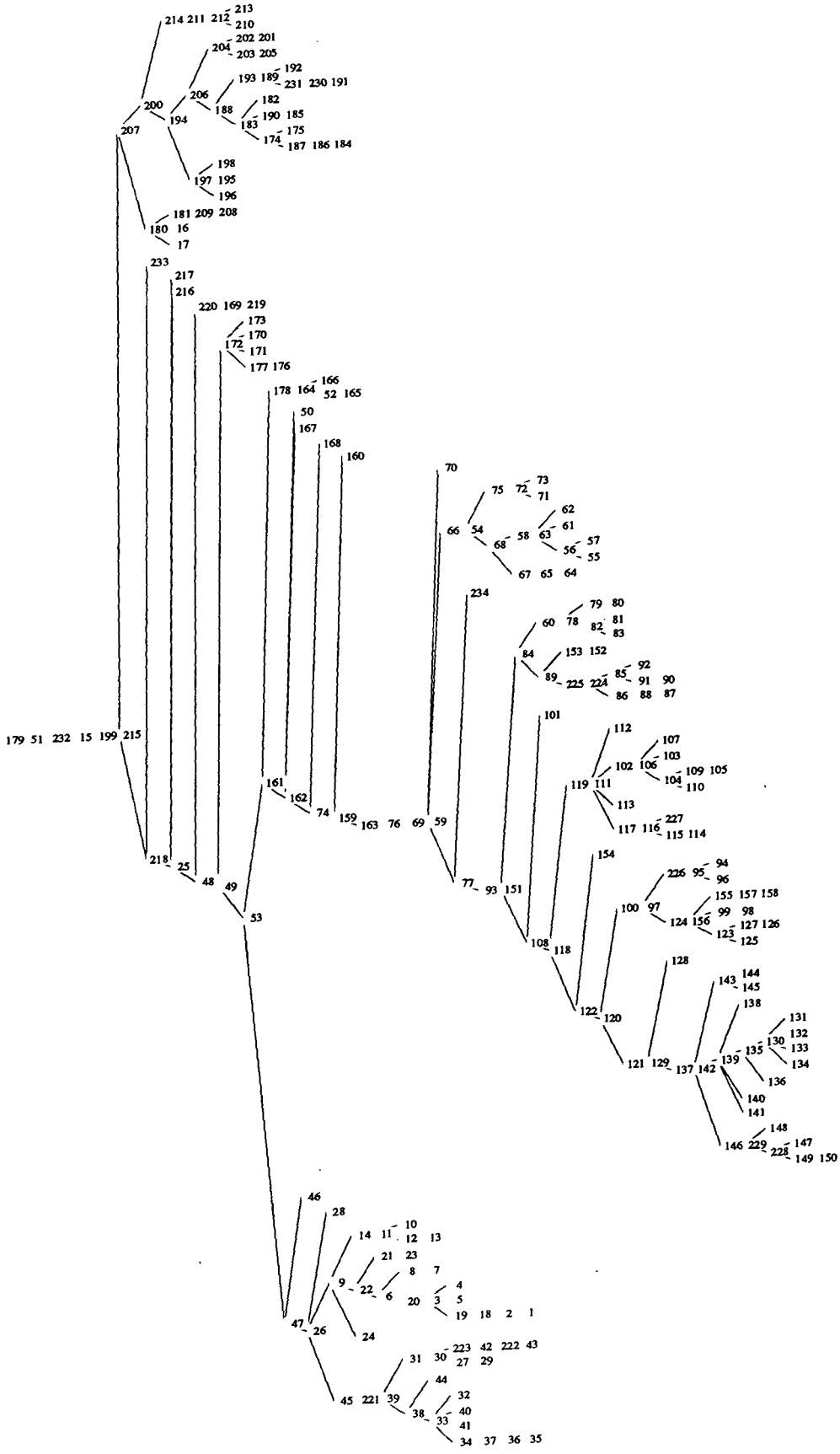


Figure B.13 Path for 234 node network, Tinney–3 ordering.

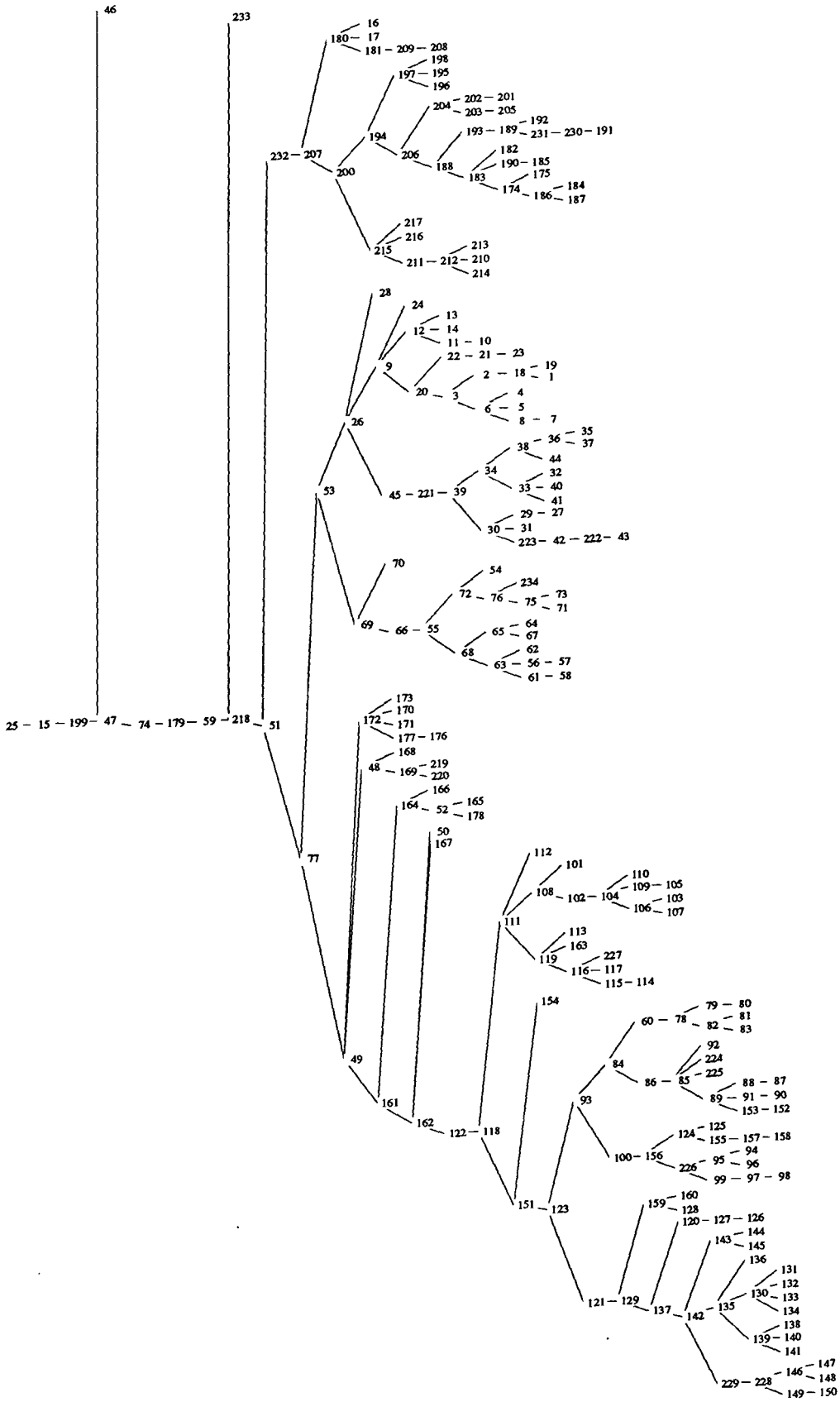


Figure B.14 Path for 234 node network, MDML ordering.

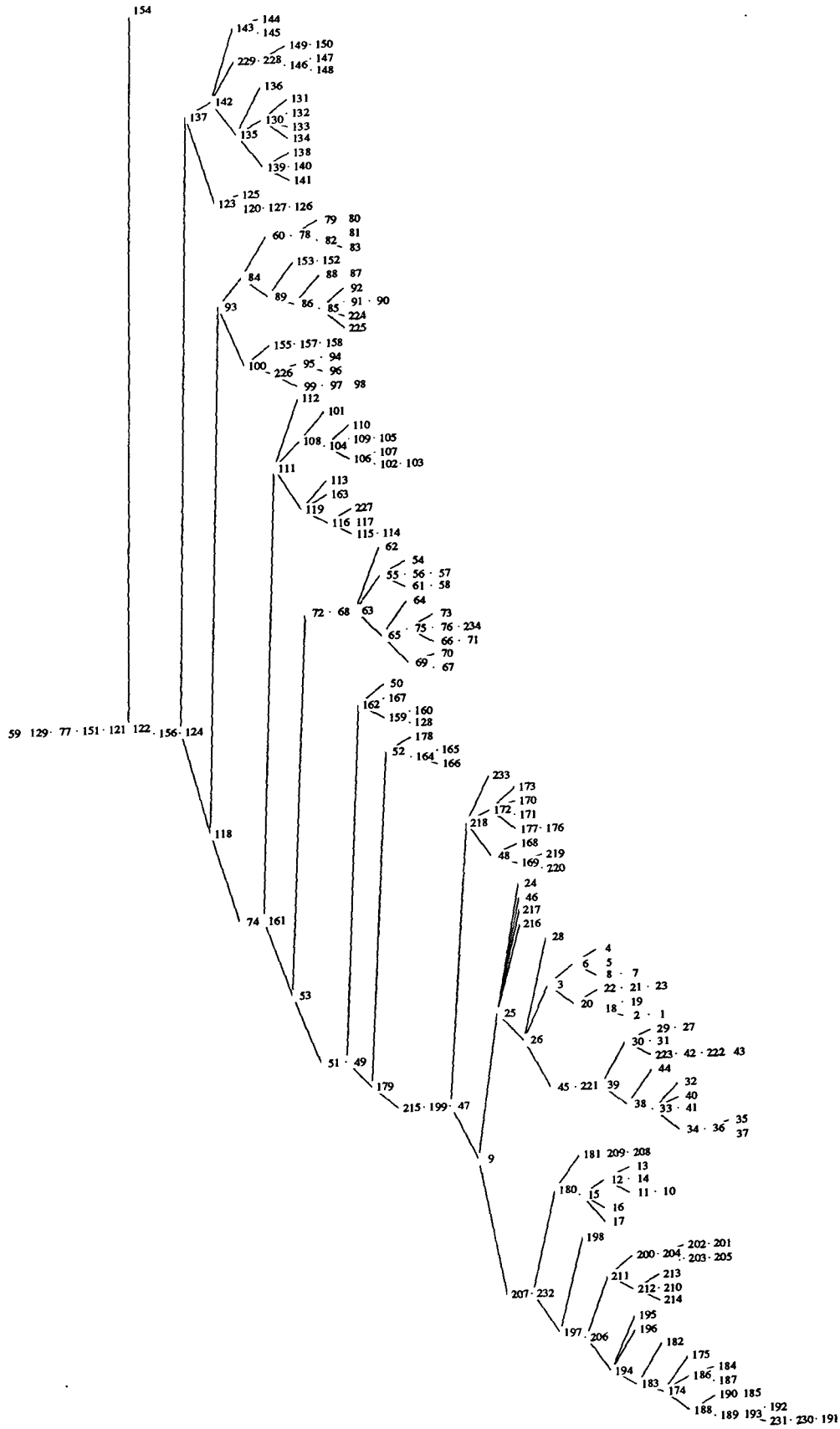


Figure B.15 Path for 234 node network, GF–1 ordering.

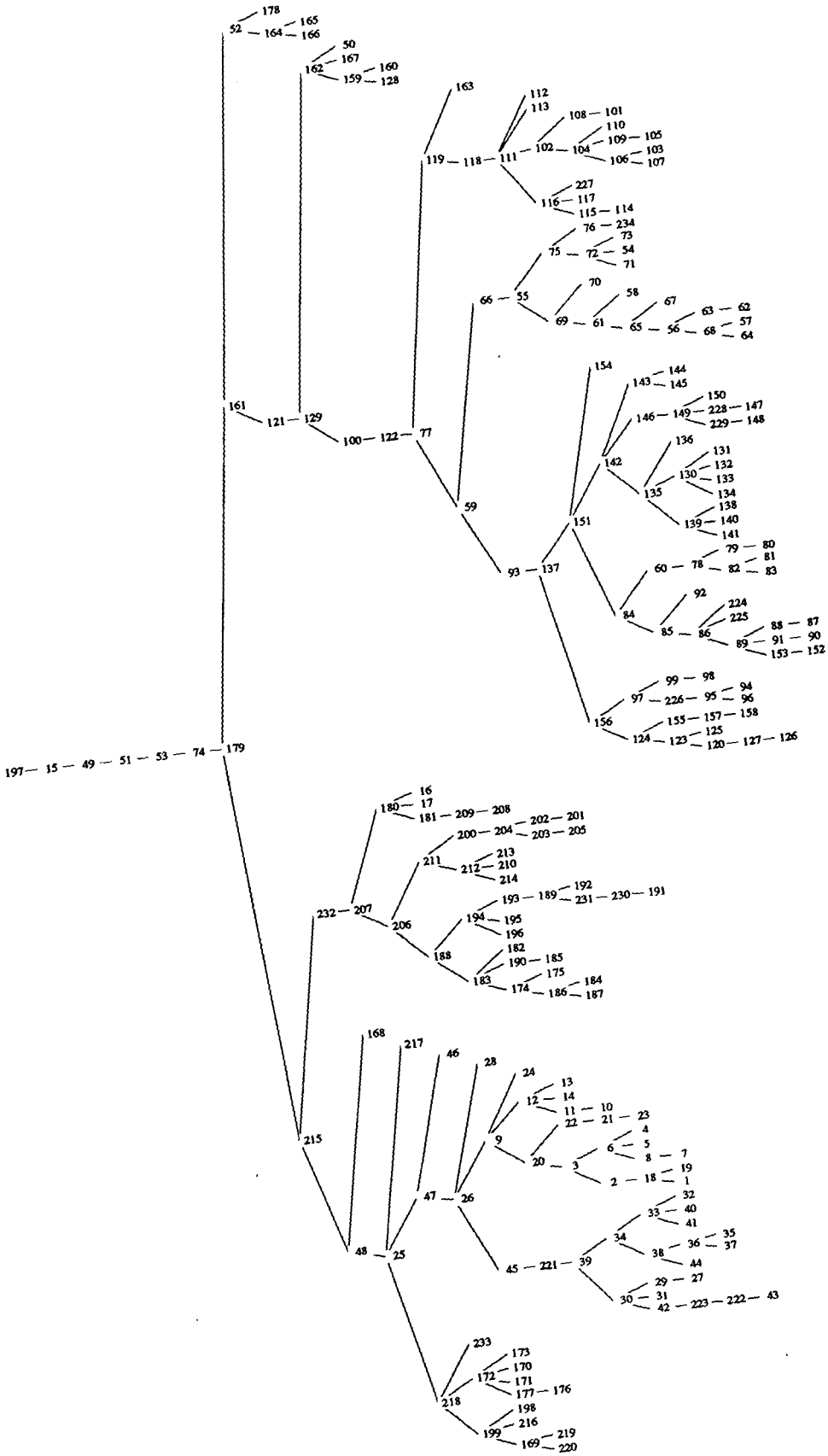


Figure B.16 Path for 234 node network, GF–2 ordering.

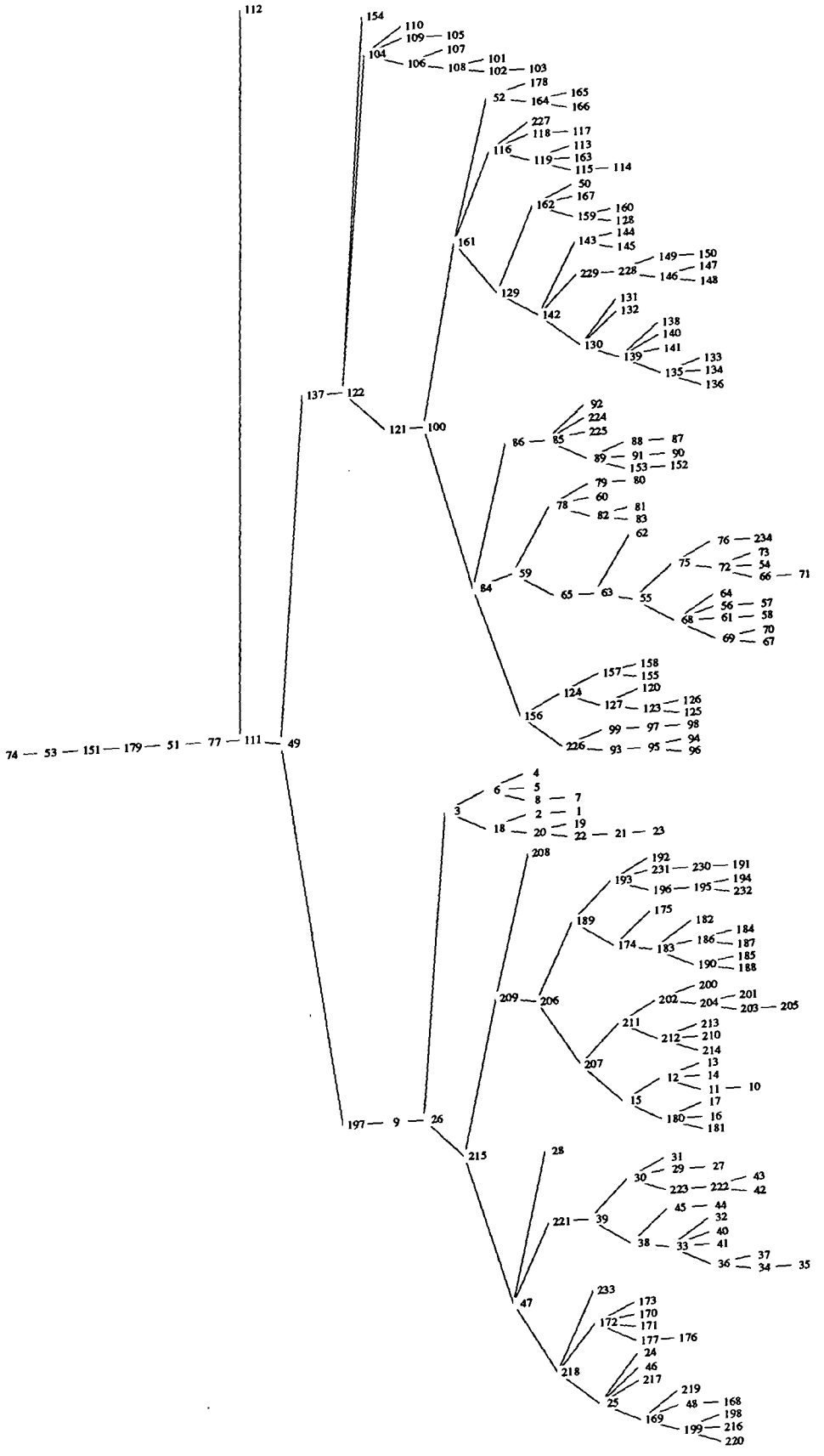


Figure B.17 Path for 234 node network, GF–3 ordering.

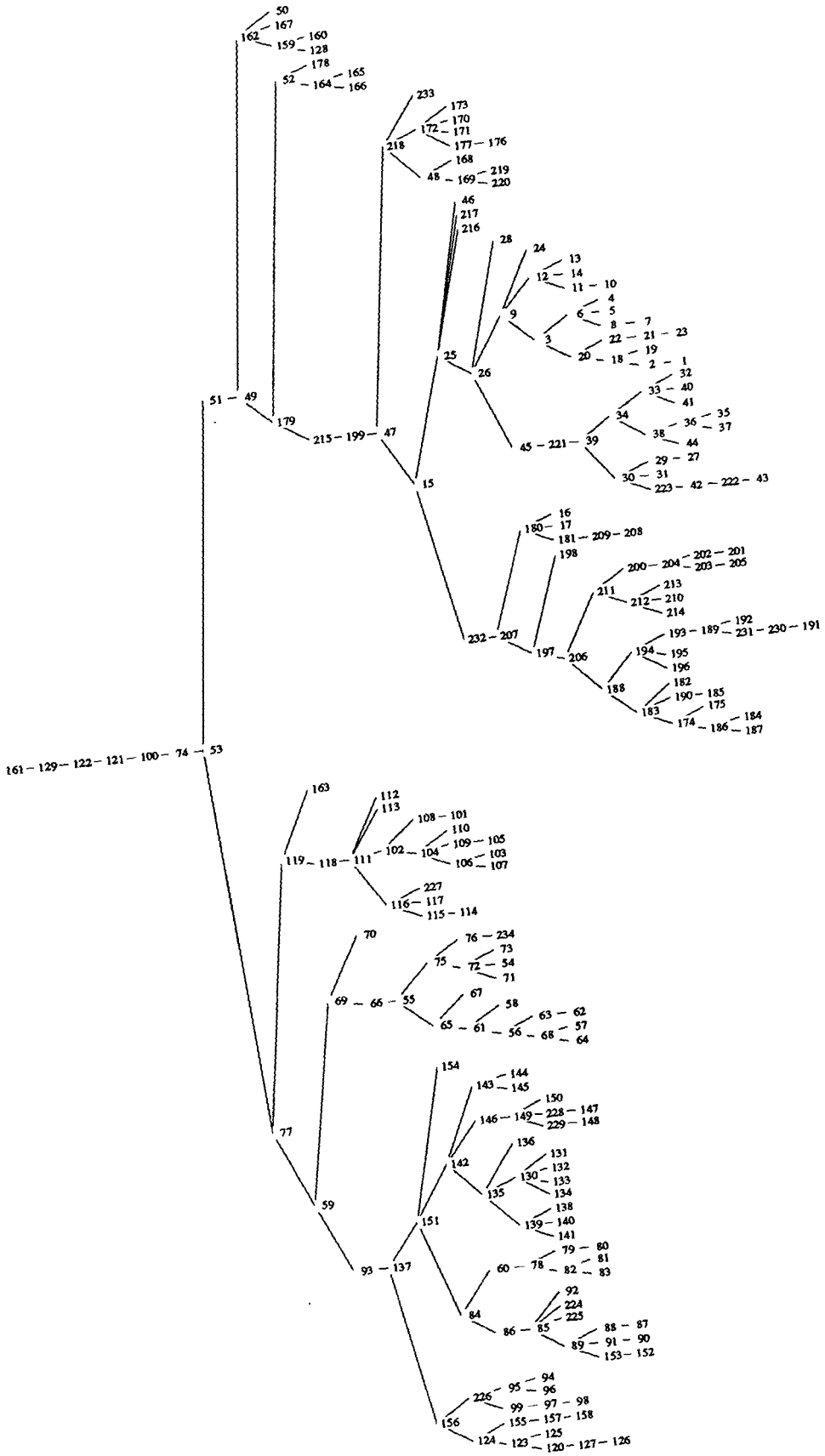


Figure B.18 Path for 234 node network, MDLRU ordering.

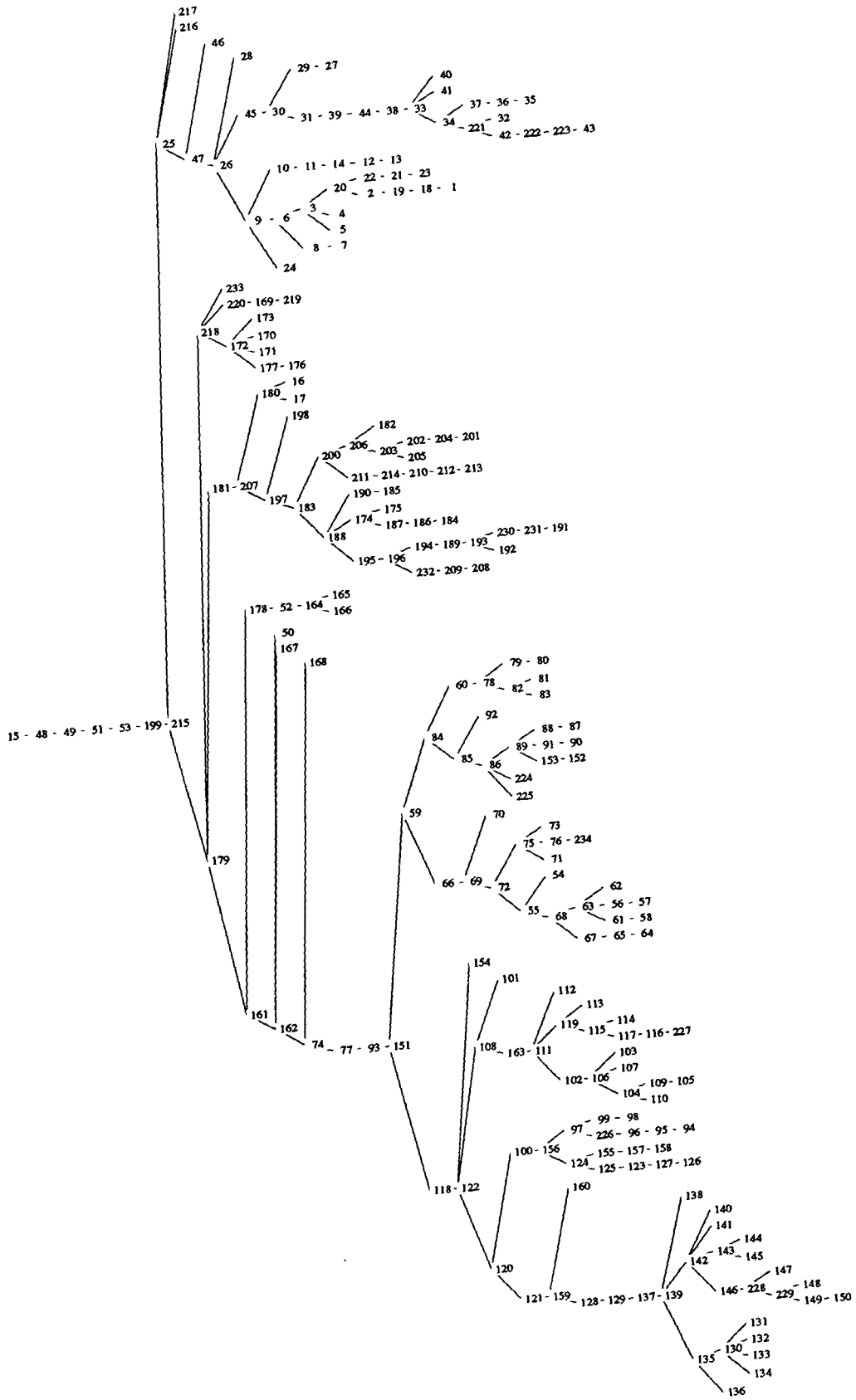


Figure B.19 Path for 234 node network, MDLRUR ordering.

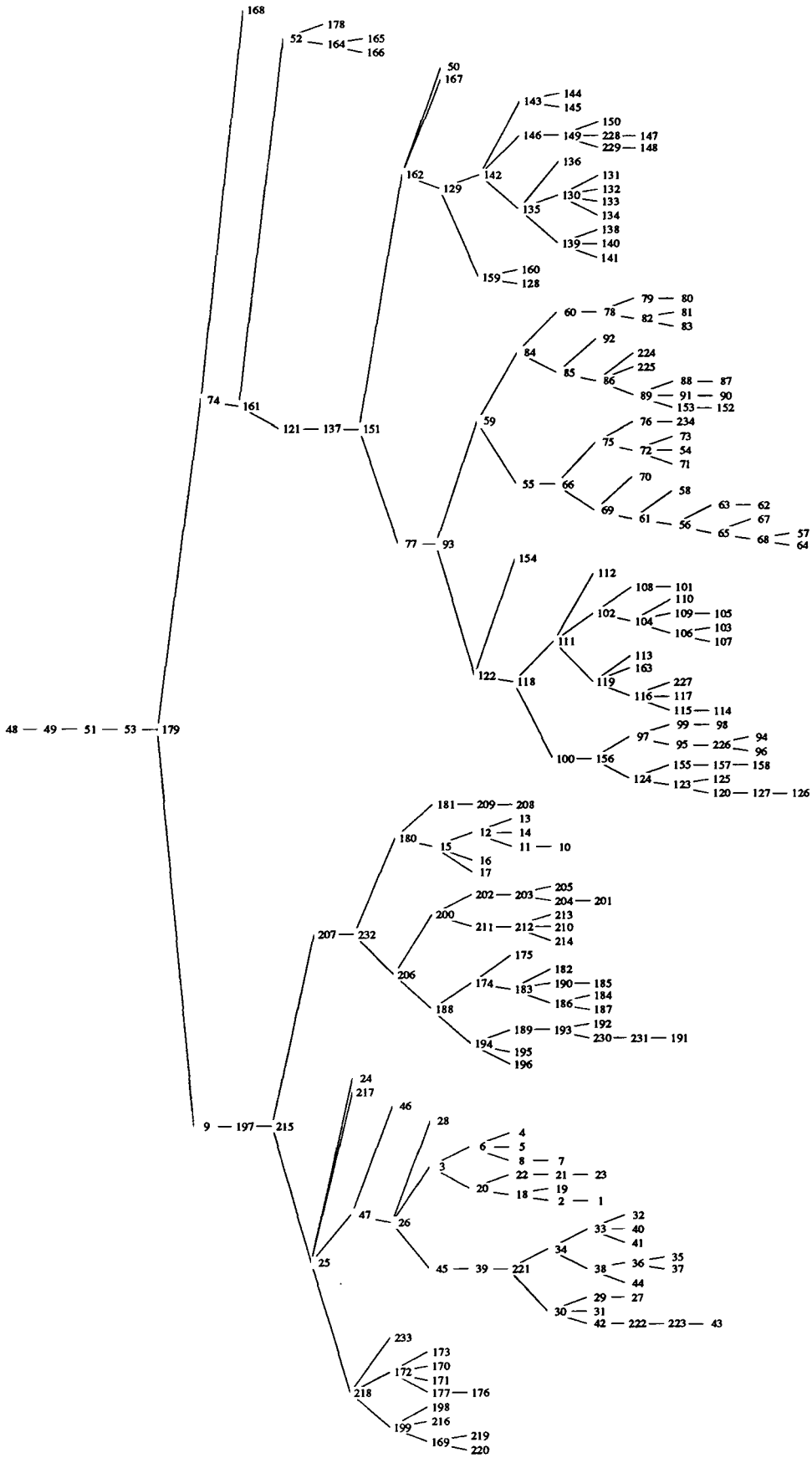


Figure B.20 Path for 234 node network, MDLRUML ordering.

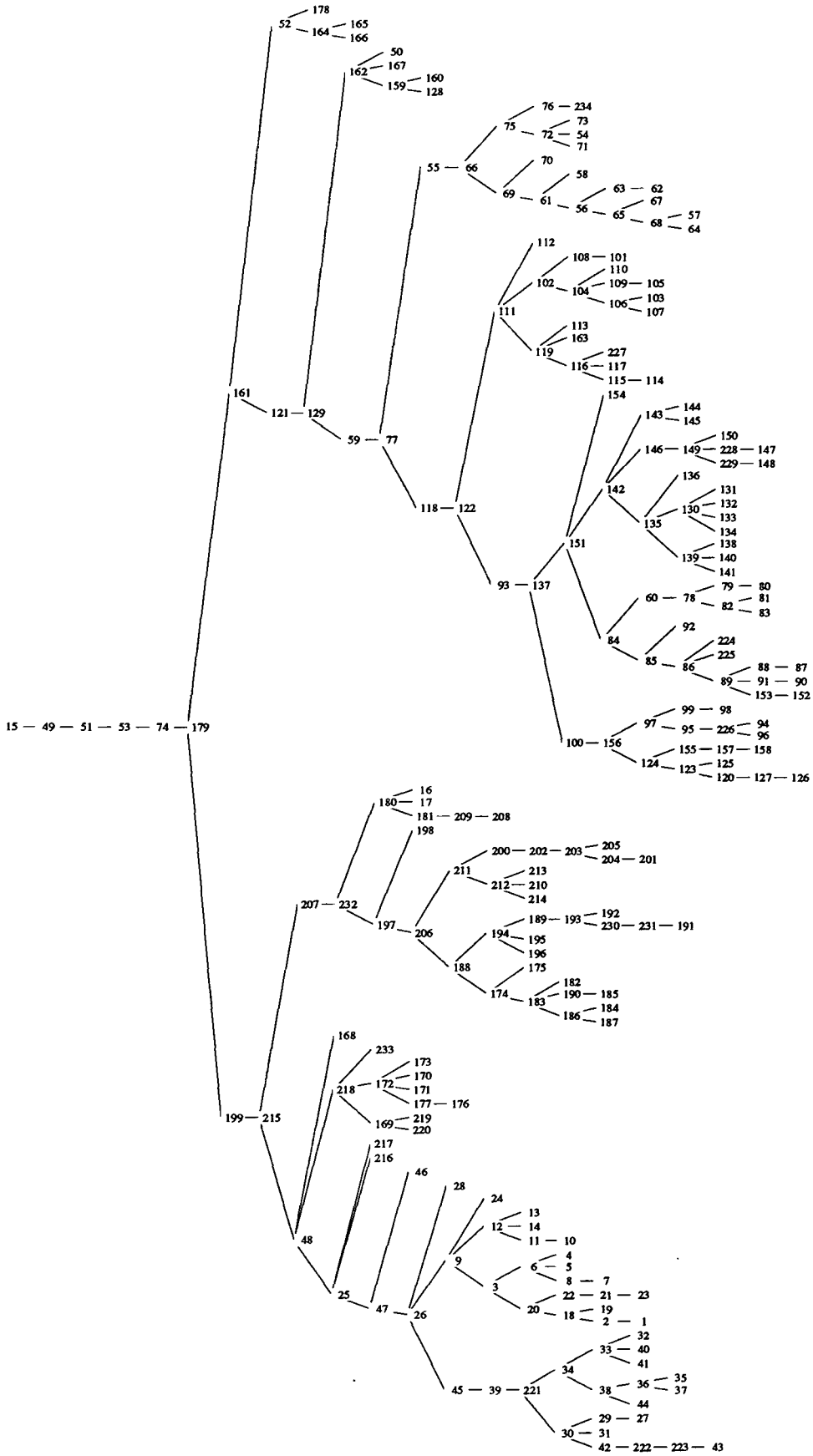


Figure B.21 Path for 234 node network, MDLRURA ordering.

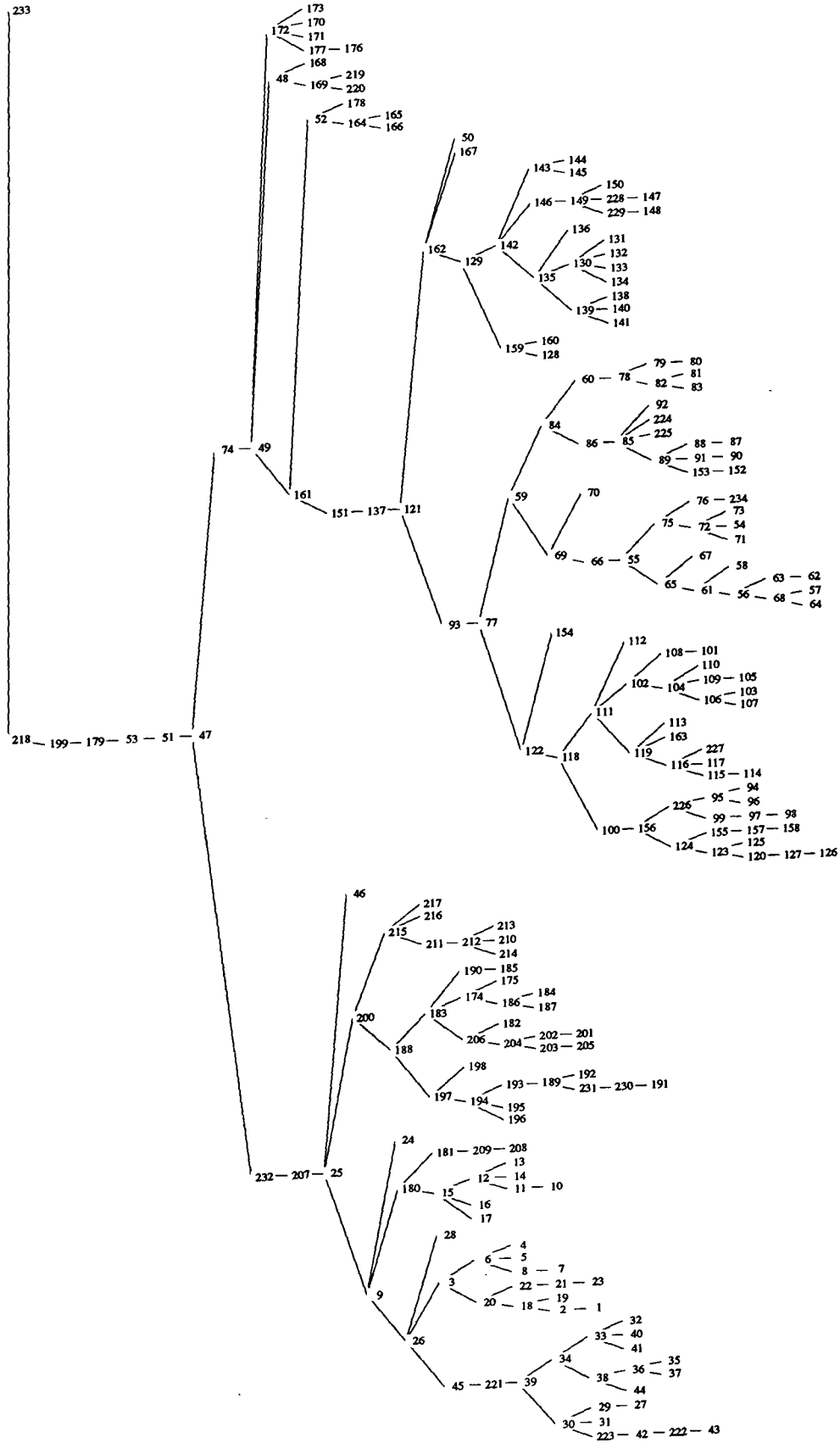
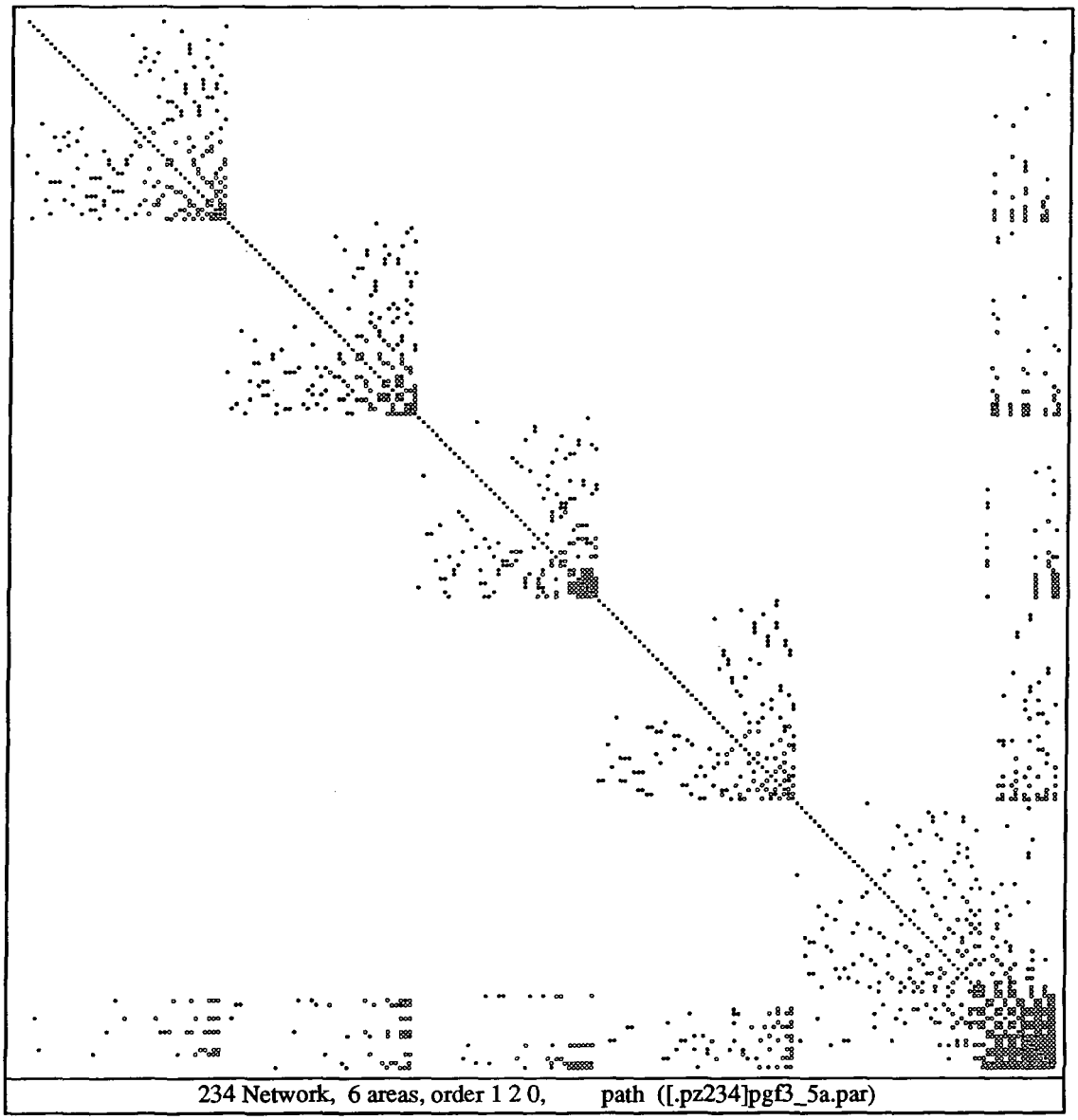


Figure B.22 Path for 234 node network, MDLRUMLA ordering.

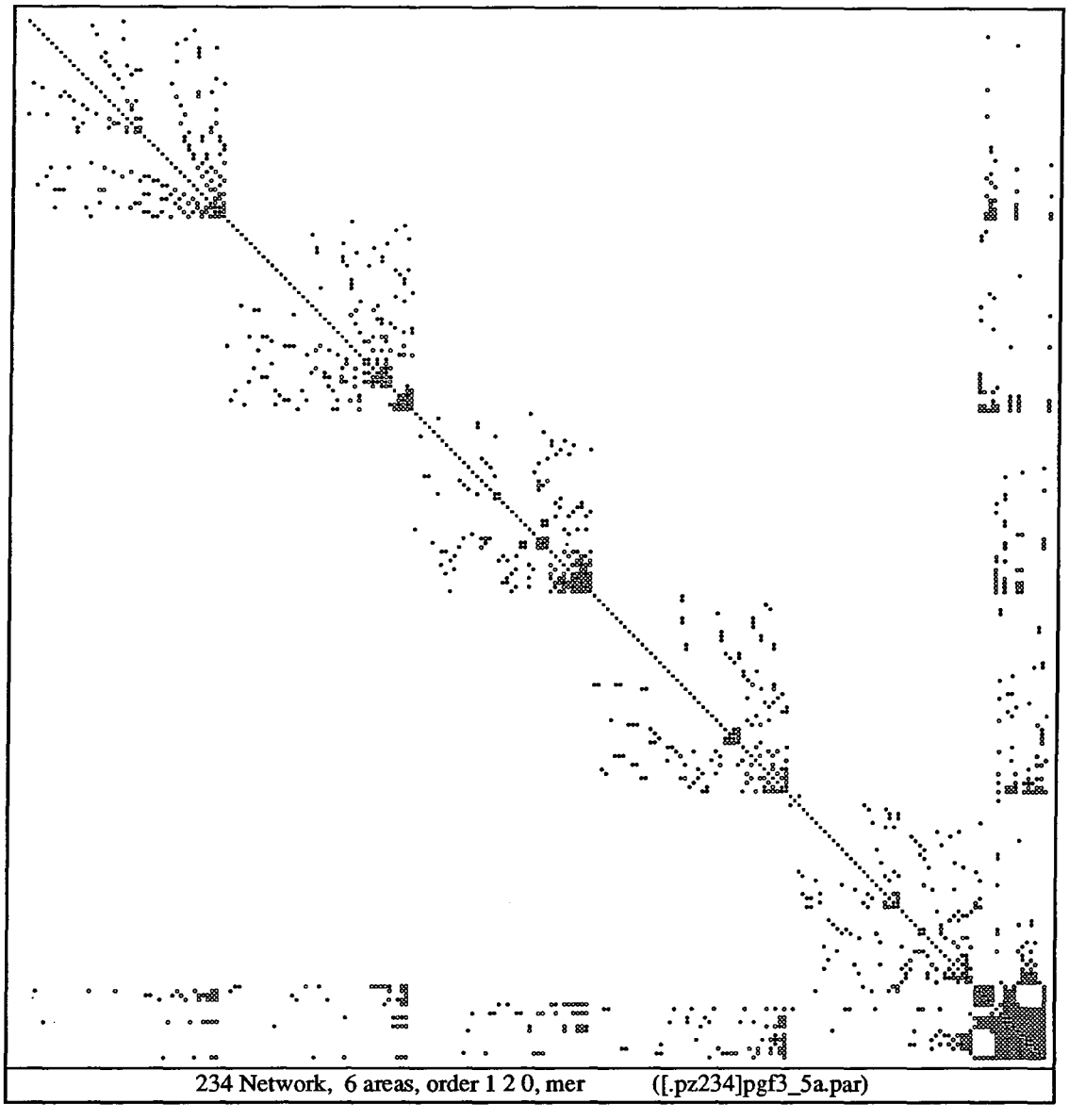
Appendix C.

Optimised Splits



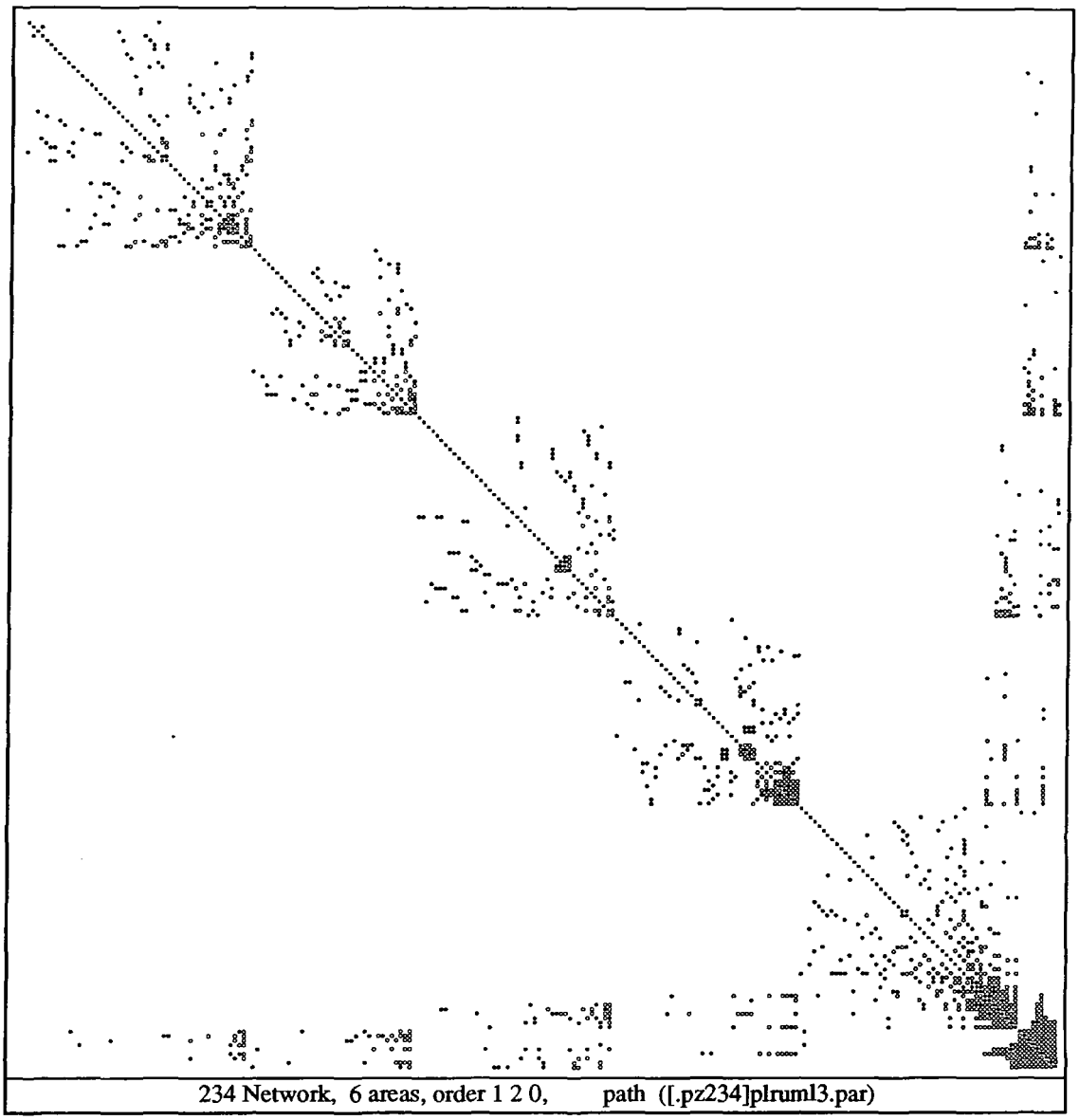

```
Ordering took                  80 ms
Pointer alteration took        10 ms
Reduction took                 40 ms
Solution took                  20 ms      160 ms
Calculation took              190 ms      350 ms
Elements:  before | after        928      956     375   108.1%
Multiplications generated       3714     1678
Additions  generated            2992     1444
Divisions  generated             234        0
Multiply-additions generated    3714     1678   37828     0.199MFlops
                                       previous          current
Muladd counts for area  1 are      0     0          520   293
Muladd counts for area  2 are      0     0          658   305
Muladd counts for area  3 are      0     0          550   277
Muladd counts for area  4 are      0     0          632   307
Muladd counts for area  5 are      0     0          510   279
Muladd counts for totals are     844   217         2870  1461
Muladd count totals are                            3714  1678
Counts required=   1871   (4406)     42.5%       2.35
```

Figure C.1 Matrix: GF–3, 5 Areas, 234 nodes, path.



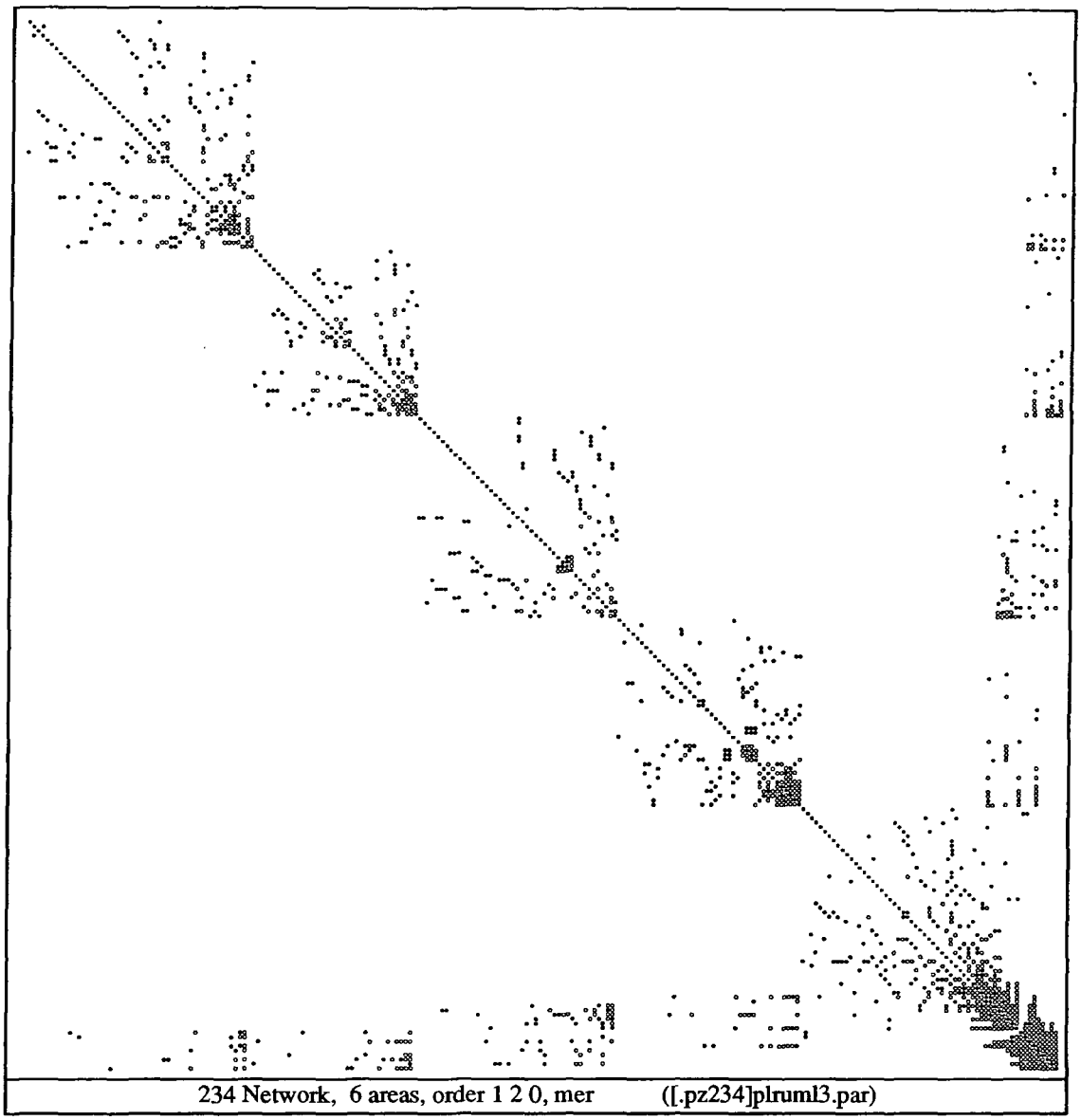

```
Ordering took                120 ms
Pointer alteration took       20 ms
Reduction took                30 ms
Solution took                 20 ms      190 ms
Calculation took             170 ms      360 ms
Elements:  before | after         928     916     335    96.5%
Multiplications generated        3328    1598
Additions  generated             2646    1364
Divisions  generated              234       0
Multiply-additions generated     3328    1598   34420     0.202MFlops
                                 previous          current          next
Muladd counts for area  0 are         0    0        462  277         0    0
Muladd counts for area  1 are         0    0        550  283         0    0
Muladd counts for area  2 are         0    0        504  269         0    0
Muladd counts for area  3 are         0    0        590  297         0    0
Muladd counts for area  4 are         0    0        446  263         0    0
Muladd counts for area  5 are       776  209          0    0         0    0
Muladd counts for totals are        776  209       2552 1389         0    0
Muladd count totals are                            3328 1598
Counts required=    1723   (4406)     39.1%      2.56
```

Figure C.2 Matrix: GF–3, 5 Areas, 234 nodes.



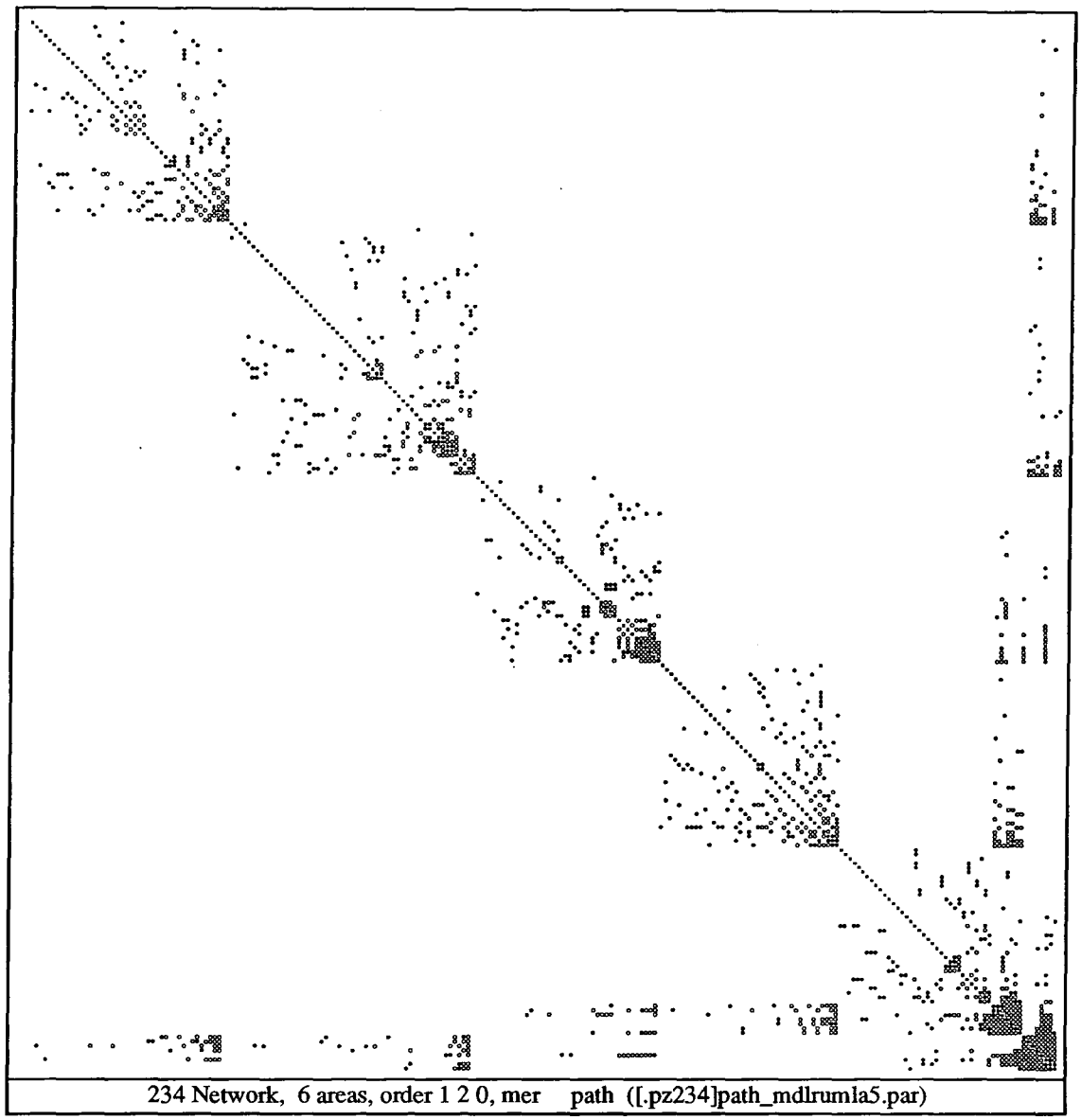

```
Ordering took                  80 ms
Pointer alteration took        20 ms
Reduction took                 30 ms
Solution took                  20 ms      150 ms
Calculation took              160 ms      310 ms
Elements:  before | after        928     892     311    89.6%
Multiplications generated       2968    1550
Additions  generated            2310    1316
Divisions  generated             234       0
Multiply-additions generated    2968    1550   31348     0.196MFlops
                                      previous        current
Muladd counts for area  1 are          0    0         438  291
Muladd counts for area  2 are          0    0         386  229
Muladd counts for area  3 are          0    0         450  271
Muladd counts for area  4 are          0    0         516  276
Muladd counts for area  5 are          0    0         584  296
Muladd counts for totals are         594  187        2374 1363
Muladd count totals are                             2968 1550
Counts required=   1513  (4406)     34.3%          2.91
```

Figure C.3 Matrix: MDLRUML, 5 Areas, 234 nodes, path.



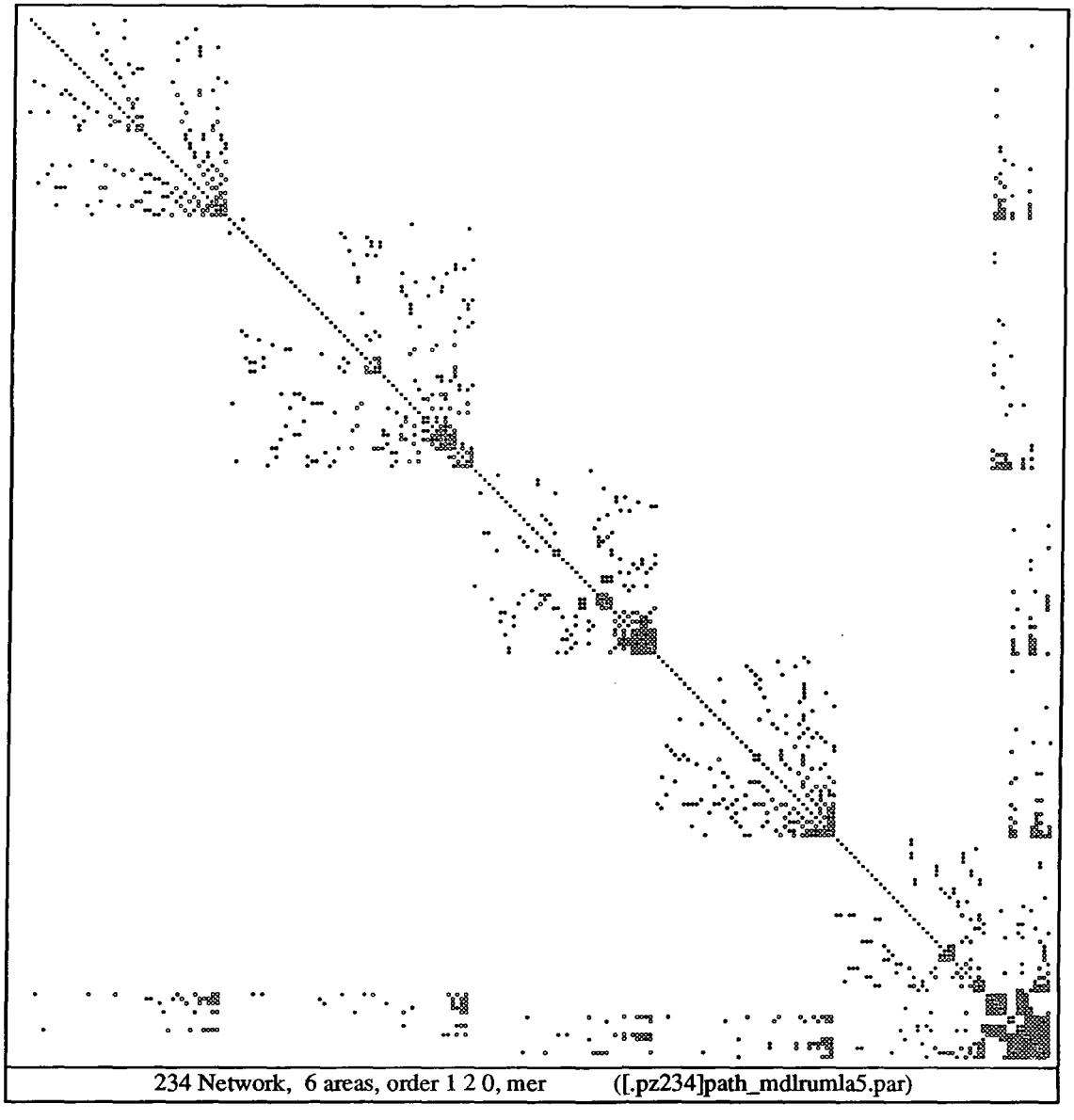

```
Ordering took                110 ms
Pointer alteration took       10 ms
Reduction took                30 ms
Solution took                 20 ms      170 ms
Calculation took             160 ms      330 ms
Elements:  before | after        928     893     312    89.9%
Multiplications generated       2980    1552
Additions  generated            2321    1318
Divisions  generated             234       0
Multiply-additions generated    2980    1552   31452     0.197MFlops
                                      previous          current          next
Muladd counts for area  0 are          0     0          438  291          0    0
Muladd counts for area  1 are          0     0          386  229          0    0
Muladd counts for area  2 are          0     0          450  271          0    0
Muladd counts for area  3 are          0     0          516  276          0    0
Muladd counts for area  4 are          0     0          596  298          0    0
Muladd counts for area  5 are        594   187            0    0          0    0
Muladd counts for totals are         594   187         2386 1365          0    0
Muladd count totals are                               2980 1552
Counts required=   1526  (4406)     34.6%     2.89
```

Figure C.4 Matrix: MDLRUML, 5 Areas, 234 nodes.



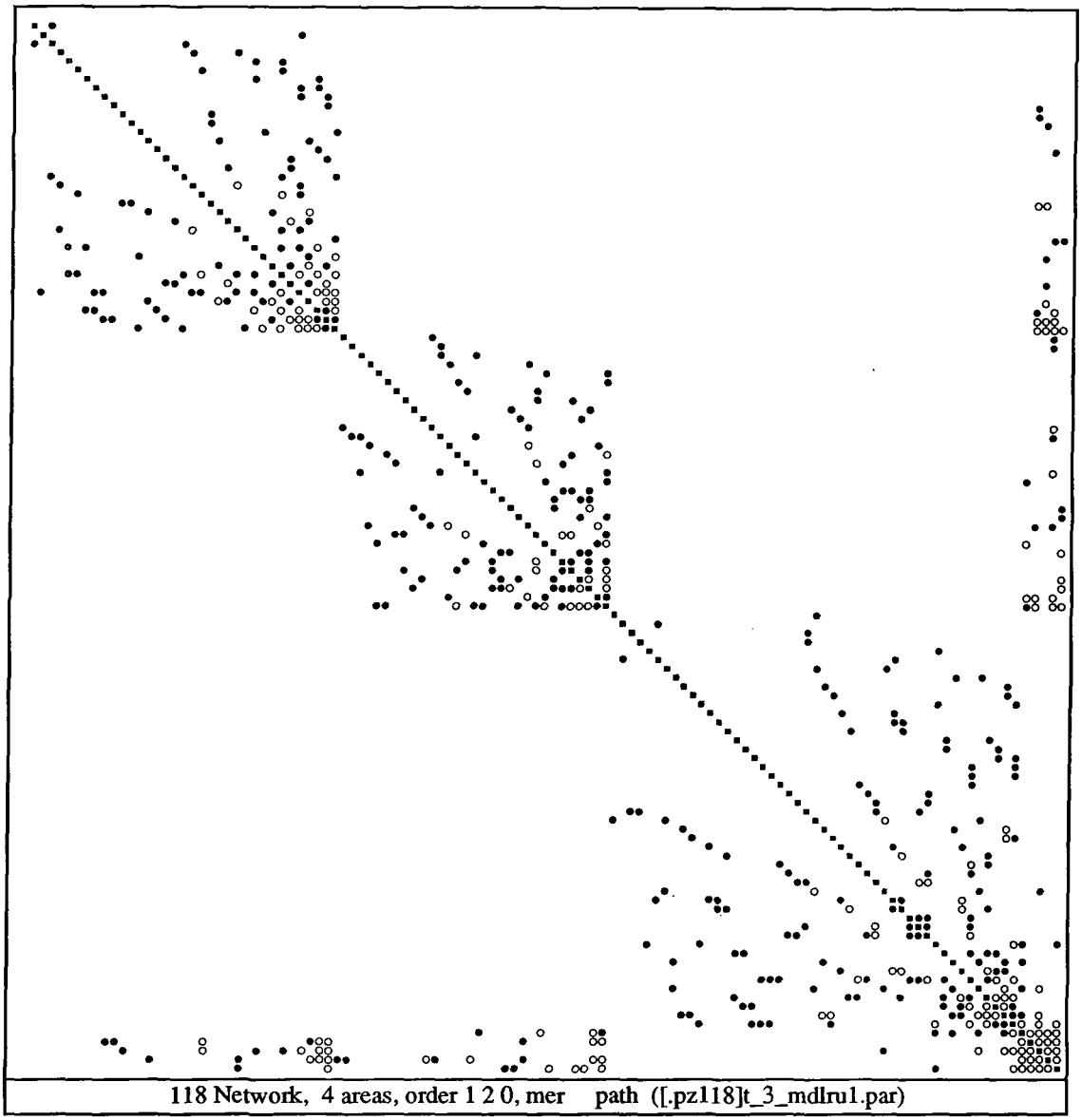

```
Ordering took                    90 ms
Pointer alteration took          20 ms
Reduction took                   30 ms
Solution took                    20 ms      160 ms
Calculation took                160 ms      320 ms
Elements:  before | after          928     887     306    88.2%
Multiplications generated         2906    1540
Additions  generated              2253    1306
Divisions  generated               234       0
Multiply-additions generated      2906    1540   30812     0.193MFlops
                                  previous          current
Muladd counts for area  1 are        0     0        462  277
Muladd counts for area  2 are        0     0        552  334
Muladd counts for area  3 are        0     0        516  276
Muladd counts for area  4 are        0     0        570  289
Muladd counts for area  5 are        0     0        352  211
Muladd counts for totals are       454   153       2452 1387
Muladd count totals are                            2906 1540
Counts required=   1326  (4406)     30.1%      3.32
```

Figure C.5 Matrix: MDLRUMLA, 5 Areas, 234 nodes, path.



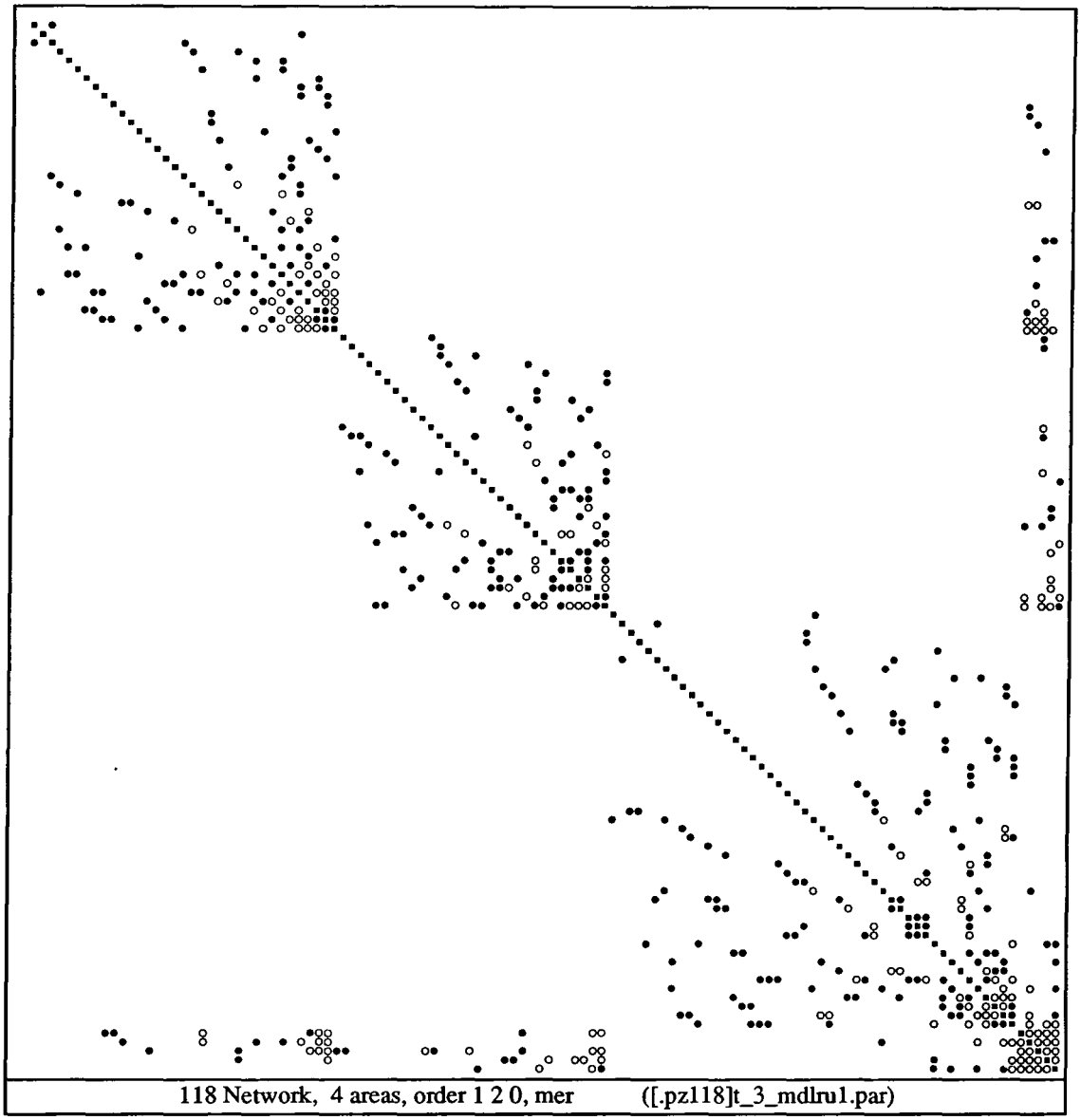

```
Ordering took                120 ms
Pointer alteration took       10 ms
Reduction took                30 ms
Solution took                 10 ms      180 ms
Calculation took             160 ms      340 ms
Elements:  before | after          928     889     308    88.8%
Multiplications generated         2926    1544
Additions  generated              2271    1310
Divisions  generated               234       0
Multiply-additions generated      2926    1544   30988     0.194MFlops
                                  previous          current          next
Muladd counts for area  0 are          0     0       462  277         0    0
Muladd counts for area  1 are          0     0       552  334         0    0
Muladd counts for area  2 are          0     0       516  276         0    0
Muladd counts for area  3 are          0     0       590  293         0    0
Muladd counts for area  4 are          0     0       352  211         0    0
Muladd counts for area  5 are        454   153         0    0         0    0
Muladd counts for totals are         454   153      2472 1391         0    0
Muladd count totals are                             2926 1544
Counts required=    1343   (4406)     30.5%      3.28
```

Figure C.6 Matrix: MDLRUMLA, 5 Areas, 234 nodes.



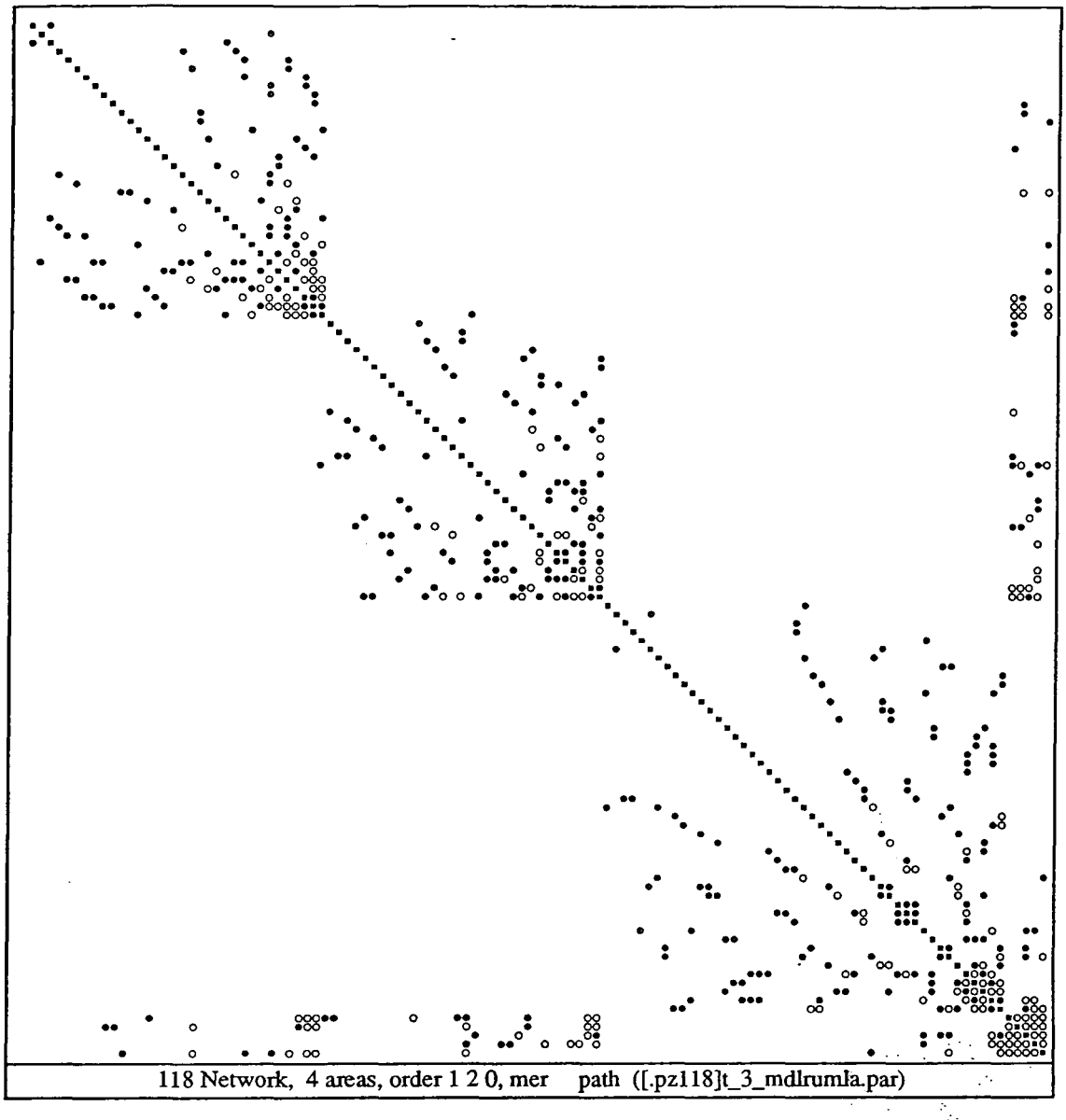

```
Ordering took                  50 ms
Pointer alteration took        00 ms
Reduction took                 10 ms
Solution took                  20 ms       90 ms
Calculation took               60 ms      150 ms
Elements:  before | after         476     383      86    48.0%
Multiplications generated         922     648
Additions  generated              657     530
Divisions  generated              118       0
Multiply-additions generated      922     648   10676     0.178MFlops
                                 previous          current
Muladd counts for area  1 are         0     0       288   197
Muladd counts for area  2 are         0     0       280   185
Muladd counts for area  3 are         0     0       314   241
Muladd counts for totals are         40    25       882   623
Muladd count totals are                             922   648
Counts required=     579   (1570)     36.9%      2.71
```

Figure C.7 Matrix: MDLRU, 3 Areas, 118 nodes, path.



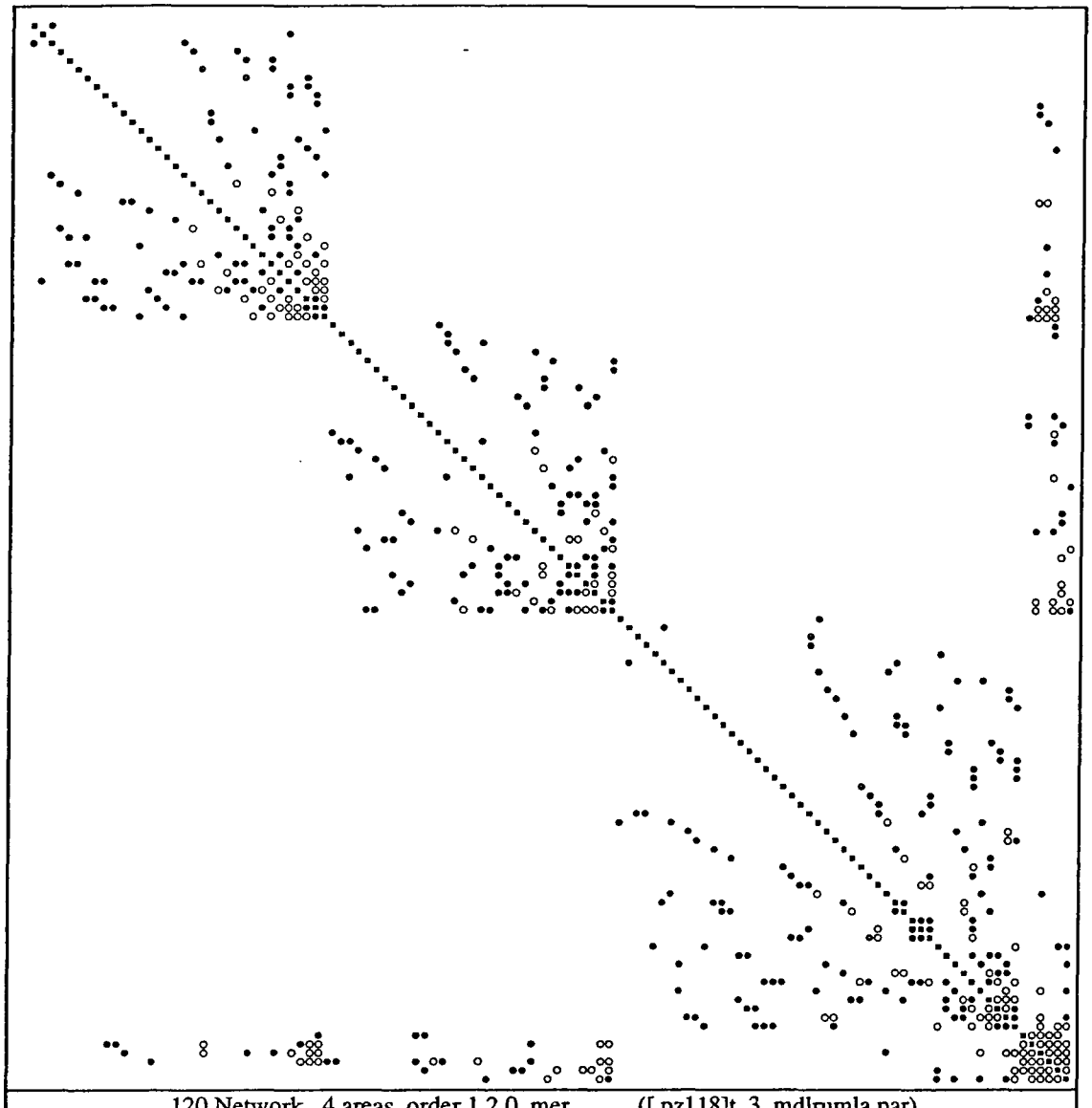

```
Ordering took                  70 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms      100 ms
Calculation took               60 ms      160 ms
Elements:  before | after          476     383      86    48.0%
Multiplications generated          922     648
Additions  generated               657     530
Divisions  generated               118       0
Multiply-additions generated       922     648   10676     0.178MFlops
                                  previous          current           next
Muladd counts for area  0 are        0     0        288   197          0     0
Muladd counts for area  1 are        0     0        280   185          0     0
Muladd counts for area  2 are        0     0        314   241          0     0
Muladd counts for area  3 are       40    25          0     0          0     0
Muladd counts for totals are        40    25        882   623          0     0
Muladd count totals are                           922   648
Counts required=    579  (1570)     36.9%      2.71
```

Figure C.8 Matrix: MDLRU, 3 Areas, 118 nodes.




```
Ordering took                60 ms
Pointer alteration took      00 ms
Reduction took               10 ms
Solution took                10 ms      100 ms
Calculation took             60 ms      160 ms
Elements:  before | after         476     384      87    48.6%
Multiplications generated         930     650
Additions  generated              664     532
Divisions  generated              118       0
Multiply-additions generated      930     650   10748     0.179MFlops
                                 previous          current
Muladd counts for area  1 are         0     0       276   190
Muladd counts for area  2 are         0     0       300   194
Muladd counts for area  3 are         0     0       314   241
Muladd counts for totals are         40    25       890   625
Muladd count totals are                             930   650
Counts required=     579   (1570)     36.9%      2.71
```

Figure C.9 Matrix: MDLRUMLA, 3 Areas, 118 nodes, path.




```
Ordering took                  70 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms      100 ms
Calculation took               60 ms      160 ms
Elements:  before | after         482     390      89    49.2%
Multiplications generated         942     660
Additions  generated              672     540
Divisions  generated              120       0
Multiply-additions generated      942     660   10896     0.182MFlops
                                previous          current          next
Muladd counts for area  0 are       0     0       276  190          0     0
Muladd counts for area  1 are      20     9       292  195          0     0
Muladd counts for area  2 are       0     0       314  241          0     0
Muladd counts for area  3 are      40    25         0    0          0     0
Muladd counts for totals are       60    34       882  626          0     0
Muladd count totals are                           942  660
Counts required=    588  (1570)     37.5%      2.67
```

Figure C.10 Matrix: MDLRUMLA, 3 Areas, 118 nodes.

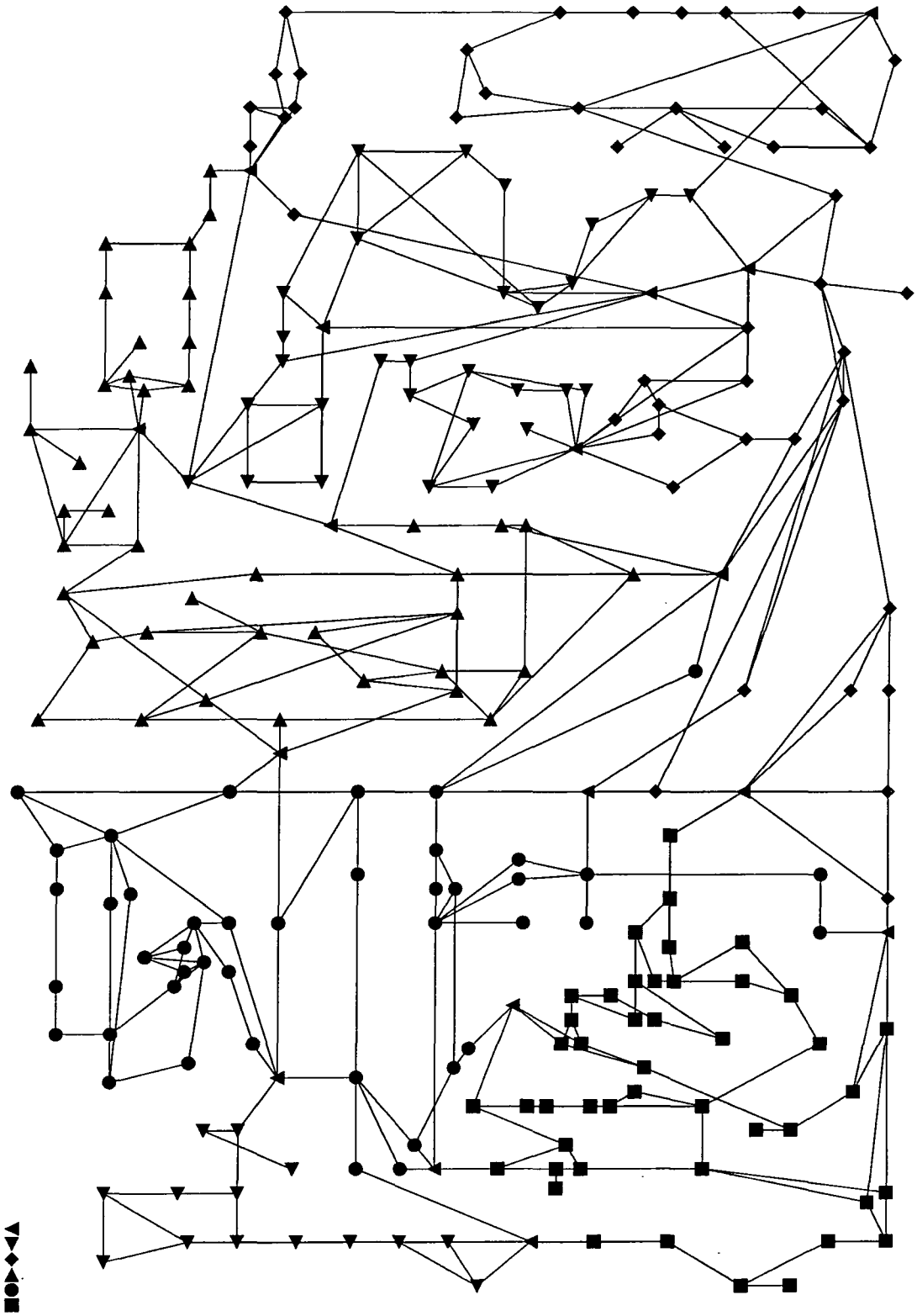


Figure C.11 Net: GF–3, 5 Areas, 234 nodes.

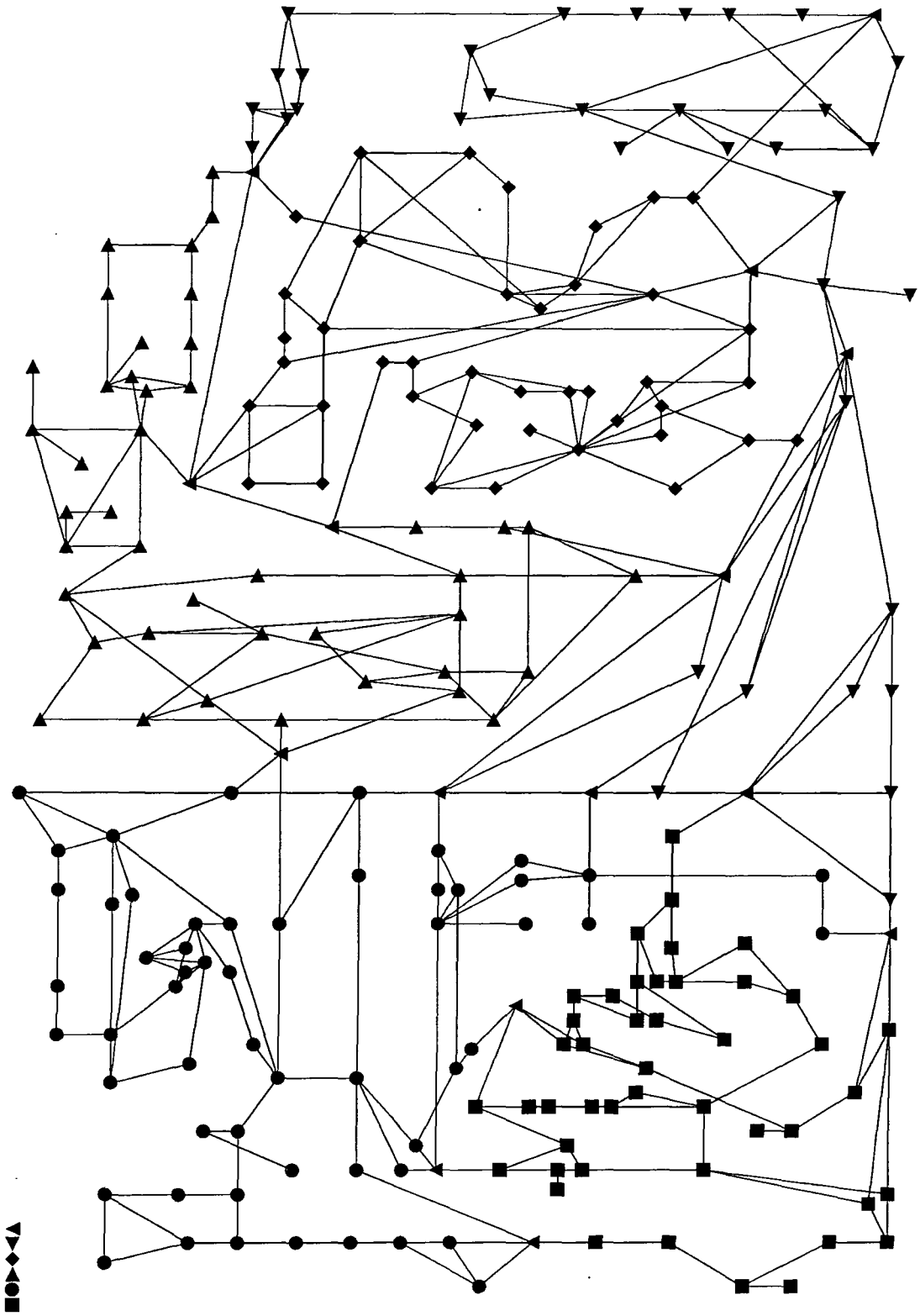


Figure C.12 Net: MDLRUML, 5 Areas, 234 nodes.

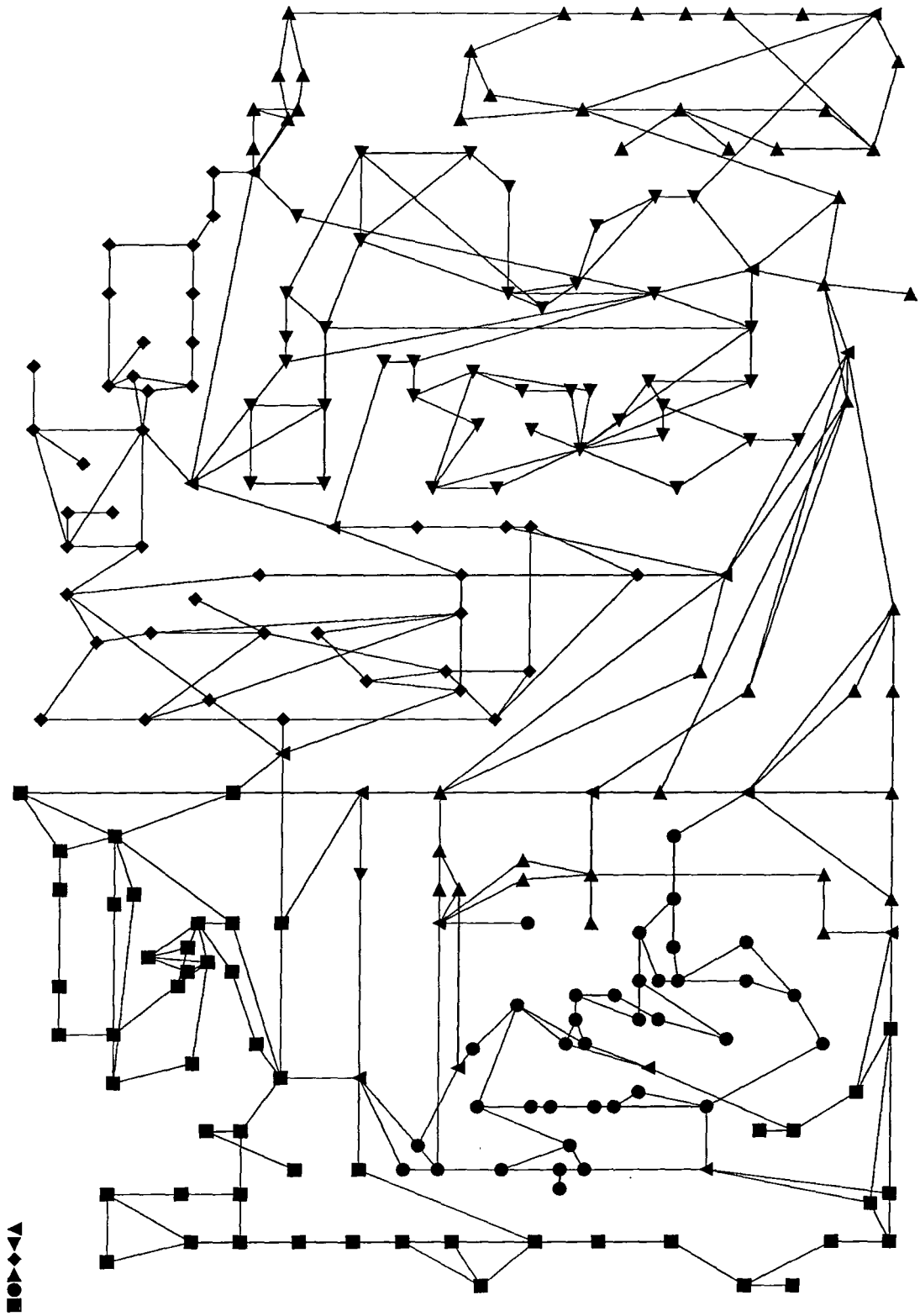


Figure C.13 Net: MDLRUMLA, 5 Areas, 234 nodes.

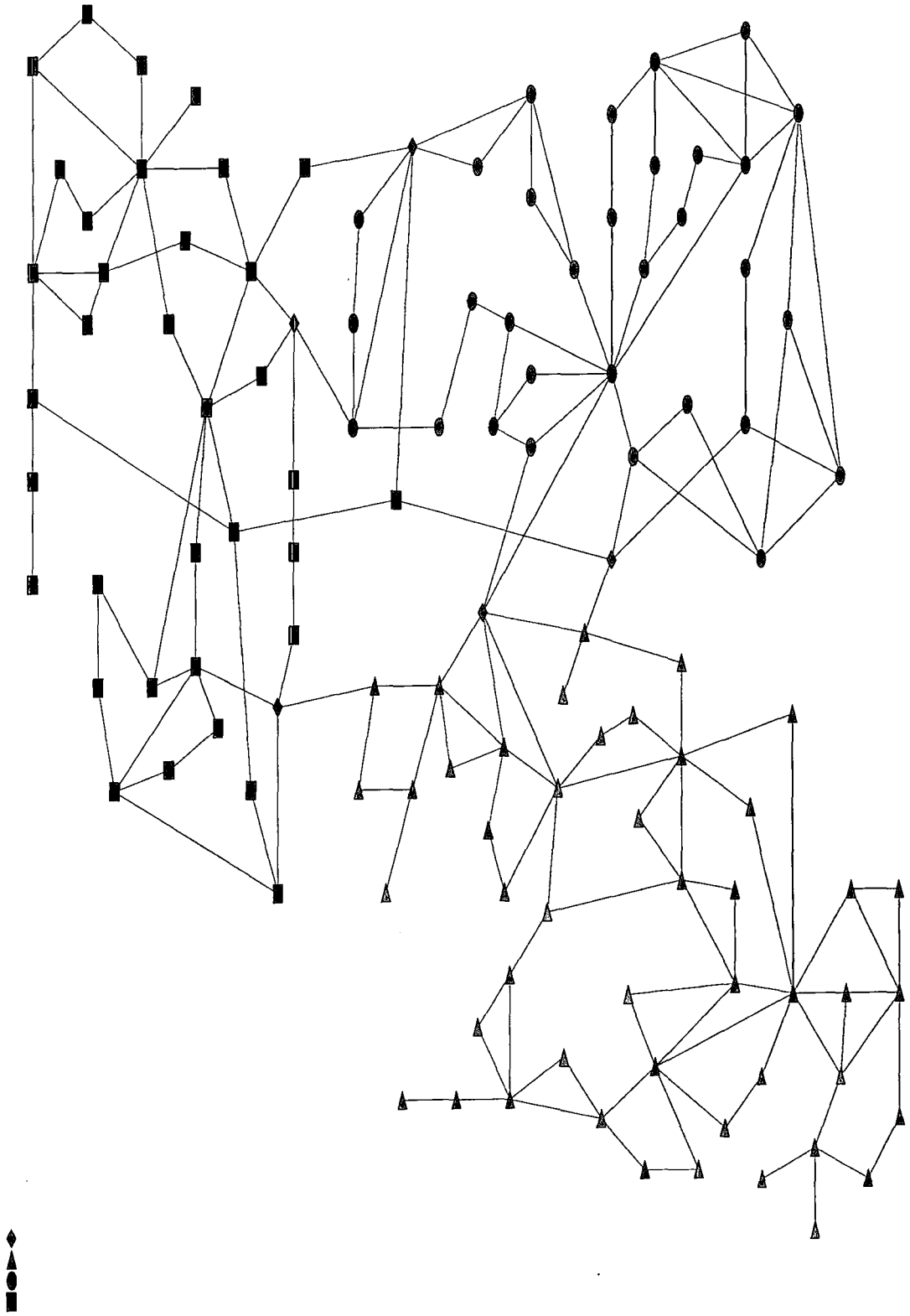


Figure C.14 Net: MDLRU, 3 Areas, 118 nodes.

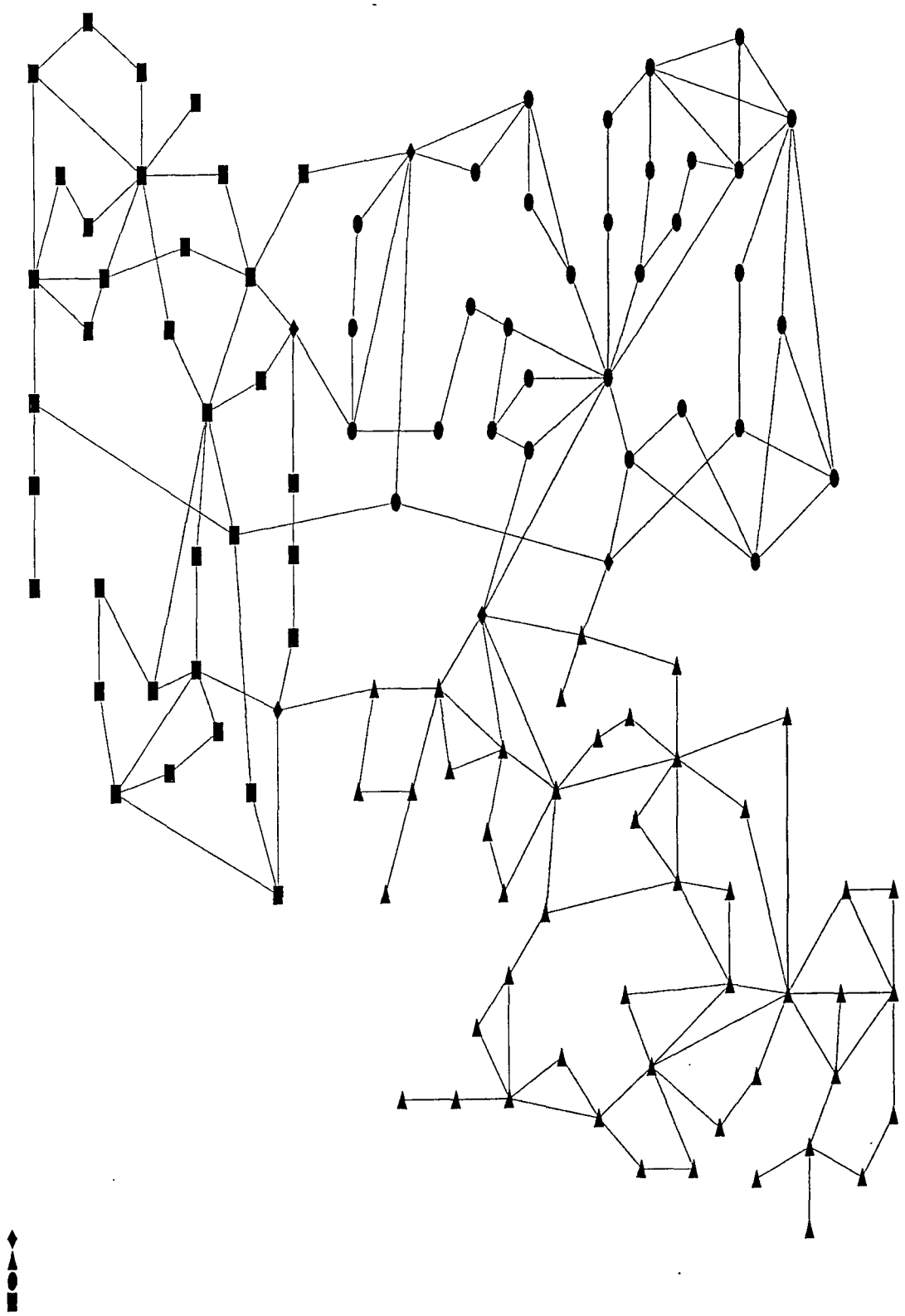
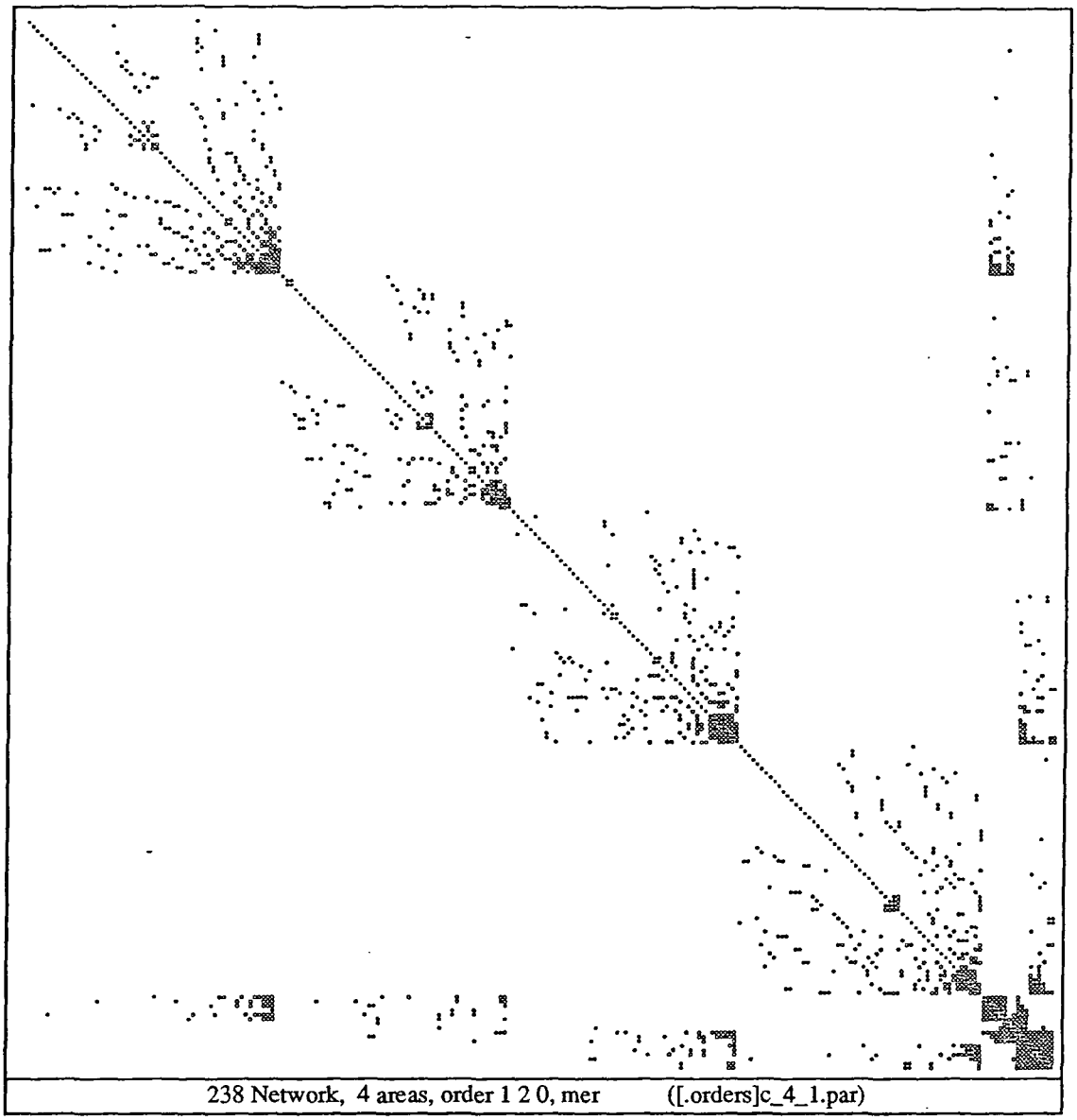


Figure C.15 Net: MDLRUMLA, 3 Areas, 118 nodes.

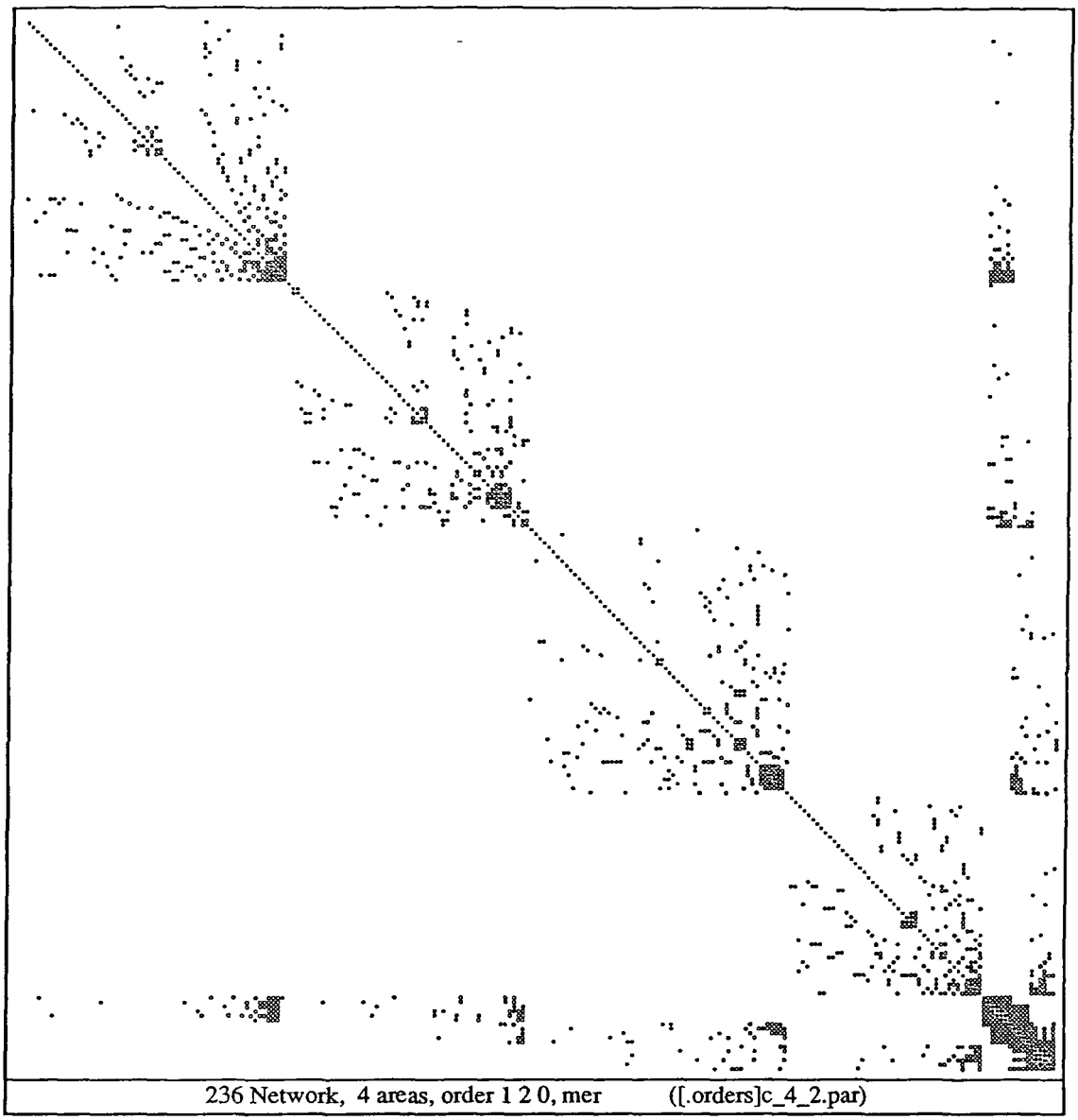
Appendix D.

Optimised Splits



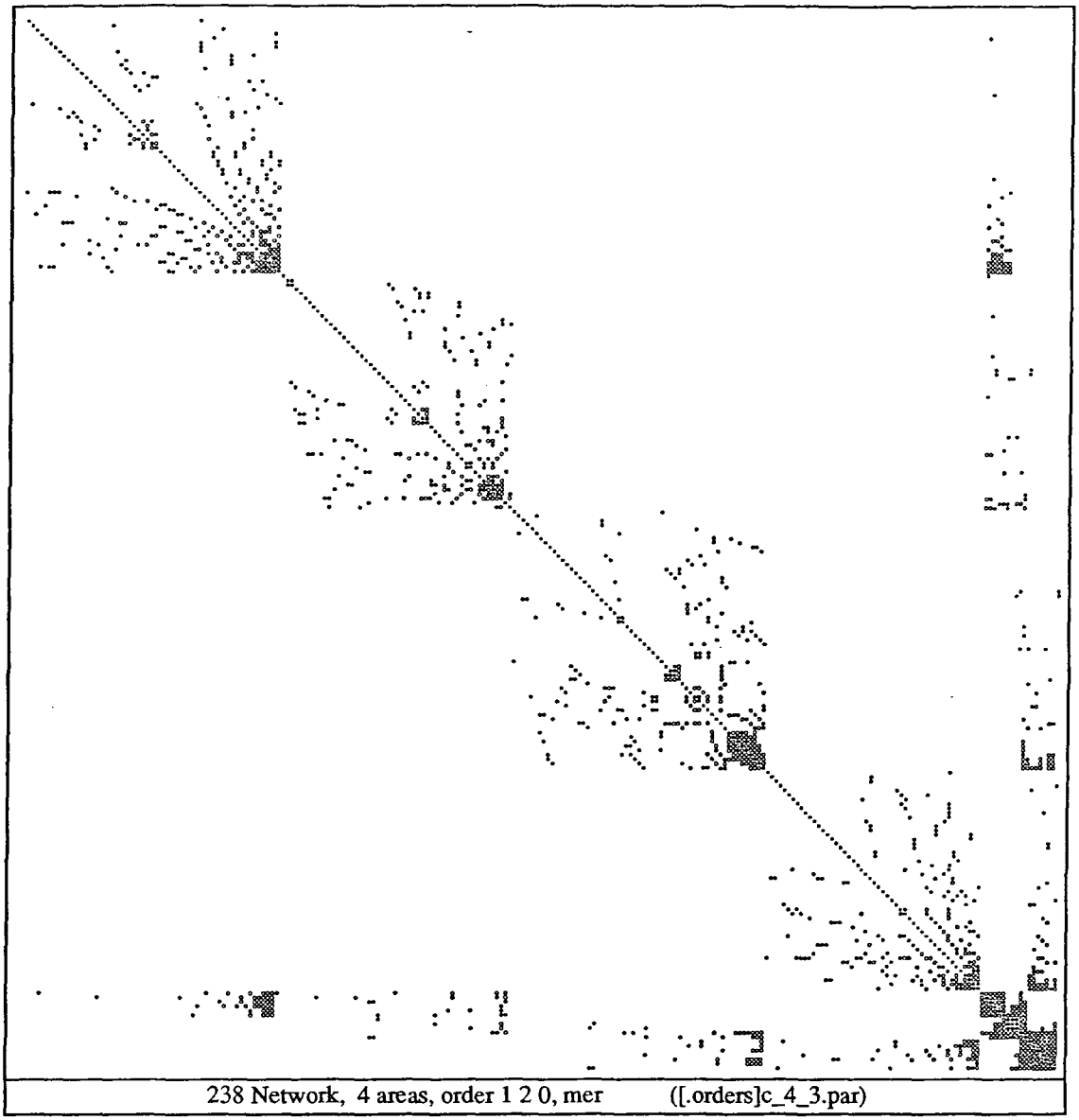

```
Ordering took                120 ms
Pointer alteration took       20 ms
Reduction took                20 ms
Solution took                 30 ms      190 ms
Calculation took             170 ms      360 ms
Elements:  before | after        940     911    322    91.7%
Multiplications generated       3030    1584
Additions  generated            2357    1346
Divisions  generated             238       0
Multiply-additions generated    3030    1584   32004     0.188MFlops
                                 previous          current          next
Muladd counts for area  0 are         0    0        760  392         0    0
Muladd counts for area  1 are       192   68        428  297         0    0
Muladd counts for area  2 are       210   63        724  363         0    0
Muladd counts for area  3 are        70   36        646  365         0    0
Muladd counts for totals are        472  167       2558 1417         0    0
Muladd count totals are                            3030 1584
Counts required=   1595  (4406)     36.2%      2.76
```

Figure D.1 Matrix: 4 Areas, 234 nodes, C–4–1a.



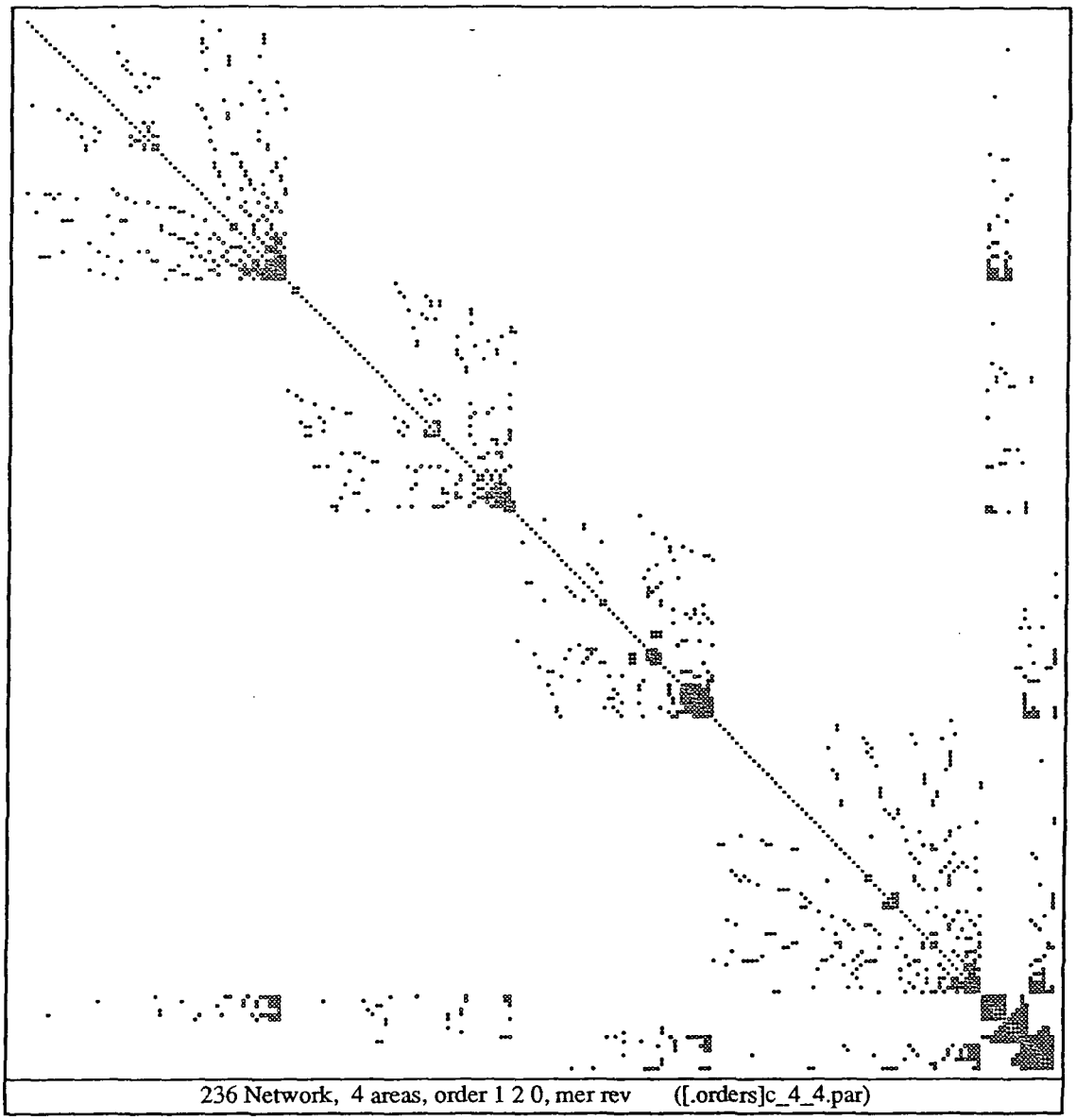

```
Ordering took                120 ms
Pointer alteration took       10 ms
Reduction took                30 ms
Solution took                 20 ms      180 ms
Calculation took             160 ms      350 ms
Elements:  before | after       934     909    324    92.8%
Multiplications generated      3092    1582
Additions  generated           2419    1346
Divisions  generated            236       0
Multiply-additions generated   3092    1582  32480     0.203MFlops
                                previous          current          next
Muladd counts for area  0 are        0    0        804  401        0    0
Muladd counts for area  1 are      272   80        578  337        0    0
Muladd counts for area  2 are      212   65        642  372        0    0
Muladd counts for area  3 are       70   36        514  291        0    0
Muladd counts for totals are       554  181       2538 1401        0    0
Muladd count totals are                          3092 1582
Counts required=   1739  (4406)    39.5%      2.53
```

Figure D.2 Matrix: 4 Areas, 234 nodes, C–4–2.



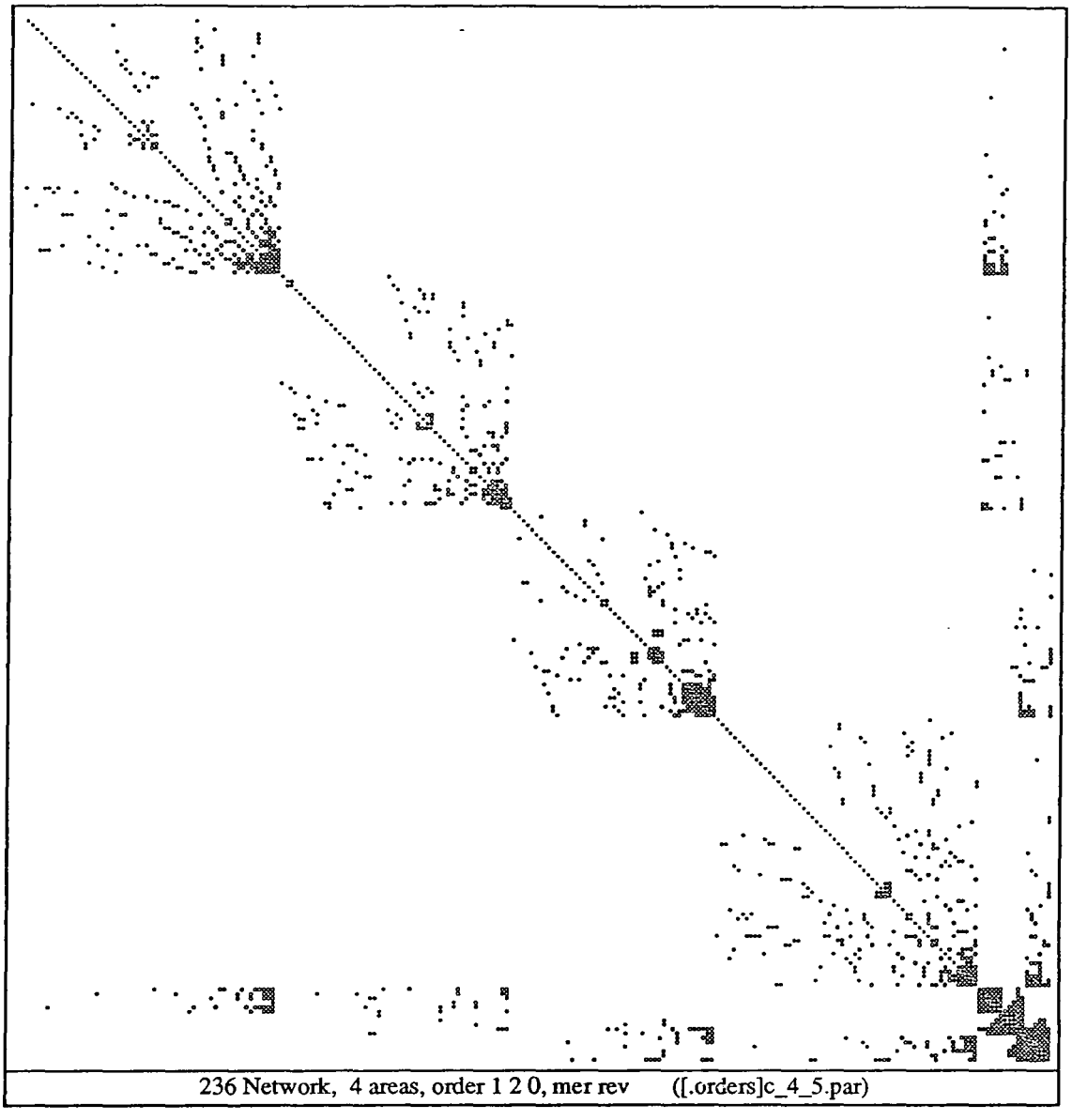

```
Ordering took                 110 ms
Pointer alteration took        20 ms
Reduction took                 30 ms
Solution took                  20 ms      180 ms
Calculation took              170 ms      350 ms
Elements:  before | after        940     921    332   94.6%
Multiplications generated       3152    1604
Additions  generated            2469    1366
Divisions  generated             238       0
Multiply-additions generated    3152    1604  33060    0.194MFlops
                                previous        current          next
Muladd counts for area  0 are        0    0      798  396        0    0
Muladd counts for area  1 are      204   70      452  303        0    0
Muladd counts for area  2 are      188   59      780  399        0    0
Muladd counts for area  3 are      112   49      618  328        0    0
Muladd counts for totals are       504  178     2648 1426        0    0
Muladd count totals are                        3152 1604
Counts required=   1678  (4406)     38.1%      2.63
```

Figure D.3 Matrix: 4 Areas, 234 nodes, C–4–3.



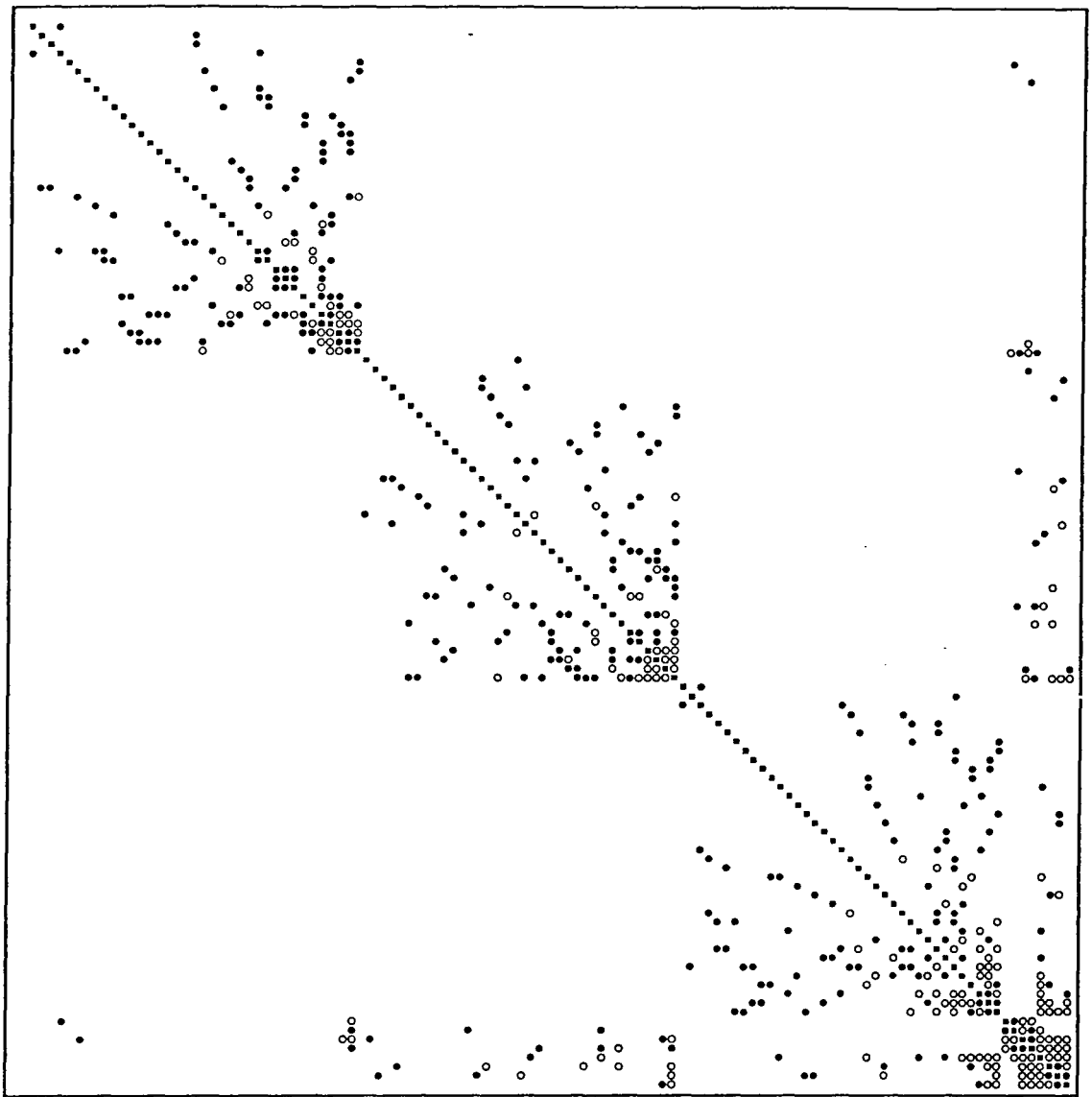

```
Ordering took                120 ms
Pointer alteration took       20 ms
Reduction took                30 ms
Solution took                 30 ms      200 ms
Calculation took             160 ms      360 ms
Elements:  before | after          934     897     312    89.4%
Multiplications generated         2918    1558
Additions  generated              2257    1322
Divisions  generated               236       0
Multiply-additions generated      2918    1558   30992     0.194MFlops
                                 previous          current          next
Muladd counts for area  0 are         0     0        772  399         0     0
Muladd counts for area  1 are       192    68        416  290         0     0
Muladd counts for area  2 are       178    59        580  306         0     0
Muladd counts for area  3 are        70    36        710  400         0     0
Muladd counts for totals are        440   163       2478 1395         0     0
Muladd count totals are                             2918 1558
Counts required=   1574  (4406)     35.7%      2.80
```

Figure D.4 Matrix: 4 Areas, 234 nodes, C–4–4.



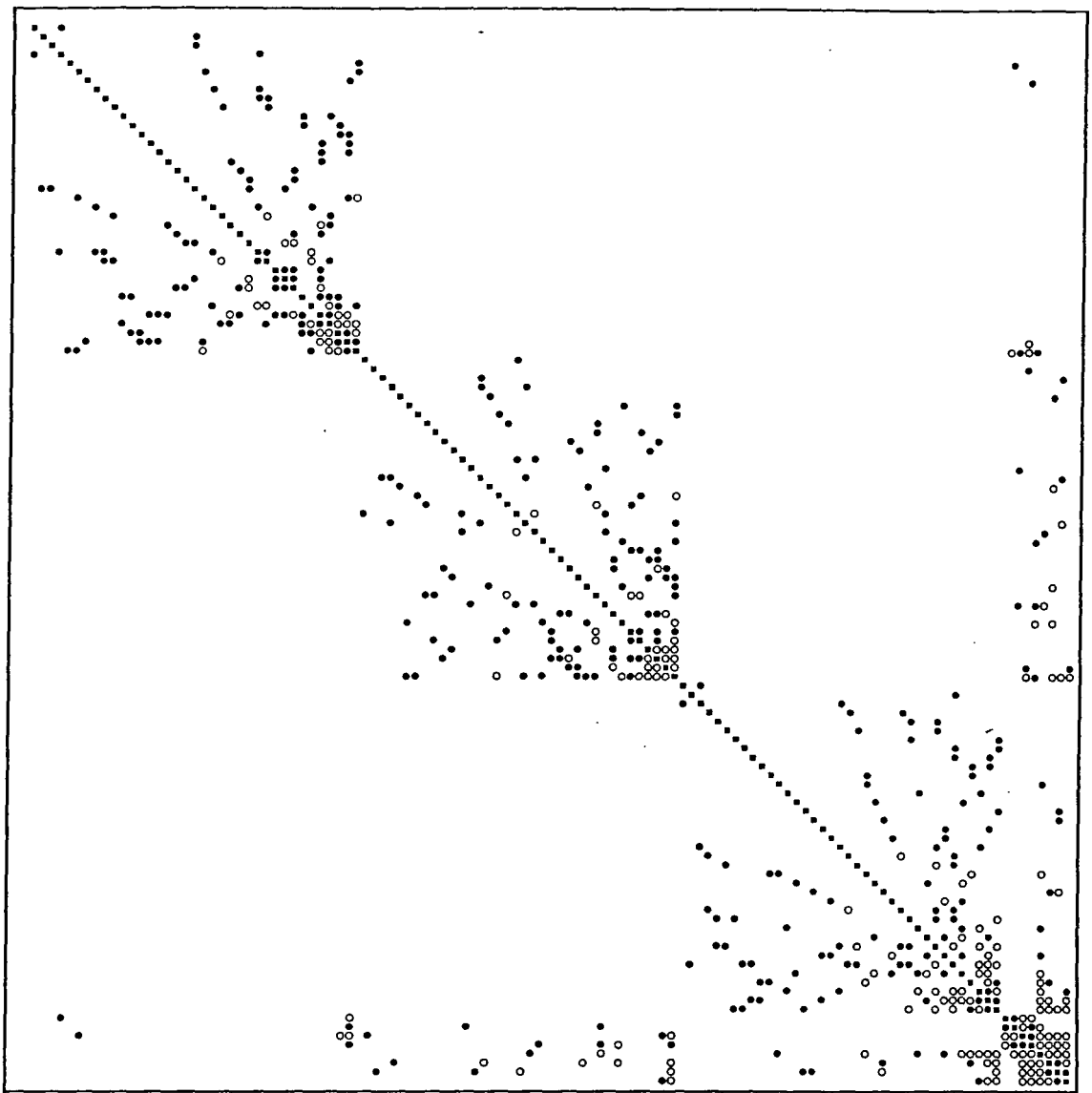

```
Ordering took                 110 ms
Pointer alteration took        10 ms
Reduction took                 30 ms
Solution took                  30 ms      180 ms
Calculation took              150 ms      330 ms
Elements:  before | after        934     895     310    88.8%
Multiplications generated       2904    1554
Additions  generated            2245    1318
Divisions  generated             236       0
Multiply-additions generated    2904    1554   30864     0.206MFlops
                                previous        current          next
Muladd counts for area  0 are        0    0       760  392        0    0
Muladd counts for area  1 are      192   68       428  297        0    0
Muladd counts for area  2 are      178   59       586  311        0    0
Muladd counts for area  3 are       70   36       690  391        0    0
Muladd counts for totals are       440  163      2464 1391        0    0
Muladd count totals are                         2904 1554
Counts required=   1559  (4406)    35.4%      2.83
```

Figure D.5 Matrix: 4 Areas, 234 nodes, C–4–5.



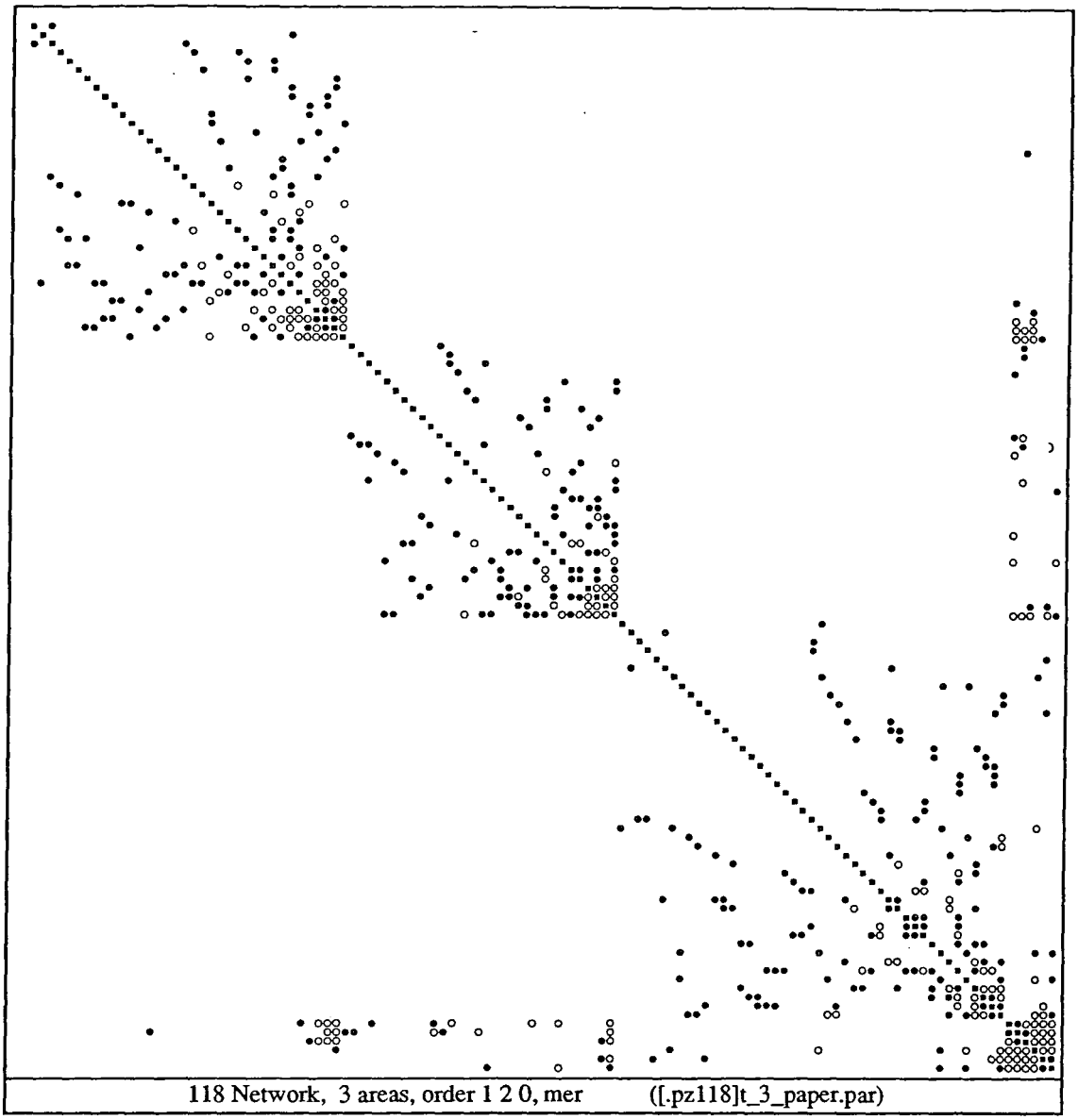

```
Ordering took                  80 ms
Pointer alteration took        00 ms
Reduction took                 10 ms
Solution took                  10 ms      100 ms
Calculation took               50 ms      160 ms
Elements:  before | after         476     387      90    50.3%
Multiplications generated         958     656
Additions  generated              689     538
Divisions  generated              118       0
Multiply-additions generated      958     656  10996     0.220MFlops
                                previous          current          next
Muladd counts for area  0 are        0    0        246  189         0    0
Muladd counts for area  1 are       74   34        310  208         0    0
Muladd counts for area  2 are       20   16        308  209         0    0
Muladd counts for totals are        94   50        864  606         0    0
Muladd count totals are                           958  656
Counts required=     568   (1570)     36.2%      2.76
```

Figure D.6 Matrix: 3 Areas, 118 nodes, T–3–1.



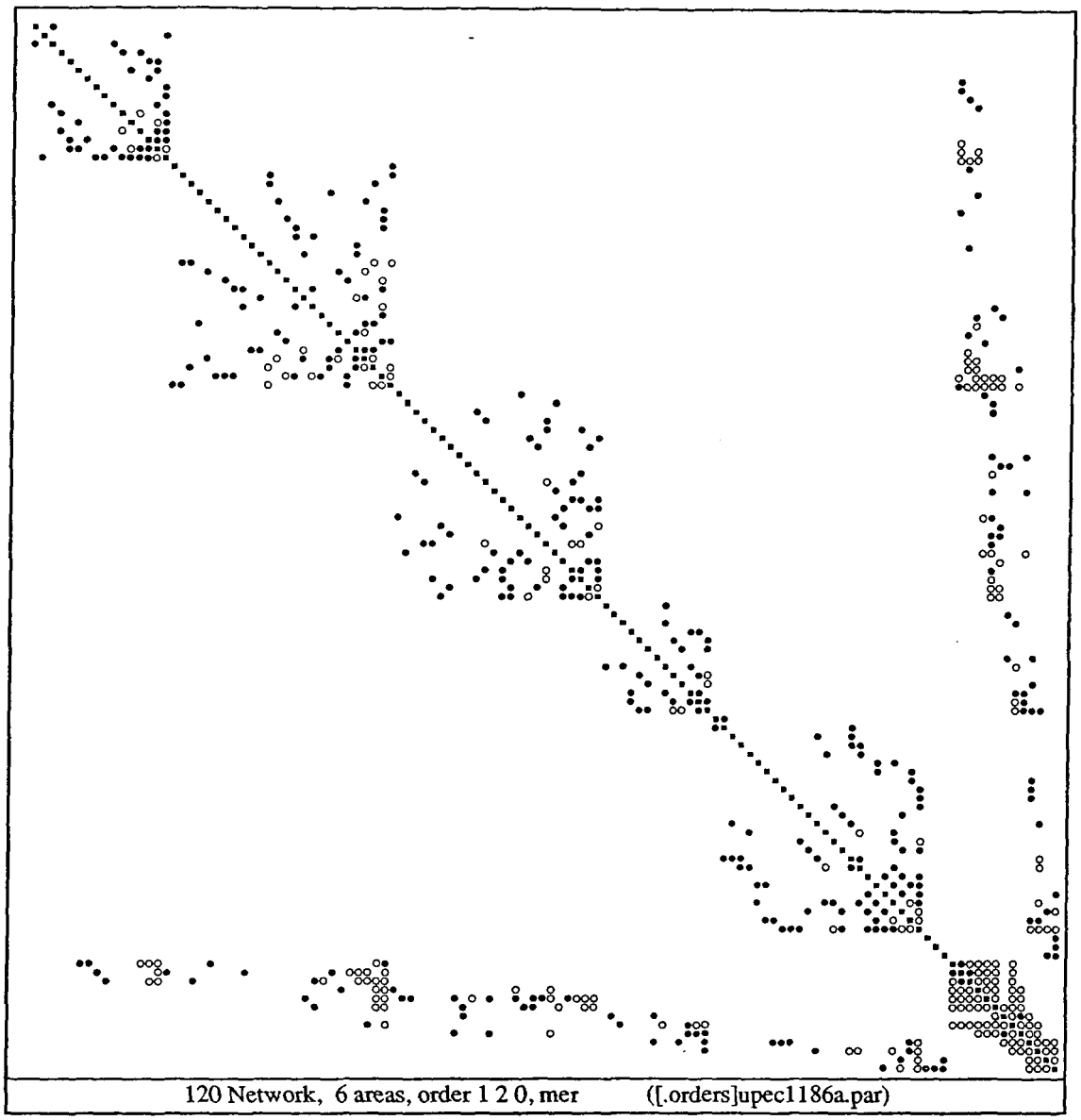

```
Ordering took                  70 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms      100 ms
Calculation took               60 ms      160 ms
Elements:  before | after        476     387     90    50.3%
Multiplications generated        958     656
Additions  generated             689     538
Divisions  generated             118       0
Multiply-additions generated     958     656  10996    0.183MFlops
                                 previous         current          next
Muladd counts for area  0 are        0    0       246  189         0    0
Muladd counts for area  1 are       74   34       310  208         0    0
Muladd counts for area  2 are       20   16       308  209         0    0
Muladd counts for totals are        94   50       864  606         0    0
Muladd count totals are                           958  656
Counts required=    568   (1570)     36.2%     2.76
```

Figure D.7 Matrix: 3 Areas, 118 nodes, UPEC1183.



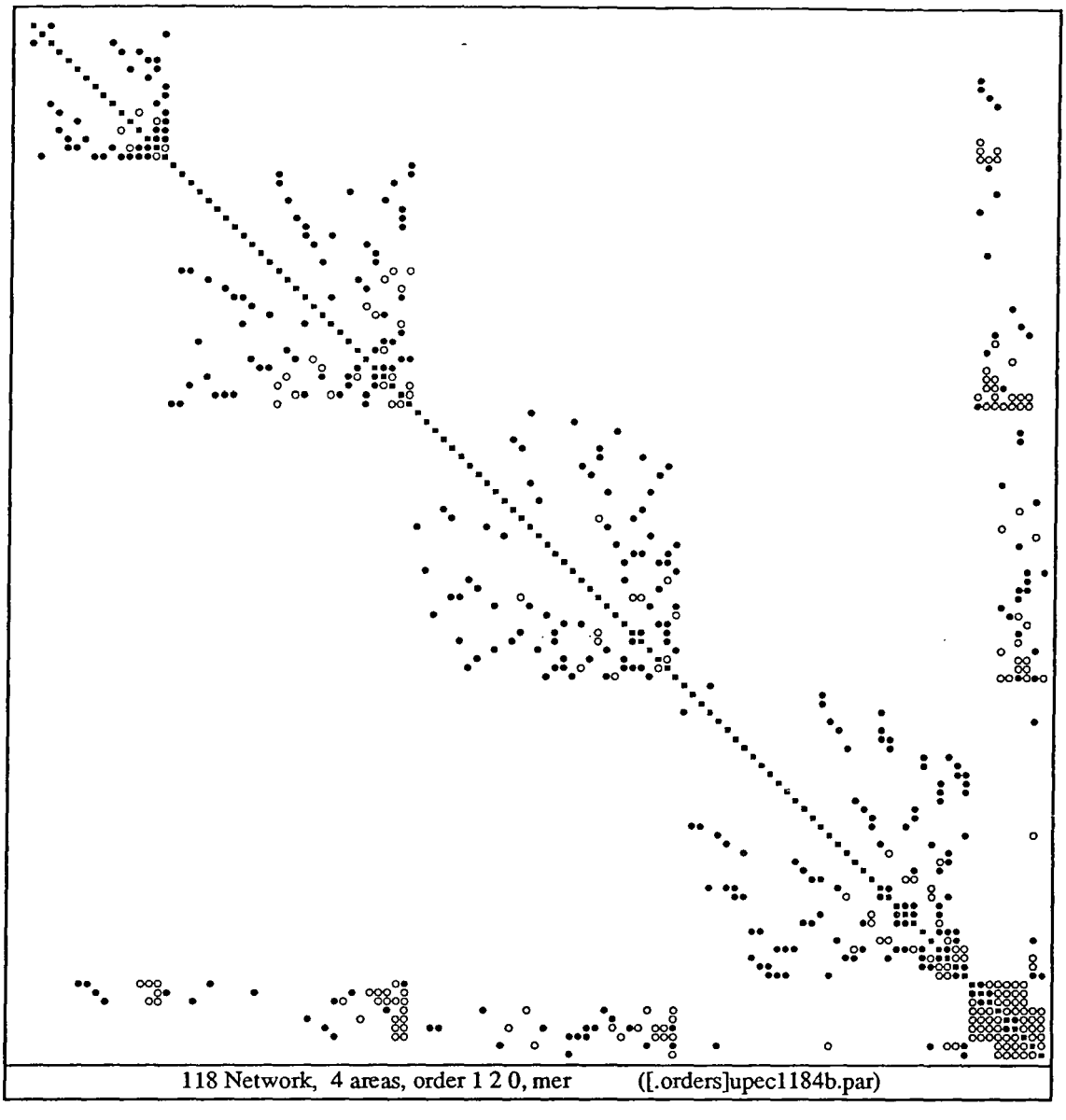

```
Ordering took                  80 ms
Pointer alteration took        00 ms
Reduction took                 10 ms
Solution took                  10 ms      100 ms
Calculation took               50 ms      160 ms
Elements:  before | after         476     387      90    50.3%
Multiplications generated         958     656
Additions  generated              689     538
Divisions  generated              118       0
Multiply-additions generated      958     656   10996    0.220MFlops
                                previous          current          next
Muladd counts for area  0 are        0    0        316  208         0    0
Muladd counts for area  1 are       62   27        282  185         0    0
Muladd counts for area  2 are        8    9        290  227         0    0
Muladd counts for totals are        70   36        888  620         0    0
Muladd count totals are                           958  656
Counts required=     560   (1570)     35.7%      2.80
```

Figure D.8 Matrix: 3 Areas, 118 nodes, T–3–PAPER.



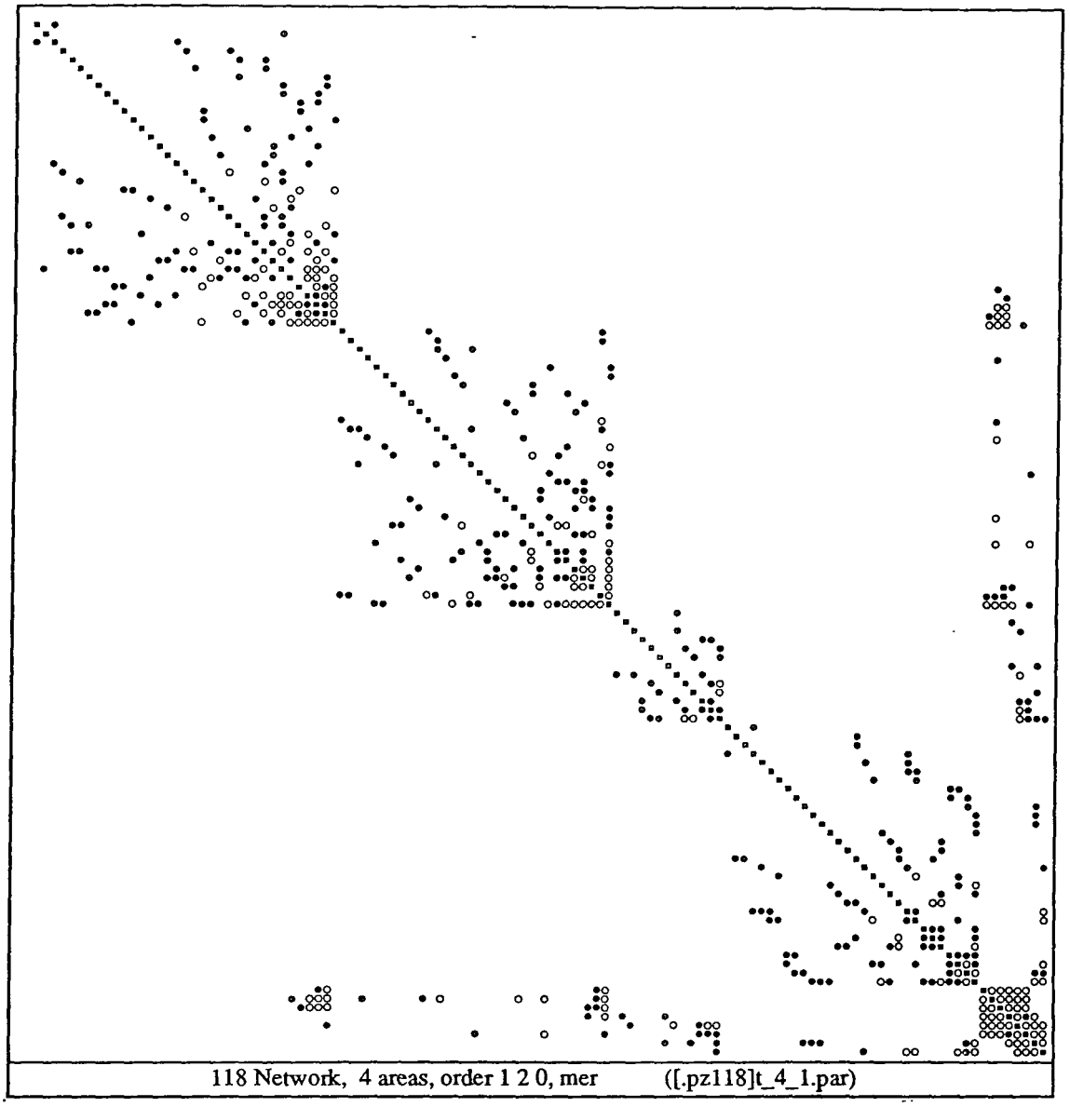

```
Ordering took                  80 ms
Pointer alteration took        00 ms
Reduction took                 20 ms
Solution took                  10 ms      110 ms
Calculation took               60 ms      170 ms
Elements:  before | after        482      407     106    58.6%
Multiplications generated       1088      694
Additions  generated             801      574
Divisions  generated             120        0
Multiply-additions generated    1088      694   12200     0.203MFlops
                                      previous          current          next
Muladd counts for area  0 are          0    0          102   80          0    0
Muladd counts for area  1 are         92   33          292  164          0    0
Muladd counts for area  2 are         44   23          210  142          0    0
Muladd counts for area  3 are         30   19           96   69          0    0
Muladd counts for area  4 are         18   12          192  137          0    0
Muladd counts for area  5 are          2    4           10   11          0    0
Muladd counts for totals are         186   91          902  603          0    0
Muladd count totals are                                1088  694
Counts required=     651   (1570)     41.5%      2.41
```

Figure D.9 Matrix: 6 Areas, 118 nodes, UPEC1186.



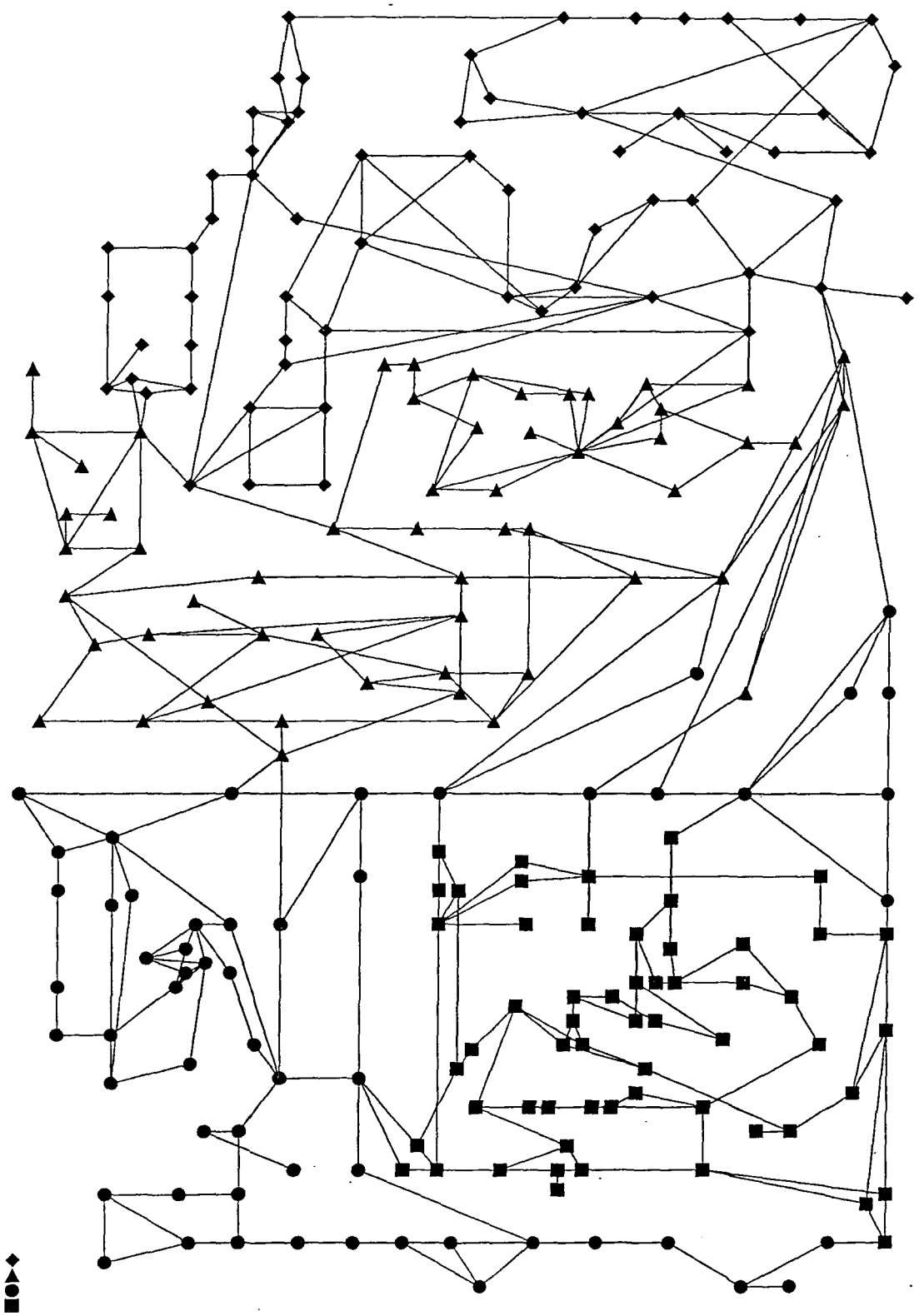

```
Ordering took                  80 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms      110 ms
Calculation took               70 ms      180 ms
Elements:  before | after        476     398     101    56.4%
Multiplications generated       1074     678
Additions  generated             794     560
Divisions  generated             118       0
Multiply-additions generated    1074     678   12012     0.172MFlops
                                previous          current          next
Muladd counts for area  0 are        0    0        102   80         0    0
Muladd counts for area  1 are       92   33        304  174         0    0
Muladd counts for area  2 are       68   32        292  185         0    0
Muladd counts for area  3 are        2    4        214  170         0    0
Muladd counts for totals are       162   69        912  609         0    0
Muladd count totals are                           1074  678
Counts required=     622   (1570)     39.6%      2.52
```

Figure D.10 Matrix: 4 Areas, 118 nodes, UPEC1184B.



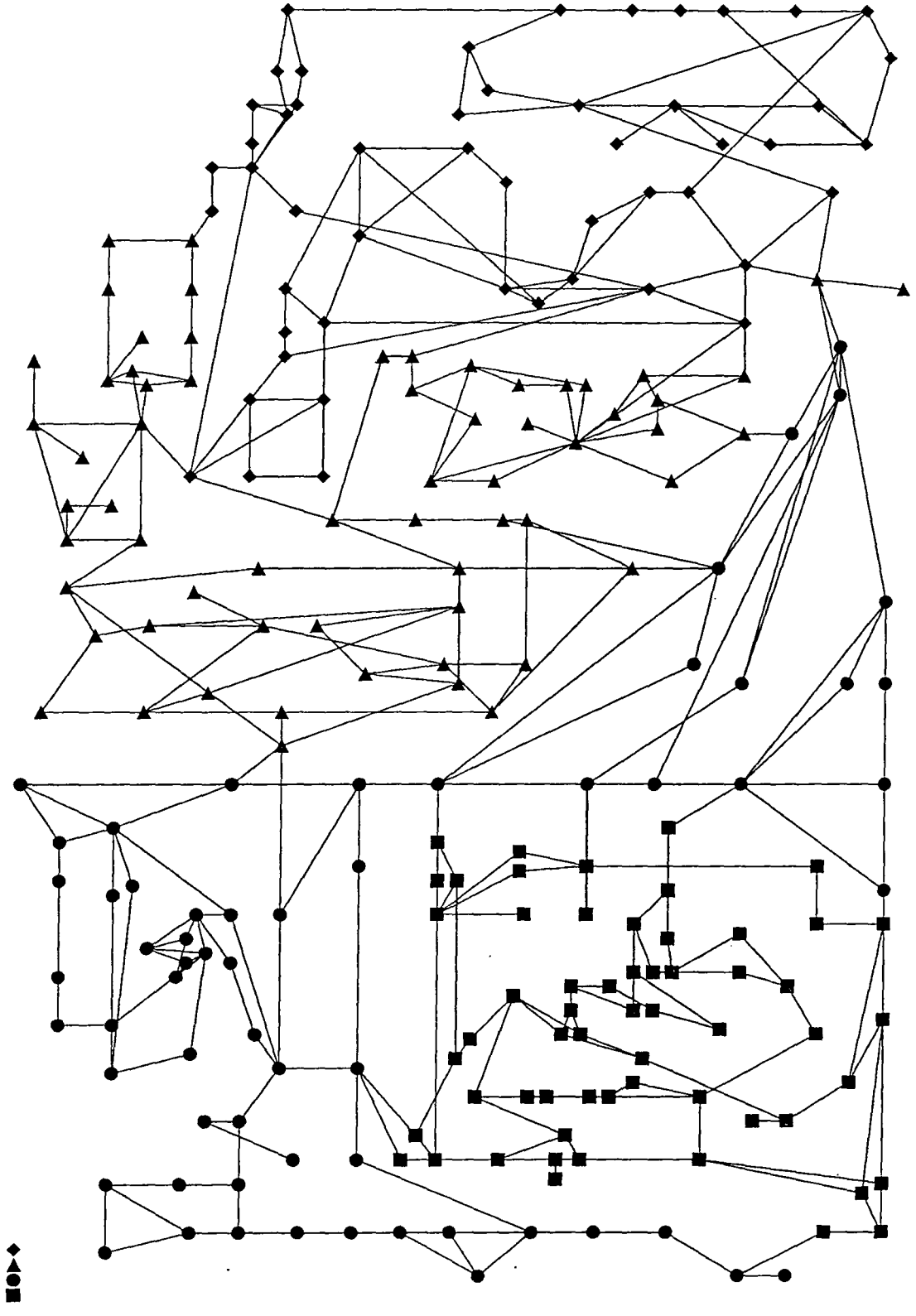

```
Ordering took                  80 ms
Pointer alteration took        10 ms
Reduction took                 10 ms
Solution took                  10 ms      110 ms
Calculation took               60 ms      170 ms
Elements:  before | after         476     391      94    52.5%
Multiplications generated         986     664
Additions  generated              713     546
Divisions  generated              118       0
Multiply-additions generated      986     664   11252    0.188MFlops
                                previous          current          next
Muladd counts for area  0 are       0    0        310  203         0    0
Muladd counts for area  1 are      62   27        302  194         0    0
Muladd counts for area  2 are      30   19         96   69         0    0
Muladd counts for area  3 are       2    4        184  148         0    0
Muladd counts for totals are       94   50        892  614         0    0
Muladd count totals are                           986  664
Counts required=    562  (1570)     35.8%      2.79
```

Figure D.11 Matrix: 4 Areas, 118 nodes, T–4–1.



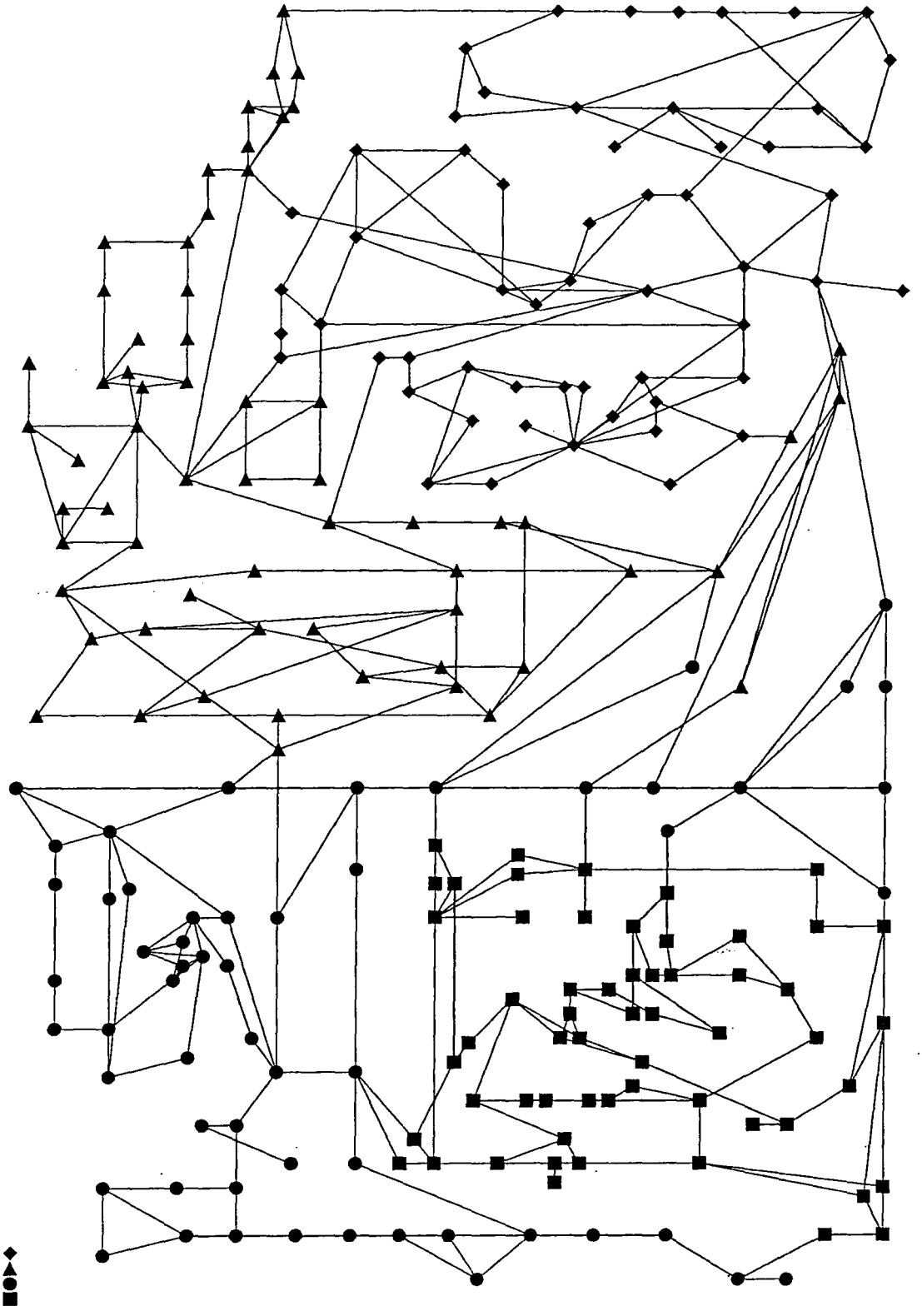
c_4_1

Figure D.12 Net: 4 Areas, 234 nodes, C–4–1.



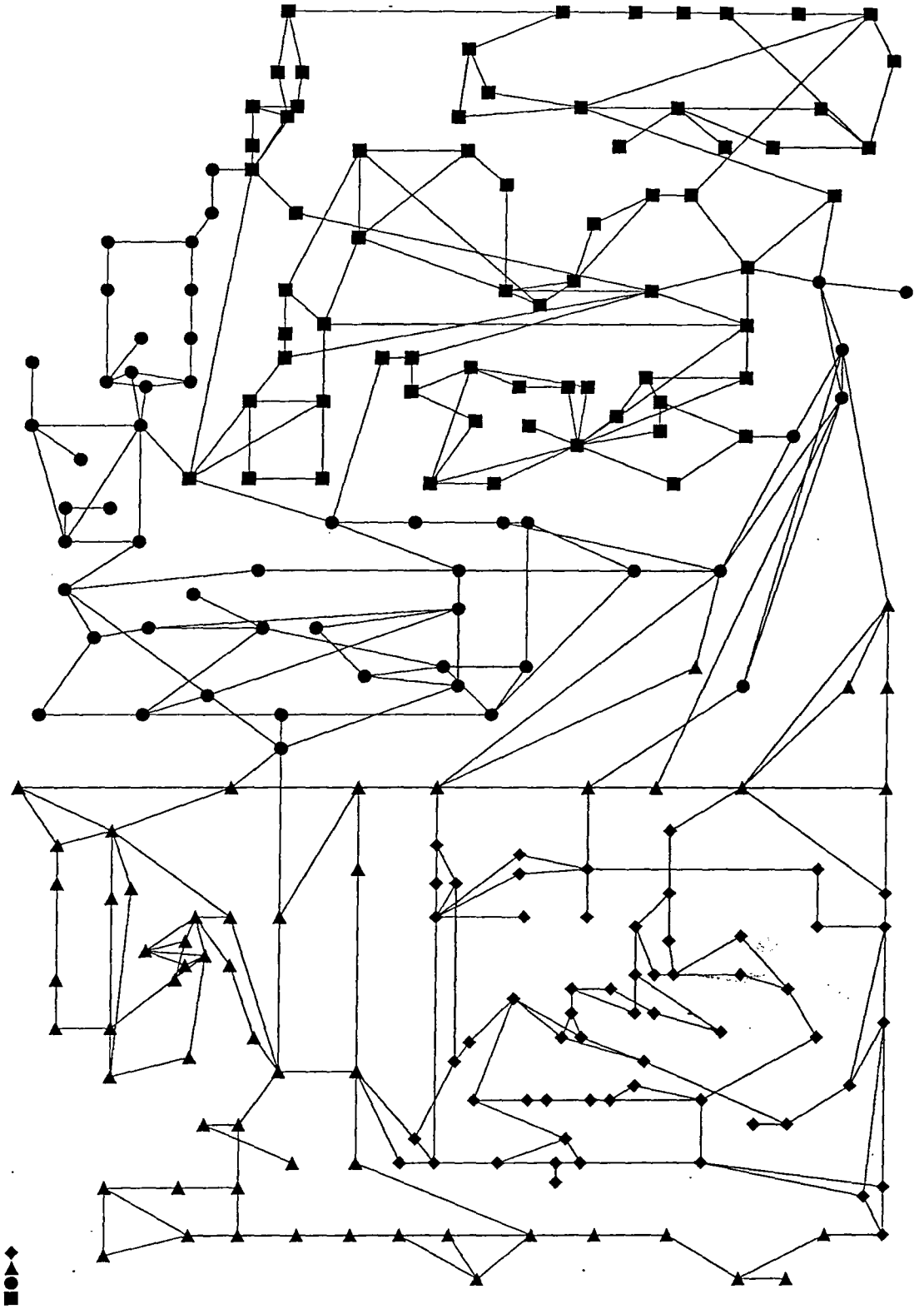
c_4_2

Figure D.13 Net: 4 Areas, 234 nodes, C–4–2.



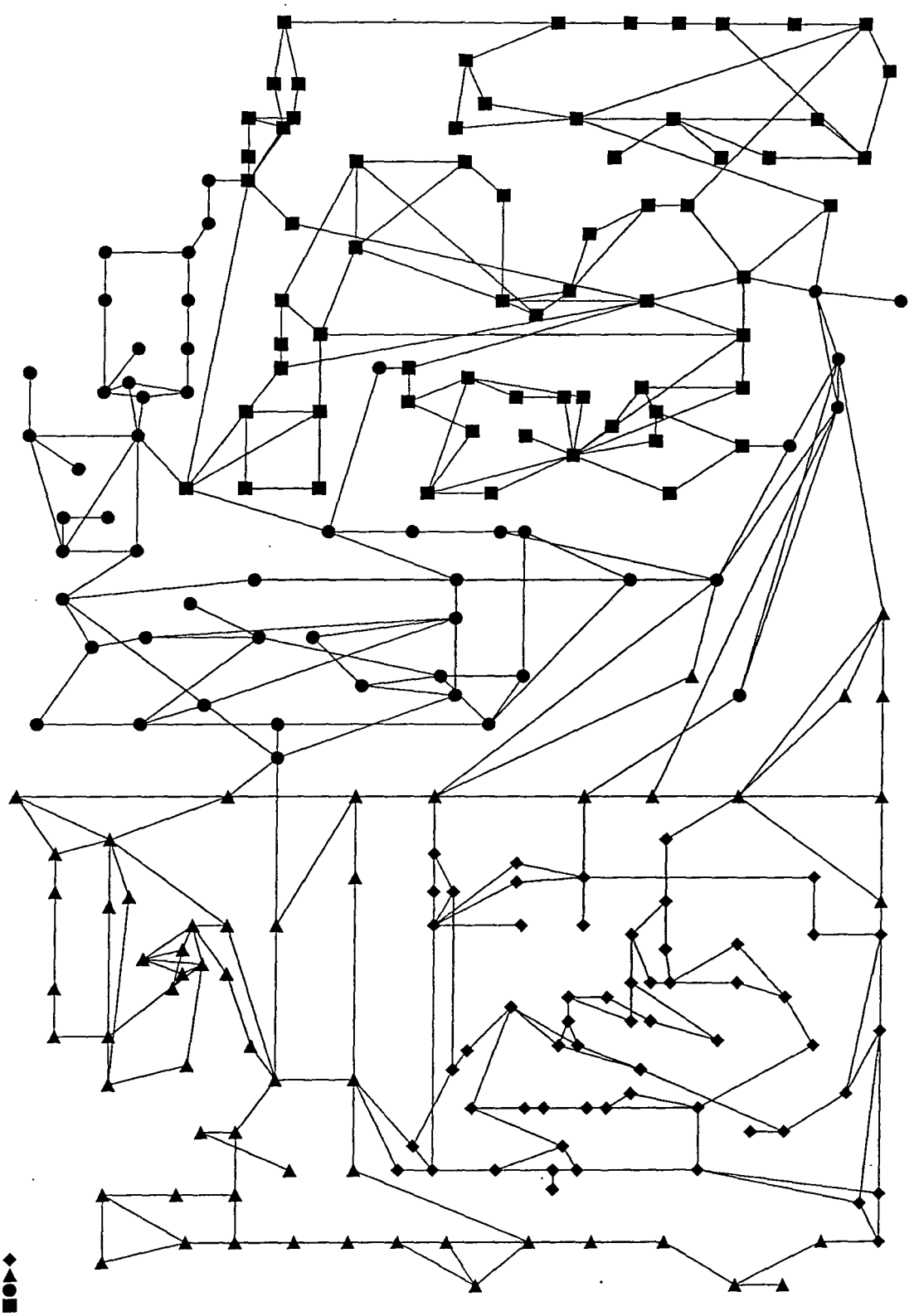
c_4_3

Figure D.14 Net: 4 Areas, 234 nodes, C–4–3.



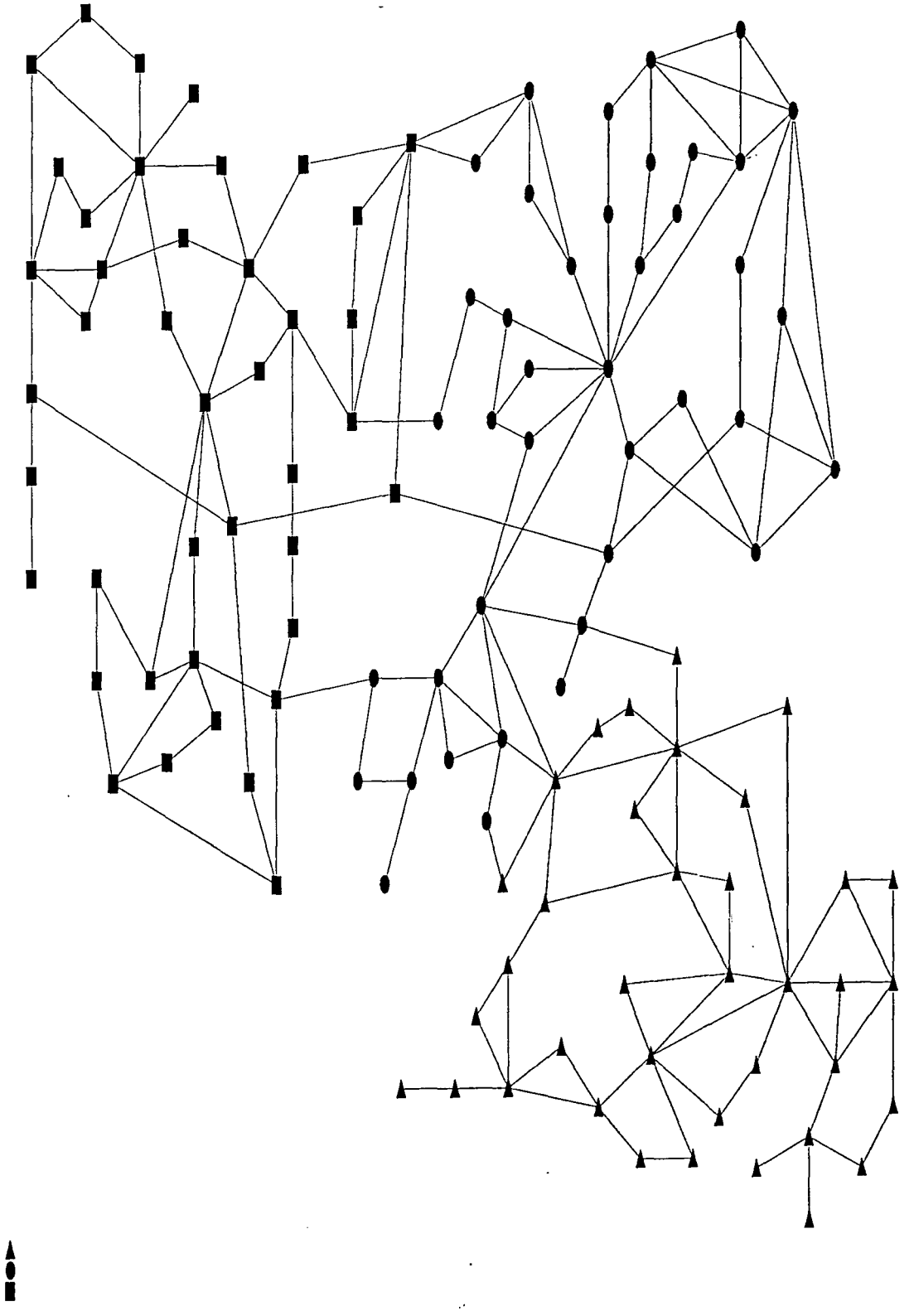
c_4_4

Figure D.15 Net: 4 Areas, 234 nodes, C–4–4.



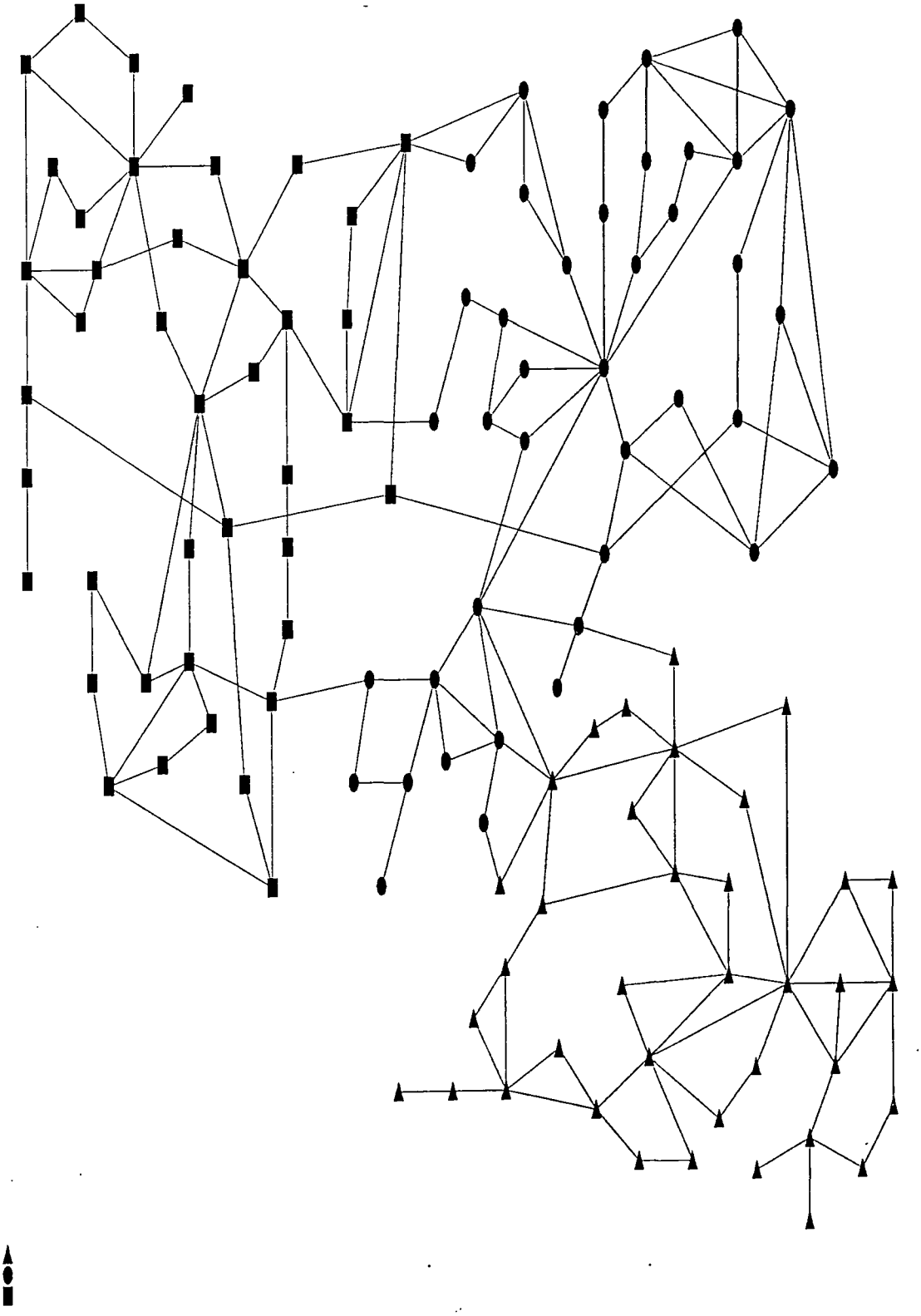
c_4_5

Figure D.16 Net: 4 Areas, 234 nodes, C–4–5.



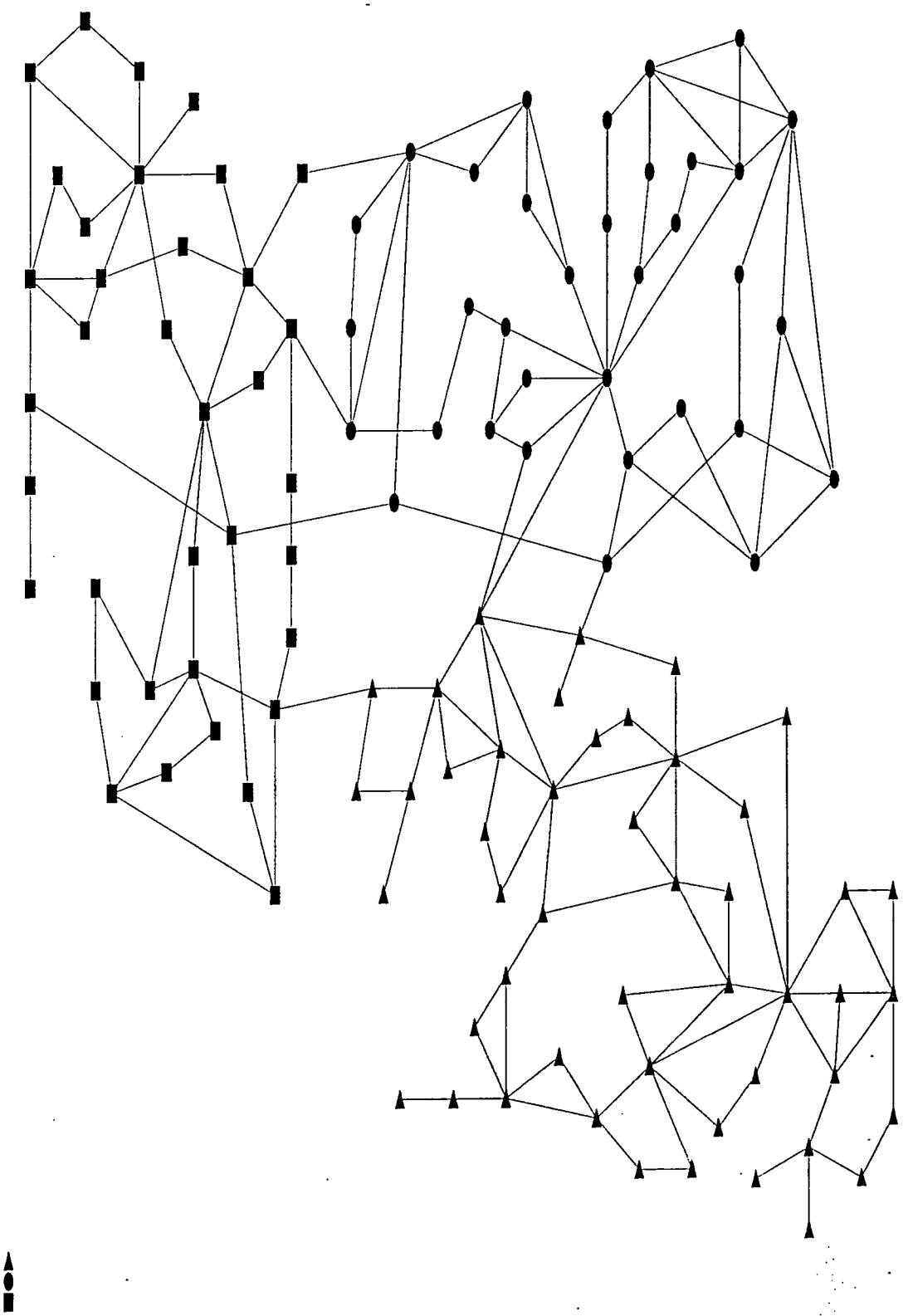
t_3_1

Figure D.17 Net: 3 Areas, 118 nodes, T–3–1.



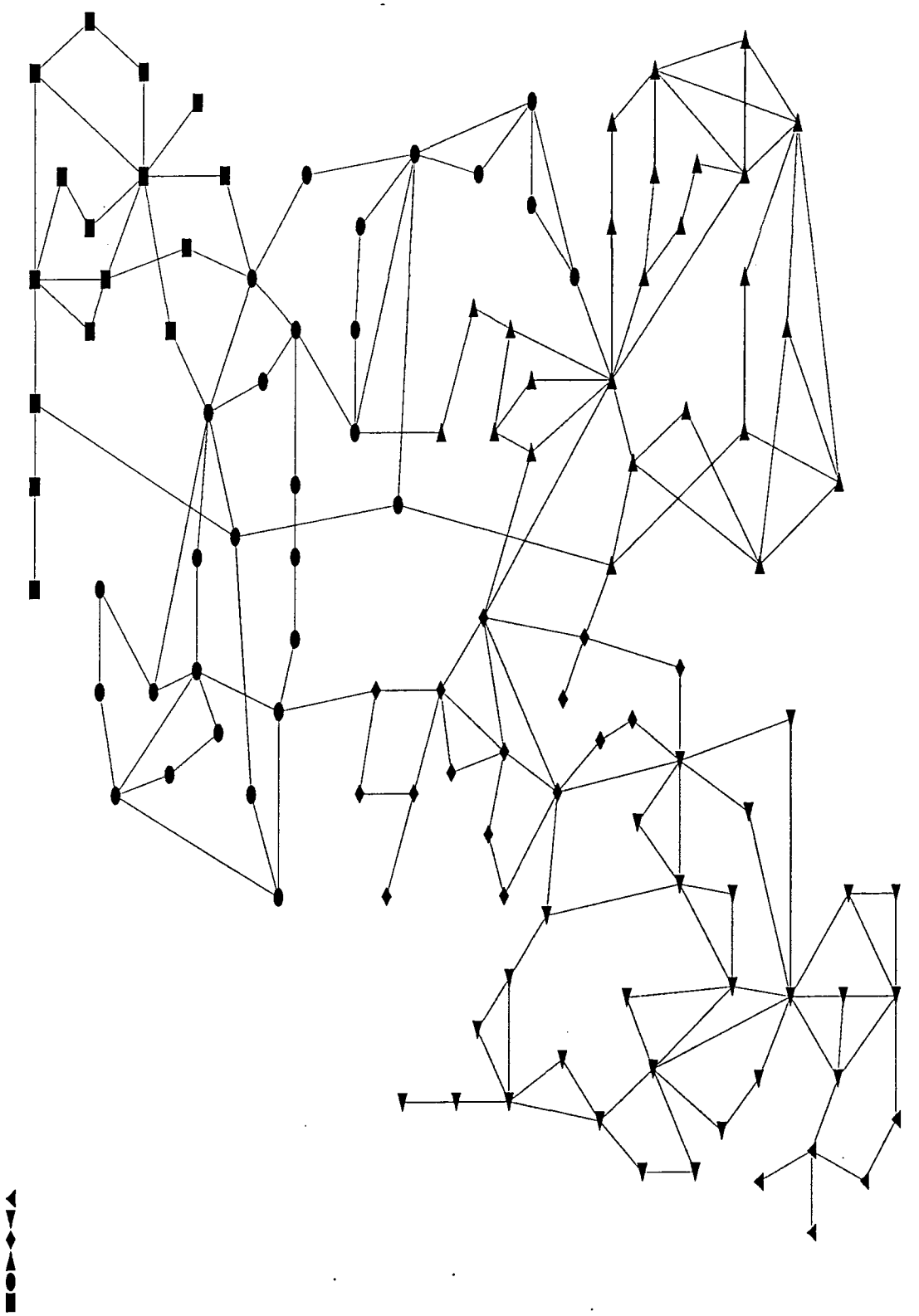
upec1183

Figure D.18 Net: 3 Areas, 118 nodes, UPEC1183.



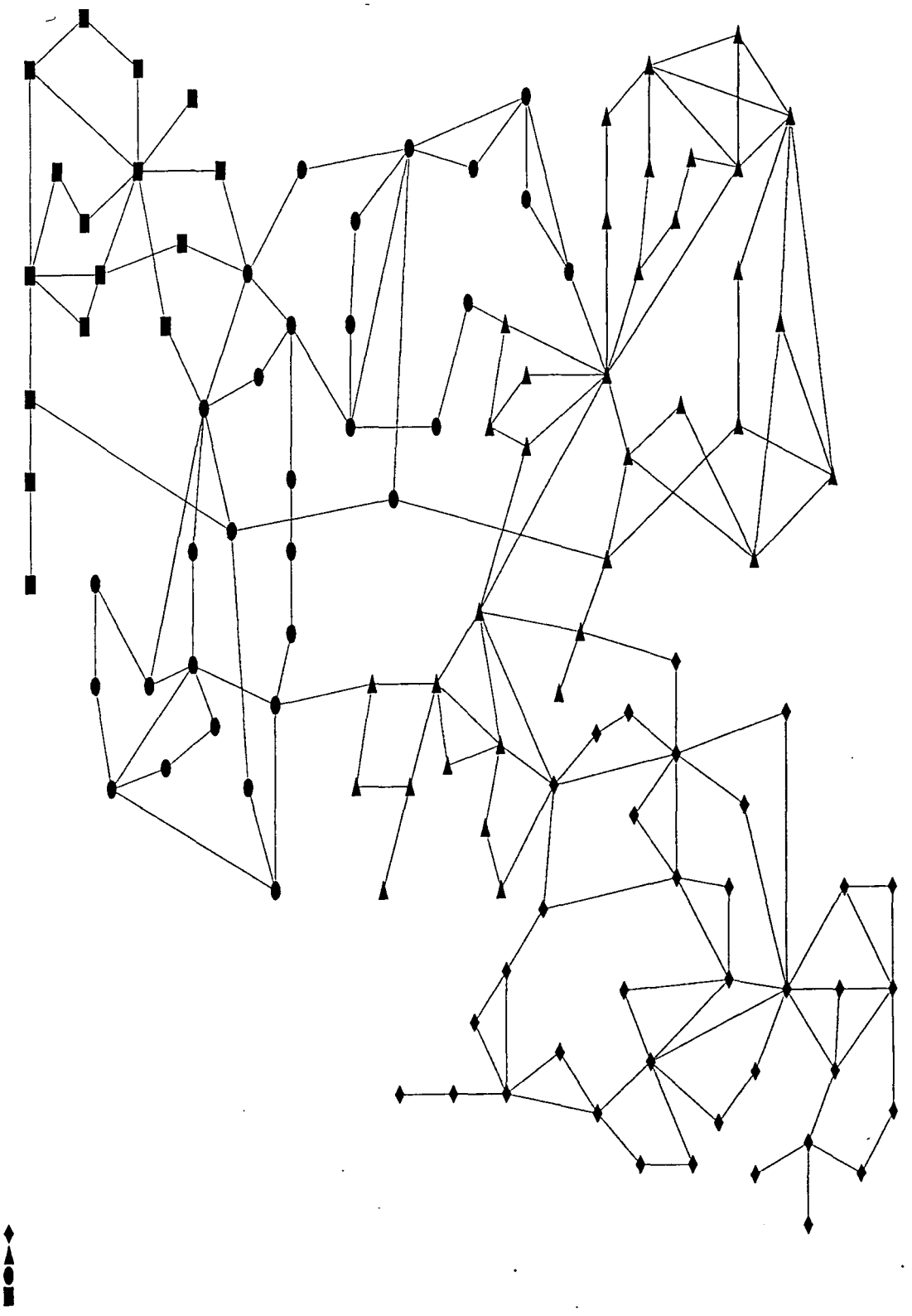
t_3_paper

Figure D.19 Net: 3 Areas, 118 nodes, T–3–PAPER.



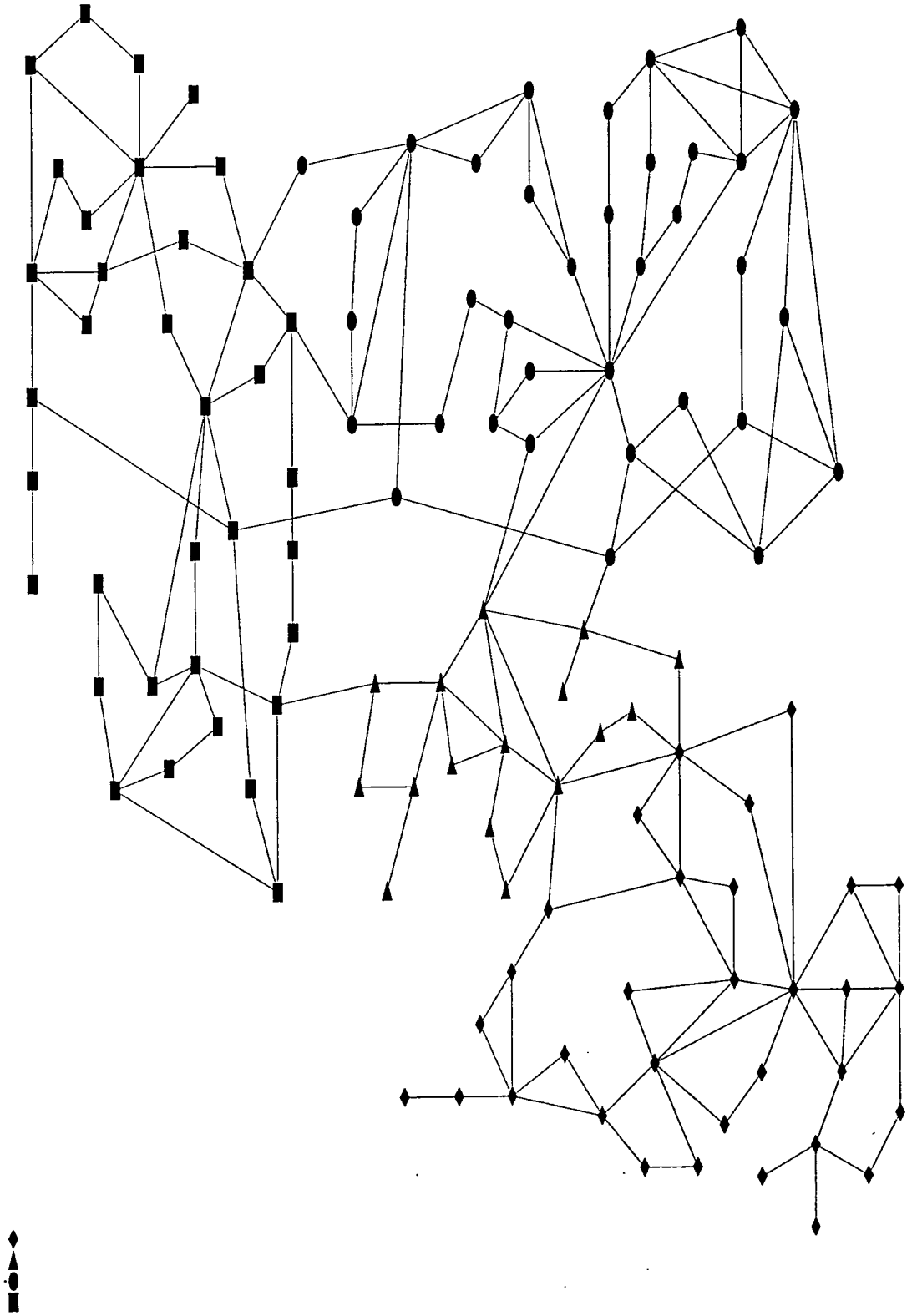
upec1186a

Figure D.20 Net: 6 Areas, 118 nodes, UPEC1186.



upec1184b

Figure D.21 Net: 4 Areas, 118 nodes, UPEC1184B.



t_4_1

Figure D.22 4 Areas, 118 nodes, T–4–1.