THE UNIVERSITY OF CALGARY

SIMULATION OF HYDRAULIC NETWORK DYNAMICS ON PARALLEL COMPUTERS

by

Prem Nath Mahana

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL ENGINEERING

CALGARY, ALBERTA

November, 1987

ISBN 0-315-42530-X

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommended to the Faculty of Graduate Studies for acceptance, a thesis entitled, "*Simulation of Hydraulic Network Dynamics on Parallel Computers*" submitted by Prem Nath Mahana in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dr. F.N. Trofimenkoff - Supervisor
Dept. of Electrical Engineering

Dr. L. E. Turner
Dept. of Electrical Engineering

Dr. J.G. Cleary
Dept. of Computer Science

Dr. E.C. Mikulcik
Dept. of Mechanical Engineering

Dr. D.A. Meneley
External Reader
University of New Brunswick

Date: 7 December 1987

Abstract

This thesis is concerned with the development of efficient parallel algorithms and the study of their speed-up and efficiency performance for the fast simulation of thermo-hydraulic network dynamics on multi-microprocessor systems. These networks are employed in nuclear power plants for the transmission of water and steam under high temperature and pressure. Simulation of the transient behaviour of these networks involves numerical integration of a large number of non-linear, coupled, time-varying, and "stiff" first-order ordinary differential equations. Since these equations are "stiff", Porsching's numerical integration algorithm which is based on Euler's backward difference formula has been chosen for parallelization. In this thesis analytical expressions for the resulting speed-ups and efficiencies of the parallelized Porsching's algorithm have been derived and it has been shown that good speed-ups and efficiencies can be achieved by using quasi-MIMD (Multiple-Instruction Stream Multiple-Data Stream) mode of operation employing a) master/slave configuration of processors, b) message-passing rather than shared memory for data communication, c) global synchronization of processors before data communication, d) a simple time-shared bus with broadcast facility as the interconnection network, and e) distributed communication memories for data communication.

Applications of this research include development of operator training simulators, interactive computer-aided design, and computer-aided emergency response and accident investigation.

Acknowledgements

The author wishes to express his sincere gratitude and a deep feeling of indebtness to Dr. F.N. Trofimenkoff for his constant support, encouragement and guidance. He is also thankful to Dr. Trofimenkoff for his helpful advice during the preparation of the manuscript.

Grateful acknowledgements are extended to Dr. M.H. Hamza and Dr. Laurence Turner for their support and encouragement.

The author would also like to thank Ms. Laura Millan for her prompt typing of this thesis.

The financial assistance received from the University of Calgary and NSERC is gratefully acknowledged.

This dissertation will not be complete without thanking my wife Neelam, my children Meenu and Ravi, my parents and my parents-in-law for their patience and support throughout the course of this work.

Table of Contents

Page No.

Table of Contents v
List of Tables viii
List of Figures ix
List of Symbols xii

1. INTRODUCTION 1
   1.1 Motivation 1
   1.2 Problem Statement 9
   1.3 Approach 9
      1.3.1 Choice of the algorithm 9
      1.3.2 Choice of parallel model of computation 10
   1.4 Overview 15
2. NUMERICAL INTEGRATION OF NETWORK DIFFERENTIAL EQUATIONS 17
   2.1 Introduction 17
   2.2 Terminology and definiations 17
      2.2.1 Hydraulic network 17
      2.2.2 Initial and terminal nodes 19
      2.2.3 Initial and terminal links 19
      2.2.4 Noncritical and critical links 19
      2.2.5 Chains 20
   2.3 Network conservation equations to be integrated 21
   2.4 Numerical integration algorithms 25
      2.4.1 One-step methods 26
      2.4.2 Multi-step methods 34
      2.4.3 Stiff differential equations 41
   2.5 Porsching's algorithm 43
      2.5.1 Introduction 43
      2.5.2 Development of the algorithm 44
3. ALGORITHM PERFORMANCE ON A SINGLE PROCESSOR 55
   3.1 Introduction 55
   3.2 Computational complexity of Porsching's algorithm 55
      3.2.1 Computation of flow rate increments from the matrix equation 55
         3.2.1.1 Structure of matrix A 55

3.2.1.2 Estimation of computation time for solving the matrix equation 65
3.2.1.2.1 Direct method I 66
3.2.1.2.2 Direct method II (Block Gaussian elimination method) 75
3.2.2 Estimation of processing time for the remaining computations 90
4. SELECTION OF A SUITABLE ARCHITECTURE FOR PARALLEL COMPUTATIONS 101
4.1 Introduction 101
4.2 Taxonomy of parallel computers 102
4.3 Selection of an architecture 106
4.3.1 Unsuitability of SIMD computers 106
4.4 Interconnection networks 112
4.4.1 Static network topologies 112
4.4.2 Dynamic network topologies 118
4.4.2.1 Single-stage networks 118
4.4.2.2 Multi-stage networks 120
4.4.2.3 Cross-bar switch 124
4.5 Selection of an interconnection network 125
4.6 Description of the chosen parallel architecture 128
4.7 On the number of processors to be used 135
5. ALGORITHM PERFORMANCE ON MULTIPLE PROCESSOR SYSTEMS 140
5.1 Introduction 140
5.2 Parallel computation of flow-rate increments from the matrix equation 141
5.2.1 Direct parallel method I 141
5.2.1.1 Number of processors equal to number of chains in the network 151
5.2.1.2 Number of processors equal to number of random links in the network 158
5.2.1.3 Number of processors equal to number of chains in the network plus a two-dimensional processor array 162
5.2.1.3.1 Fawcett's algorithm 165
5.2.2 Direct parallel method II 176
5.2.2.1 Number of processors equal to number of chains in the network 176
5.2.2.2 Number of processors equal to number of random links in the network 183
5.2.2.3 Number of processors equal to number of chains plus a two-dimensional processor array 188

5.3 Parallel processing of the remaining computations 195
5.3.1 Number of processors equal to number of chains 195
5.3.1.1 Matix solution by direct parallel method I 197
5.3.1.2 Matrix solution by direct parallel method II 205
5.3.2 Number of processors equal to number of random links in the network 205
5.4 Parallel processing performance for total computations 209
6. PERFORMANCE OF A PARALLEL ITERATIVE METHOD 224
6.1 Introduction 224
6.2 Jacobi's iterative method 224
6.3 Algorithm performance on a single processor 225
6.4 Algorithm performance on multiple processors 230
6.4.1 Parallel Jacobi's algorithm 230
6.4.2 Estimation of parallel computation time 233
6.4.3 Simulation results 239
6.4.3.1 An 8-node network example 239
6.4.3.2 A 64-node network example 244
7. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER WORK 253
7.1 Conclusion 253
7.2 Recommendations for further work 255
REFERENCES 257

List of Tables

Table No.	Title	Page No.
3.1	Six network examples for studying algorithm performance	76
5.1	Summary of processing times for solving matrix equation by direct method I	210
5.2	Summary of processing times for solving matrix equation by direct method II	211
5.3	Summary of processing times for the rest of the computations	212
5.4	Step-size values for real-time computations	219
5.5	Effect of synchronizations time on the step-size for real-time computations	222
6.1	Parameters for the 8-node network example	241

List of Figures

Figure No.	Title	Page No.
2.1	A typical hydraulic network with 11 nodes and 16 links	18
2.2	Quantities associated with a node and link	18
2.3	Classification of numerical methods	27
2.4	Region of absolute stability for forward Euler method	31
2.5	Region of absolute stability for forth-order Runge-Kutta method	35
2.6	Regions of absolute stability for first and second-order Adams-Bashforth algorithms	40
2.7	Regions of absolute stability for first and second-order Adams-Moulton algorithms	40
2.8	Values of a_{kv} elements of matrix A	52
3.1	Basic flow chart of Porsching's integration algorithm	56
3.2	Structure of matrix A for network example of Fig. 2.1	58
3.3(a)	A nonoptimal link numbering for a network example	60
3.3(b)	Structure of matrix A corresponding to this link numbering	61
3.4(a)	Improved link numbering for a network example	62
3.4(b)	Structure of matrix A corresponding to this link numbering	63
3.5	Pseudo-code for Gaussian elimination of C x = b	71
3.6	Computation times in matrix solution by direct method I	77
3.7	Flow chart for solving matrix equation by direct method II	79
3.8	Comparison of matrix equation solution times by direct methods I and II	89
3.9	Flow chart for remaining computations in Porsching's algorithm	91
3.10(a)	Computation times on a serial computer vs network size	98
3.10(b)	Computation tims as a percentage of total computing time	99
4.1	Three MIMD computer architectures	104
4.2	Vector addition on a pipelined vector computer	108
4.3	Typical performance of a pipelined vector computer as a function of vector length	110
4.4	Classification of computer interconnection networks	113
4.5	Examples of static network topologies	114
4.6	Stone's shuffle-exchange network for N = 8	119
4.7	Lawrie's Omega network for N = 8	121
4.8	Structured Multimicroprocessor System (SMS) with communication	

	memories	129
4.9(a)	Memory access mode during autonomous phase	130
4.9(b)	Memory access mode during communication phase	130
4.10	Parallel mode of computation for SMS 201 multimicroprocessor system	132
4.11	Determination of total processing time on SMS 201	133
4.12	Ware's model of parallel computer speed-up for 8, 16, 32 and 64 processors	138
5.1	Intermediate and final forms of matrix V in the forward elimination step	142
5.2(a)	kth-stage of Gauss-Jordan elimination	144
5.2(b)	Pseudo-code for sequential Gauss-Jordan elimination	144
5.3(a)	Problem partitioning in Gauss-Jordan algorithm	146
5.3(b)	Pseudo-code for parallel Gauss-Jordan elimination	147
5.4	Speed-up and efficiency for solving C x = b by Gauss-Jordan method	150
5.5	Pseudo-code for solving matrix equation on s processors by direct method I	152
5.6	Speed-up and efficiency for solving matrix equation on s processors by direct method II	157
5.7	Speed-up and efficiency for solving matrix equation on q processors by direct method I	161
5.8	Master processor with s slave processors and a q x q processor array	164
5.9	Fawcett's algorithm for solving C x = b on q x q processor array	167
5.10	Clock distribution in systolic arrays using H-tree	171
5.11	Handshaking mechanism in asynchronous processor arrays	172
5.12	Speed-up for solving matrix equation on s processors and a q x q array by direct method I	175
5.13	Pseudo-code for solving matrix equation on s processors by direct method II	177
5.14	Speed-up and efficiency for solving matrix equation on s processors by direct method II	182
5.15	Speed-up and efficiency for solving matrix equation on q processors by direct method II	187
5.16	Pseudo-code for solving matrix equation on s processors and a q x q processor array by direct method II	189
5.17	Speed-up for solving matrix equation on s processors and a q x q processor array by direct method II	194
5.18	A comparison of computing times for solving matrix equation by direct methods I and II	196
5.19	Pseudo-code for remaining computations on s processors	198
5.20	Speed-up for remaining computations on s processors	206

5.21 Speed-up and efficiency for remaining computations on q processors 208
5.22 Speed-up and efficiency for total computations on s processors using direct method I 213
5.23 Speed-up and efficiency for total computations on q processors using direct method I 214
5.24 Speed-up for total computations on s processors plus a q x q processor array using direct method I 215
5.25 Speed-up and efficiency for total computations on s processors using direct method II 216
5.26 Speed-up and efficiency for total computations on q processors using direct method II 217
5.27 Speed-up of total computations on s processors plus a q x q array using direct method II 218
5.28 Step-size versus network-size for real-time computations 220
6.1 Pseudo-code for solving A x = b by serial Jacobi's method 226
6.2 Number of iterations for Jacobi's method to be faster than direct method 229
6.3 Pseudo-code for solving A x = b by parallel Jacobi's method 231
6.4 Effect of communication time for solving matrix equation by Jacobi's method 236
6.5 Effect of synchronization time for solving matrix equation by Jacobi's method 237
6.6 Effect of p_m for solving matrix equation by Jacobi's method 238
6.7 Schematic of pressurized tank and control volume representation 240
6.8 Variation of link flow rates and pressure for 8-node example 245
6.9 Normalized parallel computing time versus number of processors for 8-node example 246
6.10 Speed-up and efficiency for 8-node example 247
6.11 Normalized parallel computing time versus number of processors for 64-node example 249
6.12 Speed-up and efficiency for 64-node example 250
6.13 Effect of communication and synchronization times on efficiency for 64-node example 251

LIST OF SYMBOLS

A_k	flow area of link k
$\mathbf{A}$	matrix of dimensions $K \times K$
a_{ij}	element (i,j) of matrix A
$\mathbf{B}_i$	matrix of dimensions $q \times p_i$
$\mathbf{b}$	a vector
$\mathbf{C}$	matrix of dimensions $q \times q$
$\mathbf{c}_i$	vector of dimensions $p_i \times 1$
d_{av}	average degree of non-chained nodes in the network
$d_{\max}$	maximum degree of a non-chained node in the network
D_k	hydraulic diameter of link k
$\mathbf{D}_i$	tridiagonal matrix of dimensions $p_i \times p_i$
$\mathbf{E}_i$	matrix of dimensions $p_i \times q$
f_k	a non-linear function
$(fr)_k$	dimensionless friction factor for link k
$\mathbf{G}_i$	a matrix of dimensions $q \times q$
$\mathbf{g}$	vector of dimension n
g	acceleration due to gravity
g_k	a non-linear algebraic function
H_ν	specific enthalpy of fluid in link ν entering node i
H_ν^*	specific enthalpy of fluid in link ν leaving node i

h	time-step size in the numerical integration algorithm
h_i	vector of dimension $q \times 1$
I	number of iterations for convergence in Jacobi's method
$\mathbf{I}$	unity matrix of dimensions $K \times K$
I_i	set of links initiating from node i
$\mathbf{J}$	Jacobian matrix of dimensions $(K + 2N) \times (K + 2N)$
K	number of normal links in the network
K^*	highest index number of a critical link in the network
k	index number of a link in the network
$\mathbf{L}$	matrix $\mathbf{L}$ in $\mathbf{L}\ \mathbf{U}$ factorization
L_k	length of link k
M_i	mass of node i
MIMD	multiple-instruction stream multiple-data stream
N	number of nodes in the network
N_c	number of chained nodes in the network
N_{nc}	number of non-chained nodes in the network
n_{av}	average degree of end-nodes minus one of all chains in the network
$n_{\max}$	maximum degree of an end-node minus one of a chain in the network
N_p	number of processors
P_i	pressure at node i
p_i	number of links in chain i
p_m	number of links in the longest chain in the network
Q_i	external heat input rate into node i

q	number of random links in the network
$\mathbf{R}_i$	matrix of dimensions $p_i \times q$
SIMD	single-instruction stream multiple-data stream
s	number of chains in the network
T	computation time on the computer
T_{dpu}	number of operations to compute derivative of a node pressure with respect to its internal energy
T_{dpm}	number of operations to compute derivative of a node pressure with respect to its mass
T_p	number of operations to compute pressure of a node
t	time
t_c	time for one data communication between the processors through bus
t_c'	time for one data communication between the processors in the processor array
t_f	one floating-point operation time on a processor
t_f'	one floating-point operation time on a processor in the array
t_{mac}	two floating-point operations plus one data communication time on the array
t_s	one synchronization time
$\mathbf{U}$	matrix U in L U factorization
U_i	total internal energy of node i
u_i	specified internal energy of node i
$\mathbf{V}$	matrix of dimension $q \times q$
v_i	specific volume of node i
W_k	fluid flow-rate in link k
$\mathbf{X}$	matrix of dimension $q \times (K - q)$

$\mathbf{x}$	vector of dimension $q \times 1$
$\mathbf{Y}$	matrix of dimension $q \times q$
$\mathbf{y}$	vector of dimension $(K + 2N) \times 1$
$\dot{\mathbf{y}}$	first derivative of $\mathbf{y}$ with respect to time
$\mathbf{z}$	vector of dimension $k \times 1$
z_k	an element of vector z

Other Symbols

ν	index number of a link
π_i	a non-linear algebraic function
ρ_k	density of fluid in link k
$\sum_{\nu \epsilon T_i}$	summation over all the links terminating at node i
$\sum_{\nu \epsilon I_i}$	summation over all the links initiating from node i .
λ	an eigen-value of a system of differential equations
σ	an angle in the unit circle in the z-plane
$\Delta \mathbf{W}$	flow-rate increments vector of dimension $K \times 1$
ΔW_{new}	new value in the Jacobi's iteration
ΔW_{old}	old value in the Jacobi's iteration
$\Delta \boldsymbol{\omega}_i$	flow-rate increments vector of dimension $p_i \times 1$
ζ_i	vector of dimension $p_i \times 1$
ΔZ_k	elevation change for link k
$\lceil \rceil$	ceiling function

CHAPTER 1

INTRODUCTION

1.1. Motivation

Large-scale technological systems, such as nuclear power plants, are so expensive that numerical simulation of their behaviour is not only an economical method of experimentation, but it is sometimes the only way to study their performance. This is specially true in those cases where experiments are difficult or impossible to perform due to safety-related reasons. The study reported in this thesis is concerned with the development of efficient parallel algorithms for the "fast" transient simulation of thermo-hydraulic network dynamics of nuclear power plants on MIMD (Multiple-Instruction Stream Multiple-Data Stream) computers. These networks are employed for the transmission of water and steam under high temperature and pressure

The study of the transient behaviour of these networks through simulations is important in a number of applications. Some of these applications are:

1. Development of training simulators for training plant operators.

   This training requires operators to understand plant processes thoroughly. A major criterion is that simulations be performed in real-time so that the operators are well-trained to handle plant malfunctions in the shortest possible time. If the operators are well-trained, then these abnormal conditions are unlikely to turn into costly accidents. As an example of a costly accident, GPU and Metropolitan Edison paid 24 million dollars to buy replacement electricity from other producers for 20 months [1].

Therefore, it is not difficult to cost-justify a high performance simulator if it prevents another accident.

2. Computer-aided emergency response to an ongoing accident.

This capability does not yet exist in present-day simulators but it may be quite useful in handling slowly varying abnormal plant conditions such as Small-Break Loss-of-Coolant (SBLOCA) accident where the operator has time to take suitable corrective action based on simulation results obtained by carrying out simulations of the plant response to a number of possible corrective actions. Obviously, this application requires faster than real-time simulation or "super simulation" capability.

3. Investigation of a recent accident.

When there is uncertainty concerning the cause of the accident, the simulator can be used to investigate the likely cause of the accident in order to find the one that produces a match to the real plant behaviour. This information may be useful in defining any future recovery procedures, reporting to the regulatory authority, and providing information to the public [2].

4. Start-up and commissioning of new plants.

The simulator can reduce plant start-up time by verifying the final procedures and developing procedural improvements.

5. Hydraulic network design and optimization.

6. Assessment of plant safety.

Results from safety-related laboratory-scale experiments can be extended to full-scale systems by the use of a simulator.

The computing requirements for some of the above applications are very stringent. For example, computations for training simulators for training plant operators have to be performed in real-time. For other applications, although real-time performance is not a definite requirement, it is still highly desirable from the point-of-view of user convenience that these simulations be performed in a fast, interactive manner. Unfortunately, these simulations require numerical integration of a large number of differential equations of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$ and, therefore, run too slowly on conventional serial computers due to their sequential mode of operation. The computing speeds necessary for achieving real-time computations are two orders of magnitude faster than those currently available on the fastest sequential computers.

In the past, the following techniques have been employed in order to achieve fast simulations:

1. Use of analog computers.

   In this implementation, the dynamics simulation problems were solved using analog computing elements such as integrators, summers, non-linear function generators, etc.. However, these analog computers suffered from the usual drawbacks of analog circuits such as low computing accuracy, non-flexibility, and the need for frequent re-calibrations/re-adjustments. The problem had to be posed in such a way that an answer could be computed with the machinery available. In other words, the answer was as much a function of computer as it was of the question [3]. The only advantage of analog computation was its high computing speed resulting from the use

of fast parallel mode of operation although analog in form.

2. Storing of pre-computed solutions in the form of look-up tables and the use of interpolation.

   Early power plant simulators made extensive use of table look-up techniques or generation of pre-computed solutions rather than obtaining solutions from the first-principles [4]. However, results obtained by these techniques may sometimes be in serious error particularly if the assumptions made in the pre-computed solutions are not valid in a particular simulation such as simulation of a highly abnormal plant operating conditions.

3. Use of coarser engineering models.

   Since the computing time depends heavily on the level of detail employed in the modelling, early simulators employed coarse models of physical processes. For example, modelling of hydaulic networks was done with as few nodes and links as possible. Obviously, this affected the accuracy of the solution obtained.

4. Use of faster fixed-point integer arithmetic instead of slower floating-point arithmetic.

   Some special-purpose computers such as Applied Dynamics AD-10 employ fixed-point arithmetic for the numerical integration of differential equations [5]. However, the problem with fixed-point arithmetic is its limited dynamic range which requires scaling of all variables in the problem solution such that they do not exceed the dynamic range of the computer. This may prove to be very tedious and time-consuming for large-scale problems.

5. Development of more efficient numerical integration algorithms.

Sometimes the use of an efficient algorithm may drastically reduce the total computation time. For example, the implicit numerical integration method developed by Porsching et al. [6] for integrating "stiff" differential equations describing the dynamic behaviour of hydraulic networks runs much faster than the explicit integration methods. However, it may still run slower than the computational requirements of a particular large-size problem such as real-time requirements for training simulators. Also, it may not always be possible to develop faster and faster algorithms for a particular problem.

6. Use of computers employing faster hardware.

In the past, almost seven orders of improvement in the computing speed of serial computers has come largely from advances in electronic technology from relays to vacuum tubes to discrete solid state circuits to integrated microelectronics. Much of this speed improvement has been achieved due to the fact that the electrons have to travel smaller distances with the reduction in the feature size of the devices. Unfortunately, this trend in the improvement of computer speed by the use of smaller and smaller devices is beginning to level off. It is becoming increasing difficult and less cost-effective to achieve faster speeds by way of miniaturization of computing devices. Present-day state-of-the-art commercial VLSI chips employ 1.25 micron feature size. Chips employing 0.8 micron feature size are just emerging from research laboratories [7]. The problem of further scaling down these MOS VLSI devices has been studied by researchers and it has been observed that scaling down the devices by a factor α results in the following difficulties [8]:

(i) Communication delays within the chip do not scale. As a result, the total computing time begins to be dominated by the delay in communicating the information rather than by the processing of the information by the devices.

(ii) Logic levels within the chip are scaled-down by a factor α due to the need to scale-down voltages by a factor α. Consequently, the noise margin is considerably reduced.

(iii) Current density in the wires is increased by α thus affecting the reliability of the devices.

(iv) Transistor off-resistance becomes very low which results in unnecessary drainage of power under off conditions.

Some of the above scaling difficulties are serious and fundamental in nature rather than temporary technology related problems.

One of the other approaches to achieving faster computers is the use of faster materials for fabricating devices such as GaAs (Gallium Arsenide) instead of the slower silicon. At the gate level, GaAs is about 7 to 8 times faster than silicon due to the higher mobility of electrons in GaAs [9]. However, this speed advantage is lost to some extent when one considers complete systems due to the relatively higher off-chip communication costs. In fact, it has been reported by one company [10] which was involved in the design of 32-bit microprocessors for GaAs technology that the improvement in speed in the GaAs-based design was only by a factor of 3. The situation was found to be even worse in the area of coprocessor design where a speedup by a factor of only 2 was reported. Also,

GaAs substrates are two orders of magnitude more expensive than silicon substrates. The other disadvantages of GaAs-based design are low yield, and difficulties in achieving VLSI levels of integration [10].

7. Development of optical digital computers.

Another approach which is being pursued by a number of research laboratories around the world is the development of optical digital computers. However, this research is still at a very early stage of development [11]. An efficient implementation of even an optical on-off switch analogous to the bistable transistor in microelectronic technology does not yet exist.

It is, therefore, clear from the above discussion that the only hope of handling such problems in the near future seems to be the introduction of parallelism to the sequential von Neumann machine where a number of processors cooperate and coordinate among themselves to solve a single problem in the shortest possible time. Here, two main approaches are possible: one of them referred to as SIMD (Single-Instruction Stream Multiple-Data Stream) mode of operation employs data level parallelism [12] where the same instruction is executed on different processors but with different data elements. Two examples of computers employing this mode of operation are the Connection Machine [12] and ILLIAC IV [13]. This mode of computation is suitable for problems involving highly structured computations on a very large number of data values such as image processing. It is not suitable for problems having irregular computational structure such as the hydraulic network problem studied in this thesis.

Another mode of parallel operation which is also usually classified as SIMD mode is the pipelined vector computing carried out on vector computers such as Cray, CDC

Cyber-205 etc. The problem with this mode of parallel computation is that good performance is heavily dependent on how much of the total computation can be vectorized. For example, a one-pipe Cyber-205 vector computer has 100 MFLOPS (Millions of Floating-Point Operations per Second) as the peak vector speed, 3.3 MFLOPS as the scalar speed [14] and if a problem with even 75 percent fully vectorized computations is run on this computer, then it is easy to show (Chapter 4) that the effective speed delivered by the machine for this problem is 12 MFLOPS which is only 12 percent of the peak rated speed. This degradation in performance is mainly due to the wide gap between the scalar and vector speeds of such vector computers. Furthermore, the computations that have been vectorized may not have long enough vector lengths to be comparable with the half-performance vector length $n_{1/2}$ (100 for CDC Cyber 205) of the computer [15]. This further reduces the effective speed of vector computers.

In the recent past, the solution of such sparse network problems has been tried on vector computers (for example, RELAP5 program on Fujitsu FACOM VP-100) but the implementations have tended to be inefficient [16]. Problems analogous to hydraulic network dynamics simulations such as electrical network dynamics simulation have also been tried on vector computers (for example SPICE program on Cray-1) and hardware utilization of 12 to 15 percent has been reported [17].

The other parallel processing approach, and one which has been employed in this thesis, involves partitioning the total computations into a number of almost independent tasks to be run on separate processors. This mode of operation is termed as MIMD (Multiple-Instruction Stream Multiple-Data Stream) mode of operation and it does not require vectorization of computations to achieve good performance. This mode of operation

has become all the more attractive and economically viable with the introduction of very powerful 32-bit microprocessors. Some of these newer 32-bit microprocessors such as Fairchild's "Clipper" architecture and Inmos's "Transputer" model T800 [18] even have on-chip floating-point operations capability in hardware. Once the initial design costs are recovered by their manufacturers, prices of these 32-bit processors are likely to fall substantially as has happened in the past. Thus, this mode of computation would be even more economically viable in the future.

1.2. Problem Statement

The study reported in this thesis is concerned with the development of efficient parallel algorithms suitable for running on MIMD computers for the "fast" transient simulation of thermo-hydraulic network dynamics which involves numerical integration of a large set of nonlinear, time-varying, "stiff", first-order ordinary differential equations of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$. This problem belongs to a class of problems referred to as "initial value problems", i.e., given the initial condition of the network at time $t = 0$, determine the temporal behaviour on an time interval $[0,t_{end}]$ of the network flowrates, masses, pressures, and energies satisfying conservation equations of momentum balance, mass balance, and energy balance. A system of equations is termed as "stiff" if it has widely separated time constants (or eigenvalues) embedded in it. This is further discussed in Chapter 2.

1.3. Approach

1.3.1. Choice of the algorithm

There are two broad classes of numerical integration algorithms for integrating systems of differential equations of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$ [19]. One class of algorithms

referred to as explicit integration algorithms belong to conditionally stable algorithms. They are called explicit algorithms because the values of the variables to be computed at the next time step is explicitly given in terms of the information available at the previous time steps. These algorithms are stable only if the integration time-step size is chosen sufficiently small in order to satisfy stability criterion. For stiff problems, this choice of step size based on stability criterion may be much smaller than required to satisfy accuracy requirements. Therefore, the explicit integration algorithms are not computationally efficient for solving stiff problems.

The other class of integration algorithms referred to as implicit integration algorithms have better stability characteristics as compared to explicit algorithms. For example, Euler's backward difference formula which is an implicit integration algorithm is unconditionally stable for stable linear system of equations for any integration step size. However, the use of the implicit integration methods invariably requires the solution of a system of linear algebraic equations at each time-step. Porsching et al. [6] have developed an integration scheme based on Euler's backward difference formula for integrating thermo-hydraulic network differential equations with considerably reduced size of the system of linear algebraic equations to be solved at each time-step. They report a speed improvement by a factor of 22 for an 8-node and 8-link network example. For this reason, this serial algorithm was chosen for parallelization in this thesis.

1.3.2. Choice of parallel model of computation

MIMD mode of parallel computation employing a master/slave configuration of processors, a simple time-shared bus with broadcast facility as the inter-connection network among the processors, and distributed communication memories for data communication

among the processors has been employed in the development of parallel algorithms. In other words, a message-passing approach rather than a shared memory approach for data communication has been employed.

In true MIMD mode of operation, processes synchronize locally among themselves without any global synchronization at a synchronization point. Data communication among them is also asynchronous and non-deterministic as far as the use of communication resources such as the bus is concerned. This can give rise to contentions and even deadlocks for the use of common resources. Moreover, true MIMD programs are more difficult to write and debug quickly due to the asynchronous nature of computations [20]. Intermediate break-points are difficult to use during the debugging process because the intermediate results are different each time the same program is run. Therefore, a modified form of the true MIMD operation, termed as quasi-MIMD mode of operation by Hoshino [20], was used in developing the algorithms. This mode of parallel computation was first described by Kober et al. [21] and Wallach [22] and has also been suggested by Tuazon et al. on the Hypercube multi-computer system [23].

There are three distinct phases of computation in the quasi-MIMD mode of operation [21] :

(i) Autonomous or computation phase where all processors are busy performing computations asynchronously using their own programs and data stored in their local memories. For some problems, all the processors may be running the same program but with different data which may be termed as quasi-SPMD (Single Program-Multiple Data stream) mode of operation.

(ii) Control or synchronization phase where all processors wait for the slowest processor to finish. This is achieved by "anding" the synchronization signals from each slave processor to the master processor. Synchronization is very fast as it is done by hardware rather than by software.

(iii) Communication phase where the master processor reads shared data from communication memories of slave processors and redistributes it to the processors as per the algorithm requirements. Here, slave processors do not send any data explicitly to any other processor.

The above quasi-MIMD mode of computation has the following advantages:

1. Since the communication is stream-lined, contentions and deadlocks for resources such as the bus cannot occur. Therefore, no time-consuming arbitration scheme is necessary.

2. Programs are easier to write and debug due to the deterministic nature of computations. Break-points can be used at the various intermediate global synchronization points to debug the programs.

3. Since the global synchronization is performed by the hardware by "anding" of slave processor synchronization signals, faster synchronization is achieved as compared to synchronization using software techniques such as the use of semaphores, etc. For example, in "PAX" model 64-J parallel processor system which employs quasi-MIMD mode of operation, global synchronization is achieved in 18.2 micro-seconds [20].

4. An added advantage is that analytical complexity results for the developed parallel algorithms similar to the complexity results for serial computers can be derived.

Thus, the performance of the parallel algorithms can be studied analytically as a function of the problem input data size without the need to carry out costly simulations.

The only requirement for good performance is that the computation time for all the processors should be about the same so that the processors do not sit idle for too long for the slowest processor to finish processing. However, data-dependencies in the Porsching's integration algorithm are such that all the processors have to wait for each other for data communication at most of the intermediate stages of the algorithm before they can proceed further with the computations. Therefore, very little extra performance can be achieved by employing true MIMD mode of computation.

The main disadvantage of this mode of computation using a simple time-shared bus as the inter-connection network is that the data communication steps in the algorithm are serialized. However, for this problem, the effect of serialization of data communication on the performance of the developed parallel algorithms is not very serious as can be seen from the speed-up and efficiency plots given in Chapter 5. This is mainly due to the following reasons:

1. The computations in the algorithms have been partitioned among the processors in such a way that a significant fraction of total data is local to the processors thus minimizing data to be communicated among the processors.
2. The granularity of computations of the parallel algorithms is coarse enough to ensure that the overhead of data communication and synchronization does not dominate the total processing time. In some of the algorithms for solving a matrix equation, this granularity of computation does become quite low. In such cases, two-dimensional

mesh-connected processors are used as a peripheral device connected to the bus in order to minimize data communication traffic on the bus.

3. The distance between any two processors connected by a bus is always the same and equal to unity. Hence, the data communication time for communicating a variable between any two processors is the same (equal to unity) as compared to variable communication time in other static networks such as near-neighbourhood mesh, hyper-cube etc. and $\log_2 N$ communication time for dynamic multi-stage interconnection networks such as Omega network where N is the number of processors.

4. On a time-shared bus, it takes the same unit amount of time whether we send data to a single processor, more than one processor, or all the processors (data broadcast). On the other hand, in the case of other static networks such as two-dimensional mesh, hyper-cube, omega network etc., it takes more than unity time to send the same data to more than one processor. The algorithms discussed in this thesis employ extensive partial and full data broadcast.

5. In the case of the bus, it is possible to use 32-bit wide data paths whereas 1-bit, 4-bit, or 8-bit wide data paths are generally employed for concurrent networks in order to keep the wiring to a manageable level. For example, hypercube Mark I multi-computer system developed at Caltech and Intel Inc. uses bit serial communication lines [24].

Apart from the above performance considerations, a simple bus is much cheaper to build than concurrent networks such as Omega network, specially for connecting a relatively large number of processors [25].

Of course, concurrent networks are the only choice when thousands of processors are to be inter-connected for problems requiring no or little data broadcast in their solution algorithm as their concurrent communication capability more than compensates for their drawbacks when the number of processors is large. In the algorithms reported in this thesis, the number of processors directly connected to the bus does not exceed 150 or so.

The above ideas are discussed further in Chapter 4.

1.4. Overview

The material in this thesis is organized as follows:

Chapter 2 gives the formulation of differential equations which govern the dynamics of thermo-hydraulic networks and describes some of the numerical integration methods for integrating these differential equations. Porsching's numerical integration scheme is then described in detail.

Chapter 3 discusses the performance of the Porsching's algorithm on conventional serial computers. It is shown that a major computational effort in this algorithm is the solution of the matrix equation $\mathbf{A}\,\Delta\mathbf{W} = \mathbf{z}$ for computing network flow rate increments ΔW.

Chapter 4 gives a critical review of some of the parallel architectures and inter-connection networks and describes a simple parallel architecture and its mode of operation for developing parallel algorithms described in Chapters 5 and 6.

Chapter 5 the parallelization of Porsching's algorithm is studied. Two direct parallel methods for solving the matrix equation $\mathbf{A}\,\Delta\mathbf{W} = \mathbf{z}$ are

described. The speed-ups and efficiencies of parallel processing as a function of hydraulic network size, data communication time, and processor synchronization time are studied.

Chapter 6 gives a parallel iterative method for solving the matrix equation $\mathbf{A}\,\Delta\mathbf{W} = \mathbf{z}$ which is preferable to direct methods for simulating large-size networks.

Finally, conclusions and recommendations for further work in this area are given in Chapter 7.

CHAPTER 2

NUMERICAL INTEGRATION OF NETWORK DIFFERENTIAL EQUATIONS

2.1. Introduction

In this chapter, the differential equations governing the dynamic behaviour of thermo-hydraulic networks are described. Algorithms available for the numerical integration of these equations and their stability properties and local truncation errors are discussed briefly. An efficient integration algorithm developed by Porsching et al. which is based on Euler's backward difference formula is described.

2.2. Terminology and Definitions

2.2.1. Hydraulic network

A hydraulic network consists of a finite number of nodes interconnected by links for the transportation of fluid mass and heat energy from one part of the network to the other part. Physically, it is an interconnection of pressurized fluid vessels and pipes.

Figure 2.1 shows a typical hydraulic network. Squares in the network represent nodes and the directed arcs represent links with the assumed flow in the direction of the arrows. Each node i has associated with it a fluid mass M_i, fluid internal energy U_i and pressure P_i. Similarly, each link k has associated with it a fluid mass flow rate W_k. This is shown in Fig. 2.2.

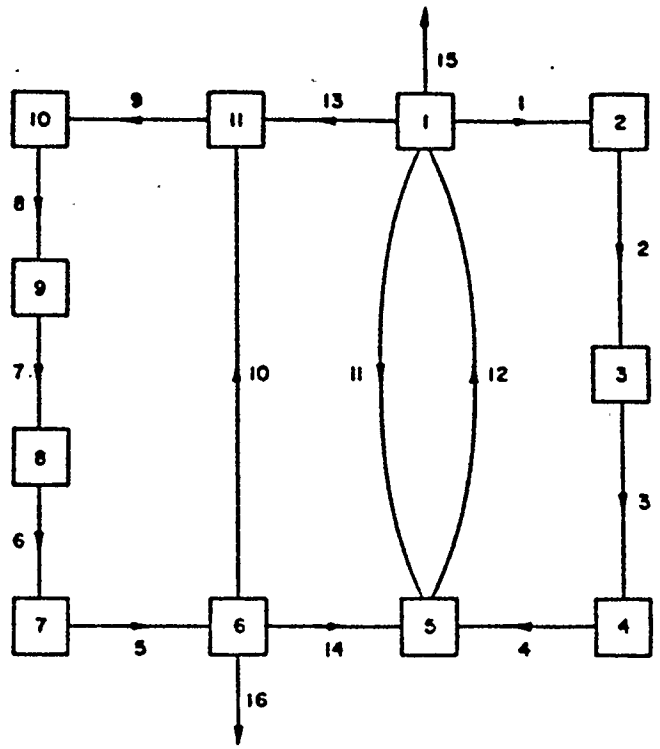


Figure 2.1 A typical hydraulic network with 11 nodes and 16 links [6].

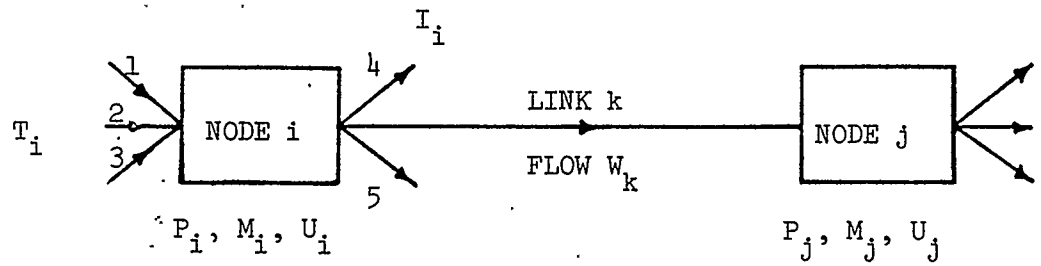


Figure 2.2 Quantities associated with a node and a link.

2.2.2. Initial and terminal nodes

A node is termed as the initial node for a link if the link initiates from that node. In Fig. 2.2, node i is the initial node for the link k.

A node is termed as the terminal node for a link if the link terminates at that node. In Fig. 2.2, node j is the terminal node for the link k.

2.2.3. Initiating and terminating links

A set of links I_i is termed as the initiating links if they initiate from node i.

A set of links T_i is termed as the terminating links if they terminate at node i.

In other words, node i has associated with it two index sets, I_i and T_i. The set I_i is the set of links for which i is the initial node and T_i is the set of links for which i is the terminal node. For example, in Fig. 2.2,

$$T_i = \{1, 2, 3\}$$

$$I_i = \{4\ 5, k\}$$

2.2.4. Noncritical and critical links

There are two kinds of flow paths in a hydraulic circuit. A link is termed as a non-critical or normal link if it has an initial and a terminal node. The flow in this case is governed by the one-dimensional momentum balance equation of the form:

$$\dot{W}_k = f_k\ (t, P_i, P_j, W_k)$$

where:

W_k = the mass flow rate in the link

P_i,P_j = Pressures at the initial and terminal nodes i and j respectively of link k

$f_k =$ a general nonlinear function

A noncritical link k having nodes i and j as initial and terminal nodes respectively is notationally represented as $k \rightarrow (i,j)$.

Critical links are geometrically similar to noncritical links except that they do not have terminal nodes. Therefore, the flow in the critical link depends only on the pressure of the initial node and can be determined by an algebraic equation of the form:

$$W_k = g_k\ (P_i)$$

where:

$P_i =$ the initial node pressure

$g_k =$ an algebraic function

Critical links are used to simulate a choked flow condition or flow originating external to the network.

In Fig. 2.1, links 15 and 16 are critical links as they do not have a terminal node. The remaining links are noncritical links.

2.2.5. Chains

A subnetwork connecting nodes i and j, $i \neq j$, is defined to be a chain of length p if it contains exactly $p + 1$ nodes (including nodes i and j) and if, for any node $l \neq i, j$, the sum of index sets T_l and I_l is exactly equal to two.

The network shown in Fig. 2.1, contains a chain of length 4 joining nodes 1 and 5 and one of length 5 joining nodes 6 and 11. Physically, a chain may represent a single vessel which is conceptually divided into a number of sections, each represented by a node

where the whole mass of the section is assumed to be concentrated. Two adjacent imaginary sections of the vessel are assumed to be connected by a link of length equal to the distance between the centres of these sections and cross sectional area equal to the area of the vessel at these two sections.

2.3. Network Conservation Equations to be Integrated

The dynamic behaviour of a thermo-hydraulic network is governed by a system of nonlinear, time-varying, coupled, first-order ordinary differential equations which account for the conservation of momentum, mass and energy of the fluid in the network.

Symbols used in the network conservation equations are first defined as follows:

i,j - node indices

k, ν - link indices

$1,2,\ldots,K$ - noncritical links in the network

$K+1,K+2,\ldots,K^*$ - critical links in the network

$1,2,\ldots,N$ - nodes in the network

$\sum_{\nu \varepsilon T_i}$ - summation over links which terminate in node i

$\sum_{\nu \varepsilon I_i}$ - summation over links which initiate from node i

M_i - Mass of node i

U_i - total internal energy of node i

P_i - Pressure of node i

W_k - Mass flow rate in link k

H_v - Specific enthalpy of the fluid in link v entering node i

H_v^* - Specific enthalpy of the fluid in link v leaving node i

ρ_k - mass density of fluid in link k

L_k - length of link k

A_k - flow area of link k

D_k - hydraulic diameter of link k

ΔZ_k - elevation change of link k

$(fr)_k$ - friction factor for the link k

g - acceleration due to gravity

Q_i - external heat input rate into node i

The basic network equations are [6]:

(i) Conservation of momentum equation for the mass flow in the normal or noncritical link $k \rightarrow (i,j)$:

$$\dot{W}_k = f_k\ (t, P_i, P_j\ W_k),\ k = 1, 2, ..., K \tag{2.1}$$

For a single phase flow, function f_k is given by

$$\dot{f}_k = \frac{A_k}{L_k}\left[(P_i - P_j) - \frac{(fr)_k\ L_k}{D_k}\ \frac{W_k\ |W_k|}{2\ \rho_k\ A_k^2} - g\ \Delta Z_k \rho_k\right] . \tag{2.2}$$

This equation is derived from the Newton's second law of dynamics which states that the rate of change of momentum is equal to the externally applied force to the body.

(ii) The flow in a critical link is a function of the initial node pressure only and is given by an algebraic relationship:

$$W_k = g_k\ (P_i),\ \ k = K + 1, K + 2, ..., K^* \tag{2.3}$$

where g_k is a general nonlinear function.

(iii) energy balance at each node i:

$$\dot{U}_i = \sum_{v \varepsilon T_i} H_v\, W_v - \sum_{v \varepsilon I_i} H_v^*\, W_v + Q_i,\ i = 1, 2, ..., N \tag{2.4}$$

i.e. the rate of change of internal energy at a node is equal to energy entering the node per unit time minus the energy leaving the node per unit time.

(iv) Mass balance at each node:

$$\dot{M}_i = \sum_{v \varepsilon T_i} W_v - \sum_{v \varepsilon I_i} W_v,\ \ i = 1, 2, ..., N \tag{2.5}$$

i.e. rate of change of mass at a node is equal to mass entering the node per unit time minus mass leaving the node per unit time.

(v) Equation of State:

$$P_i = \Pi_i\ (U_i, M_i),\ i = 1, 2, ..., N \tag{2.6}$$

where Π_i is a nonlinear function of U_i and M_i.

This equation represents the fact that the pressure at a node is not a separate variable to be integrated but is a function of internal energy and mass (more precisely specific internal energy and mass density) of the node. Function Π_i in (2.6) is generally given in the form of steam tables. If the index set I_i for a node i contains critical links, then (2.3) and (2.6) create further couplings between (2.4) and (2.5), since the corresponding flow rates

W_k in the critical links are functions of U_i and M_i. Thus the system of differential equations in (2.1), (2.4) and (2.5) are highly coupled to each other.

If we let,

$$\mathbf{y} = \text{column } [y_1, y_2, \ldots, y_{K+2N}]$$

$$= \text{column } [W_1, \ldots, W_K, U_i, \ldots, U_N, M_i, \ldots, M_N]$$

and $\mathbf{f} = \text{column } [f_1, f_2, \ldots, f_{K+2N}]$,

where the components f_i are the right-hand sides of eqns. (2.1), (2.4) and (2.5), then the conservation equations may be written in vector form as

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t) \quad (2.7)$$

In writing (2.7), it is assumed that (2.3) and (2.6) have been used to replace the flow rates in critical links by equivalent expressions in terms of masses and energies of the link's initial nodes and P_i,P_j in (2.1) are taken as functions of U_i,M_i and U_j,M_j respectively. By doing this, the right hand sides of (2.1), (2.4) and (2.5) become functions of W, U, and M only. The system of differential equations in (2.7) are:

(1) nonlinear and coupled (due to the presence of nonlinear and coupling terms)

(2) have time-varying coefficients (such as the presence of H_v in (2.4))

(3) stiff (due to the presence of widely separated time-constants)

The problem to be considered here is the following. Given an initial condition of the network at $t = 0$, determine the transient behaviour on an time interval $[o - t_f]$ of the network flow rates W_k, masses M_i and energies U_i satisfying (2.1) to (2.6). This is the well-known initial-value problem in the area of numerical integration of differential equations [19].

2.4. Numerical Integration Algorithms

There is a large body of knowledge concerning the numerical integration of systems of differential equations of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$. Broadly speaking, they can be classified into two main categories: one-step methods and multi-step methods. One-step methods are based on the Taylor series expansion of the function y around the point (y_n, t_n) to compute the value y_{n+1} at time-step $(n + 1)$. They are called one-step methods because they make use of the value of the variable y and its derivatives only at the last time-step t_n. On the other hand, multi-step methods make use of the value of the variable y and its first derivative at a number of time-steps. If the first derivatives used are at the time-step t_n or earlier, the methods are called explicit multi-step methods. On the other hand, if the derivative at the time-step t_{n+1} (which is not yet available but can be computed using Newton-Raphson or any other iterative method) is used in computing y_{n+1} , the method is called implicit multi-step method. Although implicit methods involve more computational effort at each time-step as compared to explicit methods of the same order, they are more stable than the explicit methods and therefore allow a much larger time-step size h to be chosen without making the integration method unstable. In certain problems, called "stiff", the increase in step size is truly dramatic, i.e., of the order of 1000 times or even more [33], depending upon the ratio of the largest time constant to the smallest time constant in the problem.

Examples of one-step integration methods are Euler's forward difference method and explicit Runge-Kutta methods. All one-step methods belong to the category of explicit integration methods. Examples of explicit multi-step methods are the family of Adams-Bashforth methods and examples of implicit multi-step methods are Euler's backward

difference method, the trapezoidal method, etc.

The above classification of integration methods is summarized in Fig. 2.3. A brief description of some of these numerical integration methods and their accuracy and stability properties is discussed below. Although, the methods are described for the scalar case $\dot{y} = f(y,t)$, they apply equally well to a system of differential equations $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$.

2.4.1. One-step methods

These methods are based on the Taylor series expansion of a function y about the point (y_n,t_n) to obtain the value of y at $t = t_{n+1}$, i.e. y_{n+1}. The p^{th} order Taylor algorithm contains $(p + 1)$ terms of Taylor series expansion and is given by

$$y_{n+1} = y_n + \frac{h}{1!} f(y_n,t_n) + \frac{h^2}{2!} f^{(1)}(y_n,t_n) + \cdots + \frac{h^p}{p!} f^{(p-1)}(y_n,t_n) \qquad (2.8)$$

Here, h is the time-step size, the subscript n on y and t denotes their values at time step n, the superscript on h denotes the power of h and the superscript within the small brackets on f denotes the order of the total derivative of f with respect to time t, i.e., $f^{(1)}(y_n,t_n)$ is given by

$$f^{(1)}(y_n,t_n) = \frac{df}{dt}(y_n,t_n) = \left[\frac{\partial f(y,t)}{\partial y} \cdot \frac{dy}{dt} + \frac{\partial f(y,t)}{\partial t} \cdot \frac{dt}{dt}\right]_{\substack{at \\ t=t_n}}^{y=y_n}$$

$$= \left[\frac{\partial f(y,t)}{\partial y} \cdot f(y,t) + \frac{\partial f(y,t)}{\partial t}\right]_{\substack{at \\ t=t_n}}^{y=y_n}$$

As can be seen from the above equation, computation of y_{n+1} makes use of information at time step t_n only, hence, the name one-step method.

It can be shown [26] that the local truncation error ε_n at time-step t_n of the p^{th} order Taylor's algorithm is of the order of h^{p+1}, i.e.

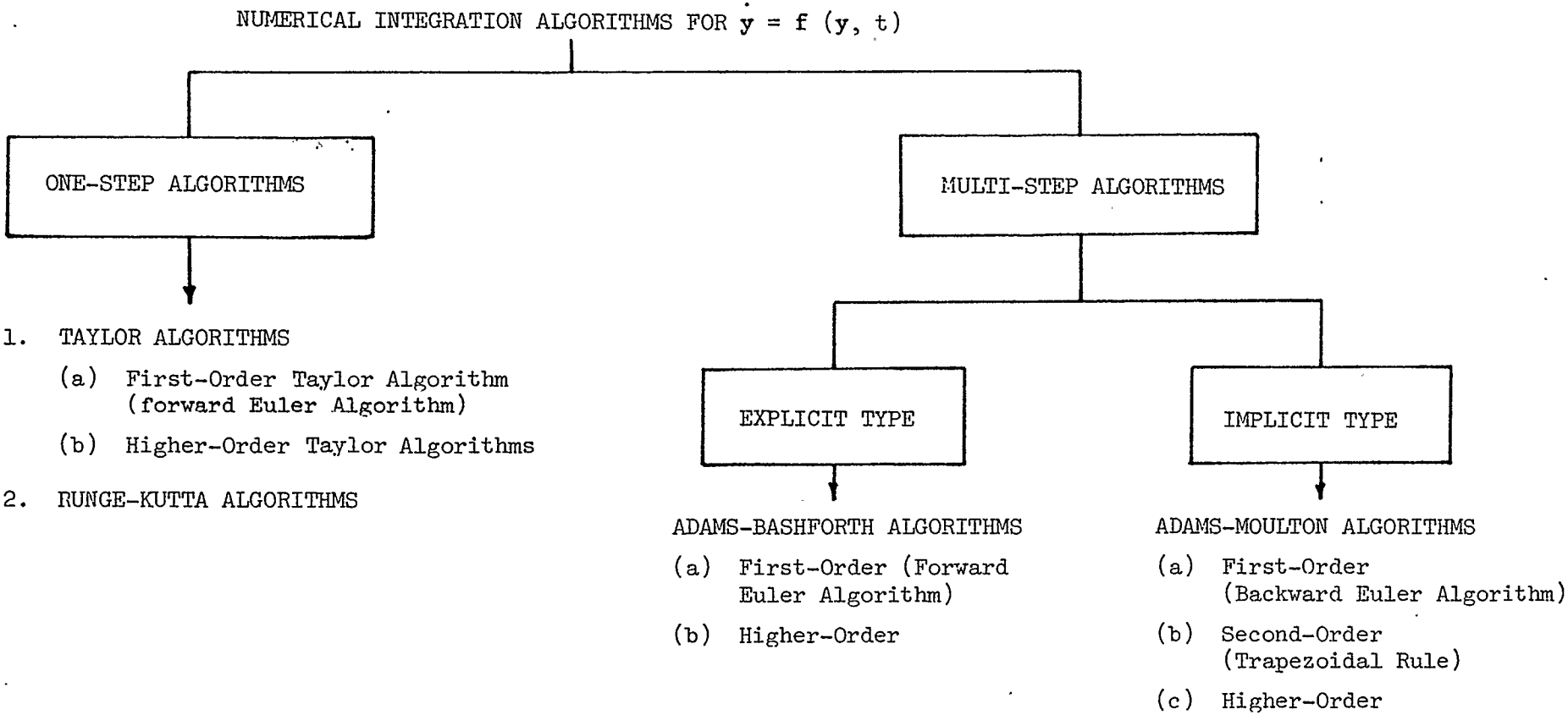


Figure 2.3 Classification of numerical methods for integrating $\dot{y} = f(y,t)$

$$\varepsilon_n = 0(h^{p+1}) \quad .$$

Also, since the local truncation error decreases as the $(p+1)$th power of step-size h, the algorithm is called p^{th}-order algorithm. Moreover, all the one-step methods given by the Taylor algorithm for different values of p are consistent since the local truncation error tends to zero as h tends to zero.

When $p=1$, we get the first-order Taylor algorithm given by

$$y_{n+1} = y_n + h\ f\ (y_n, t_n) \quad . \tag{2.9}$$

This corresponds to the first two terms of a Taylor series expansion. Equation (2.9) is also called the forward Euler algorithm. The local truncation error of this integration algorithm is proportional to h^2, i.e. $\varepsilon_n = 0(h^2)$.

An analysis of stability properties of the forward Euler algorithm is now given. Although the algorithms discussed here are intended for obtaining numerical solutions of a system of nonlinear differential equations of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},\mathbf{t})$, it is standard practice to apply each algorithm to the "test equation"

$$\dot{y} = f(y) = -\lambda y \tag{2.10}$$

associated with a first-order linear system with time constant $\frac{1}{\lambda}$. The exact analytical solution to (2.10) is given by

$$y(t) = y_o\ e^{-\lambda t} \quad , \quad t \geq 0 \tag{2.11}$$

where $y_o = y(0)$ is the initial condition. The reason for choosing (2.10) as the test equation is not only because it has an exact analytical solution to which other numerical solutions can be compared, but also because, incrementally speaking, most solutions to a

differential equation can be approximated by a portion of an exponential. Therefore, if an algorithm fails to solve (2.10), it is very likely that it will also fail when applied to other differential equations.

Now applying forward Euler's method given by (2.9) to test equation (2.10),

$$y_{n+1} = y_n + h(-\lambda\, y_n)$$

or

$$y_{n+1} = (1 - h\lambda)\, y_n \tag{2.12}$$

substituting $n = n - 1$ in (2.12), we have

$$y_n = (1 - h\lambda)\, y_{n-1}$$

substituting the above value of y_n in (2.12)

$$y_n = (1 - h\lambda)^2\ y_{n-1}$$

Repeating the above sequences, we have

$$y_{n+1} = (1 - h\lambda)^n\ y_o$$

for the solution to decay with time,

$$|1 - h\lambda| < 1 \tag{2.13}$$

or

$$0 < h\lambda < 2 \quad , \quad for\ \ real\ \ \lambda.$$

Therefore, the algorithm to be stable, time-step size h should be less $\frac{2}{\lambda}$, i.e. less than twice the time constant. If the value of the time constant is very small, time-step size required to maintain stability will be also very small. Therefore, for small time constant

problems, the choice of step size is governed by stability considerations rather than accuracy considerations. For complex values of λ, (2.13) represents a circle of unity radius and centre (1, 0) in $h\lambda$ plane. The same results can also be obtained by using the z-transform technique by substituting the solution $y_n = cz^n$ in the difference equation (2.12) giving

$$z^{n+1} = (1 - h\lambda)\, z^n$$

or

$$z = (1 - h\lambda) \cdot \tag{2.14}$$

For this algorithm to be absolutely stable, the set of values $\sigma = h\lambda$ should be such that the roots of z in (2.14) lie within or on the unit circle $|z| = 1$ in the z-plane so that the solution $y_n = cz^n$ does not grow with time. The set of values for which the integration method is absolutely stable is called the region of absolute stability. This mapping for $h\lambda$ for the forward Euler Algorithm is obtained by substituting $z = e^{j\theta}$ in (2.14) and solving for $\sigma = h\lambda$. This gives $\sigma = h\lambda = 1 - e^{j\theta}$ which is the interior area bounded by the circle of radius unity centred at (1, 0) in the $\sigma = h\lambda$ plane and is shown in Fig. 2.4.

Application of forward Euler algorithm to the hydraulic network differential equations gives the following set of integration formulas. For the sake of notational convenience for dealing with the system of differential equations, subscript n, $(n+1)$ etc. have been moved to superscript positions, i.e. y^n instead of y_n represents the value of y at time step n.

$$W_k^{n+1} = W_k^n + h\, f_k\, (t^n, p_i^n, p_j^n, W_k^n),\ k = 1, 2, \ldots, K \tag{2.15}$$

$$U_i^{n+1} = U_i^n + h \left[\sum_{v \in T_i} H_v^n\, W_v^n - \sum_{v \in I_i} H_v^{*n}\, W_v^n + Q_i^n \right],\ i = 1, 2, \ldots, N \tag{2.16}$$

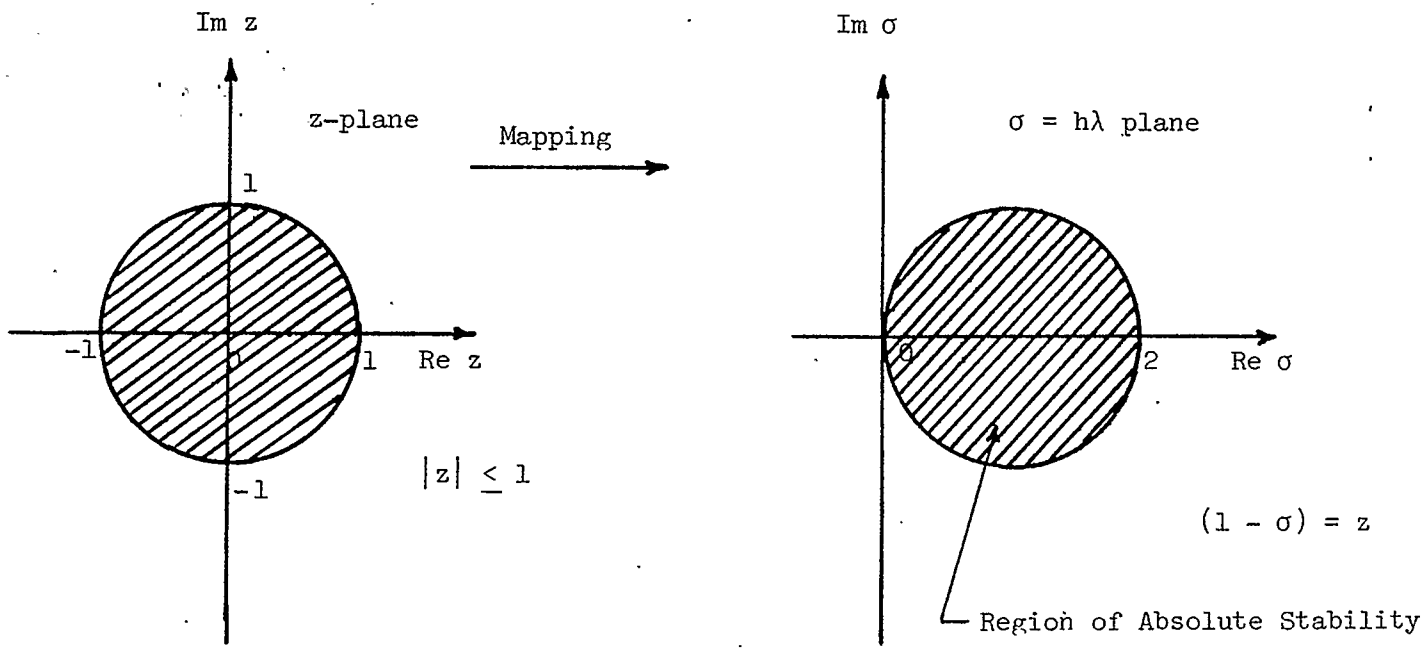


Figure 2.4 Region of absolute stability for forward Euler Method.

$$M_i^{n+1} = M_i^n + h\left[\sum_{v\varepsilon T_i} W_v^n - \sum_{v\varepsilon I_i} W_v^n\right], \quad i = 1, 2, ..., N \tag{2.17}$$

$$P_i^{n+1} = \Pi_i\ (U_i^{n+1}, M_i^{n+1}) \quad , \quad i = 1, 2, ..., N \tag{2.18}$$

From the point of view of concurrent processing, the set of equations (2.15), (2.16) and (2.17) are decoupled from each other and therefore, can be computed in parallel. After these computations, computations in (2.18) can also be performed concurrently.

If we include more terms from (2.8), we get higher-order Taylor algorithms with accompanying smaller local truncation errors. However, this will also require computation of higher-order derivatives of $f(y,t)$ which may be very time-consuming for complicated functions. Moreover, it is extremely error-prone, especially if $f(y,t)$ is not available in explicit analytic form. For this reason, higher order Taylor algorithms are generally not used. Instead, a modification of Taylor algorithm called Runge-Kutta algorithm, which avoids computation of higher-order derivatives of $f(y,t)$ but still retains the same order of accuracy is commonly used. The basic idea [26] consists of replacing the second and higher terms in (2.8) by another function $hK^p(y_n, t_n, h)$ requiring no partial derivatives of $f(y,t)$ such that

$$\left|\frac{h}{1!} f(y_n,t_n) + \frac{h^2}{2!} f^{(1)}(y_n,t_n) + \cdots + \frac{h^p}{p!} f^{(p-1)}(y_n,t_n) - hK^p(y_n,t_n,h)\right| \le R\ h^p$$

where R is some constant independent of h. In view of the above, the modified algorithm

$$y_{n+1} = y_n + h\ K_p(y_n,t_n,h) \tag{2.19}$$

has precisely the same order of magnitude of truncation error as the corresponding Taylor's algorithm. Therefore, (2.19) is called the p^{th}-order Runge-Kutta algorithm. For example, for $p = 4$, we get the fourth-order Runge-Kutta algorithm given by

$$y_{n+1} = y_n + h\ K_4(y_n, t_n, h) \tag{2.20}$$

where

$$K_4(y_n, t_n, h) = \frac{1}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]$$

$$k_1 = f(y_n, t_n)$$

$$k_2 = f\left(y_n + \frac{h}{2}\ k_1,\ t_n + \frac{h}{2}\right) \tag{2.21}$$

$$k_3 = f\left(y_n + \frac{h}{2}\ k_2,\ t_n + \frac{h}{2}\right)$$

$$k_4 = f(y_n + h\ k_3, t_n + h)$$

The local truncation error of the algorithm is $0(h^5)$ but requires the evaluation of function $f(y,t)$ four times per time-step which is undesirable particularly for complicated functions.

In order to determine the region of absolute stability in the $h\lambda$-plane of the fourth-order Runge-Kutta method consider the test equation $\dot{y} = f(y) = -h\lambda$. After computing the values of k_1, k_2, k_3, and k_4 from (2.21) and substituting in (2.20), we have

$$y_{n+1} = \left[1 + h\lambda + \frac{h^2\lambda^2}{2} + \frac{h^3\lambda^3}{6} + \frac{h^4\lambda^4}{24}\right] y_n$$

substituting $y_n = cz^n$ as the solution to the above difference equation, we have

$$z = \left[1 + h\lambda + \frac{h^2\lambda^2}{2} + \frac{h^3\lambda^3}{6} + \frac{h^4\lambda^4}{24}\right]$$

substituting $h\lambda = \sigma$ in the above equation, we have

$$z = \left[1 + \sigma + \frac{\sigma^2}{2} + \frac{\sigma^3}{6} + \frac{\sigma^4}{24}\right]$$

for the solution to be absolutely stable,

$$|z| \leq 1 \cdot$$

Therefore, $\left|\left[1 + \sigma + \frac{\sigma^2}{2} + \frac{\sigma^3}{6} + \frac{\sigma^4}{24}\right]\right| \leq 1$. This region of absolute stability in the $\sigma = h\lambda$ plane is shown in Fig. 2.5 [19]. As can be seen, it is only slightly larger as compared to the stability region of the forward Euler algorithm.

2.4.2. Multi-step methods

In multi-step numerical integration methods, unlike Taylor's algorithm, past information from previous time-steps is also utilized to compute y_{n+1}. The general form of the multi-step algorithms is given by the following equation.

$$\begin{aligned} y_{n+1} &= a_0 y_n + a_1 y_{n-1} + \cdots + a_p y_{n-p} + h\,[b^{-1} f(y_{n+1}, t_{n+1}) \\ &\quad + b_o f(y_n, t_n) + \cdots + b_p f(y_{n-p}, t_{n-p})] \\ &= \sum_{i=0}^{p} a_i\, y_{n-i} + h \sum_{i=-1}^{p} b_i\, f(y_{n-i}, t_{n-i}) \end{aligned} \tag{2.22}$$

These algorithms are also called polynomial approximation algorithms since they are capable of calculating the exact value $y(t_{n+1})$ of y at $t = t_{n+1}$ for any initial value problem having an exact solution in the form of a k^{th} degree polynomial in t. If the integration formula gives an exact solution for a polynomial of order less than or equal to k in t, it is called a numerical-integration formula of order k. The local truncation error ε_n of the k^{th}-order multi-step method is of the order of h^{k+1}, i.e. $\varepsilon_n = 0(h^{k+1})$ [26]. Since $\varepsilon_n \to 0$

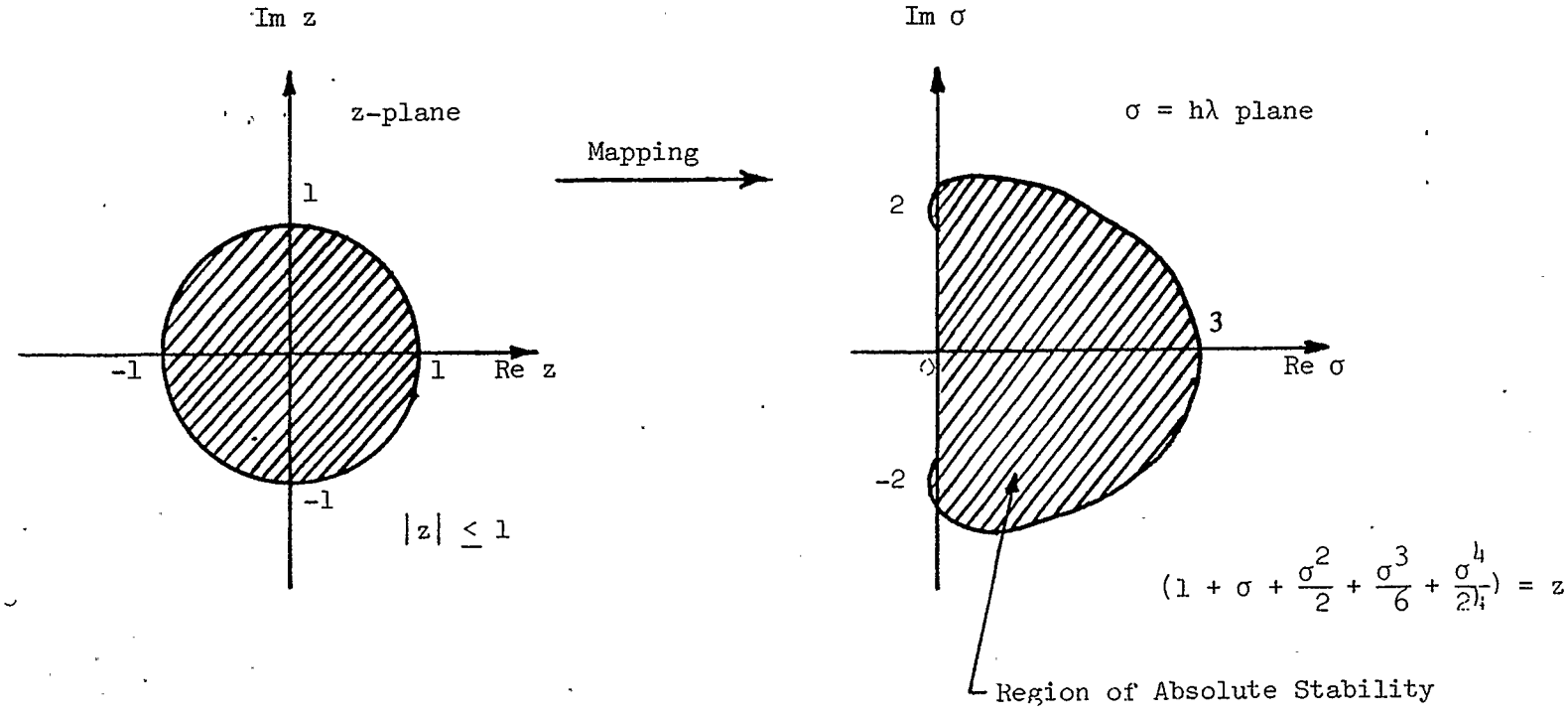


Figure 2.5 Region of absolute stability for the fourth-order Runge-Kutta Method.

as $h \rightarrow 0$, all multi-step methods are consistent.

When $b_{-1} = 0$, the integration method is called an explicit or open method and (2.22) gives y_{n+1} explicity in terms of previously determined values. When $b_{-1} \neq 0$, the method is called an implicit or closed method since y_{n+1} occurs on both sides of (2.22) and is determined only in an implicit manner such as the use of functional iteration.

An important class of multi-step explicit algorithms is called the Adams-Bashforth algorithm. The k^{th} order Adams-Bashforth algorithm is obtained by setting

$$p = k - 1, \; a_1 = a_2 = \cdots = a_{k-1} = 0, \; b_{-1} = 0$$

in (2.22), i.e.

$$\begin{aligned} y_{n+1} &= a_o \, y_n + h \, [b_o \, f(y_n, t_n) + b_1 \, f(y_{n-1}, t_{n-1}) + \cdots \\ &\quad + b_{k-1} \, f(y_{n-k+1}, t_{n-k+1})] \\ &= a_o \, y_n + h \sum_{i=0}^{k-1} b_i \, f(y_{n-i}, t_{n-i}) \; . \end{aligned} \tag{2.23}$$

The $k+1$ coefficients $a_o, b_o, b_1, \ldots, b_{k-1}$ are determined so that (2.23) is exact for all polynominal solutions of degree less than or equal to k. The coefficient a_o is always unity for all values of k in order to satisfy this condition.

When $k=1$ it can be shown that $b_o = 1$ to satisfy the above constraint and (2.23) degenerates into one-step integration formula called the forward Euler formula described in section 2.41, i.e.

$$y_{n+1} = y_n + h \, f(y_n, t_n)$$

and therefore has the same local truncation error and the same region of absolute stability as the forward Euler formula and is shown in Fig. 2.6(a).

Higher order Adams-Bashforth algorithms are obtained by using higher values of k in (2.23) and then finding the a_i and b_i coefficients by the method of undetermined coefficients such that (2.23) gives the exact solution for polynomial of degree less than or equal to k. For example, the second-order Adams-Bashforth formula is given by

$$y_{n+1} = y_n + h\left[\frac{3}{2} f(y_n, t_n) - \frac{1}{2} f(y_{n-1}, t_{n-1})\right] . \tag{2.24}$$

Its region of absolute stability for the test equation can be determined by substituting the two values of f from (2.10) and substituting $y_n = c\ z^n$ as the solution of the difference equation (2.24). After these substitutions in (2.24), we have

$$h\lambda = \sigma = \frac{-2z(z-1)}{(3z-1)} . \tag{2.25}$$

The root-locus of $h\lambda$ in the $h\lambda$-plane corresponding to the unit circle $|z| = 1$ in the z-plane can be obtained by substituting $z = e^{j\theta}$ in (2.25). Therefore,

$$h\lambda = \sigma(\theta) = \frac{-2e^{j\theta}(e^{j\theta}-1)}{(3e^{j\theta}-1)} . \tag{2.26}$$

This is plotted in Fig. 2.6(b).

As can be seen from this figure, the region of absolute stability for the second-order Adams-Bashforth formula is small compared to the corresponding region for the first-order formula. These stability regions for the higher-order Adams-Bashforth algorithms can also be similarly obtained from (2.23) after substituting the values of f from (2.10), then substituting the solution $y_n = c\ z^n$, and finally solving the resulting equation for $h\lambda$. After this, we have

$$h\lambda = \sigma(\theta) = \frac{(a_o - e^{jp\theta})}{(b_o\ e^{j(k-1)\theta} + b_1\ e^{j(k-2)\theta} + \cdots + b_{k-2}\ e^{j\theta} + b_{k-1})} . \tag{2.27}$$

These stability regions shrink monotonically as the order k of the algorithm is increased. Therefore, as the order of the Adams-Bashforth algorithms increases, its region of absolute stability decreases even though its local truncation error also decreases as $0(h^{k+1})$. As a result, for an initial-value problem having very small time constants, the step size in the Adams-Bashforth integration algorithm is determined by stability considerations rather than accuracy considerations. This is a serious limitation of all explicit integration algorithms.

This difficulty can be overcome by using implicit multi-step integration formulas obtained from (2.22) when the coefficient b_{-1} is non-zero. An important class of implicit multi-step integration algorithms, called Adams-Moulton algorithms, is obtained by setting

$$p = k - 2 \quad , \quad a_1 = a_2 = \cdots = a_{k-2} = 0$$

in (2.22). Therefore,

$$\begin{aligned} y_{n+1} = a_o\, y_n + h \Big[& b_{-1}\, f(y_{n+1},t_{n+1}) + b_o\, f(y_n,t_n) + b_1\, f(y_{n-1},t_{n-1}) \\ & + \cdots + b_{k-2}\, f(y_{n-k+2},t_{n-k+2}) \Big] \end{aligned} \tag{2.28}$$

The local truncation error of Adams-Moulton algorithm of order k is $0(h^{k+1})$. As an example, when $k = 1$, $a_o = 1$, $b_1 = 1$ with rest of the coefficients equal to zero, we get the first-order Adams-Moulton algorithm also commonly known as the backward Euler algorithm or Euler's backward difference formula,

$$y_{n+1} = y_n + h\, f(y_{n+1},t_{n+1}) \tag{2.29}$$

Since (2.29) cannot be solved explicity for y_{n+1}, it is an implicit algorithm. It is usually solved by applying one iteration of the Newton-Raphson algorithm for finding the zero of a non-linear algebraic equation [19]. This gives rise to a system of linear equations of the

form $\mathbf{Ax} = \mathbf{b}$ in the multi-variable case which are to be solved at each time step. This is further discussed in section 2.52 of this chapter. The region of absolute stability of (2.29) is given by

$$h\lambda = \sigma(\theta) = e^{-j\theta} - 1 \quad , \quad 0 \le \theta \le 2\pi \cdot$$

which is the entire $h\lambda$ plane minus the area enclosed by a unit circle centred at (−1, 0) as shown in Fig. 2.7(a).

When $k = 2$, $a_o = 1$, $b_{-1} = b_o = \frac{1}{2}$, we get the second-order Adams-Moulton formula better known as the Trapezoidal Rule and is given by

$$y_{n+1} = y_n + h\left[\frac{1}{2} f(y_{n+1}, t_{n+1}) + \frac{1}{2} f(y_n, t_n)\right] \cdot \tag{2.30}$$

The boundary of the region of absolute stability for Trapezoidal rule is given by

$$h\lambda = \sigma(\theta) = \frac{2(1 - e^{j\theta})}{(1 + e^{j\theta})} \quad , \quad 0 \le \theta \le 2\pi \cdot$$

which encloses the complete right-half of $h\lambda$ plane as shown in Fig. 2.7(b).

A comparison with Figs. 2.5 and 2.6 shows that the stability regions for the Adams-Moulton algorithms are much larger than for explicit algorithms. In fact, for any given $\lambda > 0$ the implicit first-order backward Euler Formula and the second-order Trapezoidal rule will be stable for any step size. Hence, the choice of step size h for the backward Euler and Trapezoidal algorithms is restricted only by accuracy, i.e. the maximum allowable local truncation error and not by stability considerations. In view of the above observation, the Euler's backward difference formula and the Trapezoidal rule are generally considered some of the best algorithms for solving the initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$ particularly when the problem contains a mixture of very small and very large time-constants.

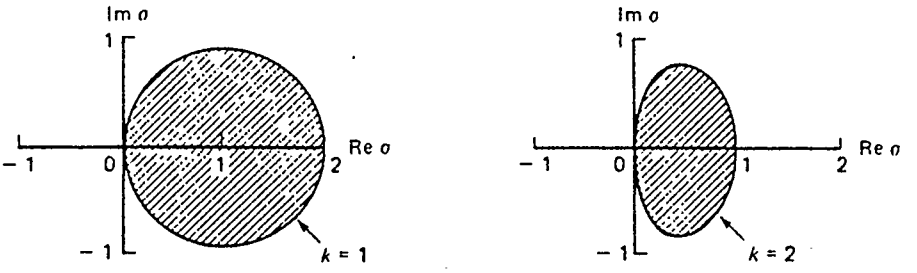


Figure 2.6 Regions of absolute stability (shaded regions) for the first and second-order Adams-Bashforth explicit integration algorithms.

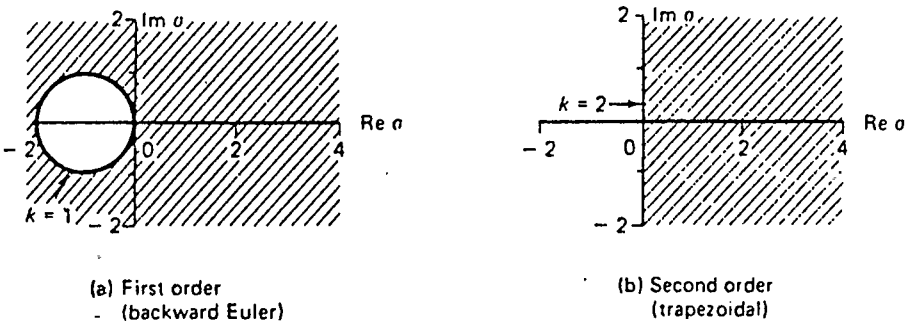


Figure 2.7 Regions of absolute stability (shaded regions) for the first- and second-order Adams-Moulton implicit integration algorithms.

2.4.3. Stiff differential equations

A system of differential equations whose solution contains both "very fast" and "very slow" components is said to be stiff.

As an example, consider the numerical integration of the following system of uncoupled linear equations

$$\begin{aligned} \dot{y}_1 &= -\lambda_1 y_1 \\ \dot{y}_2 &= -\lambda_2 y_2 \end{aligned} \tag{2.31}$$

with $\lambda_1 = 10^4$ and $\lambda_2 = 1$ so that the time constants $T_1 = \frac{1}{\lambda_1} = 10^{-4}$ sec and $T_2 = \frac{1}{\lambda_2} = 1$ sec. The exact solution is given by

$$y_1 = c_1 e^{-\lambda_1 t}$$

$$y_2 = c_2 e^{-\lambda_2 t}$$

If we integrate (2.31) by the forward Euler method, then to ensure numerical stability we must restrict the step-size to $h < \frac{2}{\lambda_1} = 2 \times 10^{-4}$ seconds for y_1, and $h < \frac{2}{\lambda_2} = 2$ seconds for y_2 as given by (2.13). For this uncoupled system, we could have solved it as two separate problems using two separate step-sizes. However, in general, the variables are coupled to one another, and instead of (2.31), we have

$$\dot{y}_1 = -a_{11}y_1 - a_{12}y_2$$

$$\dot{y}_2 = -a_{21}y_1 - a_{22}y_2 \qquad (2.32)$$

or

$$\dot{\mathbf{y}} = \mathbf{A}\,\mathbf{y}$$

where

$$A = \begin{bmatrix} -a_{11} & -a_{12} \\ -a_{21} & -a_{22} \end{bmatrix}.$$

Assume that the matrix A is such that its eigenvalues are $-\lambda_1 = -10^4$ and $-\lambda_2 = -1$ as before. It is well known [27] that equation of the form (2.32) can be transformed into the canonical or decoupled form $\dot{\mathbf{x}} = \Lambda\mathbf{x}$ by the similarity transformation $\mathbf{y} = \mathbf{P}\mathbf{x}$, where

$$\Lambda = P^{-1}\,A\,P$$

$$= \begin{bmatrix} -\lambda_1 & 0 \\ 0 & -\lambda_2 \end{bmatrix}$$

i.e. the eigenvalues of the system of differential equations of the form (2.32) remain invarient under the similarly transformation $\mathbf{y} = \mathbf{P}\,\mathbf{x}$.

P is called the permutation matrix whose columns are the eigenvectors $\mathbf{p}_i$ corresponding to the eigenvalues λ_i. Now the transformed system of equations $\dot{\mathbf{x}} = \Lambda\,\mathbf{x}$ is of the decoupled form of (2.31). Integrating the transformed system by the forward Euler algorithm will therefore lead to the same step size restrictions as before. However, since the complete solution y_1 and y_2 are linear sums of the solution of the transformed variables x_1 and x_2, i.e. $\mathbf{y} = \mathbf{P}\,\mathbf{x}$, a common step size must be chosen to be the smaller of the two, i.e. $h \leq 2 \times 10^{-4}$ in order to maintain stability throughout the problem solution time. This is

necessary even when the faster solution component y_1 has decayed to insignificant values which is after about five time constants, i.e. 5×10^{-4} seconds in the present example. Therefore, the best choice to solve such problems is the implicit numerical integration methods such as backward Euler algorithm or the Trapezoidal rule which are stable for any value of λ in the positive half-plane.

In the case of nuclear power plant hydraulic networks, time constants as small as 10^{-4} seconds occur when the high velocity pressure waves travel through short length network links [6]. Therefore, the system of differential equations describing the network dynamics are stiff. An efficient implicit algorithm based on the Euler's backward difference formula has been developed by Porsching et al. [6] and is described in the next section.

2.5. Porsching's Algorithm

2.5.1. Introduction

Porsching's algorithm is based on the Euler's backward difference formula. Since it is an implicit algorithm, it permits a larger step size to be chosen for the numerical integration of equations, but involves the solution of a system of linear algebraic equations of the form $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ at each time-step. However, in the Porsching algorithm, the variables ΔU_i and ΔM_i are first eliminated algebraically from the linear equations before proceeding with the solution of these system of equations. This reduces the dimension of the systems of equations from $(K + 2N)$ to K. For example, for a 150-node and 200-link network problem, the size of the matrix equation to be solved at each time-step is 200×200 instead of 500×500. This results in a considerable reduction in the computational effort at each

time-step.

Moreover, if the links in the chains of the network are numbered first and in a monotonically increasing order and the remaining links, termed as random links, are then numbered higher than the chained links, then the matrix **A** in the solution of system of linear equations $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, assumes a bordered block-diagonal form. The block structure of the matrix **A** can be exploited in the Gaussian elimination process resulting in further reduction in the computations involved at each time-step.

The above features of the Porsching's algorithm make it very attractive for the numerical integration of large network problems and a number of practical computer codes [28-31] based on this technique have been developed and used in nuclear power plant safety analysis. This technique has been extended in [32] to take into account the nonhomogeneous flow conditions accurately (improved slip model) in the network but still involves the solution of the matrix equation of the form $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ at each time-step which is the dominatant computation for large networks as will be shown in Chapter 3.

2.5.2. Development of the algorithm

In this section, the derivation of Porsching's algorithm for integrating the systems of differential equations (2.1) through (2.6), which are of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$, is given. Since **y** is now a vector, the value of the vector **y** at time nh is represented by the symbol $\mathbf{y}^n$ instead of $\mathbf{y}_n$ in order to avoid confusion with the individual elements $y_1, y_2, \cdots$ of the vector **y**. $y_1^n, y_2^n, \cdots$ etc. now represent the values of individual elements of y at time nh. In other words, the subscript n in section 2.4 is now the superscript n.

The system of equations to be integrated is of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$, where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ \cdots \\ \cdots \\ y_K \\ y_{K+1} \\ y_{K+2} \\ \cdots \\ \cdots \\ \cdots \\ y_{K+N} \\ y_{K+N+1} \\ y_{K+N+2} \\ \cdots \\ \cdots \\ \cdots \\ y_{K+2N} \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \cdots \\ \cdots \\ \cdots \\ W_K \\ U_1 \\ U_2 \\ \cdots \\ \cdots \\ \cdots \\ U_N \\ M_1 \\ M_2 \\ \cdots \\ \cdots \\ \cdots \\ M_N \end{bmatrix} \tag{2.33}$$

$$\mathbf{f} = \begin{bmatrix} \mathrm{f}_1 \\ \mathrm{f}_2 \\ \cdots \\ \cdots \\ \cdots \\ \mathrm{f}_K \\ \mathrm{f}_{K+1} \\ \mathrm{f}_{K+2} \\ \cdots \\ \cdots \\ \cdots \\ \mathrm{f}_{K+N} \\ \mathrm{f}_{K+N+1} \\ \mathrm{f}_{K+N+2} \\ \cdots \\ \cdots \\ \cdots \\ \mathrm{f}_{K+2N} \end{bmatrix} = \begin{bmatrix} f_1(t,P_i,P_j,W_1) \\ f_2(t,P_i,P_j,W_2) \\ \cdots \\ \cdots \\ f_K(t,P_i,P_j,W_k) \\ \sum_{v\varepsilon T_1} H_v W_v - \sum_{v\varepsilon I_1} H_v^* W_v + Q_1 \\ \sum_{v\varepsilon T_2} H_v W_v - \sum_{v\varepsilon I_2} H_v^* W_v + Q_2 \\ \cdots \\ \cdots \\ \sum_{v\varepsilon T_N} H_v W_v - \sum_{v\varepsilon I_N} H_v^* W_v + Q_N \\ \sum_{v\varepsilon T_1} W_v - \sum_{v\varepsilon I_2} W_v \\ \sum_{v\varepsilon T_2} W_v - \sum_{v\varepsilon I_2} W_v \\ \cdots \\ \cdots \\ \sum_{v\varepsilon T_N} W_v - \sum_{v\varepsilon I_N} W_v \end{bmatrix} \tag{2.34}$$

Euler's backward difference formula is given by

$$\mathbf{y}^{n+1} = \mathbf{y}^n + hf(\mathbf{y}^{n+1}, t^{n+1})$$

or

$$(\mathbf{y}^{n+1} - \mathbf{y}^n) - h\ \mathbf{f}(\mathbf{y}^{n+1}\ ,\ \mathrm{t}^{n+1}) = 0 \tag{2.35}$$

$\mathbf{y}^{n+1}$ cannot be explicity solved from the above equation. One solution to the problem is to perform a functional iteration of the form

$$\left.\begin{aligned} \left[\left[\mathbf{y}^{n+1}\right]^{m+1} - \mathbf{y}^{n}\right] - h\ \mathbf{f}\left[t^{n+1}\ ,\ \left[\mathbf{y}^{n+1}\right]^{m}\right] = 0 \\ \left[\mathbf{y}^{n+1}\right]^{m} = \left[\mathbf{y}^{n+1}\right]^{m+1} \end{aligned}\right\}$$

where $(\mathbf{y}^{n+1})^{m+1}$ denotes the value of $\mathbf{y}^{n+1}$ at iteration $(m+1)$, till the maximum difference between any two corresponding elements in $(\mathbf{y}^{n+1})^{m+1}$ and $(\mathbf{y}^{n+1})^{m}$ is less than a predetermined value ε. However, this may require a very small value of step size h in order to ensure convergence [33] and hence, the advantage of the Euler's implicit method is lost.

A different approach is to employ only one iteration of Newton-Ralphson method for finding the zero of the nonlinear function, i.e. the value of y^{n+1} in (2.35).

The Newton-Ralphson iteration for computing the value of variable x satisfying $\mathbf{S}(\mathbf{x}) = 0$ is given by

$$\mathbf{x}^{n+1} = \mathbf{x}^{n} - \left[\frac{d\mathbf{S}(\mathbf{x})}{d\mathbf{x}}\bigg|_{\mathbf{x}=\mathbf{x}^{n}}\right]^{-1} \mathbf{S}(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}^{n}} .$$

In our case, the function whose zero is to be estimated is given by the left hand side of (2.35) and the variable in the equation is $\mathbf{y}^{n+1}$. Therefore,

$$S(\mathbf{y}^{n+1}) = (\mathbf{y}^{n+1} - \mathbf{y}^{n}) - h\ \mathbf{f}(\mathbf{y}^{n+1}, t^{n+1}) .$$

Using Newton-Ralphson iteration,

$$\mathbf{y}^{n+1} = \mathbf{y}^{n} - \left[\mathbf{I} - h\frac{d\mathbf{f}}{d\mathbf{y}}(\mathbf{y}^{n}, t^{n})\right]^{-1} \left[(\mathbf{y}^{n} - \mathbf{y}^{n}) - h\ \mathbf{f}(\mathbf{y}^{n}, t^{n})\right]$$

or

$$\mathbf{y}^{n+1} = \mathbf{y}^{n} + h[\mathbf{I} - h\ \mathbf{J}^{n}]^{-1}\ \mathbf{f}(\mathbf{y}^{n}, t^{n}) \tag{2.36}$$

where

$I =$ identity matrix of size $(K + 2N) \times (K + 2N)$

$$J^n = \frac{\mathbf{df}}{\mathbf{dy}} (\mathbf{y}^n, t^n) = \left[\frac{df_i}{dy_j} (\mathbf{y}^n, t^n)\right]$$

and is called the Jacobian matrix for $\mathbf{f}(\mathbf{y},t)$. Equation (2.36) could also have been obtained by first linearizing (2.7) and then applying the Euler's backward difference formula to this linearized equation. It can be shown [19] that the numerical integration formula given by (2.36) is consistent and stable, therefore, convergent.

Since computing the inverse of a matrix involves more floating point operations than the solution of a system of linear equations of the same order, (2.36) is not evaluated in the present form but is rearranged as the solution of a system of equations.

Rewriting (2.36) as

$$(\mathbf{y}^{n+1} - \mathbf{y}^n) = h[\mathbf{I} - h\ \mathbf{J}^n]^{-1}\ \mathbf{f}(\mathbf{y}^n, t^n)$$

and premulitplying both sides by $(\mathbf{I} - h\ \mathbf{J}^n)$ and replacing $(\mathbf{y}^{n+1} - \mathbf{y}^n)$ by the incremental vector $\Delta\ \mathbf{y}^{n+1}$, we have

$$(\mathbf{I} - h\ \mathbf{J}^n)\ \Delta\ \mathbf{y}^{n+1} = h\ \mathbf{f}(\mathbf{y}^n, t^n) \quad (2.37)$$

The determination of $\mathbf{y}^{n+1}$ then amounts to the determination of the increment $\Delta\mathbf{y}^{n+1}$, the solution of the linear system (2.37).

Because of the simple nature of (2.4) and (2.5), the quantities $\Delta\ U_i^{n+1}$ and $\Delta\ M_i^{n+1}$ can be eliminated from (2.37). The resulting system of equations contains the variables $\Delta\ W_k^{n+1}$ only and is of order K as opposed to $(K + 2N)$, the order of (2.37). This results in considerable saving in computation time for large size networks.

Elimination of ΔU_i^{n+1} and ΔM_i^{n+1} from (2.37) is done by first solving $(K + i)$th and $(K + N + i)$th equations in (2.37) for ΔU_i^{n+1} and ΔM_i^{n+1} and then substituting these values into the first K equations of (2.37)

After the above elimination procedure [6], the variables ΔU_i^{n+1} and ΔM_i^{n+1} are given by

$$\Delta U_i^{n+1} = h \left[\gamma_{1i} + \sum_{\nu \varepsilon T_i} (\alpha_{1i} + \beta_{1i} H_\nu) \Delta W_\nu^{n+1} - \sum_{\nu \varepsilon I_i \,,\, \nu \le K} (\alpha_{1i} + \beta_{1i} H_\nu^*) \Delta W_\nu^{n+1} \right] \tag{2.38}$$

$$\Delta M_i^{n+1} = h \left[\gamma_{2i} + \sum_{\nu \varepsilon T_i} (\alpha_{2i} + \beta_{2i} H_\nu) \Delta W_\nu^{n+1} - \sum_{\nu \varepsilon I_i \,,\, \gamma \le K} (\alpha_{2i} + \beta_{2i} H_\nu^*) \Delta W_\nu^{n+1} \right] \tag{2.39}$$

and (2.37) is reduced to the matrix equation

$$\mathbf{A} \Delta \mathbf{W}^{n+1} = \mathbf{z} \tag{2.40}$$

where

$$\alpha_{1i} = \frac{-h}{D_i} \frac{\partial HG}{\partial M_i} \tag{2.41}$$

$$\beta_{1i} = \frac{\left[1 + h \frac{\partial G}{dM_i} \right]}{D_i} \tag{2.42}$$

$$\gamma_{1i} = (\beta_{1i} f_{K+i} + \alpha_{1i} f_{K+N+i}) \tag{2.43}$$

$$\alpha_{2i} = \frac{\left[1 + h \frac{\partial HG}{\partial U_i}\right]}{D_i} \tag{2.44}$$

$$\beta_{2i} = \frac{-h}{D_i} \frac{\partial G}{\partial U_i} \tag{2.45}$$

$$\gamma_{2i} = \left[\alpha_{2i} f_{K+N+i} + \beta_{2i} f_{K+i}\right] \tag{2.46}$$

$$\frac{\partial HG}{\partial M_i} = \frac{\partial P_i}{\partial M_i} \sum_{\nu \in I_i, \nu > K} H_\nu^* \frac{\partial g_\nu}{\partial P_i} \tag{2.47}$$

$$\frac{\partial HG}{\partial U_i} = \frac{\partial P_i}{\partial U_i} \sum_{\nu \in I_i, \nu > K} H_\nu^* \frac{\partial g_\nu}{\partial P_i} \tag{2.48}$$

$$\frac{\partial G}{\partial M_i} = \frac{\partial P_i}{\partial M_i} \sum_{\nu \in I_i, \nu > K} \frac{\partial g_\nu}{\partial P_i} \tag{2.49}$$

$$\frac{\partial G}{\partial U_i} = \frac{\partial P_i}{\partial U_i} \sum_{\nu \in I_i, \nu > K} \frac{\partial g_\nu}{\partial P_i} \tag{2.50}$$

$$D_i = \left(1 + h \frac{\partial HG}{\partial U_i}\right)\left(1 + h \frac{\partial G}{\partial M_i}\right) - h^2 \frac{\partial HG}{\partial M_i} \frac{\partial G}{\partial U_i} \tag{2.51}$$

and the elements of vector **z** and matrix **A** are given by

$$z_k = h f_k(\mathbf{y}^n, t^n) + h^2 \left[\frac{\partial f_k}{\partial U_i} \gamma_{1i} + \frac{\partial f_k}{\partial U_j} \gamma_{1j} + \frac{\partial f_k}{\partial M_i} \gamma_{2i} + \frac{\partial f_k}{\partial M_j} \gamma_{2j}\right], \quad k = 1, 2, \ldots, K \tag{2.52}$$

$$a_{kk} = 1 - h\,\frac{\partial f_k}{\partial W_k} - h^2\left[\frac{\partial f_k}{\partial U_j}\,(\alpha_{1j} + \beta_{1j}\,H_k) + \frac{\partial f_k}{\partial M_j}\,(\alpha_{2j} + \beta_{2j}\,H_k)\right.$$
$$\left. - \frac{\partial f_k}{\partial U_i}\,(\alpha_{1i} + \beta_{1i}\,H_\nu^*) - \frac{\partial f_k}{\partial M_i}\,(\alpha_{2i} + \beta_{2i}\,H_\nu^*)\right],\ k = 1, 2, \ldots, K \qquad (2.53)$$

The values of a_{kk} elements for $k \neq \nu$ are given by (2.54) which is shown in Fig. 2.8. The seven possible expressions for $a_{k\nu}$ values correspond to seven possible positions of link ν with respect to link k as shown in Fig. 2.8.

Therefore, the element $a_{k\nu}$ of matrix **A**, which has the row index k and column index ν, will be nonzero only if links k and ν share a common node. Otherwise, its value will be equal to zero.

Most of the nodes in a typical network are connected to normal or non-critical links. Therefore, for a normal link whose initial and terminal nodes have no critical links,

$$\frac{\partial HG}{\partial M_i} = 0\ ;\ \frac{\partial HG}{\partial U_i} = 0, \qquad \text{from (2.47) and (2.48)}$$

$$\frac{\partial G}{\partial M_i} = 0\ ;\ \frac{\partial G}{\partial U_i} = 0, \qquad \text{from (2.49) and (2.50)}$$

$$\alpha_{1i} = 0\ ;\ \beta_{1i} = 1, \qquad \text{from (2.41) and (2.42)}$$

$$\gamma_{1i} = f_{K+i}\,(\mathbf{y}^n, t^n), \qquad \text{from (2.43)}$$

$$\alpha_{2i} = 1\ ,\ \beta_{2i} = 0, \qquad \text{from (2.44) and (2.45)}$$

$$\gamma_{2i} = f_{K+N+i}\,(\mathbf{y}^n, t^n), \qquad \text{from (2.46)}$$

$$D_i = 1 \qquad \text{from (2.51).}$$

After substituting the above values in (2.38),(2.39),(2.52), (2.53) and (2.54), we have

$$a_{k\nu} = \begin{cases} -h^2\left[\dfrac{\partial f_k}{\partial U_i}(\alpha_{1i} + \beta_{1i}H_\nu) + \dfrac{\partial f_k}{\partial M_i}(\alpha_{2i} + \beta_{2i}H_\nu)\right] & \text{if } \nu \in T_i, \nu \notin I_j, \\ h^2\left[\dfrac{\partial f_k}{\partial U_i}(\alpha_{1i} + \beta_{1i}H^*_\nu) + \dfrac{\partial f_k}{\partial M_i}(\alpha_{2i} + \beta_{2i}H^*_\nu)\right] & \text{if } \nu \in I_i, \nu \notin T_j, \\ -h^2\left[\dfrac{\partial f_k}{\partial U_j}(\alpha_{1j} + \beta_{1j}H_\nu) + \dfrac{\partial f_k}{\partial M_j}(\alpha_{2j} + \beta_{2j}H_\nu)\right] & \text{if } \nu \in T_j, \nu \notin I_i, \\ h^2\left[\dfrac{\partial f_k}{\partial U_j}(\alpha_{1j} + \beta_{1j}H^*_\nu) + \dfrac{\partial f_k}{\partial M_j}(\alpha_{2j} + \beta_{2j}H^*_\nu)\right] & \text{if } \nu \in I_j, \nu \notin T_i, \\ -h^2\left[\dfrac{\partial f_k}{\partial U_i}(\alpha_{1i} + \beta_{1i}H_\nu) + \dfrac{\partial f_k}{\partial M_i}(\alpha_{2i} + \beta_{2i}H_\nu)\right. & \\ \left. -\dfrac{\partial f_k}{\partial U_j}(\alpha_{1j} + \beta_{1j}H^*_\nu) - \dfrac{\partial f_k}{\partial M_j}(\alpha_{2j} + \beta_{2j}H^*_\nu)\right] & \text{if } \nu \in I_j, \nu \in T_i, \\ -h^2\left[\dfrac{\partial f_k}{\partial U_j}(\alpha_{1j} + \beta_{1j}H_\nu) + \dfrac{\partial f_k}{\partial M_j}(\alpha_{2j} + \beta_{2j}H_\nu)\right. & \\ \left. -\dfrac{\partial f_k}{\partial U_i}(\alpha_{1i} + \beta_{1i}H^*_\nu) - \dfrac{\partial f_k}{\partial M_i}(\alpha_{2i} + \beta_{2i}H^*_\nu)\right] & \text{if } \nu \in I_i, \nu \in T_j, \\ 0, \quad \textit{otherwise.} & \end{cases} \tag{2.54}$$

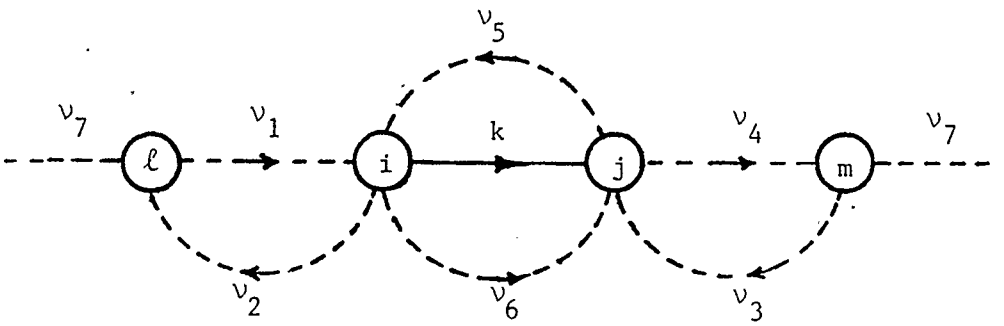


Figure 2.8 Values of $a_{k\nu}$ elements of matrix A for $\nu \neq k$.

$$\Delta U_i^{n+1} = h\left[\sum_{\nu \epsilon T_i} H_\nu W_\nu^{n+1} - \sum_{\nu \epsilon I_i} H_\nu^* W_\nu^{n+1} + Q_i\right], \quad i = 1, 2, \ldots, N \tag{2.55}$$

$$\Delta M_i^{n+1} = h\left[\sum_{\nu \epsilon T_i} W_\nu^{n+1} - \sum_{\nu \epsilon I_i} W_\nu^{n+1}\right], \quad i = 1, 2, \ldots, N \tag{2.56}$$

$$\begin{aligned} z_k = h\, f_k\,(\mathbf{y}^n, t^n) + h^2 \left[\frac{\partial f_k}{\partial U_i} f_{K+i}\,(\mathbf{y}^n, t^n) + \frac{\partial f_k}{\partial U_j} f_{K+N+j}\,(\mathbf{y}^n, t^n)\right. \\ \left. + \frac{\partial f_k}{\partial M_i} f_{K+N+i}\,(\mathbf{y}^n, t^n) + \frac{\partial f_k}{\partial M_j} f_{K+N+j}\,(\mathbf{y}^n, t^n)\right], \quad k = 1, 2, \ldots, K \end{aligned} \tag{2.57}$$

$$\begin{aligned} a_{kk} = 1 - h\,\frac{\partial f_k}{\partial W_k} - h^2\left[\frac{\partial f_k}{\partial U_j} H_k + \frac{\partial f_k}{\partial M_j} - \frac{\partial f_k}{\partial U_i} H_k^*\right. \\ \left. - \frac{\partial f_k}{\partial M_i}\right], \quad k = 1, 2, \ldots, K \end{aligned} \tag{2.58}$$

$a_{k\nu}$ for $\nu \neq k$ is given by one of the expressions shown on the next page.

$$
a_{k\nu} = \begin{cases}
-h^2\left[\dfrac{\partial f_k}{\partial U_i} H_\nu + \dfrac{\partial f_k}{\partial M_i}\right], & \text{if } \nu\varepsilon T_i,\ \nu \in I_j \\
h^2\left[\dfrac{\partial f_k}{\partial U_i} H_\nu^* + \dfrac{\partial f_k}{\partial M_i}\right], & \text{if } \nu\varepsilon I_i,\ \nu \in T_j, \\
-h^2\left[\dfrac{\partial f_k}{\partial U_j} H_\nu + \dfrac{\partial f_k}{\partial M_j}\right], & \text{if } \nu\varepsilon T_j,\ \nu \in I_i, \\
h^2\left[\dfrac{\partial f_k}{\partial U_j} + \dfrac{\partial f_k}{\partial M_j}\right], & \text{if } \nu\varepsilon I_j,\ \nu \in T_i, \\
h^2\left[\dfrac{\partial f_k}{\partial U_i} H_\nu + \dfrac{\partial f_k}{\partial M_i} - \dfrac{\partial f_k}{\partial U_j} H_\nu^* - \dfrac{\partial f_k}{\partial M_j}\right], & \text{if } \nu\varepsilon I_i,\ \nu\varepsilon T_i, \\
-h^2\left[\dfrac{\partial f_k}{\partial U_j} H_\nu + \dfrac{\partial f_k}{\partial M_j} - \dfrac{\partial f_k}{\partial U_i} H_\nu^* - \dfrac{\partial f_k}{\partial M_i}\right], & \text{if } \nu\varepsilon I_i,\ \nu\varepsilon T_j, \\
0, & \text{otherwise}
\end{cases}
\qquad (2.59)
$$

$k = 1, 2, \ldots, K$

This completes the derivation of Porsching's algorithm. The computational aspects of the algorithm on a serial computer are given in Chapter 3. The parallelization of the algorithm and its performance analysis on mulitple computers is given in Chapters 5 and 6.

CHAPTER 3

ALGORITHM PERFORMANCE ON A SINGLE PROCESSOR

3.1. Introduction

Porshcing's algorithm for the numerical integration of hydraulic network differential equations was described in the previous chapter. In this chapter, the computational complexity of the algorithm on a single processor is discussed.

The flow chart of the computations involved in the Porsching's algorithm is shown in Fig. 3.1 As will be shown in section 3.2, the major computational effort at each time-step in the Porsching's algorithm is expended in the solution of ΔW^{n+1} from the matrix equation $A \Delta W^{n+1} = z$. It is therefore important that this matrix equation be solved efficiently at each time-step. This is achieved by making the sparse matrix A assume a desirable structure as discussed in the next section.

3.2. Computational Complexity of Porsching's Algorithm

3.2.1. Computation of flow rate increments from the matrix equation (2.40)

3.2.1.1. Structure of matrix A

An element $a_{k\nu}$ of matrix A is non-zero whenever its row number k and column number ν are adjacent links. Since large hydraulic networks encountered in nuclear plants have very sparse structure because of low degree of the network nodes (degree of a node is defined as the sum of the initiating and terminating links to the node), the matrix A is very sparse for large networks, i.e. it has a few non-zero elements (less than, say, 10 per row).

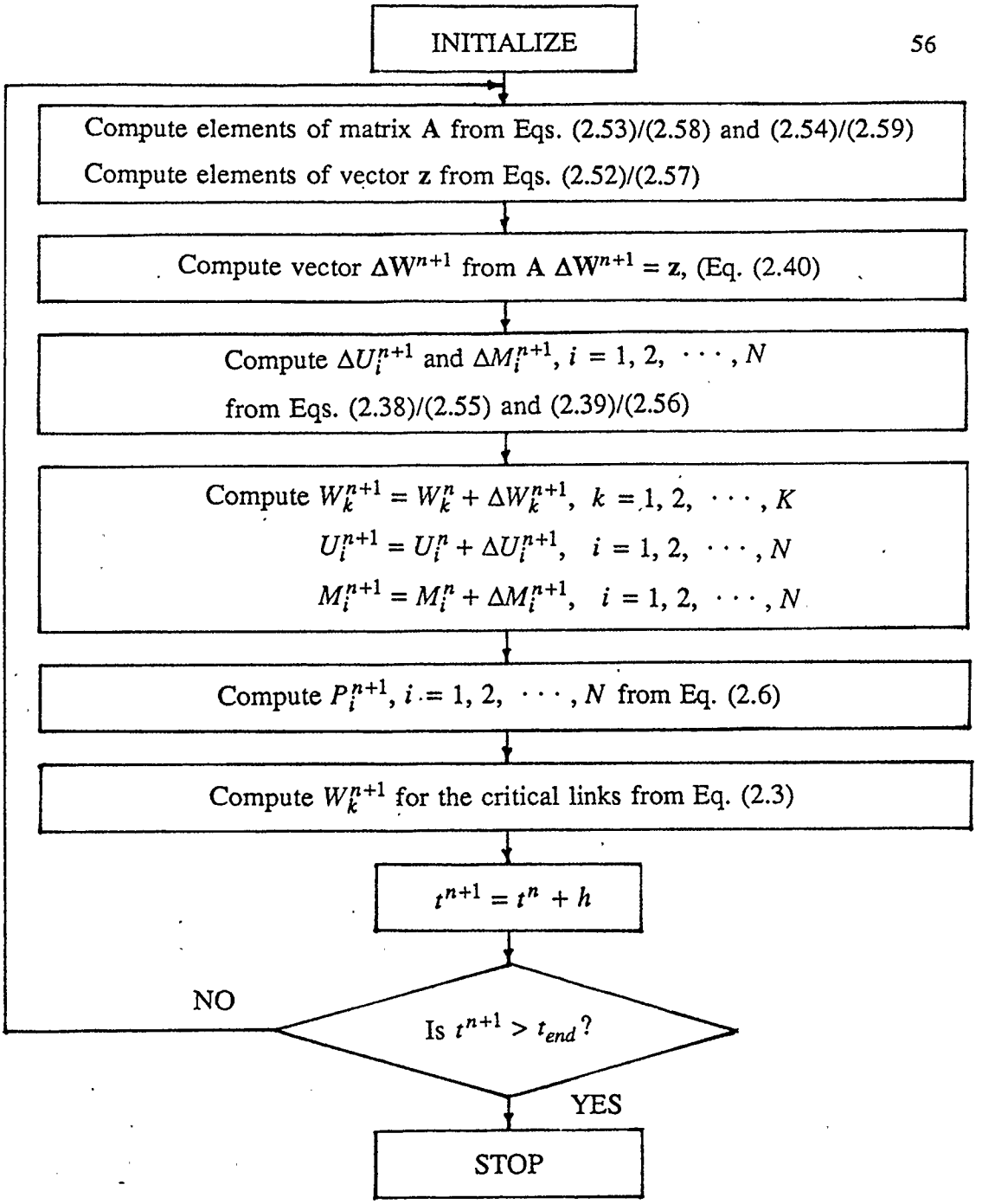


Figure 3.1 Basic flow chart of Porsching's algorithm for the numerical integration of hydraulic network differential equations.

If the network links are numbered at random, the position of non-zero elements a_{kv} in the matrix will also be at random. When the matrix solution is being obtained by the sparse Gaussian elimination method, many of the zero elements in the matrix will become non-zero in the elimination process and the sparse matrix becomes almost full after a few elimination steps. This phenomena is called "fill-in" [34] and may result in large computation time even for a sparse matrix.

However, if the links of a given chain are consecutively numbered, then the principal submatrix of A corresponding to that chain will be tridiagonal. Moreover, submatrices corresponding to disjointed chains are themselves disjointed. The random links in the network (such as interconnecting links, by-pass links etc.) are then numbered higher than the chained links. This procedure reduces the amount of "fill-in" in the elimination process. It also decouples the principal submatrices corresponding to disjointed chains among themselves which is an essential requirement for concurrent processing on a multiple processor system. Also tri-diagonal blocks of equations are easiest to solve by Gaussian elimination since they take time proportional to size of the problem as compared to full matrices which take time proportional to cube of the problem size.

Figure 3.2 illustrates the structure of matrix A for the network of Fig 2.1. It has two tri-diagonal blocks corresponding to two disjointed chains in the network, one of length 4 and one of length 5. Also, only the rightmost column block and bottom row block have submatrices with non-zero elements. This matrix structure is called the bordered block-diagonal form [34] of the matrix and is a desirable structure for sparse matrices. It is also desirable to have as small a width of the borders as possible for an efficient solution of the matrix equation.

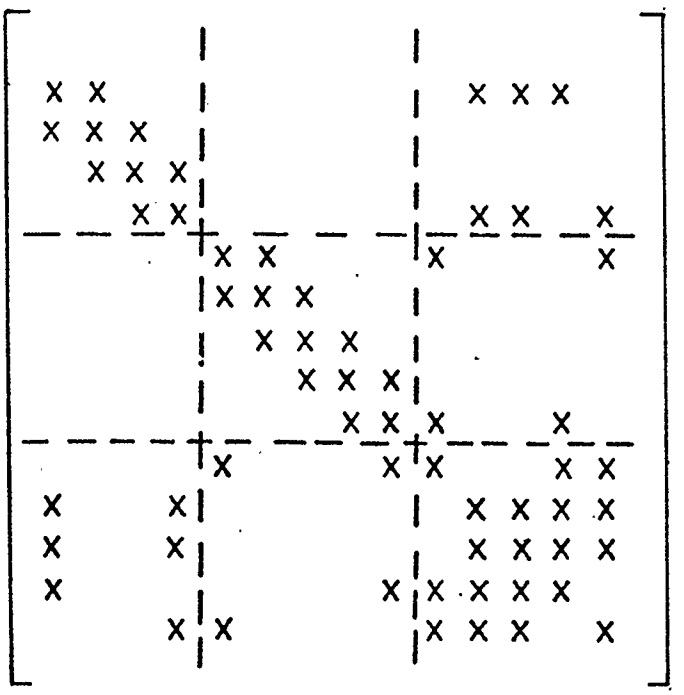


Figure 3.2 Structure of matrix A for the network of Fig. 2.1. It has two tridiagonal blocks corresponding to two disjointed chains in the network.

As a second example, consider the network of Fig 3.3(a). It has two neighbouring chains (links 1 to 7) and (links 13 to 18) sharing the two end nodes and a disjointed chain (links 8 to 12). The corresponding structure of matrix A is shown in Fig. 3.3(b). It has two disjointed tri-diagonal blocks and the width of the border is 9 elements wide. However, if the two end links of the neighbouring chains are considered as random links and not as a part of the chains, then the neighbouring chains can be converted to disjointed chains. This improved numbering of links for the network of Fig. 3.3(a) is shown in Fig. 3.4(a) and the corresponding structure of matrix A is shown in Fig 3.4(b). As can be seen from Fig. 3.4(b), we now have three disjointed tri-diagonal blocks corresponding to the three chains in the network. This increases the amount of parallelism in the problem as all the tri-diagonal blocks can be computed in parallel. Also the width of the border is now 7 elements as compared to 9 in the previous link numbering. The same numbering scheme can also be used if the two chains share only one end node instead of both the end nodes.

In general, suppose the network contains s disjointed chains of length $p_1, p_2, \ldots, p_s$ and let q be the number of random links in the network, then

$$q = K - \sum_{i=1}^{s} p_i \qquad (3.1)$$

If the links in the chains are consecutively numbered as described before, then, the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$ can be made to assume the form,

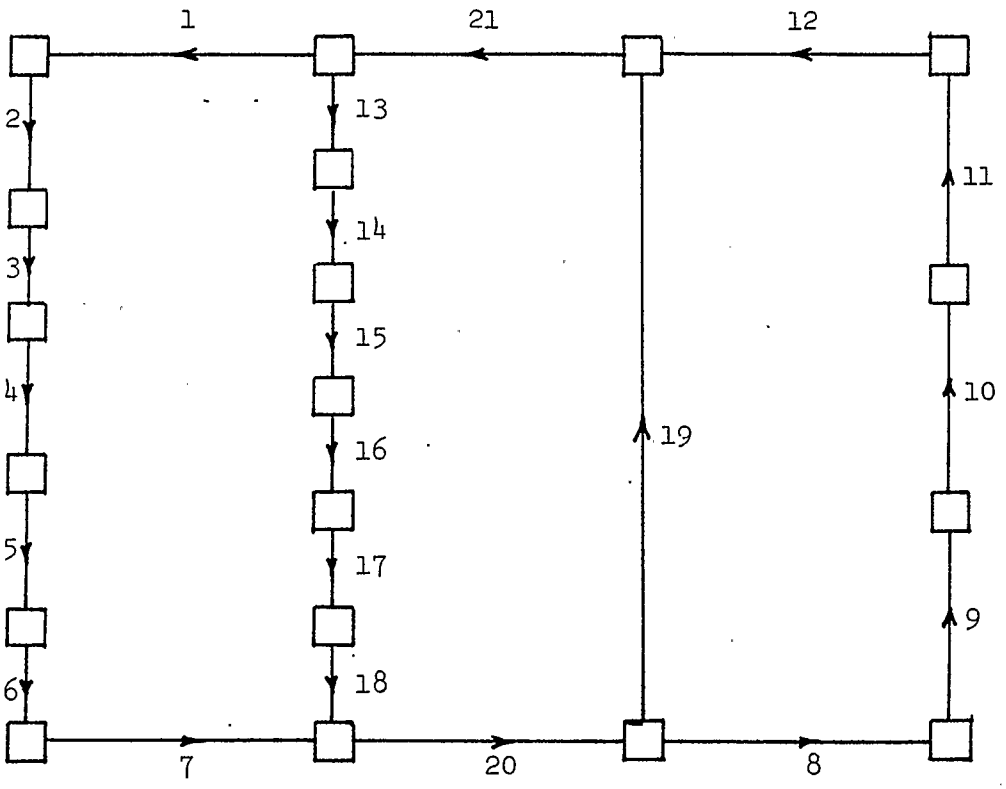


Figure 3.3(a) A non-optimal link numbering for a network have two neighbouring chains and one disjointed chain.

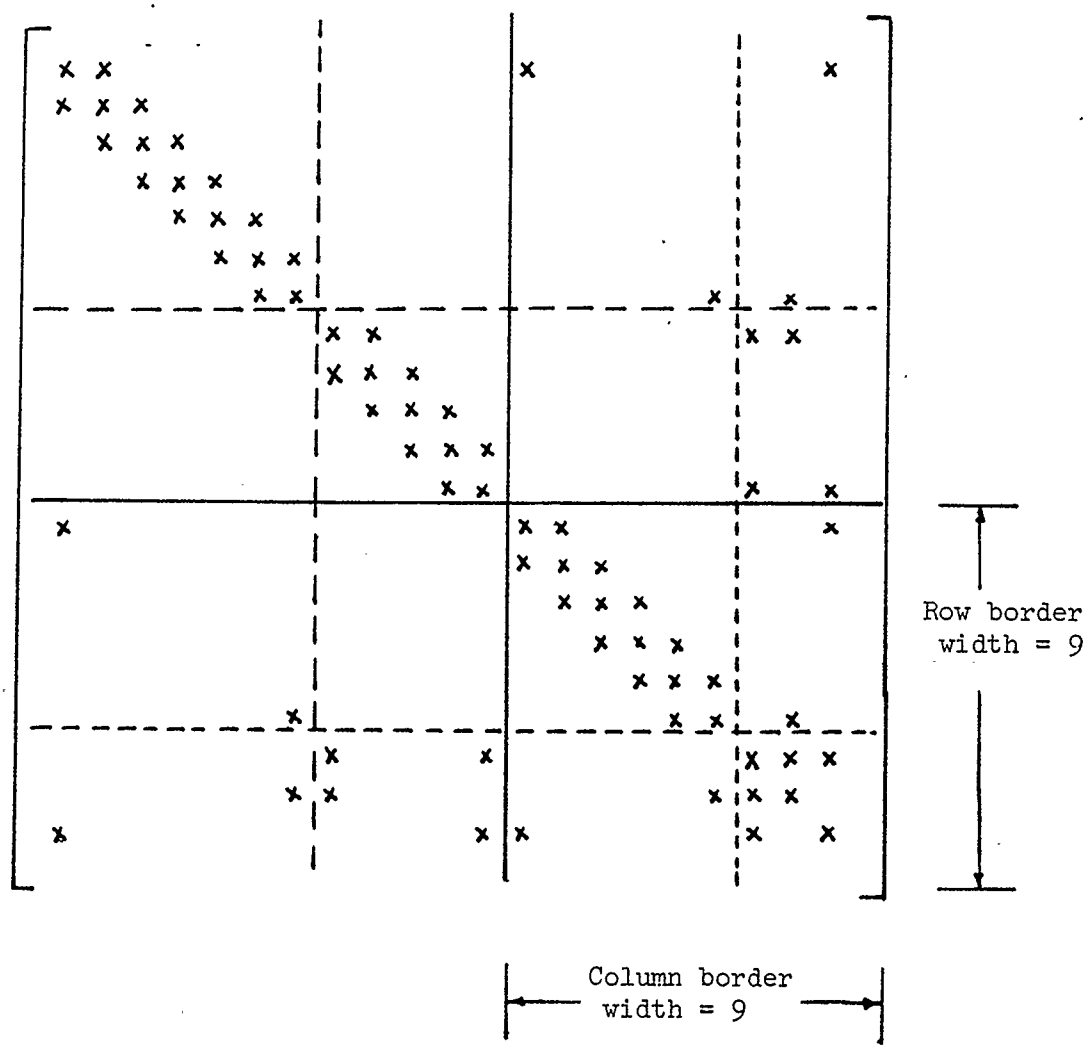


Figure 3.3(b) Structure of matrix A corresponding to network of Fig. 3.3(a). The number of tridiagonal blocks is two.

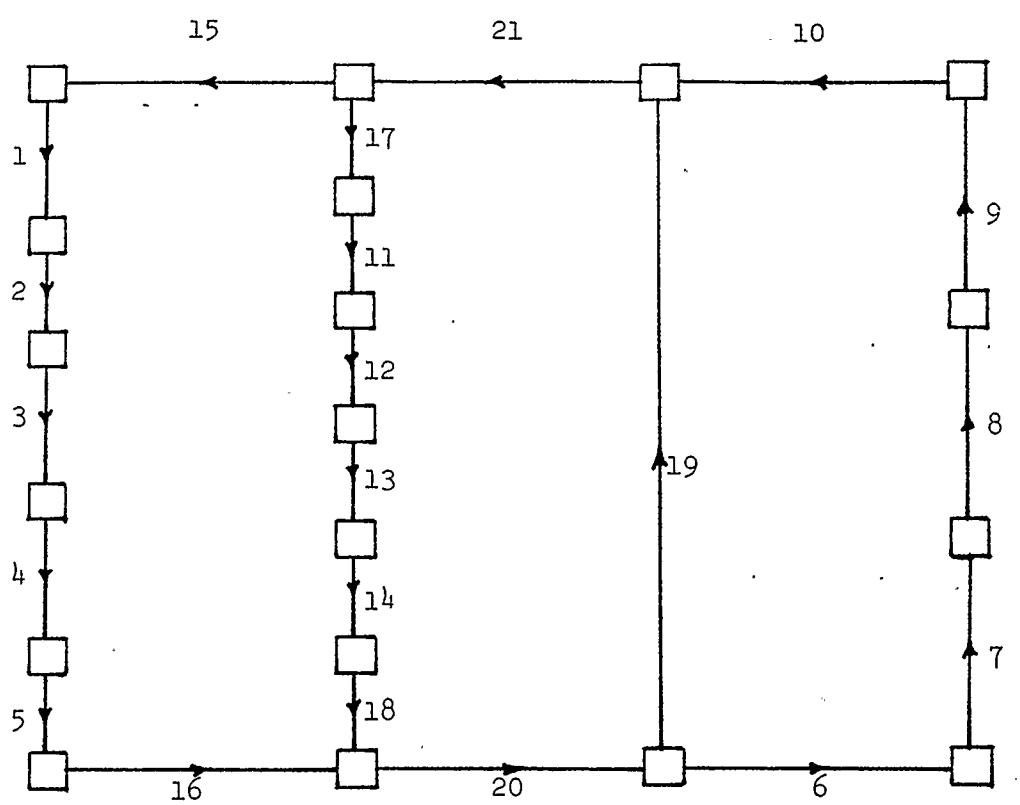


Figure 3.4(a) Improved link numbering for a network having two neighbouring chains and one disjointed chain.

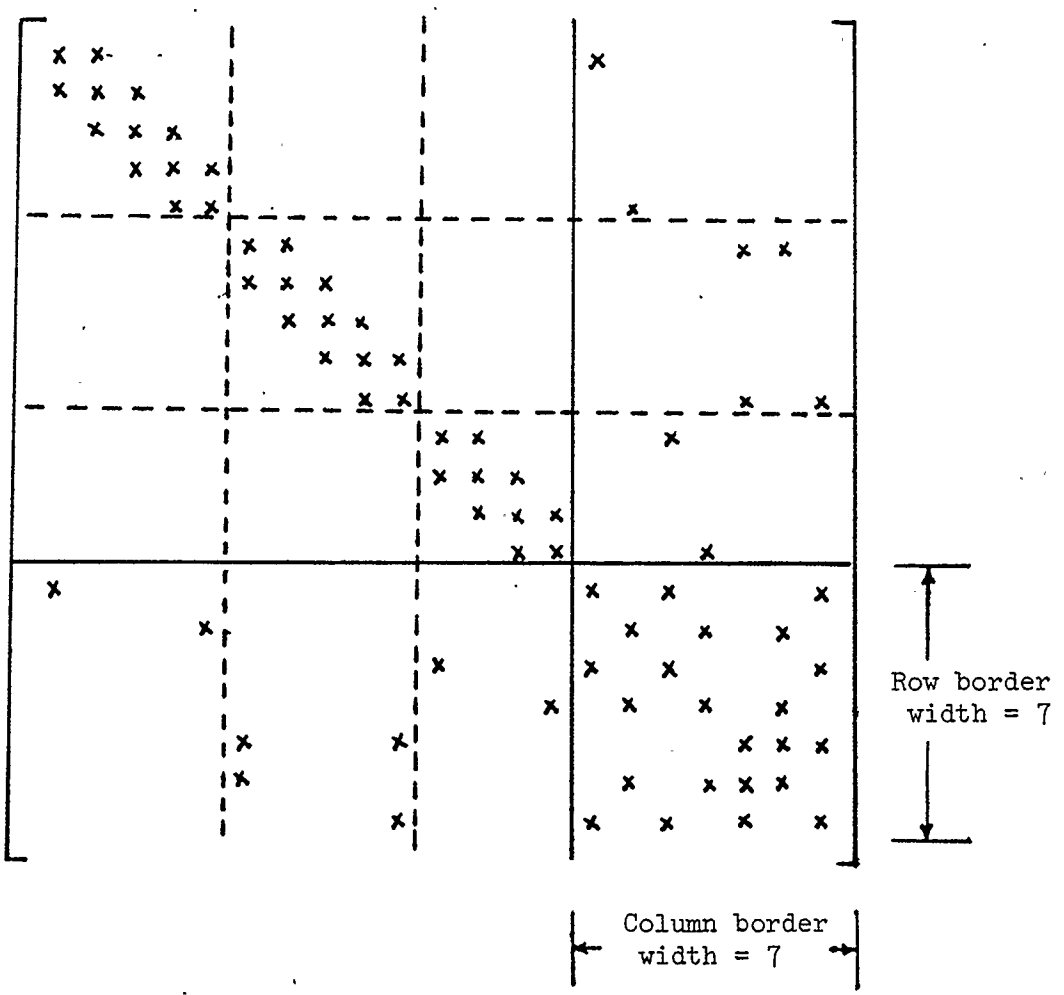


Figure 3.4(b) Structure of matrix A corresponding to network labeling of Fig. 3.4(a). The number of tridiagonal blocks is now three.

$$\begin{bmatrix} \mathbf{D}_1 & & & & & \mathbf{R}_1 \\ & \mathbf{D}_2 & & & & \mathbf{R}_2 \\ & & & & & \cdots \\ & & & & & \cdots \\ & & \mathbf{D}_i & & & \mathbf{R}_i \\ & & & & & \cdots \\ & & & & & \cdots \\ & & & & \mathbf{D}_s & \mathbf{R}_s \\ \mathbf{B}_1 & \mathbf{B}_2 & \cdots & \mathbf{B}_i & \cdots & \mathbf{B}_s \quad \mathbf{V} \end{bmatrix} \begin{bmatrix} \Delta\,\omega_1 \\ \Delta\,\omega_2 \\ \cdots \\ \cdots \\ \Delta\,\omega_i \\ \cdots \\ \cdots \\ \Delta\,\omega_s \\ \Delta\,\omega_{s+1} \end{bmatrix} = \begin{bmatrix} \zeta_1 \\ \zeta_2 \\ \cdots \\ \cdots \\ \zeta_i \\ \cdots \\ \cdots \\ \zeta_s \\ \zeta_{s+1} \end{bmatrix} \tag{3.2}$$

as shown for the example networks. Various submatrices of matrix **A** and subvectors of vectors $\Delta\,\mathbf{W}$ and **z** have the following features:

(i) $\mathbf{D}_i$ is a $p_i \times p_i$ tri-diagonal matrix corresponding to chain i.

(ii) $\mathbf{R}_i$ is a $p_i \times q$ submatrix corresponding to chain i and has possible non-zero elements in the first and last row only. These non-zero elements correspond to the random links sharing the two end nodes of the chain i.

(iii) $\mathbf{B}_i$ is a $q \times p_i$ submatrix corresponding to chain i and has possible non-zero elements in the first and last columns only. These non-zero elements correspond to the random links sharing the two end nodes of the chain i. Matrix $\mathbf{B}_i$ is the transpose of matrix of $\mathbf{R}_i$ with respect to its element locations but not their values.

(iv) **V** is a $q \times q$ sparse random matrix with non-zero diagonal elements corresponding to the random links. The matrix is symmetric with respect to its element locations but not with respect to their values.

(v) $\Delta\,\omega_i$ and ζ_i are $p_i \times 1$ full vectors for $i = 1, 2, \ldots, s$. $\Delta\,\omega_{s+1}$ and $\Delta\,\zeta_{s+1}$ are full vectors of length q each.

3.2.1.2. Estimation of computation time for solving the matrix equation

In this section, the sequential complexity of computing $\Delta\ \mathbf{W}^{n+1}$ from $\mathbf{A}\ \Delta\mathbf{W}^{n+1} = \mathbf{z}$ on a single processor is derived. Only floating-point add, multiply and divide operations are counted in obtaining the complexity results. Time spent in conditionals such as "if" statements is neglected. Also, in order to get simpler expressions for computational complexity, all the floating-point operations are assumed to take an equal amount of time t_f. In practice, floating point add and multiply take more or less equal time in present-day computers but divide takes almost twice as long as an add or a multiply operation. However, the number of divide operations in typical computations is much less than the number of additions and multiplications. Therefore, the error caused by this assumption is not very serious. Moreover, this thesis is more concerned with the ratios of computation times (such as speed-ups obtained by running algorithms on multiple processors as compared to a single processor) rather than absolute values of computation times. This further reduces the error caused by the above two assumptions.

Two methods for solving $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ are given below and their computation times on a single processor are estimated. Since, matrices $\mathbf{B}_i$ and $\mathbf{R}_i$ are sparse, they are assumed to be sparse-stored and operated upon. This is done by storing the values of the non-zero matrix elements only along with their position in the matrix by the use of index vectors. The tri-diagonal portion is stored as three one dimensional full vectors one for the main leading diagonal and the other two for the sub- and super-diagonal. Matrix V is assumed to be stored as full because of the large "fill-in" of V to almost a full matrix during the elimination process.

3.2.1.2.1. Direct method I

In this method, matrix **A** is a triangularized into an upper triangular form by sparse forward elimination step of the Gaussian elimination method. At each step of the forward elimination procedure, the diagonal element a_{kk} of **A** is taken as the pivot element and any non-zero element a_{ik} present below it is eliminated. Any other non-zero elements a_{ij} present in row i are modified as

$$\hat{a}_{ij} = a_{ij} - \left[\frac{a_{ik}}{a_{kk}}\right] a_{kj} \quad . \tag{3.3}$$

After all the sub-matrices $\mathbf{B}_i$ have been eliminated in this fashion, matrix equation (3.2) assumes the form

$$\begin{bmatrix} \hat{\mathbf{D}}_1 & & & & \hat{\mathbf{R}}_1 \\ & \hat{\mathbf{D}}_2 & & & \hat{\mathbf{R}}_2 \\ & & & & \cdots \\ & & & & \cdots \\ & & \hat{\mathbf{D}}_i & & \hat{\mathbf{R}}_i \\ & & & & \cdots \\ & & & & \cdots \\ & & & \hat{\mathbf{D}}_s & \hat{\mathbf{R}}_s \\ & & & & \\ & & 0 & & \hat{\mathbf{V}} \end{bmatrix} \begin{bmatrix} \Delta\,\omega_1 \\ \Delta\,\omega_2 \\ \cdots \\ \cdots \\ \Delta\,\omega_i \\ \cdots \\ \cdots \\ \Delta\,\omega_s \\ \\ \Delta\,\omega_{s+1} \end{bmatrix} = \begin{bmatrix} \zeta_1 \\ \zeta_2 \\ \cdots \\ \cdots \\ \zeta_i \\ \\ \\ \zeta_s \\ \\ \zeta_{s+1} \end{bmatrix} \tag{3.4}$$

where $\hat{\mathbf{D}}_i$ are now bi-diagonal submatrices having one leading diagonal and one leading super-diagonal. Each sub-matrix $\hat{\mathbf{R}}_i$ has full columns of non-zero elements where ever a non-zero element existed in the first row of the original sub-matrix $\mathbf{R}_i$. Submatrix $\hat{\mathbf{V}}$ is now an almost full matrix.

Matrix equation (3.4) can be broken down into the following set of matrix equations:

$$\hat{D}_i \Delta \omega_i + \hat{R}_i \Delta \omega_{s+1} = \hat{\zeta}_i \, , i = 1, 2, ..., s$$

or

$$\Delta \omega_i = (\hat{D}_i)^{-1} [\hat{\zeta}_i - \hat{R}_i \Delta \omega_{s+1}], \; i = 1, 2, ..., s \tag{3.5}$$

$$\hat{V} \Delta \omega_{s+1} = \hat{\zeta}_{s+1} \tag{3.6}$$

Vector $\Delta \omega_{s+1}$ is first computed from (3.6) by Gaussian elimination after which vectors $\Delta \omega_i$, i = 1, 2, ..., s can be computed from (3.5). In (3.5), $(\hat{D}_i)^{-1}$ is not computed explicitly due to reasons of computational efficiency. Rather the following matrix equation is solved by performing the back substitution since $\hat{D}_i$ is upper triangular (bi-diagonal) in form.

$$\hat{D}_i \; \Delta \omega_i = [\hat{\zeta}_i - \hat{R}_i \Delta \omega_{s+1}], \; i = 1, 2, ..., s \tag{3.7}$$

All elements of $\Delta \mathbf{W}^{n+1}$ have now been computed and we can proceed with the remaining computations for the time step t^{n+1}.

An estimation of computation time for solving $\mathbf{A} \Delta \mathbf{W}^{n+1} = \mathbf{z}$ on a single processor is now obtained as follows:

Let

t_f = time for one floating-point operation (i.e. add, subtract, multiply or divide)

n_{av} = average number of non-zero elements in a non-zero row of $\mathbf{R}_i$ or non-zero column of $\mathbf{B}_i$, $i = 1, 2, ..., s$ (This is also equal to the average degree of an end node of a chain minus one)

(i) Forward elimination of $\mathbf{B}_i$ and lower diagonal of $\mathbf{D}_i$, $i = 1, 2, ..., s$.

In the forward elimination process, new non-zero elements are created in the rows

below the first non-zero row of $\mathbf{R}_i$ and in the same column positions. Similarly new non-zero elements are created in the adjacent columns to the first column of $\mathbf{B}_i$ and in the same row positions. In other words, the first row of $\mathbf{R}_i$ and the first column of $\mathbf{B}_i$ duplicate themselves in their non-zero element positions although with different values of the non-zero elements. New non-zero elements are also created in matrix $\mathbf{V}$ due to the presence of non-zero elements in rows of $\mathbf{R}_i$. Elements of $\mathbf{A}$ below the pivot row are modified according to (3.3). Similarly, elements of $\mathbf{z}$ below the pivot row are modified according to (3.3). In fact, the Gaussian elimination is carried out on the augmented matrix $[\mathbf{A}|\mathbf{z}]$ of dimensions k x $(k + 1)$ rather than matrix $\mathbf{A}$. Elimination of first column of $\mathbf{D}_i$ (which has only one non-zero element just below the pivot element) involves one division, $(n_{av} + 2)$ multiplications and $(n_{av} + 2)$ subtractions. Therefore, computation time to eliminate first column of $\mathbf{D}_i$ is $= [1 + (n_{av} + 2) + (n_{av} + 2)]\ t_f = (2\ n_{av} + 5)\ t_f$.

Elimination of a non-zero element in the first column of $\mathbf{B}_i$ involves one division, $(n_{av} + 2)$ multiplications and $(n_{av} + 2)$ subtractions. There are a total of n_{av} non-zero elements in the first column of $\mathbf{B}_i$. Therefore computation time to eliminate the first column of $\mathbf{B}_i$ is

$$n_{av}[1 + (n_{av} + 2) + (n_{av} + 2)]\ t_f$$

$$= (2\ n_{av}^2 + 5\ n_{av})\ t_f \quad .$$

There are a total of p_i columns in $\mathbf{B}_i$ or $\mathbf{D}_i$ where p_i is the length of the chain i.

Therefore, the computation time to eliminate the first $(p_i - 1)$ columns of $\mathbf{D}_i$ and $\mathbf{B}_i$ below the pivots is

$$(p_i - 1)\,(2\,n_{av} + 5 + 2\,n_{av}^2 + 5\,n_{av})\,t_f$$

$$= (p_i - 1)\,(2\,n_{av}^2 + 7\,n_{av} + 5)\,t_f \quad .$$

When the last diagonal element of $\mathbf{D}_i$ is the pivot, there is no element below it in $\mathbf{D}_i$ and there are $2\,n_{av}$ non-zero elements in the last column of $\mathbf{B}_i$ due to the creation of n_{av} new non-zero elements in the last column of $\mathbf{B}_i$ in the previous elimination step. There were already n_{av} non-zero elements in the last column of $\mathbf{B}_i$ in the original matrix $\mathbf{B}_i$. Similarly there are $2\,n_{av}$ non-zero elements in the last row of $\mathbf{R}_i$ as compared to n_{av} in the previous elimination steps. Therefore, computation time to eliminate the last column of $\mathbf{B}_i$ is

$$2\,n_{av}[1 + (2\,n_{av} + 1) + (2\,n_{av} + 1)]\,t_f$$

$$= (8\,n_{av}^2 + 6\,n_{av})\,t_f \quad .$$

Hence, the total time required for the forward elimination of $\mathbf{B}_i$ and lower diagonal of $\mathbf{D}_i$ is

$$(p_i - 1)\,(2\,n_{av}^2 + 7\,n_{av} + 5)\,t_f + (8\,n_{av}^2 + 6\,n_{av})\,t_f$$

$$= p_i\,(2\,n_{av}^2 + 7\,n_{av} + 5)\,t_f + (6\,n_{av}^2 - n_{av} - 5)\,t_f .$$

The above estimation of computation time has not taken into account the overhead due to manipulation of pointers or indices for the generation of addresses for fetching the sparsely stored elements of matrices $\mathbf{B}_i$ and $\mathbf{R}_i$. In typical sparse codes, the value of this overhead has been found to be substantial and is of the order of 100% of the actual floating-point operation count.

Therefore, taking the overhead of pointer manipulation as 100% of floating-point arithmetic operation count, we have total computation time for forward elimination of $\mathbf{B}_i$

and lower diagonal of $\mathbf{D}_i$ equal to

$$2\,p_i(2\,n_{av}^2 + 7\,n_{av} + 5)\,t_f + 2(6\,n_{av}^2 - n_{av} - 5)\,t_f \quad .$$

The total number of such eliminations is s for each $\mathbf{B}_i$, $\mathbf{D}_i$, $i = 1, 2, ..., s$, where s is the number of chains in the network. Therefore, total computation time for forward elimination of all $\mathbf{B}_i$ and lower diagonal of $\mathbf{D}_i$ is

$$\sum_{i=1}^{s} 2\,p_i\,(2\,n_{av}^2 + 7\,n_v + 5)\,t_f + \sum_{i=1}^{s} 2(6\,n_{av}^2 - n_{av} - 5)\,t_f$$

or, after substituting $\sum_{i=1}^{s} p_i = (K - q)$ from (3.1),

$$T_{1a} = \left[(4\,n_{av}^2 + 14\,n_{av} + 10)\,(K - q) + (12\,n_{av}^2 - 2\,n_{av} - 10)s\right] t_f \quad . \qquad (3.8)$$

(ii) Solution of $\Delta\,\hat{\omega}_{s+1}$ from $\hat{\mathbf{V}}\,\Delta\,\boldsymbol{\omega}_{s+1} = \boldsymbol{\zeta}_{s+1}$. This matrix equation is solved by full Gaussian elimination as the $q \times q$ matrix $\hat{\mathbf{V}}$ is almost full. The pseudo code of the algorithm for the general matrix equation $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ of order q is given in Fig. 3.5. In our case, $\mathbf{C} = \hat{\mathbf{V}}$, $\mathbf{x} = \Delta\,\boldsymbol{\omega}_{s+1}$ and $\mathbf{b} = \boldsymbol{\zeta}_{s+1}$.

An estimate of computation time for Gaussian elimination is obtained as follows (refer to Fig. 3.5)

Forward Elimination Phase

for $k = 1$ to $(q - 1)$ do

  for $i = (k + 1)$ to q do

    $\text{temp} = \dfrac{c(i, k)}{c(k, k)}$

    $b(i) = b(i) - \text{temp} \times b(k)$

    for $j = (k + 1)$ to q do

      $c(i\ j) = c(i, j) - \text{temp} \times (c(k, j)$

    end do {loop j}

  end do {loop i}

end do {loop k}

Back-Substitution Phase

(Now the matrix equation is of the form **U x = g** where matrix **U** is upper triangular in form)

$x_q = \dfrac{g_q}{u_{qq}}$

for $i = (q - 1)$ to 1 do

  for $j = (i + 1)$ to q do

    $g(i) = g(i) - u(i, j) \times x(j)$

  end do {loop j}

  $x(i) = \dfrac{g(i)}{u(i, i)}$

end do {loop i}

Figure 3.5 Pseudo-code for the Gaussian elimination of **C x = b.**

$$\text{forward elimination time} = \sum_{k=1}^{(q-1)} (q-k)\,\{2(q-k)+3\}t_f$$

$$= \left\{2\sum_{k=1}^{(q-1)} (q-k)^2 + 3\sum_{k=1}^{(q-1)} (q-k)\right]t_f$$

$$= \Big[2\,\{1^2 + 2^2 + \cdots + (q-1)^2\}$$

$$+ 3\{1 + 2 + \cdots + (q-1)\}\Big]t_f$$

$$= \left[\frac{2\,(q-1)\,(q)\,\{2(q-1)+1\}}{6} + \frac{3(q-1)\,q}{2}\right]t_f$$

$$\text{Since } 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)\,(2n+1)}{6}$$

$$\text{and } 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Therefore,

$$\text{forward elimination time} = \left[\frac{2}{3}\,q^3 + \frac{q^2}{2} - \frac{7}{6}\,q\right]t_f$$

$$\text{back–substitution time} = \left[1 + \sum_{i=(q-1)}^{1} \{2(q-i)+1\}\right]t_f$$

$$= \Big[1 + 2\{1 + 2 + \cdots + (q-1)\} + (q-1)\Big]\,t_f$$

$$= \left[q + 2\,\frac{(q-1)\,q}{2}\right]t_f$$

$$= q^2\,t_f$$

Therefore, total time for Gaussian elimination is

$$\left[\frac{2}{3}\,q^3 + \frac{q^2}{2} - \frac{7}{6}q + q^2\right] t_f$$

or

$$T_{1b} = \left[\frac{2}{3}\,q^3 + \frac{3}{2}\,q^2 - \frac{7}{6}\,q\right] t_f \quad . \tag{3.9}$$

(iii) Solution of vectors $\Delta\,\omega_i$, $i = 1, 2, \ldots, s$ from back-subtraction in (3.7).

Evaluation of $\mathbf{x} = (\hat{\zeta}_i - \mathbf{R}_i \cdot \Delta\,\omega_{s+1})$ involves n_{av} multiplications and n_{av} subtractions per row for the first $(p_i - 1)$ rows and 2 n_{av} multiplications and 2 n_{av} subtractions for the last row. Assuming 100% overhead of sparse pointer manipulation,

Computation time for evaluating $\left(\hat{\zeta}_i - \hat{\mathbf{R}}_i\,\Delta\,\omega_{s+1}\right)$

$$2\left[2\,n_{av}\,(p_i - 1) + 4\,n_{av}\right] t_f$$

$$= \left(4\,n_{av}\,p_i + 4\,n_{av}\right) t_f \quad .$$

Matrices $\hat{\mathbf{D}}_i$ are upper triangular with one leading main diagonal and one superdiagonal above the main diagonal. Evaluation of $\Delta\,\omega_i$ from $\hat{\mathbf{D}}_i\,\Delta\,\omega_i = \mathbf{x}$ is done by back-substitution and involves only one division for the last element of $\Delta\,\omega_i$ and one multiplication, one subtraction, and division for the remaining $(p_i - 1)$ elements.

Therefore, time to compute $\Delta\,\omega_i$ from $\hat{\mathbf{D}}_i\,\Delta\,\omega_i = \mathbf{x}$ is

$$[1 + 3\,(p_i - 1)]\,t_f$$

$$= (3\,p_i - 2)\,t_f \quad .$$

Hence, total time for evaluating one $\Delta\,\omega_i$ is

$$\left[4\, n_{av}\, p_i + 4\, n_{av} + 3\, p_i - 2\right] t_f$$

$$= \left[(4\, n_{av} + 3)p_i + (4\, n_{av} - 2)\right] t_f \quad .$$

A total of s such evaluations are to be done for each $i = 1, 2, ..., s$. Therefore, total time to evaluate all $\Delta\, \omega_i$, $i = 1, 2, ..., s$ is

$$\sum_{i=1}^{s} \left[(4\, n_{av} + 3)p_i + (4\, n_{av} - 2)\right] t_f$$

or

$$T_{1c} = \left[(4\, n_{av} + 3)\, (K - q) + (4\, n_{av} - 2)s\right] t_f \quad . \tag{3.10}$$

Now the total time T_1(matrix) to compute $\Delta\, \mathbf{W}^{n+1}$ from $\mathbf{A}\, \Delta\, \mathbf{W}^{n+1} = \mathbf{z}$ on a single processor is the sum total of times T_{1a}, T_{1b} and T_{1c} given by (3.8), (3.9) and (3.10) respectively or

$$T_1(matrix) = T_{1a} + T_{1b} + T_{1c}$$

$$= \left[(4\, n_{av}^2 + 14\, n_{av} + 10)\, (K - q) + (12\, n_{av}^2 - 2\, n_{av} - 10)s\right.$$

$$\left. + \frac{2}{3}\, q^3 + \frac{3}{2}\, q^2 - \frac{7}{6}\, q + (4\, n_{av} + 3)\, (K - q) + (4\, n_{av} - 2)s\right] t_f \quad .$$

or

$$T_1(\text{matrix}) = \left[\frac{2}{3}\, q^3 + \frac{3}{2}\, q^2 + (4\, n_{av} + 18\, n_{av} + 13)\, (K - q)\right.$$

$$\left. + (12\, n_{av}^2 + 2\, n_{av} - 12)s - \frac{7}{6}\, q\right] t_f \quad . \tag{3.11}$$

The break-down of total matrix solution time T_1(matrix) for a number of network

examples given in Table 3.1 is plotted in Fig. 3.6 as a percentage of total matrix solution time T_1(matrix). As can be seen from this plot, the total computation time is dominated by the time T_{1b}.

3.2.1.2.2. Direct method II (Block Gaussian elimination method)

In this method, various submatrices of matrix **A** are treated as block sub-matrices throughout the solution process. The computational procedure is as follows [6].

Matrix (2.49) can be written in an equivalent form as a collection of $(s + 1)$ lower-order matrix equations involving block submatrices $\mathbf{D}_i$, $\mathbf{B}_i$, $\mathbf{R}_i$ and **V**, i.e.,

$$\mathbf{D}_i \, \Delta \, \omega_i + \mathbf{R}_i \, \Delta \, \omega_{s+1} = \zeta_i, \; i = 1, 2, ..., s \tag{3.12}$$

$$\sum_{i=1}^{s} \mathbf{B}_i \, \Delta \, \omega_i + \mathbf{V} \, \Delta \, \omega_{s+1} = \zeta_{s+1} \tag{3.13}$$

Equation (3.12) can be solved for vector $\Delta \, \omega_i$ as

$$\Delta \, \omega_i = \mathbf{D}_i^{-1} \, \zeta_i - \mathbf{D}_i^{-1} \, \mathbf{R}_i \, \Delta \, \omega_{s+1}, \; i = 1, 2, ..., s$$

or

$$\Delta \, \omega_i = \mathbf{c}_i - \mathbf{E}_i \, \Delta \, \omega_{s+1}, \; i = 1, 2, ..., s \tag{3.14}$$

where

$$\mathbf{c}_i = \mathbf{D}_i^{-1} \, \zeta_i, \; i = 1, 2 ..., s \tag{3.15}$$

$$\mathbf{E}_i = \mathbf{D}_i^{-1} \, \mathbf{R}_i, \; i = 1, 2, ..., s \tag{3.16}$$

Here $\mathbf{c}_i$ are $(p_i \times 1)$ vectors and $\mathbf{E}_i$ are $(p_i \times q)$ matrices. After substituting $\Delta \, \omega_i$ from (3.14) into (3.13), we have

Table 3.1

Six network examples used in this thesis for studying the performance of algorithms.

Network example	Number of total links K	Number of random links q	Number of chains s	Number of nodes N
#1	50	15	3	45
#2	100	30	5	90
#3	200	60	10	180
#4	300	90	15	270
#5	400	120	20	360
#6	500	150	25	450

Average number of non-zero elements in a row of $\mathbf{B}_i = n_{av} = 2.0$.

Maximum number of non-sero elements in any row of $B_i = n_{\max} = 6$.

Average degree of a node connected to a random link = $d_{av} = 3.0$.

Maximum degree of any node connected to a random link = $d_{\max} = 7$.

Maximum length of a chain in the network = $p_m = 20$.

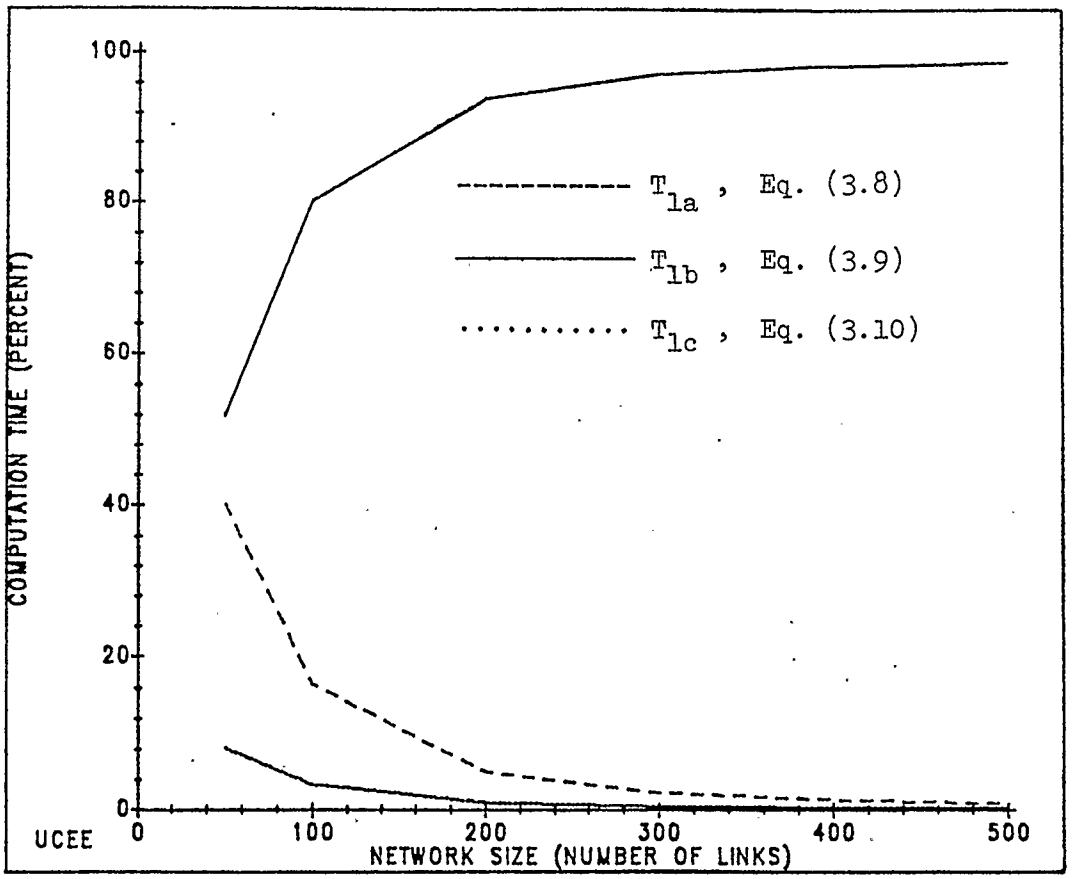


Figure 3.6 Computation times in the three stages of matrix equation solution by direct method I.

$$\sum_{i=1}^{s} \mathbf{B}_i (\mathbf{c}_i - \mathbf{E}_i \Delta \omega_{s+1}) + \mathbf{V} \Delta \omega_{s+1} = \zeta_{s+1}$$

or

$$(\mathbf{V} - \sum_{i=1}^{s} \mathbf{B}_i \mathbf{E}_i) \Delta \omega_{s+1} = \zeta_{s+1} - \sum_{i=1}^{s} \mathbf{B}_i \mathbf{c}_i$$

or

$$(\mathbf{V} - \sum_{i=1}^{s} \mathbf{G}_i) \Delta \omega_{s+1} = (\zeta_{s+1} - \sum_{i=1}^{s} \mathbf{h}_i) \tag{3.17}$$

where

$$\mathbf{G}_i = \mathbf{B}_i \mathbf{E}_i, \; i = 1, 2, ..., s \tag{3.18}$$

$$\mathbf{h}_i = \mathbf{B}_i \mathbf{c}_i, \; i = 1, 2, ..., s \tag{3.19}$$

Here, $\mathbf{G}_i$ are $(q \times q)$ matrices and $\mathbf{h}_i$ are $(q \times 1)$ vectors. Equation (3.17) can now be written as

$$\mathbf{Y} \Delta \omega_{s+1} = \mathbf{x} \tag{3.20}$$

where

$$\mathbf{Y} = \mathbf{V} - \sum_{i=1}^{s} \mathbf{G}_i \tag{3.21}$$

$$\mathbf{x} = \zeta_{s+1} - \sum_{i=1}^{s} \mathbf{h}_i \tag{3.22}$$

The flowchart of computational sequence is given in Fig. 3.7.

An estimation of computational time for solving $\mathbf{A} \Delta \boldsymbol{\omega} = \mathbf{z}$ by block Gaussian elimination method can be obtained as follows.

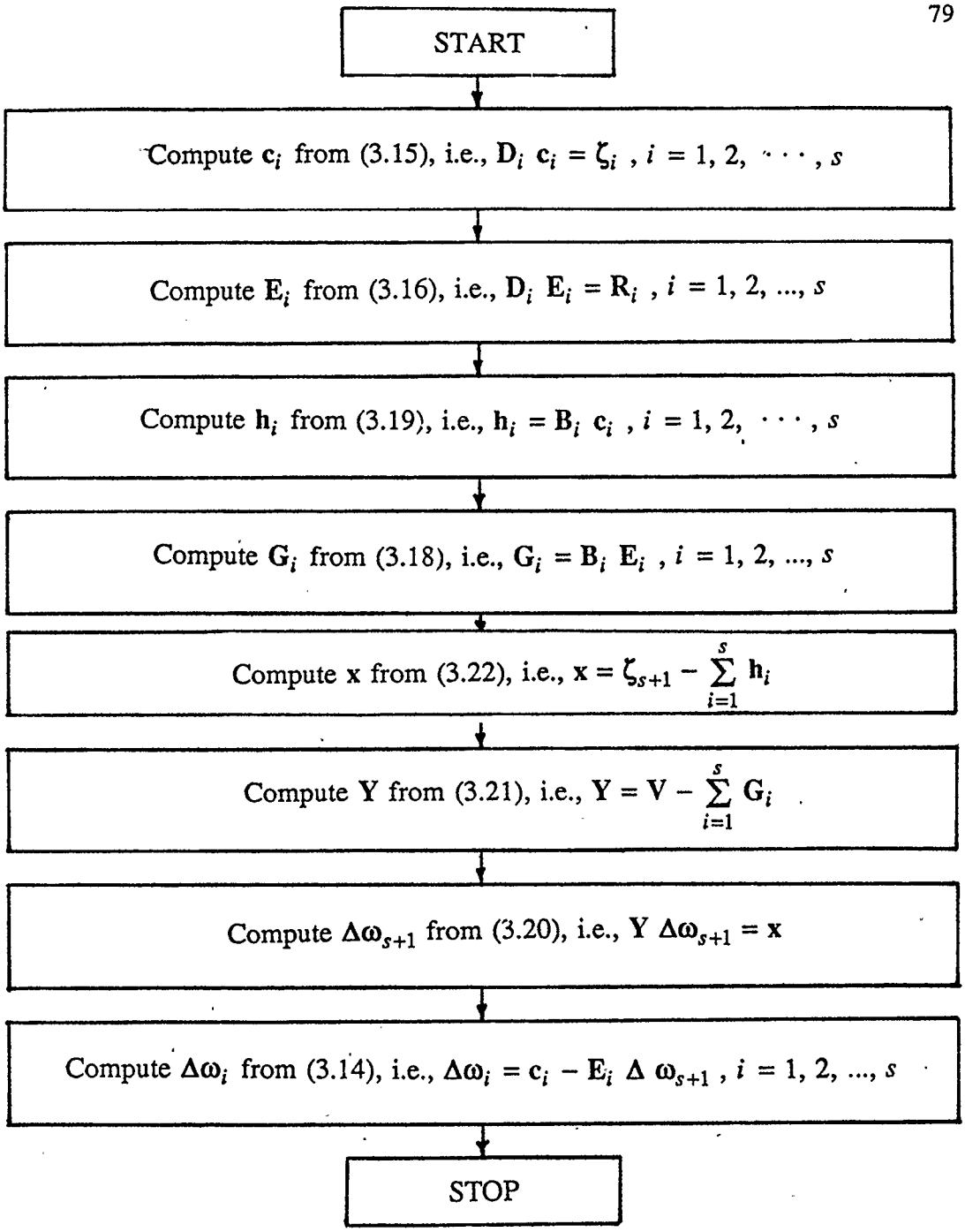


Figure 3.7 Flow chart for computing $\Delta\ \mathbf{W}^{n+1}$ from $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ by direct method II (block Gaussian elimination method)

(i) Computation of vectors $\mathbf{c}_i$ from $\mathbf{c}_i = \mathbf{D}_i^{-1}\,\zeta_i,\ i = 1, 2, ..., s$

In this computation, the inverse of $\mathbf{D}_i$ is not computed explicitly. Rather, a system of equations $\mathbf{D}_i, \mathbf{c}_i, = \zeta_i$ is solved for $\mathbf{c}_i$ by Gaussian elimination which is more efficient than computing the inverse of $\mathbf{D}_i$.

Let us now consider the solution of a general system of tridiagonal equations. $\mathbf{T}\,\mathbf{x} = \mathbf{b}$ of order n by **L U** factorization. In this method, matrix **T** is written as a product of two matrices **L** and **U** where **L** is lower-triangular and **U** is upper-triangular in structure. Hence,

$$\mathbf{L}\,\mathbf{U}\,\mathbf{x} = \mathbf{b}$$

$$\mathbf{L}\ \mathbf{g} = \mathbf{b} \tag{3.23}$$

and

$$\mathbf{U}\,\mathbf{x} = \mathbf{g} \tag{3.24}$$

Therefore, if the **L** and **U** factors of **T** are known, vector **x** can be obtained by first computing intermediate vector **g** from (3.23) by forward substitution and then computing **x** from (3.24) by back substitution. One form of **L U** factors also known as "Crout reduction" [35] is given by

$$\begin{bmatrix} t_{11} & t_{12} & & & & & \\ t_{21} & t_{22} & t_{23} & & & & \\ & t_{32} & t_{33} & t_{34} & & & \\ & & & & & & \\ & & & & & & \\ & & & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} & \\ & & & & t_{n,n-1} & t_{n,n} & \end{bmatrix} =$$

$$\begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ & & & \\ & & l_{n,n-1} & l_{n,n} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & & \\ & 1 & u_{23} & \\ & & & \\ & & 1 & u_{n-1,n} \\ & & & 1 \end{bmatrix}$$

By equating the corresponding elements on both sides of the above matrix equation, we have the following recursions:

$$\left.\begin{aligned} & l_{11} = t_{11} \ , \ u_{12} = t_{12}/l_{11} \\ & l_{i,i-1} = t_{i,i-1} \ , \ i = 2, 3, ..., n \\ & l_{ii} = t_{ii} - l_{i,i-1} \ , u_{i-1,i} \ , \ i = 2, 3, ..., n \\ & u_{i,i+1} = t_{ii}/l_{ii} \ , \ i = 2, 3, ..., n-1 \end{aligned}\right\} \tag{3.25}$$

Therefore, operation count for obtaining **L U** factors is

$$1 + 2(n-1) + (n-2)$$

$$= (3n - 3) \ .$$

The intermediate solution vector **g** is obtained from (3.23) by the following equations obtained by forward substitution.

$$\left.\begin{aligned} & g_1 = \frac{b_1}{l_{11}} \\ & g_i = \frac{(b_i - l_{i,i-1}\, g_{i-1})}{l_{ii}} \ , \ i = 2, 3, .., n \end{aligned}\right\} \tag{3.26}$$

Operation count for obtaining **g** is

$$1 + 3(n - 1)$$

$$= (3n - 2) \cdot$$

The final solution **x** is recovered from (3.24) as

$$\left.\begin{aligned} x_n &= g_n \\ x_i &= g_i - u_{i,i+1}\, x_{i+1} \quad , \quad i = n - 1,\, n - 2 \ldots,\, 1 \end{aligned}\right\} \cdot \qquad (3.27)$$

Operation count for backward substitution to obtain vector **x** is

$$= 2(n - 1) \cdot$$

Therefore, total operation count to solve $\mathbf{T}\,\mathbf{x} = \mathbf{b}$ for vector **x** is

$$(3n - 3) + (3n - 2) + 2(n - 1)$$

$$= (8n - 7) \cdot$$

If t_f is the time for one floating-point operation, then total computation time is $(8n - 7)\, t_f$. In our case, $n = p_i$ and therefore, time to compute all $\mathbf{c}_i$ vectors is

$$\sum_{i=1}^{s} (8\, p_i - 7)\, t_f$$

or

$$T_{1a} = [8(K - q) - 7s]\, t_f \cdot \qquad (3.28)$$

(ii) Computation of matrices $\mathbf{E}_i$ from $\mathbf{E}_i = \mathbf{D}_i^{-1}\ \mathbf{R}_i$, $i = 1, 2, .., s$

As was done in computing $\mathbf{c}_i$, $\mathbf{E}_i$ are computed by solving the matrix equation

$$\mathbf{D}_i \mathbf{E}_i = \mathbf{R}_i$$

However, the **L** and **U** factors of $\mathbf{D}_i$ have already been computed while computing $\mathbf{c}_i$ in part (i) and need not be computed again. Therefore, only forward and

backward substitutions as given by (3.23) and (3.24) are to be performed for each right hand-side column of $\mathbf{R}_i$. Only the first and last rows of $\mathbf{R}_i$ have non-zero elements in them. Assuming the average of n_{av} non-zero elements in the first row and the same number in the last row of $\mathbf{R}_i$, at most 2 n_{av} number of columns of $\mathbf{R}_i$ will have a non-zero element in them. Rest of the $(q - 2\ n_{av})$ columns of $\mathbf{R}_i$ are null-vectors, i.e. have only zero elements in them. Therefore, the matrix equation is to be solved only for those columns of $\mathbf{R}_i$ which have a non-zero element in them. The matrix solution is a null-vector for those columns of $\mathbf{R}_i$ which contain all zeros since they give rise to a homogeneous system of equations which have a null-vector as a solution vector [36]. As an example, consider

$$\mathbf{R}_i = \begin{bmatrix} x & o & o & x & o & o \\ o & o & o & o & o & o \\ o & o & o & o & o & o \\ o & o & o & o & o & o \\ o & o & o & o & o & x \end{bmatrix}$$

where "x" denotes a non-zero element.

The matrix equation $\mathbf{D}_i\mathbf{E}_i = \mathbf{R}_i$ has to be solved only for the first, fourth and last column of $\mathbf{R}_i$. The solution will be a null-vector for the remaining three columns of $\mathbf{R}_i$.

For those columns of $\mathbf{R}_i$ which have a non-zero element in the first row position and remaining zeroes (such as column one or four in the above example), (3.26) for forward substitution are modified as

$$g_1 = \frac{b_1}{l_{11}}$$

$$g_i = -\frac{l_{i,i-1}}{l_{ii}} \cdot g_{i-1} \quad , \quad i = 2, 3, \ldots, n \quad .$$

Therefore, computation time for one forward substitution is $[1 + 2\ (n - 1)]\ t_f$

$$= (2n - 1)\ t_f \quad .$$

Equations (3.27) for backward substitution do not change in this case. Time for backward substitution is $2(n - 1)\ t_f$. Therefore, total solution time is

$$(2n - 1 + 2n - 2)\ t_f$$

$$= (4n - 3)\ t_f \quad .$$

In our case $n = p_i$ and n_{av} such equations are to be solved. Therefore, the total computation time for n_{av} such vectors of $\mathbf{R}_i$ is

$$n_{av}\ (4\ p_i - 3)\ t_f \quad .$$

For those columns of R_i which contain a non-zero element in their last row, (3.26) and (3.27) are modified as

$$g_i = 0 \quad , \quad i = 1, 2, \ldots, n - 1$$

$$g_n = \frac{b_n}{l_{nn}}$$

$$x_n = g_n$$

$$x_i = -u_{i,i+1}\ x_{i+1} \quad , \quad i = n - 1, \ldots, 1 \quad .$$

Total computation time for n_{av} such vectors of $\mathbf{R}_i$ is

$$n_{av}\ p_i\ t_f \cdot$$

Therefore, total time to compute $\mathbf{E}_i$ is $n_{av}\ (5\ p_i - 3)\ t_f$. Hence, total time to compute all $\mathbf{E}_i$, $i = 1, 2, ..., s$ is $\sum_{i=1}^{s} n_{av}\ (5\ p_i - 3)\ t_f$

$$\text{or} \qquad T_{1b} = [5\ n_{av}\ (K - q) - 3\ n_{av}\ s]\ t_f \cdot \tag{3.29}$$

Matrix $\mathbf{E}_i$ for the example matrix $\mathbf{R}_i$ discussed previously, will have the following form

$$\mathbf{E}_i = \begin{bmatrix} x & o & o & x & o & x \\ x & o & o & x & o & x \\ x & o & o & x & o & x \\ x & o & o & x & o & x \\ x & o & o & x & o & x \end{bmatrix} \cdot$$

(iii) Computation of $\mathbf{h}_i$ from $\mathbf{h}_i = \mathbf{B}_i \mathbf{c}_i$, $i = 1, 2, ..., s$:

There are n_{av} non-zero elements in the first column of $\mathbf{B}_i$ and the same number of non-zero elements in the last column of $\mathbf{B}_i$. The rest of the columns of $\mathbf{B}_i$ are all zero. Taking 100% over-head for sparse operations,

the total computation time for computing all $\mathbf{h}_i$ is

$$\sum_{i=1}^{s} 2\ (n_{av} + n_{av})\ t_f$$

or

$$T_{1c} = 4\ n_{av} \cdot s\ t_f \cdot \tag{3.30}$$

There will be $2\ n_{av}$ non-zero elements in the vector $\mathbf{h}_i$.

(iv) Computation of $\mathbf{G}_i = \mathbf{B}_i\ \mathbf{E}_i$, $i = 1, 2, ..., s$:

A total of $2\ n_{av}$ columns of each $\mathbf{E}_i$ are full vectors. The rest of the columns are null-vectors. Therefore, matrix $\mathbf{G}_i$ will have $2\ n_{av}$ non-zero columns with $2\ n_{av}$ non-zero elements per column. The rest of the columns of $\mathbf{G}_i$ will be null-vectors. Moreover, $\mathbf{G}_i$ will be symmetric with respect to the position of non-zero elements. For the example matrix $\mathbf{R}_i$ of section (ii), since the structure of $\mathbf{B}_i$ is the transpose of that of $\mathbf{R}_i$, we have,

$$\mathbf{G}_i = \mathbf{B}_i \mathbf{E}_i$$

$$= \begin{bmatrix} x & o & o & o & o \\ o & o & o & o & o \\ o & o & o & o & o \\ x & o & o & o & o \\ o & o & o & o & o \\ o & o & o & o & x \end{bmatrix} \begin{bmatrix} x & o & o & x & o & x \\ x & o & o & x & o & x \\ x & o & o & x & o & x \\ x & o & o & x & o & x \\ x & o & o & x & o & x \end{bmatrix}$$

$$= \begin{bmatrix} x & o & o & x & o & x \\ o & o & o & o & o & o \\ o & o & o & o & o & o \\ x & o & o & x & o & x \\ o & o & o & o & o & o \\ x & o & o & x & o & x \end{bmatrix} .$$

Taking 100% overhead for sparse operations, total time to compute all $\mathbf{G}_i$ matrices is

$$\sum_{i=1}^{s} 2\ [(2\ n_{av})\ n_{av} + (2\ n_{av})\ n_{av}]\ t_f$$

or

$$T_{1d} = 8\ n_{av}^2\ s\ \ t_f \quad . \tag{3.31}$$

(v) Computation of vector $\mathbf{x} = \zeta_{s+1} - \sum_{i=1}^{s} \mathbf{h}_i$

$2\, n_{av}$ elments of $\mathbf{h}_i$ are non-zero. Taking 100% overhead due to sparse computations, time to compute all $\mathbf{h}_i$ is $\sum_{i=1}^{s} 2(2\, n_{av})\, t_f$

$$or \qquad T_{1e} = 4\, n_{av}\, s\, t_f \quad . \tag{3.32}$$

(vi) Computation of $\mathbf{Y}$ from $\mathbf{Y} = \mathbf{V} - \sum_{i=1}^{s} \mathbf{G}_i$

Matrix $\mathbf{G}_i$ has $2\, n_{av}$ non-zero columns with $2\, n_{av}$ non-zero elements per column.

Therefore, total time to compute $\mathbf{Y}$ is $\sum_{i=1}^{s} 2(2\, n_{av})\, (2\, n_{av})\, t_f$

$$or \qquad T_{1f} = 8\, n_{av}^2\, s\, t_f \quad . \tag{3.33}$$

(vii) Computation of vector $\Delta\, \omega_{s+1}$ from $\mathbf{Y}\, \Delta\, \omega_{s+1} = \mathbf{x}$

This is done by normal Gaussian elimination as described in section 3.2.2.1. The computation time is given by (3.9). Therefore, time to compute $\Delta\, \omega_{s+1}$ is $\left[\frac{2}{3}\, q^3 + \frac{3}{2}\, q^2 - \frac{7}{6}\, q\right] t_f$

$$or \qquad T_{1g} = \left[\frac{2}{3}\, q^3 + \frac{3}{2}\, q^2 - \frac{7}{6}\, q\right] t_f \quad . \tag{3.34}$$

(viii) Computation of $\Delta\, \omega_i$ from $\Delta\, \omega_i = \mathbf{c}_i - \mathbf{E}_i\, \Delta\, \omega_{s+1}$, $i = 1, 2, \ldots, s$

Matrix $\mathbf{E}_i$ has $2\, n_{av}$ number of full column vectors and the remaining $(q - 2\, n_{av})$ columns are null vectors. Therefore, computation of one element of $\Delta\, \omega_i$ requires $2\, n_{av}$ multiplications and $2\, n_{av}$ subtractions. Taking 100% overhead for sparse index

manipulations,

total time to compute all $\Delta\,\omega_i$ is $\sum_{i=1}^{s} 2(2\,n_{av} + 2\,n_{av})\,p_i\,t_f$

$$or \qquad T_{1h} = 8\,n_{av}\,(K-q)\,t_f \qquad (3.35)$$

Now the total time to solve the matrix equation $\mathbf{A}\,\Delta\mathbf{W}^{n+1} = \mathbf{z}$ for $\Delta\,\mathbf{W}^{n+1}$ will be the sum of all the above computation times, i.e.

$$T_1(\text{matrix}) = T_{1a} + T_{1b} + T_{1c} + T_{1d} + T_{1e} + T_{1f} + T_{1g} + T_{1h}$$

or

$$\begin{aligned} T_1(\text{matrix}) = [&8(K-q) - 7s + 5\,n_{av}\,(K-q) - 3\,n_{av}\,s + 4\,n_{av}\,s \\ &+ 8\,n_{av}^2\,s + 4\,n_{av}\,s + 8\,n_{av}^2\,s \\ &+ \frac{2}{3}\,q^3 + \frac{3}{2}\,q^2 - \frac{7}{6}\,q \\ &+ 8\,n_{av}\,(K-q)]\,t_f \end{aligned}$$

or

$$\begin{aligned} T_1(\text{matrix}) = [&\frac{2}{3}\,q^3 + \frac{3}{2}\,q^2 + (13\,n_{av} + 8)\,(K-q) \\ &+ (16\,n_{av}^2 + 5\,n_{av} - 7)s - \frac{7}{6}\,q]\,t_f \end{aligned} \qquad (3.36)$$

A comparison of these two methods with respect to the processing times for solving $\mathbf{A}\,\Delta\mathbf{W} = \mathbf{z}$, i.e. eqns. (3.11) and (3.36) for a number of network examples is shown in Fig. 3.8. The number of random links in the network has been taken to be 30% of the total links in the network which is typical of nuclear plant hydraulic networks [37]. As can be seen from Fig. 3.8, both methods take about the same time for large networks.

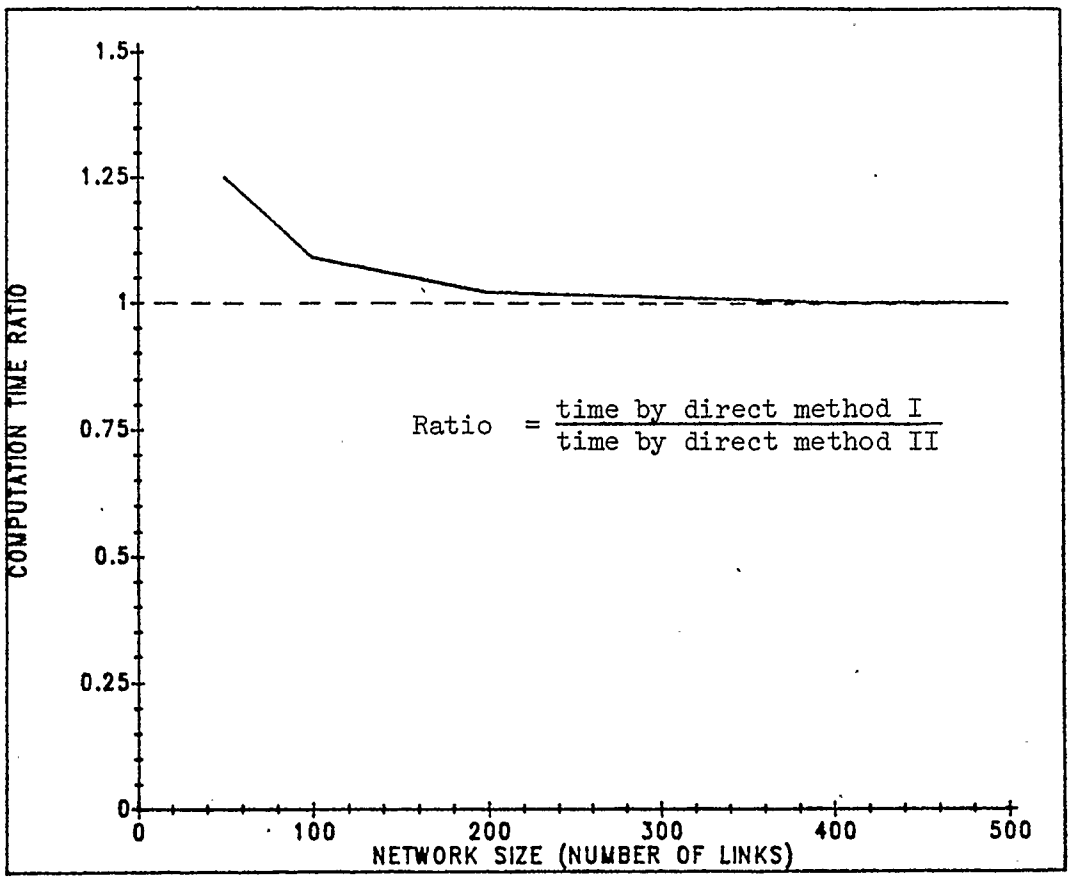


Figure 3.8 Comparision of matrix equation solution times by direct method I and direct method II.

3.2.2. Estimation of processing time for the remaining computations

In this section, an estimate of processing time for the remaining computations (after getting $\Delta \mathbf{W}^{n+1}$ from (2.40)) is obtained. Since the number of critical links in a network is usually very small as compared to non-critical or normal links, computation time required to obtain flow rates in the critical links has been neglected. The flow chart for these computations (after computing $\Delta \mathbf{W}^{n+1}$) is given in Fig. 3.9.

Let

d_{av} = average degree of a non-chained node in the network

N_c = Total number of nodes of degree 2 (i.e. chained nodes) in the network

N_{nc} = Total number of non-chained nodes in the network

Now the total number of chained links in the network = $(K - q)$. Therefore,

$$N_c = (K - q - s) \tag{3.37}$$

and

$$N_{nc} = N - N_c$$

or

$$N_{nc} = (N - K + q + s) \cdot \tag{3.38}$$

An estimate of computation time for the computations given in flow chart of Fig. 3.9 can now be obtained as follows:

1. Computation of ΔU_i^{n+1}, $i = 1, 2, ..., N$ from (2.55):

   Computation of ΔU_i^{n+1} for a node of degree 2, i.e. chained node, requires three multiplications and two additions. Therefore, computation time for computing ΔU_i^{n+1}

Compute the following at time step t^{n+1} :

ΔU_i^{n+1} , ΔM_i^{n+1} , $i = 1, 2, ..., N$ {from (2.55) and (2.56)}

$W_k^{n+1} = W_k^n + \Delta W_k^{n+1}$, $k = 1, 2, \cdots, K$

$U_i^{n+1} = U_i^n + \Delta U_i^{n+1}$, $i = 1, 2, \cdots, N$

$M_i^{n+1} = M_i^n + \Delta M_i^{n+1}$, $i = 1, 2, \cdots, N$

$u_i^{n+1} = U_i^{n+1}/M_i^{n+1}$, $i = 1, 2, \cdots, N$ {u_i is the specific internal energy of node i}

$v_i^{n+1} = \dfrac{V_i}{M_i^{n+1}}$, $i = 1, 2, \cdots, N$ {v_i is the specific volume and V_i is the volume of node i}

P_i^{n+1} {from (2.6)}

$H_i^{n+1} = u_i^{n+1} + P_i^{n+1} v_i^{n+1}$, $i = 1, 2, \cdots, N$

$\dfrac{\partial P_i}{\partial U_i}$, $\dfrac{\partial P_i}{\partial M_i}$, $i = 1, 2, \cdots, N$ {from steam tables}

f_k , $k = 1, 2, \cdots, K$ {from right–hand side of (2.2)}

f_{K+i} , $i = 1, 2, \cdots, N$ {from right–hand side of (2.4)}

f_{K+N+i} , $i = 1, 2, \cdots, N$ {from right–hand side of (2.5)}

$\dfrac{\partial f_k}{\partial U_i}$, $\dfrac{\partial f_k}{\partial U_j}$, $\dfrac{\partial f_k}{\partial M_i}$, $\dfrac{\partial f_k}{\partial M_j}$, $k = 1, 2, \cdots, K$ {by differentiating right–hand side of (2.2)}

z_k , $k = 1, 2, ..., K$ {from (2.57)}

a_{kk} , $k = 1, 2, \cdots, K$ {from (2.58)}

a_{kv} , $k = 1, 2, \cdots, K$ {from (2.59)}

Figure 3.9 Flow chart for the remaining computations in the Porsching's Algorithm.

for all the chained nodes is

$$5\ N_c\ t_f$$

$$= 5(K - q - s)\ t_f \quad .$$

Computation of $\Delta\ U_i^{n+1}$ for a non-chained node requires $(d_{av} + 1)$ multiplications and d_{av} additions/subtractions. Therefore, computation time for computing $\Delta\ U_i^{n+1}$ for all the non-chained nodes is

$$(2\ d_{av} + 1)\ N_{nc}\ t_f$$

$$= (2\ d_{av} + 1)\ (N - K + q + s)\ t_f \quad .$$

Hence, total computation time for all $\Delta\ U_i^{n+1}$ is

$$[5\ K - q - s) + (2\ d_{av} + 1)\ (N - K + q + s)]\ t_f \quad .$$

2. Computation of $\Delta\ M_i^{n+1}, i = 1, 2, ..., N$ from (2.56):

Computation time for all ΔM_i^{n+1} is

$$[3(K - q - s + (d_{av} + 1)\ (N - K + q + s)]\ t_f \quad .$$

3. Computation of $W_k^{n+1} = W_k^n + \Delta\ W_k^{n+1}$, $k = 1, 2, ..., K$:

computation time is $K\ t_f$.

4. Computation of $U_i^{n+1} = U_i^n + \Delta\ U_i^{n+1}$, $i = 1, 2, ..., N$:

computation time is $N\ \ t_f$.

5. Computation of $M_i^{n+1} = M_i^n + \Delta\ M_i^{n+1}$, $i = 1, 2, ..., N$:

computation time is $N\ \ t_f$.

6. Compuation of $u_i^{n+1} = U_i^{n+1}/M_i^{n+1}$, $i = 1, 2, ..., N$:

computation time is $N\ t_f$.

7. Computation of v_i^{n+1} from $v_i^{n+1} = \dfrac{V_i}{M_i^{n+1}}$, $i = 1, 2, ..., N$:

computation time is $N\ t_f$.

8. Computation of P_i from (2.6):

computation time is $N\ T_p\ t_f$.

where $T_p\ t_f$ = average computation time to compute pressure at a node from steam tables.

9. Computation of $H_i^{n+1} = u_i^{n+1} + P_i^{n+1}\ v_i^{n+1}$, $i = 1, 2, ..., N$:

computation time is $2\ N\ t_f$.

10. Computation of $\dfrac{\partial P_i}{\partial U_i}, \dfrac{\partial P_i}{\partial M_i}$, $i = 1, 2, ..., N$:

computation time is $N(T_{dpu} + T_{dpm})\ t_f$.

where, $T_{dpu}\ t_f$ = time to compute one $\dfrac{\partial P_i}{\partial U_i}$

$T_{dpm}\ t_f$ = time to compute one $\dfrac{\partial P_i}{\partial M_i}$.

11. Computation of f_k, $k = 1, 2, ..., K$ from right-hand side of (2.2):

The constants $\left[\frac{A_k}{L_k}\right]$, $\left[\frac{(fr)_k\ L_k}{2\ D_k\ A_k^2}\right]$, and $(g\ \Delta\ Z_k)$ are computed only once in the initialization step and not at every time-step. Therefore, these operations will not be counted.

computation time is $8\ K\ t_f$.

12. Computation of f_{K+i} , $i = 1, 2, ..., N$ from the right-hand side of (2.4):

computation time for the chained nodes is $4\ N_c\ t_f = 4(K - q - s)\ t_f$.

computation time for the non-chained nodes is $2\ d_{av}\ N_{nc}\ t_f$

$$= 2\ d_{av}\ (N - K + q + s)\ t_f.$$

Total computation time for all f_{K+i} is $[4(K - q - s) + 2\ d_{av}\ (N - K + q + s]\ t_f$.

13. Computation of f_{K+N+i} , $i = 1, 2, ..., N$ from the right-hand side of (2.5): from (2.5),

total computation time for all f_{K+N+i} is $[(K - q - s) +$

$$(d_{av} - 1)\ (N - K + q + s)]\ t_f.$$

14. Computation of $\frac{\partial f_k}{\partial U_i}$, $\frac{\partial f_k}{\partial U_j}$, $k = 1, 2, ..., K$: from right-hand side of (2.2),

$$\frac{\partial f_k}{\partial U_i} = \left[\frac{A_k}{L_k}\right] \frac{\partial P_i}{\partial U_i}$$

$$\frac{\partial f_k}{\partial U_j} = \left[-\frac{A_k}{L_k}\right] \frac{\partial P_j}{\partial U_j} \quad .$$

The derivatives $\frac{\partial P_i}{\partial U_i}$, $\frac{\partial P_j}{\partial U_j}$ have already been computed in step 10.

Therefore, computation time is $2\ K\ t_f$.

15. Computation of $\frac{\partial f_k}{\partial M_i}$, $\frac{\partial f_k}{\partial M_j}$, $k = 1, 2, ..., K$:

From the right-hand side of (2.2),

for a positive flow in the link k,

$$\frac{\partial f_k}{\partial M_i} = \left[\frac{A_k}{L_k}\right] \left[\frac{\partial P_i}{\partial M_i} + \left[\frac{(fr)_k\ L_k\ v_i}{2\ D_k\ A_k^2}\right] \frac{W_k \cdot |W_k|}{M_i^2} - \left[\frac{g\ \Delta Z_k}{v_i}\right]\right]$$

$$\frac{\partial f_k}{\partial M_j} = \left[-\frac{A_k}{L_k}\right] \frac{\partial P_i}{\partial M_j} \quad .$$

for the negative flow in the link k,

$$\frac{\partial f_k}{\partial M_i} = \left[\frac{A_k}{L_k}\right] \frac{\partial P_i}{\partial M_i}$$

$$\frac{\partial f_k}{\partial M_j} = \left[\frac{A_k}{L_k}\right] \left[-\frac{\partial P_j}{\partial M_j} + \left[\frac{(fr)_k\ L_k\ v_j}{2\ D_k\ A_k^2}\right] \frac{W_k |W_k|}{M_j^2} - \left[\frac{g\ \Delta Z_k}{v_j}\right]\right] \quad .$$

In either case, computation of each $\frac{\partial f_k}{\partial M_i}$, $\frac{\partial f_k}{\partial M_j}$ pair requires six multiplications and two

additions/subtractions. Constant terms within the small brackets are computed only once in the initialization phase and not at every time-step.

Therefore, total time is $8\ K\ t_f$.

16.

Computation of z_k , $k = 1, 2, ..., K$ from (2.57):

All the functions and derivatives in (2.57) have been computed in the previous steps. h^2 is computed in the initialization step.

computation time to compute all z_k is $9\ K\ t_f$.

17. Computation of a_{kk} , $k = 1, 2, ..., K$ from (2.58):

Except for $\frac{\partial f_k}{\partial W_k}$, all the other derivatives have already been computed in the previous steps. Now from the right-hand side of (2.2),

$$\frac{\partial f_k}{\partial W_k} = \left[-\frac{A_k}{L_k} \cdot \frac{(fr)_k\ L_k}{D_k\ A_k^2} \right] \cdot \frac{|W_k|}{\rho_k} \ .$$

The term within the brackets is constant and is computed only once at the time of initialization.

computation time to compute all a_{kk} is $11\ K\ t_f$.

18. Computation of a_{kv} , $k = 1, 2, ..., K$ from (2.59):

Number of chained links connected to nodes of degree two is equal to $(K - q - 2s)$. Each such link gives rise to two a_{kv} elements of matrix A, i.e. $a_{k,k-1}$, $a_{k,k+1}$

corresponding to two neighbouring links $(k-1)$ and $(k+1)$ of link k. Therefore, total number of $a_{k\nu}$ elements due to these links = $2(K-q-2s)$. Each end link of a chain gives rise to d_{av} number of $a_{k\nu}$ elements and there are $2s$ such links. Therefore, total number of $a_{k\nu}$ elements due to end links of chains is $2\,d_{av}\,s$. Finally, there are q number of random links in the network. On an average, each end of a random link is connected to a node of degree d_{av}.

Therefore, number of links which share the two end nodes of a random link is

$$(d_{av}-1)+(d_{av}-1)$$

$$=(2\,d_{av}-2)\ .$$

Therefore, the number of $a_{k\nu}$ elements due to all the random links in the network is $(2\,d_{av}-2)q$.

Hence, the total number of $a_{k\nu}$ elements in matrix A is

$$2(K-q-2s)+2\,d_{av}\,s+(2\,d_{av}-2)q$$

$$=2K+(2\,d_{av}-4)\,(q+s)\ .$$

Now from (2.59), computation of each $a_{k\nu}$ requires 3 floating-point operations assuming link ν is not a parallel or bypass link to link k.

Therefore, total computation time to compute all $a_{k\nu}$ is

$$3\,[2\,K+(2\,d_{av}-4)\,(q+s)]\,t_f$$

$$=6[K+(d_{av}-2)\,(q+s)]\,t_f\ .$$

Now the total time to perform all the above computations is the sum total of computation times in steps 1 through 18.

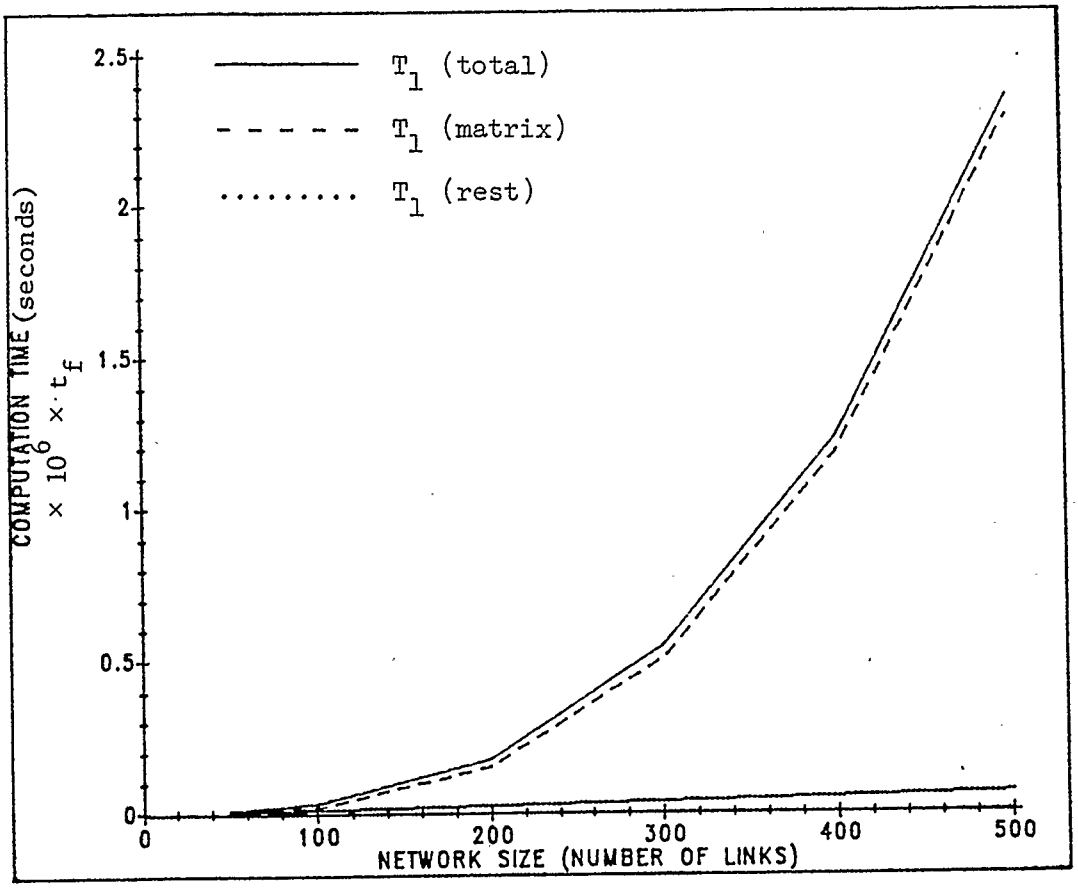


Figure 3.10(a) Computation times on a serial computer versus network size (number of links).

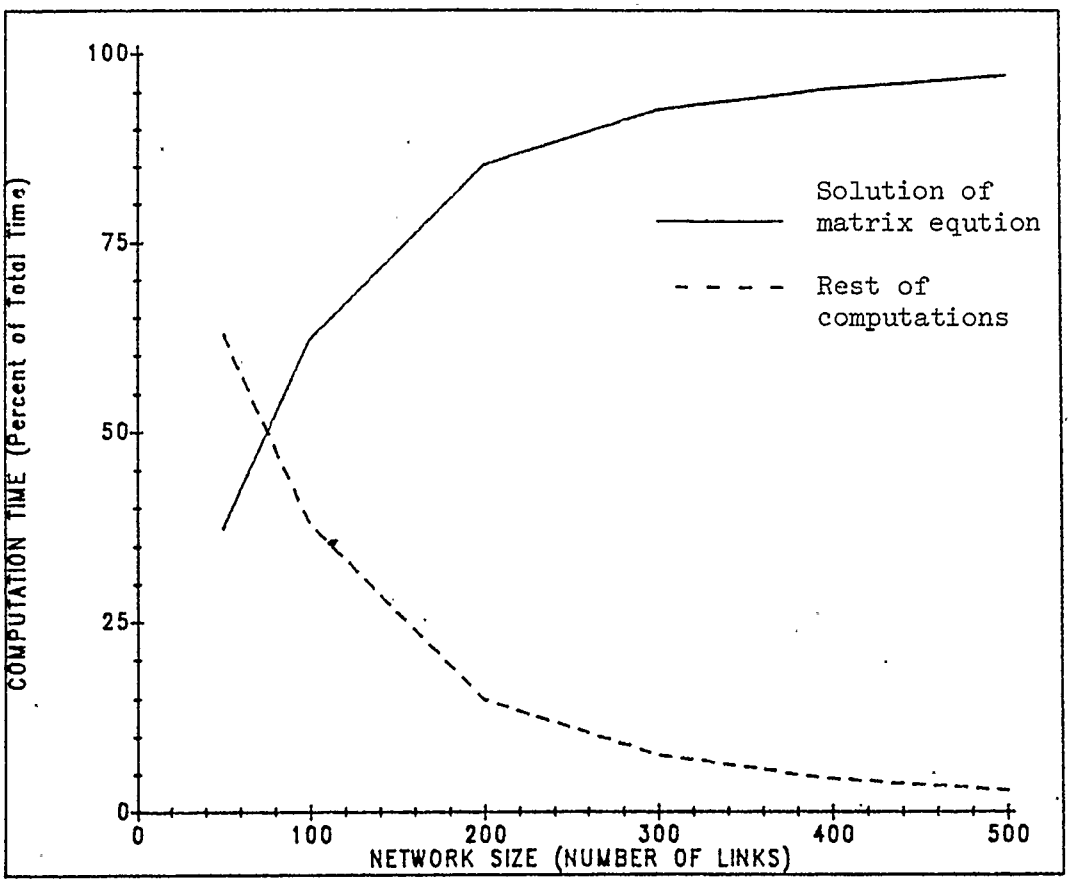


Figure 3.10(b) Components of computation times on a serial computer as a percentage of total computing time.

Hence, total computation time other than the matrix solution time

$$T_1(\text{rest}) = \Big[(57 - 6\ d_{av})K + (6\ d_{av} + T_p + T_{dpu} + T_{dpm} + 7)N + (12\ d_{av} - 24)\ (q + s)\Big]\ t_f \quad . \tag{3.39}$$

Taking $T_{dpu} = T_{dpm} = T_p = 25$,

$$T_1(\text{rest}) = \Big[(57 - 6\ d_{av})K + (6\ d_{av} + 82)N + (12\ d_{av} - 24)\ (q + s)\Big]\ t_f \quad . \tag{3.40}$$

The total computation time per time-step will be the sum of time T_1(matrix) to solve the matrix equation as given by (3.11) or (3.36) and the time T_1(rest) as given by (3.40) to perform the rest of the computations.

Therefore, $T_1(\text{total}) = T_1(\text{matrix}) + T_1(\text{rest})$.

The total computation time T_1(total), matrix solution time T_1(matrix), and the computation time T_1(rest) for performing the rest of the computations are plotted in Figs. 3.10(a) and 3.10(b) as a function of the size of the problem. As can be seen from Fig. 3.10, the total computation time is dominated by the matrix solution time for large size problems.

Therefore, it is important to compute the solution of the matrix equation $\mathbf{A}\ \Delta\mathbf{W}^{n+1} = \mathbf{z}$ efficiently on a single processor system as well as by parallel processing on a multiple processor system. The performance of the algorithms on multiple processors is discussed in Chapters 5 and 6.

CHAPTER 4

SELECTION OF A SUITABLE ARCHITECTURE FOR PARALLEL COMPUTATIONS

4.1. Introduction

In order to determine the complexity of a sequential algorithm for implementation on a serial computer, it is only necessary to count the number of floating-point operations involved in the algorithm and therefore a knowledge about the architectural aspects of the computer on which the algorithm will run is not required. However, this is not so for determining the complexity of a parallel algorithm. Here, total computation time is determined not only by the way the problem is partitioned or parallelized and the floating-point operations involved in the parallel algorithm but also by the communication and synchronization overhead involved which does not exist in the case of sequential algorithms. In order to determine this partitioning and the communication and synchronization overhead, one has to have some knowledge, at least at the abstract level, of the architecture of the parallel computer on which the problem is to be solved. This is also necessary for the optimal design of the parallel algorithm such that it can exploit all the good architectural features of the parallel computer and at the same time also takes into account the various constraints of the parallel architecture. Otherwise, the performance of the algorithm may be very poor due to loss of parallelism and increased communication and synchronization overhead despite the high amount of inherent parallelism in the problem. For the purpose of choosing a suitable parallel architecture for our problem, this chapter first describes some of the important features of parallel architectures using Flynn's classification scheme

[38,39]. This is followed by a brief description of interconnection networks used by these parallel computers. Finally, a very simple but useful model of parallel computation is chosen for our problem and its architectural implementation is described in some detail. A brief discussion about the proper choice of number of processors to be used in solving a problem is also given.

4.2. Taxonomy of Parallel Computers

Of all the classification schemes for parallel computer architectures [38-42], Flynn's taxonomy [38,39] is the most commonly used in literature. Flynn characterized the concurrency of data operation with respect to instruction streams by dividing architectures into four categories based on the number of instruction streams and the number of data streams. Single (SI) or multiple (MI) instruction streams are combined with single (SD) or multiple (MD) data streams to form four architectural categories:

(i) Single-instruction stream single-data stream (SISD)

(ii) Single-instruction stream multiple-data stream (SIMD)

(iii) Multiple-instruction stream single-data stream (MISD)

(iv) Multiple-instruction stream multiple-data stream (MIMD)

SISD computer organization:

In SISD computer organization, instructions are executed sequentially but may be overlapped in their execution stages, i.e. may be pipelined. There is only one control unit (CU) and one processor unit (PU) which may have more than one functional unit in it. This is the conventional serial von Neumann computer and represents most serial computers built in the past. Examples are: IBM 7090, CDC 6600, VAX 11/780.

SIMD computer organization:

This computer organization has a single control unit and hence, a single stream of instructions acting on multiple streams of data called vectors. Each element of the vector is a member of a separate data stream. This classification by Flynn includes all machines with vector instructions.

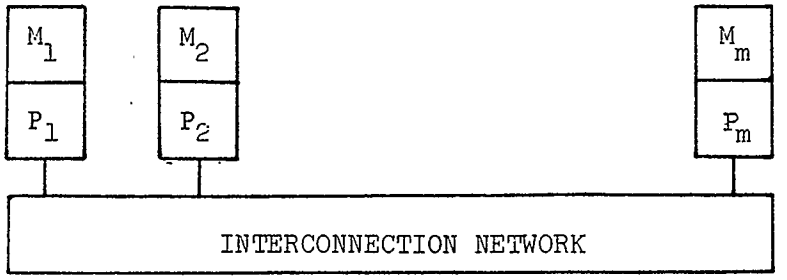
There are two further sub-classifications of SIMD computers [38]: Pipe-lined vector computers and processor arrays. Examples of pipelined vector computers are Cray-1 and Cyber-205. Processor arrays have multiple processing elements (PEs) supervised by the same control unit. All PEs receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams. Examples of processor arrays are: ILLIAC IV [43,44], ICL's Distributed Array Processor (DAP) [45], Goodyear Aerospace Massively Parallel Processor (MPP) [46], and the Connection Machine [12] from Thinking Machines Inc.

MISD computer organization:

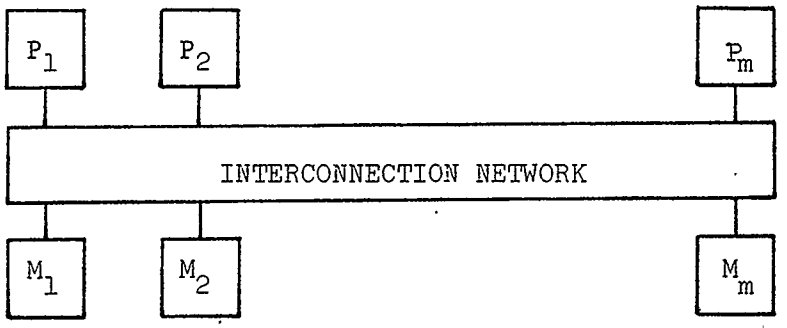
In this organization, there are n instruction streams operating on a single data stream. This organization seems to be of no practical importance and no real implementation of this architecture exists at present.

MIMD computer organization:

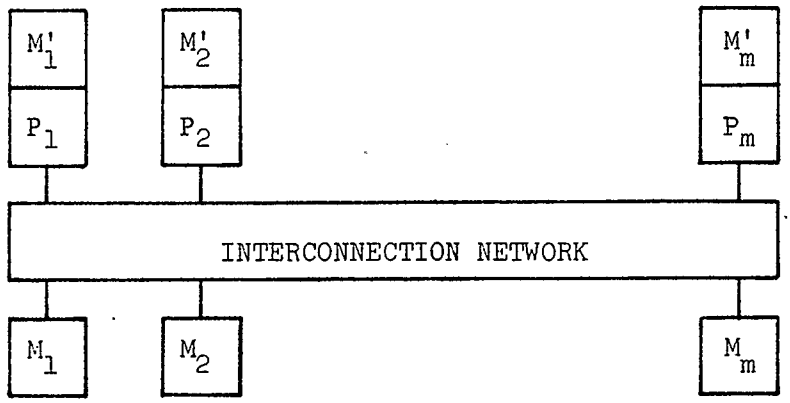
Here, each processor executes its own stream of instructions and the data communication among the processors is either by message-passing or via shared memory. If the data communication is by message-passing, the resulting MIMD architecture is termed as "loosely coupled" as shown in Fig. 4.1(a). On the other hand, if the processors communi-



(a) Loosely-coupled MIMD architecture



(b) Tightly-coupled multiprocessor without local memory



(c) Tightly-coupled multiprocessor with local memory

Figure 4.1 Three MIMD computer architectures

cate via a shared memory, the resulting architecture is termed as "tightly coupled". Most multiprocessors belong to this category. Two further sub-classifications of tightly-coupled multiprocessors can be made. In one case shown in Fig. 4.1(b), the processors have no local memory. They fetch their instructions and data from the shared memory. This structure is suitable only if the number of processors is small. An example of this class of multiprocessors is the supercomputer Cray-XMP with 2 or 4 processors. In the other case shown in Fig. 4.3(c), processors have their own local memories for storing their programs and local data and only shared variables are stored in the shared memory. This has the effect of significantly reducing the data communication traffic on the interconnection network resulting in an improved multiprocessor performance. Almost all large multiprocessor systems have local or cashe memories associated with their processors apart from shared memories. Most of the current research effort is being directed to develop this class of parallel computers because of their greater flexibility to solve a wider variety of problems than is possible with SIMD computers. MIMD-type parallel computers built in the past or presently being built for research purposes are: C.mmp [47], Cm* [48], Siemens SMS201 [49], Denelcor HEP [50], Ultracomputer [25], PACS [51], Cedar [52], Hypercube [24], BBN Butterfly [53] and IBM RP3 [54]. Of the above, SMS-201, PACS, and Hypercube are based on the message-passing model for data communication and can, therefore, be classified as loosely-coupled MIMD multi-computers. The rest of the above MIMD computers employ the shared-memory model for data communication among the processors and can, therefore, be classified as tightly-coupled multiprocessors.

The above classification scheme covers most of the parallel computers employing a von-Neumann model of computation although classification for some of the architectures may not be as clear-cut. For example, systolic/wavefront-type architectures may employ

both multiprocessing and pipelining. Another parallel architecture which cannot be classified under the above scheme is the data flow computer [55] which does not have the concept of data streams or instruction streams. Instead, computing nodes containing instructions fire as soon as their input data becomes available [55].

4.3. Selection of an Architecture

Of all the architectures described above, MIMD type of architecture is most suitable for our problem. SISD type computers, i.e. the usual serial computers, are too slow. MISD type computers do not exist. This leaves us with the choice of SIMD- or MIMD-type parallel computers. In the following, it is shown that SIMD architecture is not very suitable for our problem which has a sparse structure and, therefore, does not give rise to a highly vectorized code with long vectors.

4.3.1. Unsuitability of SIMD computers

There are two sub-classifications of SIMD computers: processor arrays which are operated in a lock-step manner by the single control unit, and the vector computers which operate on vectors in a pipe-lined fashion.

Processor arrays, such as, ILLIAC IV, ICL DAP employ a two-dimensional array of identical processors connected in a nearest-neighbourhood mesh manner. All the processors perform the identical operation such as a multiplication on their own operands in a lock-step manner. They are, therefore, suitable for highly-structured computations as are, for example, encountered in the low-level image processing algorithms.

Vector computers are suitable for problems which give rise to vector operations with very long vector lengths. For short vectors or for scalar operations, their performance

becomes very poor as compared to their peak vector performance. This can be seen by carrying out the following analysis [15] for the "add" pipe-line shown in Fig. 4.2.

Let

$\tau =$ clock period

$L =$ length of the add pipe

A,B = $N \times 1$ vectors

$s\tau =$ overhead of starting the pipeline computation

The operation to be performed is the addition of two vectors **A** and **B** of length N each. The operation of adding two scalar numbers is carried out in L stages as shown in Fig. 4.2. The time to fill the pipeline with operands is $L\ \tau$. Therefore, time to add first two elements of vectors **A** and **B** is $(s\ \tau + L\ \tau)$. After the pipeline is filled, one element of resultant vector **C** comes out of the pipeline every clock cycle τ. Therefore, the remaining $(N - 1)$ elements of resultant vector **C** will take $(N - 1)\ \tau$ time.

Hence, total time to add two $N \times 1$ vectors on a pipelined computer

$$t_{pipe} = s\tau + L\tau + (N - 1)\tau \quad (4.1)$$

On a serial computer without any pipelining, this time will be

$$T_{serial} = L\ \tau\ N \quad (4.2)$$

The speed-up which is defined as a ratio of time taken on a serial computer to time on a pipelined computer is given by

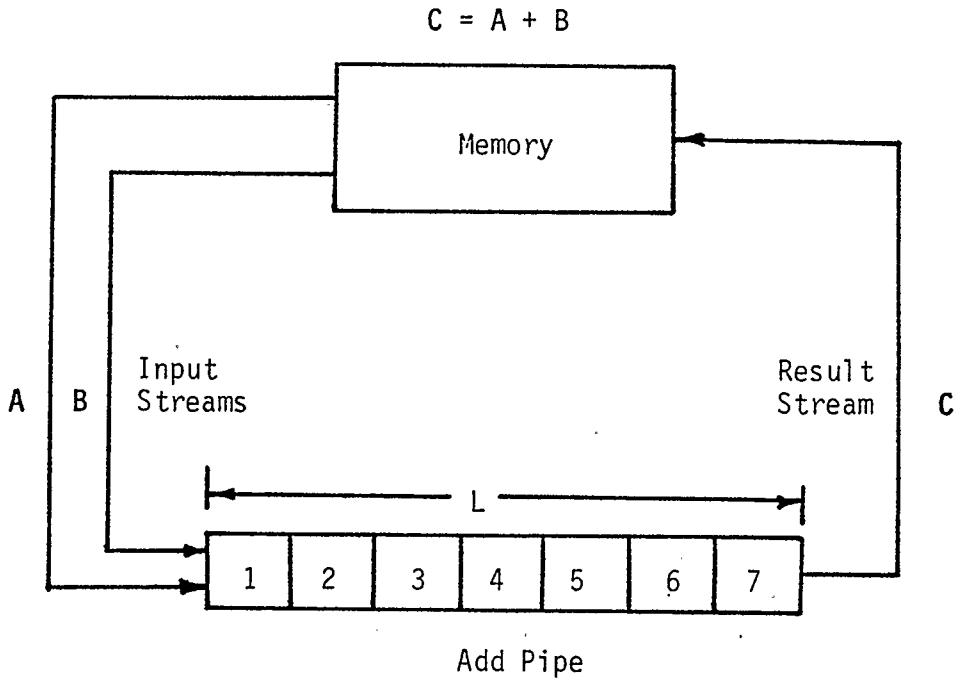


Figure 4.2 Vector addition on a pipelined vector computer

$$\text{speed–up} = \frac{t_{\text{serial}}}{t_{\text{pipe}}} = \frac{L\tau N}{s\tau + L\tau + (N-1)\tau}$$

$$= \frac{LN}{s + L + (N-1)} = \frac{L}{\frac{(s+L)}{N} + \frac{(N-1)}{N}} \tag{4.3}$$

only when $N \rightarrow \infty$ (very large N), the speed-up tends to the ideal value of L. This is so because the overhead of starting the vector operation and filling the pipeline becomes negligible as compared to the actual computing time of the resultant vector C.

The performance of the CDC Cyber-205 vector computer for adding two vectors of length N is shown in Fig. 4.3 as a function of the vector length N. As can be seen, the performance of the vector computer for vectors of length 100 is only 50% of the peak performance. It is much smaller for vectors of shorter length. Hockney [15] defines a parameter $n_{1/2}$ to denote half-performance vector length required to achieve half the maximum performance. Since our problem does not give rise to very long vectors, the performance of the algorithm on a vector computer will not be very satisfactory.

Another point to be noted is that some stages in an algorithm may vectorize to long vectors while the remaining stages may not be vectorized at all. As shown below [56], the overall performance of the algorithm in such a case is determined by the slower scalar portion of the algorithm rather than the faster highly-vectorized portion of the algorithm. This is a direct consequence of the Amdahl's law as discussed in section 4.7 of this chapter.

Let

F_{scalar} = fraction of the total computations that can be processed in scalar mode only

F_{vector} = $(1 - F_{\text{scalar}})$ = fraction of the total computations that

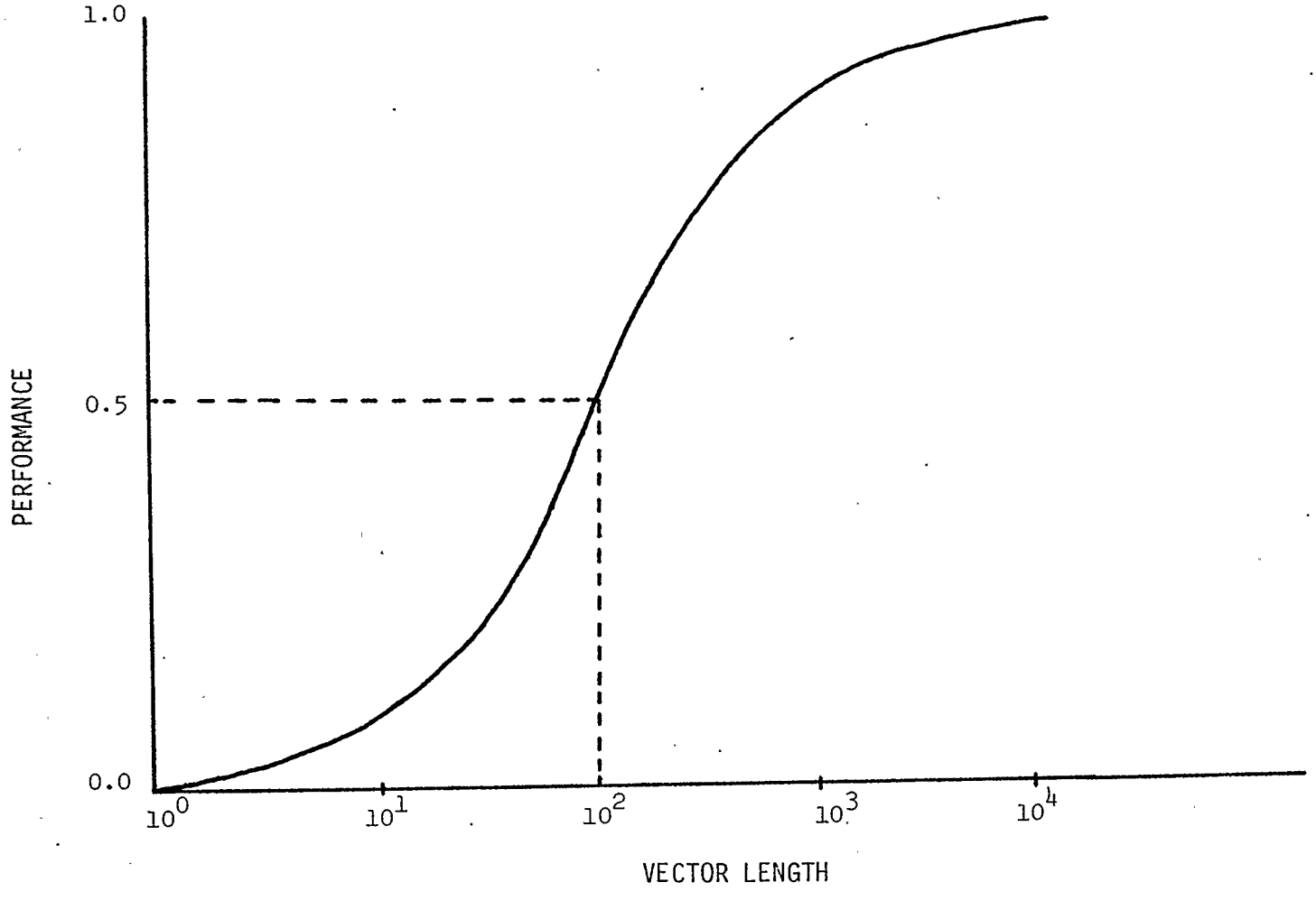


Figure 4.3 Typical performance of a pipelined vector computer as a function of vector length

can be processed in vector mode with long vectors

S_{scalar} = speed of the vector computer for scalar operations

S_{vector} = speed of the vector computer for vector operations

S_{eff} = effective speed of the vector computer for solving the complete problem

Then, the total computation time T is proportional to

$$T \alpha \frac{F_{scalar}}{S_{scalar}} + \frac{F_{vector}}{S_{vector}} = \frac{1}{S_{eff}} \cdot \tag{4.4}$$

For example, for CDC Cyber-205 [56],

S_{scalar} = 3.3 MFLOPS (millions of floating-point operations per second)

S_{vector} = 100 MFLOPS (Single pipe)

Even if 75% of the algorithm code is vectorized to long vectors with $S_{vector} = 100$ *MFLOPS*, the overall *MFLOP* rate S_{eff} obtained is given by

$$\frac{1}{S_{eff}} = \frac{0.25}{3.3} + \frac{0.75}{100}$$

or

$$S_{eff} = 12 \text{ MFLOPS}$$

which is much closer to S_{scalar} than to S_{vector}. Therefore, the above two factors, namely, shorter vector lengths and the presence of slower scalar computations in the workload, are mainly responsible for vector computers not being suitable for unstructured problems such as our network problem.

The performance of MIMD-type computers, on the other hand, does not depend upon the ability to vectorize the computations completely. Rather, it depends upon the ability to suitably partition the problem is such a manner that processors share the computational load as equally as possible with a minimum of interaction among themselves during computations. Therefore, this mode of parallel computation is quite suitable for unstructured problems as long as proper partitioning of the algorithm among the processors is possible.

A slight variation of the true MIMD operation known as quasi-MIMD mode has been chosen for our problem and is described in section 4.6 of this chapter.

4.4. Interconnection Networks

Interconnection networks play an important role in the performance of parallel computers. They can be broadly classified as regular and irregular networks as shown by the classification scheme given in Fig. 4.4 [57]. Only regular networks are important from the point-of-view of parallel processing. Regular network topologies are further subdivided as static or dynamic. In a static topology, links between two processors are dedicated and cannot be reconfigured for direct connections to other processors. On the other hand, links in the dynamic category can be reconfigured by setting the network's active switching elements.

4.4.1. Static network topologies

Examples of a few static networks are shown in Fig. 4.5. In this figure, circles, represent the processors and lines represent the dedicated communication links between them.

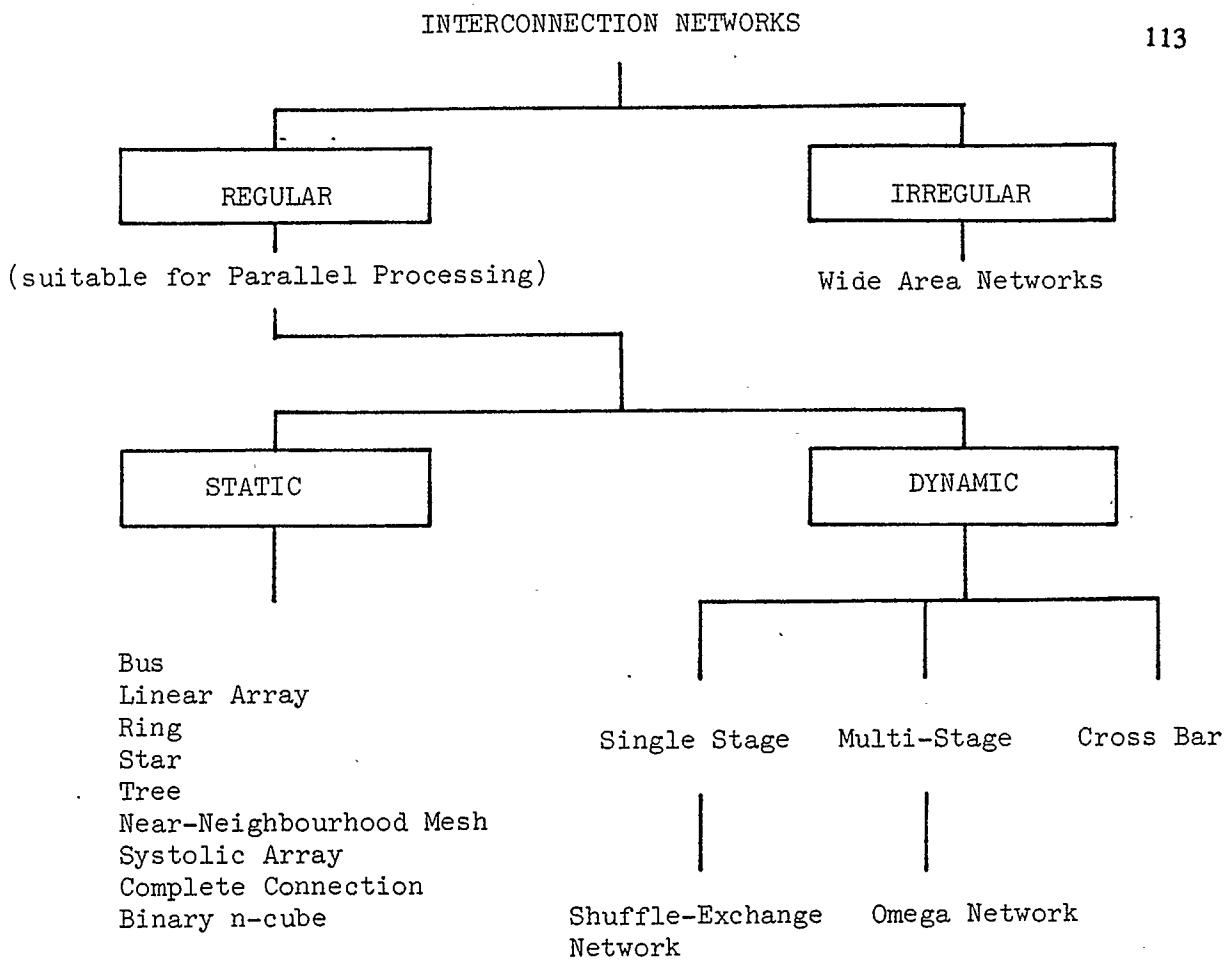
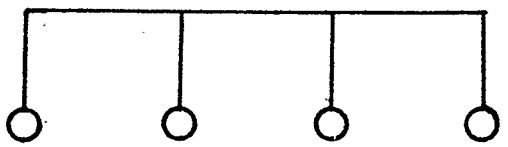
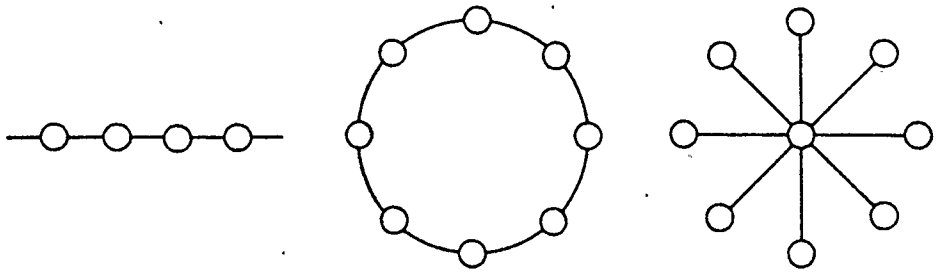


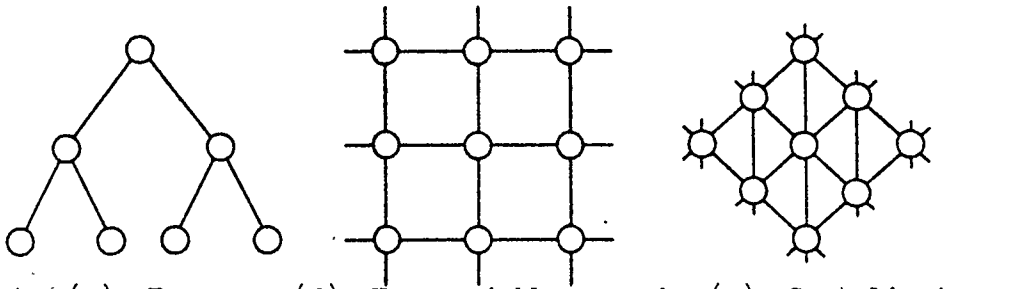
Figure 4.4 Classification of computer interconnection networks



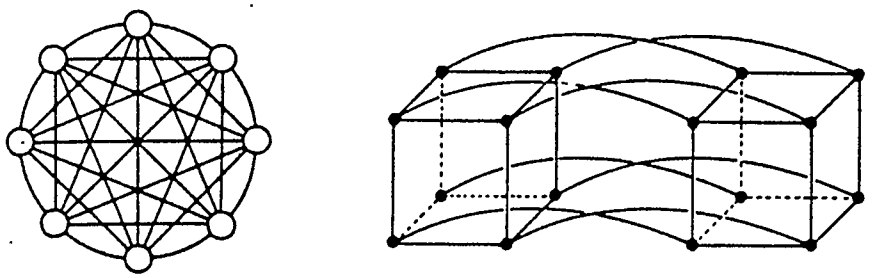
(a) time-shared bus



(b) Linear Array (c) ring (d) star



(e) Tree (f) Near-neighbour mesh (g) Systolic Array



(h) completely connected (i) hypercube ($n = \log_2 N = 4$)

Figure 4.5 Examples of static network topologies

Although drawn differently, the topologies of time-shared bus shown in Fig. 4.5(a) and star shown in Fig. 4.5(d) are quite close to each other. Except for the time-shared bus and the complete connection topology [Fig. 4.5(h)] the performance of these regular networks depends upon the structure of problem being solved. If the problem structure exactly matches with the network topology, the performance of the algorithm will be very good due to low communication overhead. On the other hand, if the two structures do not match well, the performance of the algorithm may be quite poor. For example, the near-neighbourhood mesh topology shown in Fig. 4.5(f) is ideally suited to certain low-level image processing algorithms such as smoothing for noise removal because the network architecture matches exactly with the problem structure. This network is also suitable for certain matrix problems due to local communication patterns involved in the algorithms. However, the same network architecture will not be suitable for problem requiring global communication. For this network, for example, the ratio of communication times for communication between the processors farthest apart and between the two nearest processors is $2\sqrt{N}$ where N is the total number of processors in the network. Furthermore, there may also be problems of contention over the use of dedicated links when heavy non-local communication patterns occur which further increases the communication overhead.

Similarly, the tree topology, shown in Fig. 4.5(e), is suitable for certain sorting and searching algorithms which take $0(\log_2 N)$ time on this architecture including the communication time. However, if an algorithm involves heavy global communication patterns, the performance of the algorithm may be quite poor due to saturation occurring at the root of the tree since much of the global communication may have to pass through the root node.

Another static network topology is the complete connection network as shown in Fig. 4.5(h). Performance-wise, this topology has the lowest communication overhead as there is a complete connection among the processors, i.e. each processor is connected to every other processor in the network by a separate dedicated link. However, from the point-of-view of hardware cost, this network is one of the most expensive, particularly for large networks due to its $0(N^2)$ cost-dependence on the number of processors N in the network. Therefore, complete connection is used only when the number of processors is small, say less than ten or so.

The final example of static networks is the binary n-cube or hypercube [24] shown in Fig. 4.5(i). Each processor in this topology is connected to n of its nearest neighbours corresponding to n $(= \log_2 N)$ independent directions or axes in the hypercube where N is the total number of processors or nodes in the network. This architecture is ideally suited for certain problems such as FFT computations since the data topology in FFT butterfly (combination of two data elements at a FFT butterfly node which differ only in one bit position in the binary representation of their addresses which corresponds to a dedicated direct link between the processors in the binary n-cube) matches exactly with the network topology. For non-local communications, the data has to be routed through the intermediate nodes. The maximum distance between any two nodes in the network or the diameter of the network is $\log_2 N$ which is better than the near-neighbourhood mesh $(dia = 2\sqrt{N})$ but worse than the time-shared bus (diameter = 1). The hardware complexity of the network grows as $0(N \log_2 N)$ which is better than the $0(N^2)$ complexity of complete connection network of Fig. 4.5(h). $I/0$ port complexity of each node grows as $0(\log_2 N)$.

It is to be noted that if the problem structure does not exactly match with the network topology, such as a tree, mesh or hypercube, an attempt can be made to assign the computational modules of the problem in such a way that pairs of modules that communicate with each other are placed, as far as possible, on processors that are directly connected. This assignment of modules to processors is called mapping and the problem of maximizing the number of pairs of communicating modules that fall on pairs of directly connected processors is called the mapping problem [58]. The problem of finding the best mapping is, in general, very difficult. It has been shown [58] to be equivalent to the graph isomorphism problem which is one of the classical unsolved combinatorial problems and also equivalent to the problem of bandwidth reduction of sparse matrices which is also known to be NP-complete. Moreover, the best mapping from the point of view of least communication overhead may not be the best mapping from the point of view of load-balancing among the processors.

In connection with the mapping problem, it is important to make a distinction between the problem structure and its parallel algorithm data communication structure. In general, it is the algorithm communication structure which is to be mapped on the network topology and not the problem structure. In some cases, the problem structure and its algorithm communication structure may be the same. However, in general, it will not always be the case. For example, in the parallel version of the Porsching's algorithm for the numerical integration of hydraulic networks as discussed in Chapter 5, the algorithm communication structure for computing the elements of matrix $\mathbf{A}$ and vector $\mathbf{z}$ is very close to the problem structure, i.e. the network structure. However, the data communication structure for solving the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ involves a series of data broadcasts which is not related to the problem structure at all.

Except for the time-shared bus and the equivalent star topology (which are suitable for shared-memory as well as message-passing models of computation), the remaining static network topologies shown in Fig. 4.5 are suitable only for the message-passing model of parallel computation.

4.4.2. Dynamic network topologies

Dynamic networks can be further classified as single-stage, multi-stage and cross-bar networks as shown in Fig. 4.4.

4.4.2.1. Single-stage networks

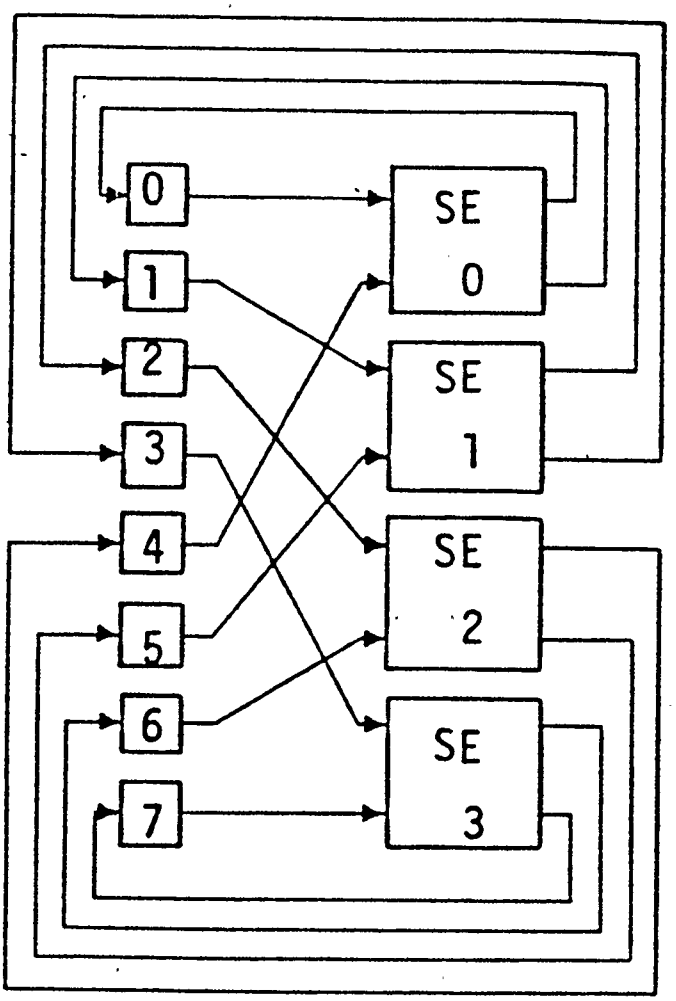
A single-stage network is composed of a stage of switching elements cascaded to a link connection pattern. An example of a single-stage network is the shuffle-exchange network [59], based on a perfect-shuffle connection cascaded to a stage of switching elements as shown in Fig. 4.6 for $N = 8$. In this figure, the squares numbered 0, 1, 2, ..., 7 are the processors and the squares numbered SE0, SE1, SE2, SE3 are the switching elements. The single-stage network is also called a recirculating network because data items may have to recirculate through the single-stage several times before reaching their final destination.

The "perfect shuffle" operation is given by

$$f(s_1\, s_2\, \cdots\, s_n) \Longrightarrow s_2\, s_3\, \cdots\, s_n\, s_1$$

where $s_1\, s_2\, \cdots\, s_n$ is the binary representation of the processor number. In other words, the perfect shuffle operation is the same as the left rotate operation on the binary address of the processor.

The "exchange" function is given by



(a) direct connection (b) cross connection

Switching Element (SE)

Figure 4.6 Stone's shuffle-exchange network for $N = 8$

$$E(s_1\, s_2\, \cdots\, s_{n-1}\, s_n) \Longrightarrow s_1\, s_2\, \cdots\, s_{n-1}\, \overline{s_n}\ .$$

This interconnection network has been shown [59] to be suitable for FFT computations and matrix transposition. However, for certain connections, the network has to go through many cycles to complete the required connection. For example, three network cycles are required to connect processor no. 1 to processor no. 7. This delay will be more in larger networks.

4.4.2.2. Multi-stage networks

Many stages of interconnected switches form a multi-stage network. For example, n stages of shuffle-exchange network connected in series form a multi-stage network known as Omega network [60]. A three-stage network for connecting eight processors ($N = 2^3$) is shown in Fig. 4.7. For a multiprocessor configuration, the processors are connected on the left-hand side or the input side of the network and the same number of memory modules are connected on the right-hand side or output side of the network as shown in Fig. 4.7. The switches can have four configurations: straight, interchange, upper broadcast and lower broadcast. With this interconnection network, any processor can be connected to any memory module in a fixed time proportional to the number of network stages, i.e. $\log_2 N$, where N is the number of processors in the network provided there is no contention or blocking in the network. The hardware complexity of the network is $0(N \log_2 N)$ and the theoretical bandwidth of the non-pipelined network is $(N/\log_2 N)$ where N is the number of inputs to the network.

One of the most attractive features of the network is its distributed control of the switches in the network as opposed to the centralized control which may be too time-consuming for large networks and therefore may result in poor network performance. This

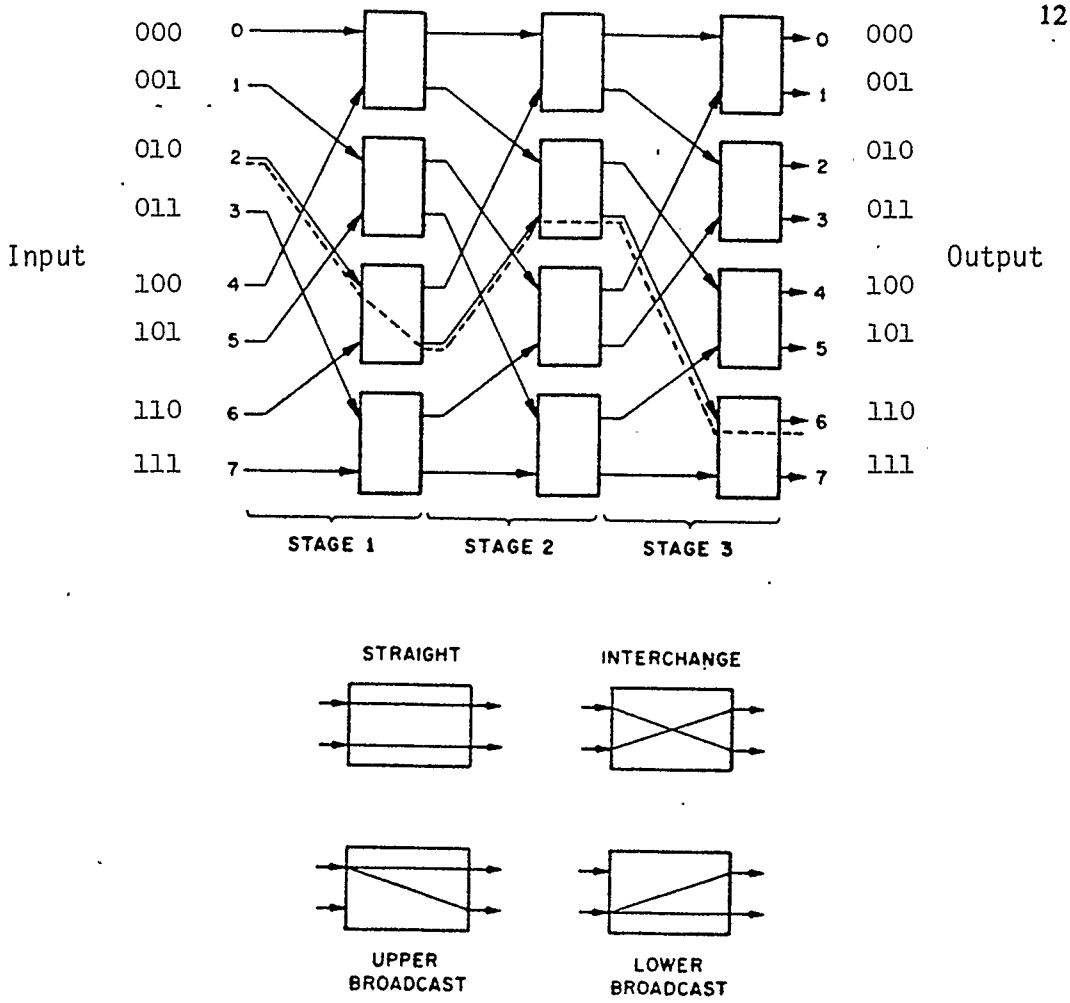


Figure 4.7 Lawrie's Omega network for $N = 8$

distributed routing of data or, in other words, distributed control is carried out as follows: let $D = d_1\, d_2 \cdots d_n$ be the destination tag, i.e. the binary representation of the output number to which input number S is to be connected, and let $S = s_1\, s_2 \cdots s_n$ be the source tag, i.e. binary representation of the input number. The first switch to which s is connected is set to switch input s to the upper output if $d_1 = 0$ or the lower output if $d_1 = 1$. This is shown in Fig. 4.7 for $s = 010$, $D = 110$. In the next stage, the input is again switched to the upper output if $d_2 = 0$ or to the lower output if $d_2 = 1$. This procedure is continued at each stage until we get to the output end of the network which is the destination tag. The data read from the memory at the destination tag is routed back to the source tag by following the same algorithm as described for forward routing.

It is to be noted that there is a unique path for each input-output connection through the network. Thus, there is a possibility of contention or conflict occurring if two sets of input-output connections have a common intermediate communication link through which the data has to be routed. For example, $000 \rightarrow 000$ and $100 \rightarrow 010$ paths in Fig. 4.7 share a common connection link at the output of the first stage. Such networks are called blocking networks. For such situations, the data routing has to be serialized at those switches where the conflicting requests arrive by providing buffers or queues in the switches. If the multiple sources need access to a single destination address, then "combining" can be used at the intermediate switch at which the different access requests meet [25]. This improves the performance of the network by eliminating the serialization of multiple requests at the intermediate switches.

A modified version of Omega network which employs "combining" of multiple requests to the same memory location is being developed for two large-scale research

multiprocessors- "Ultra Computer" [25], being developed at New York University and RP3 [54], being developed at IBM's T.J. Watson Research Centre.

The performance of Omega network has been analysed by Kruskal and Snir [61] by assuming uniform network traffic in space and time and infinitely long queues at the switches. Under such unrealistic assumptions, the performance of the Omega network has been shown to be quite satisfactory [61]. Simulation experiments have also been carried out by IBM researchers considering a non-uniform traffic pattern consisting of a single "hot spot" of higher access rate caused by shared lock and synchronization data superimposed on a background of uniform traffic [62]. Their simulations revealed a severe degradation of network performance even with a moderate hot spot traffic. Also, the degradation of network performance is global, i.e. it affects all memory access and not just the accessing of shared lock data. This degradation in network performance has been shown to be due to the "tree saturation" effect caused by distributed routing and finite length of queues at the switches. The use of an additional network which employs combining of messages at the switches has been proposed to solve this problem caused by shared synchronization data. However, the extra hardware needed to support this feature is estimated to increase the switch size and cost by a factor of between 6 and 32 [62]. Even without this additional network, the total cost of the multiprocessor, as noted by Gottlieb et al. [25], is dominated by the network cost and not the cost of processors and memory. Moreover, a problem similar to hot spot is still likely to occur if the algorithm memory references for shared data are not uniformly distributed but are localized to a few memory locations. It has not been shown in [62], whether message combining is effective in the case of general non-uniform memory traffic generated by accessing the program's shared data.

Due to the large amount of non-local wiring in the network, 32-bit wide links to connect switches are not practical. Therefore, a bit-serial or a byte-serial approach has to be used which reduces the effective bandwidth of the network. For example, for bit-serial communication, the effective bandwidth of the non-pipelined Omega network is given by

$$\text{effective bandwidth} = \frac{N}{32 \log_2 N} \quad (4.5)$$

This effective bandwidth has to be greater than 1 for Omega network in order to be faster than a 32-bit wide time-shared bus, i.e.

$$\frac{N}{32 \log_2 N} \geq 1$$

or $N \geq 256$.

Therefore, roughly speaking the bit-serial Omega network is faster than a 32-bit bus only when the number of processors in the multiprocessor is of the order of 256 or more.

For the ideal fully pipelined Omega network, the bandwidth is equal to N. For the bit-serial Omega network to be faster than a 32-bit bus,

$$\frac{N}{32} \geq 1 \quad \text{or} \quad N \geq 32 .$$

4.4.2.3. Crossbar switch

In a crossbar switch every input port can be connected to a free output port without blocking. The number of switches employed in the network grows as N^2 where N is the number of input or output ports in the network. Therefore, this network is suitable for connecting a relatively small number of processors. The C.mmp multi-processor [47] employed a 16 ×16 crossbar switch to connect 16 minicomputers to 16 memory modules.

4.5. Selection of an Interconnection Network

The implicit numerical integration method of Porsching is described in section 2.5 of Chapter 2 and section 3.2 of Chapter 3. This algorithm consists of two main stages, i.e. calculation of elements of matrix **A** and vector **z** from hydraulic network data and solution of the matrix equation $\mathbf{A}\,\Delta\mathbf{W}^{n+1} = \mathbf{z}$ for determining $\Delta\,\mathbf{W}^{n+1}$ at each time-step. The data communication structure for the first part of the algorithm is very similar to the hydraulic network structure. However, the solution of the matrix equation (which is the dominant part of the total computation for large size problems) by the parallel Gauss-Jordon algorithm, as described in Chapter 5, involves a series of data broadcasts to other processors for which a bus architecture with data broadcast facility is the best architecture since it can broadcast a data to all the processors in unit time. Although data broadcast can also be carried out by other networks such as a tree, Omega network, or binary n-cube, the communication time is longer and hardware complexity is significantly higher than a simple bus connection network. Other advantages of the time-shared bus architecture can be summarized as follows:

(i) The diameter of the bus network, i.e. the maximum distance between two nodes in the network is unity. In other words, the data communication delay between two neighbouring processors is the same as the communication delay between two processors which are physically farthest apart. As a result, the difficult problem of mapping the problem structure to multi-processor communication structure does not exist in this case and the algorithm can be partitioned among the processors to satisfy the requirement of load balancing only among the processors without any concern for the optimal mapping of the problem from the point of

view of minimizing total communication delay.

On the other hand, the diameter or the maximum communication delay for the case of two-dimensional mesh grows as $\sqrt{N}$ and for the case of tree, binary n-cube, and Omega network it grows as $\log_2 N$. Therefore, except for the case of Omega network which has constant delay of $\log_2 N$ for all memory references, the problem has to be mapped to the processor communication structure in order to minimize the total communication delay and it is quite likely that this mapping is different from the mapping resulting from load balancing considerations. Moreover, these networks generally use bit-serial communication links due to large non-local wiring requirements which further reduces their effective communication bandwidth. Therefore, although these networks have high communication bandwidth due to their concurrent communication capability as compared to serial communication on a time-shared bus, this bandwidth is greatly reduced due to the above two reasons of non-local communication patterns in the problem and bit-serial communication links in the network. Hence, these network connections are to be preferred when connecting a very large number of processors together (say more than 256) so that their concurrent communication capability (proportional to N) more than compensates their drawbacks of large communication delay (proportional to $\log_2 N$) and bit-serial nature of data communication.

(ii) The hardware complexity of bus communication network increases linearly with the number of processors N as compared to N^2 or ($N \log_2 N$) complexity of other networks.

(iii) Incremental expansion of the bus-based architecture is relatively easy as compared to other architectures based on complex interconnection networks.

Considering the above factors, a time-shared bus-based architecture has been chosen for our problem because:

(a) Much of the data communication for solving the large-size network problems (i.e. solution of matrix equation) involves data broadcast to other processors for which bus network is ideal (Chapter 5).

(b) The other data communication in the problem (for example, calculation of matrix A coefficients) requires communication among non-local processors.

(c) The number of processors that can be used for the parallel solution of the problem is not very large. In the first parallel algorithm, this number is equal to the number of chains (less than 25 in a typical large network) in the network. In the second parallel algorithm, it is equal to number of random links in the network which is less than 150 in typical large hydraulic networks.

(d) The granularity of computations in the parallel algorithms remains quite high so that the total communication overhead does not dominate the total processing time. Hence, the choice of interconnection network although important is not very critical for achieving good performance of these two parallel algorithms.

The following section describes the architecture of the bus-based multiple processor system finally chosen for our problem.

4.6. Description of the Chosen Parallel Architecture

The parallel architecture used in this thesis for determining the complexity of parallel algorithms is a simple time-shared bus-based multi-microprocessor system called Structured Multiprocessor System (*SMS*) proposed and actually built by Siemens researchers [49,63-66]. It consists of a number of microcomputer modules [128 in models SMS-101 and SMS-201] each with its own local memory for program and data and a communication memory (*CM*) for data communication among the processor modules as shown in Fig. 4.8. The communication memories are connected either to slave processors or to the main processor by the switches operated by main processor. These slave modules are supervised by the master or main processor. There are three distinct phases in the execution of a parallel program on this parallel processor system:

1. Control Phase: The master processor signals the slave processors to start execution of their assigned tasks.

2. Autonomous Phase: In this phase, the slave processors work independently by executing programs stored in their program memories. They write the values of computed shared variables in the local communication memories which are connected to the slave processors as shown in Fig. 4.9(a): This phase terminates when all the processors have finished their tasks. The synchronization of all processors to indicate completion of their tasks is done in hardware by "anding" their individual task completion interrupt signals to the master processor.

3. Communication Phase: In this phase, the master processor has access to the communication memories (*CM*s) of the slave processors as shown in Fig. 4.9(b). The master processor can transfer data from one *CM* to another *CM* or between a *CM*

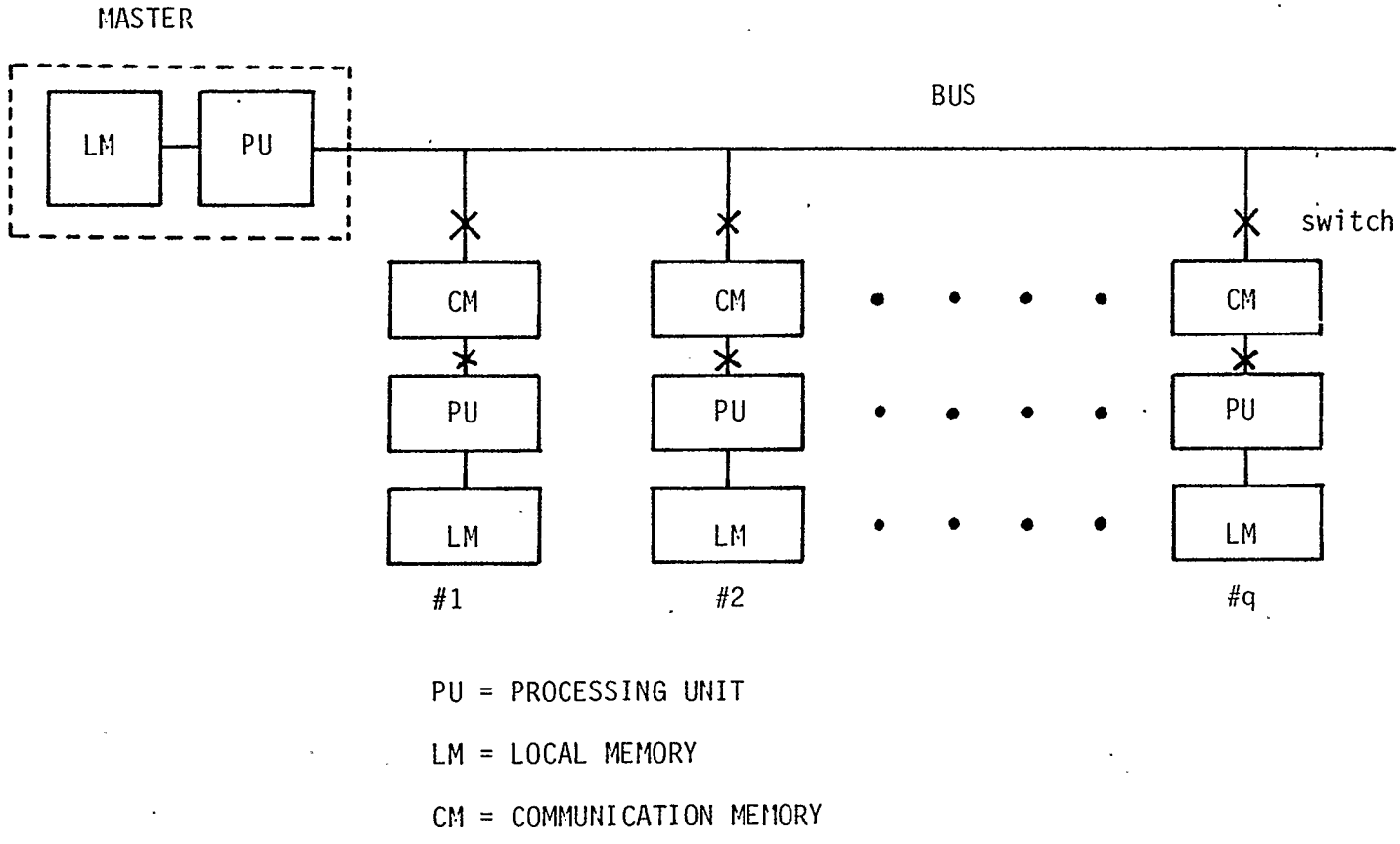


Figure 4.8 Structured Multimicroprocessor System (SMS) with communication memories.

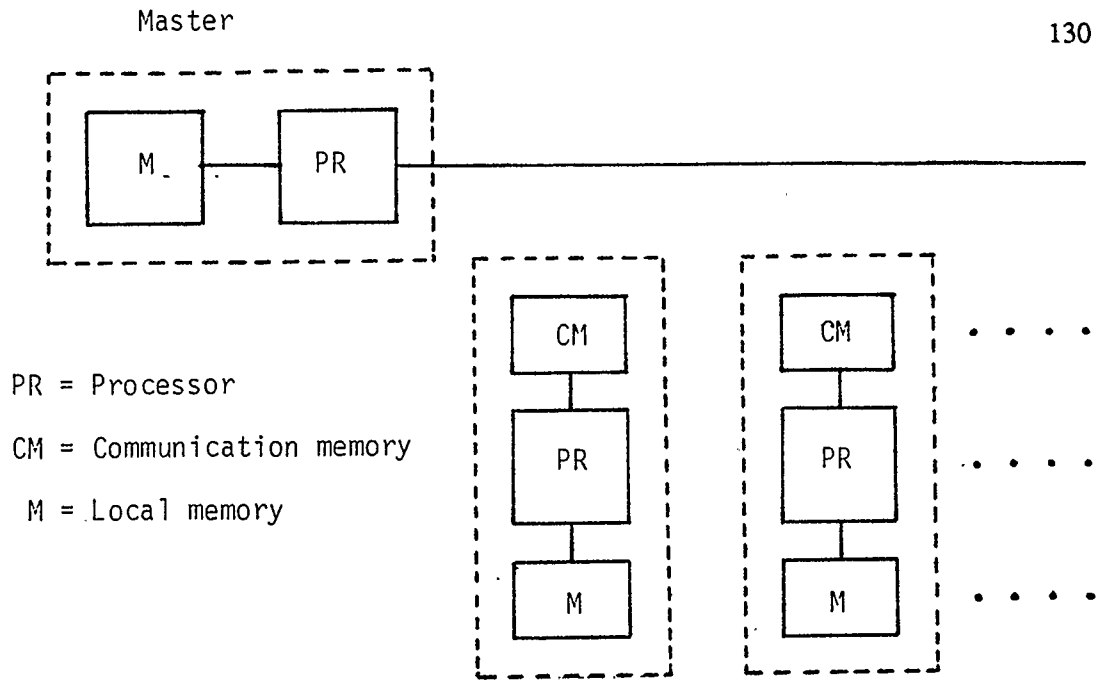


Figure 4.9(a) Memory access mode during autonomous phase

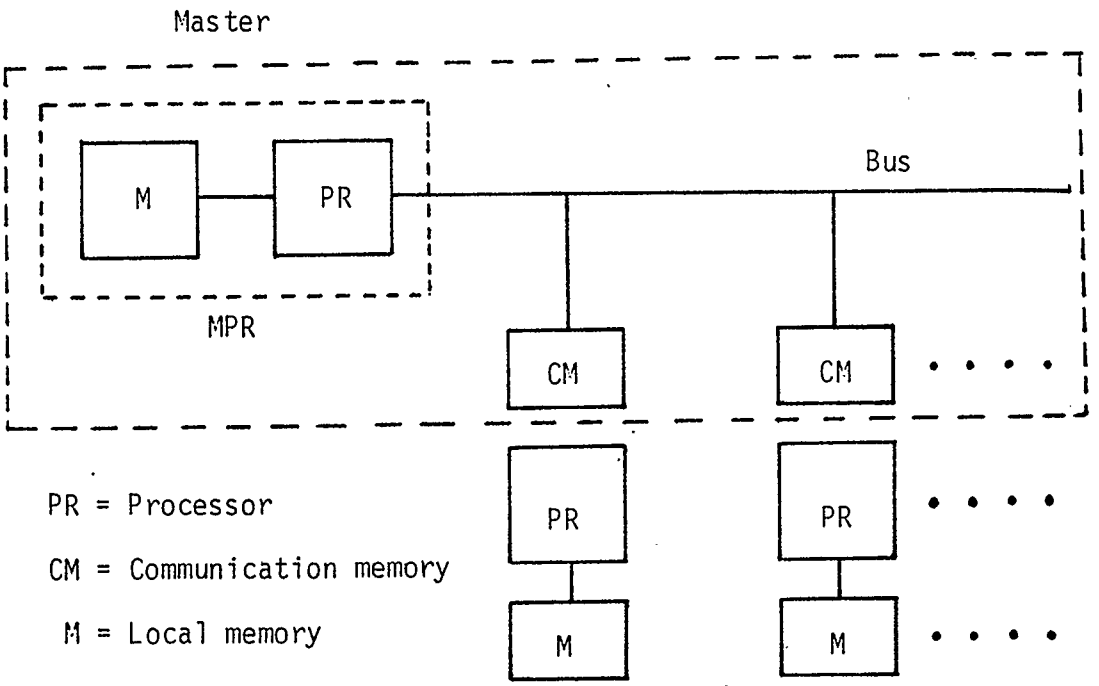


Figure 4.9(b) Memory access mode during communication phase

and the main processor memory. It can also broadcast a data from one *CM* to all the remaining *CM*s in a single command cycle [65].

Another possible method for communicating shared data in this architecture is as follows: first, the master processor reads all the shared data from the communication memories of all the processors. After this, the master processor changes the switch positions of all the processors so that they are directly connected to the bus for reading their needed data on the bus. Then the master processor broadcasts shared data on the bus which is read by those processors which require it and is ignored by the remaining processors.

Since slave processors do not transfer data themselves on the bus, bus contention cannot occur with the result that no time is wasted in the arbitration of bus requests and also arbitration hardware is not required.

The above model of parallel computation for the *SMS* multi-microprocessor system is depicted in Fig. 4.10. Total processing time for one processing cycle of *SMS* 201 which consists of computation phase, synchronizing phase, communication phase and control phase is given by Fig. 4.11,

$$\text{Total processing time} = \text{maximum of } [T_f(1), T_f(2), \cdots T_f(N)] + T_{sync} + T_{comm} \tag{4.6}$$

where,

$T_f(i) =$ Total computation time of task i on processor i

$T_{sync} =$ Synchronization and control phase time

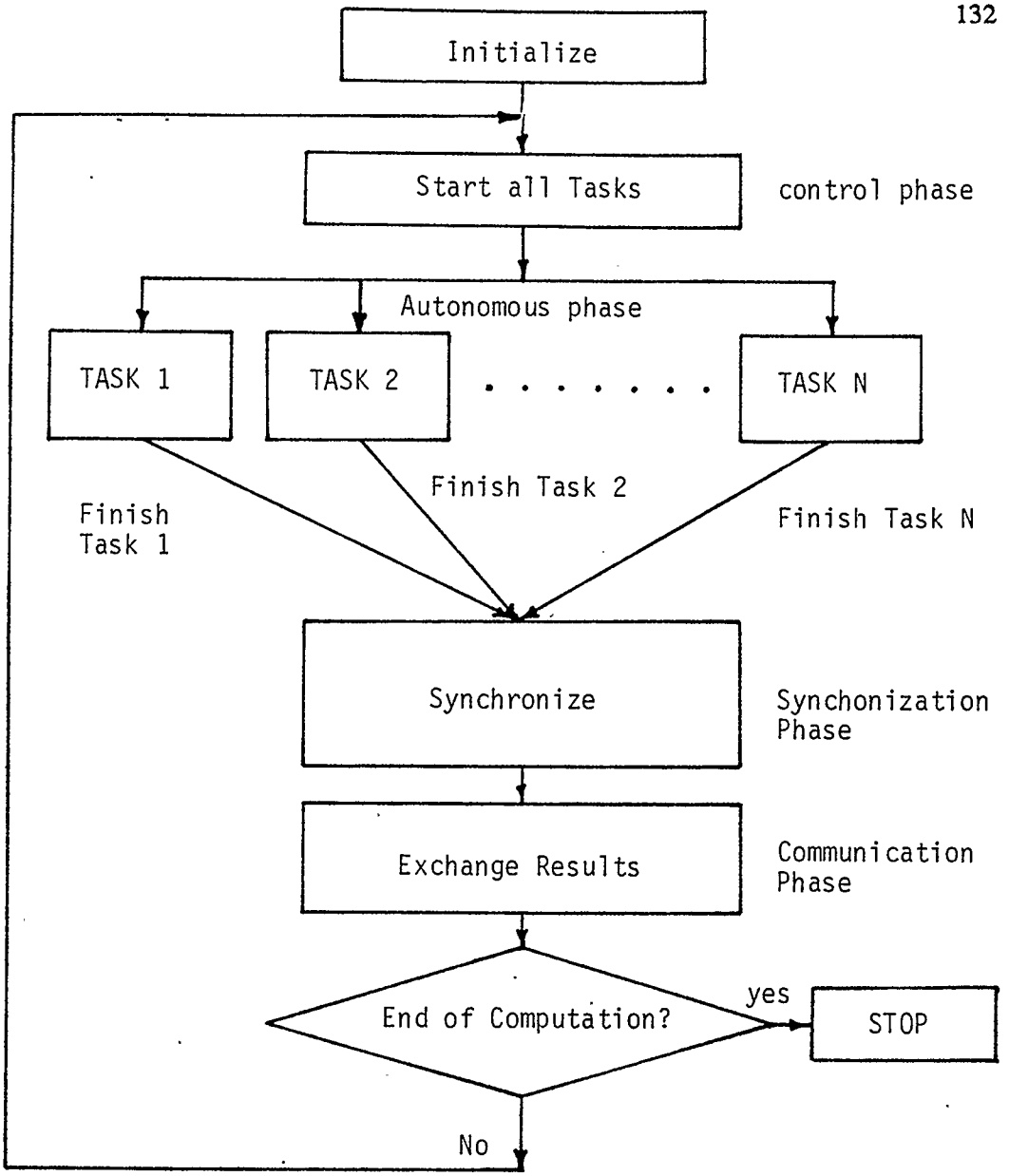


Figure 4.10 Parallel mode of computation for SMS-201 multi-microprocessor system.

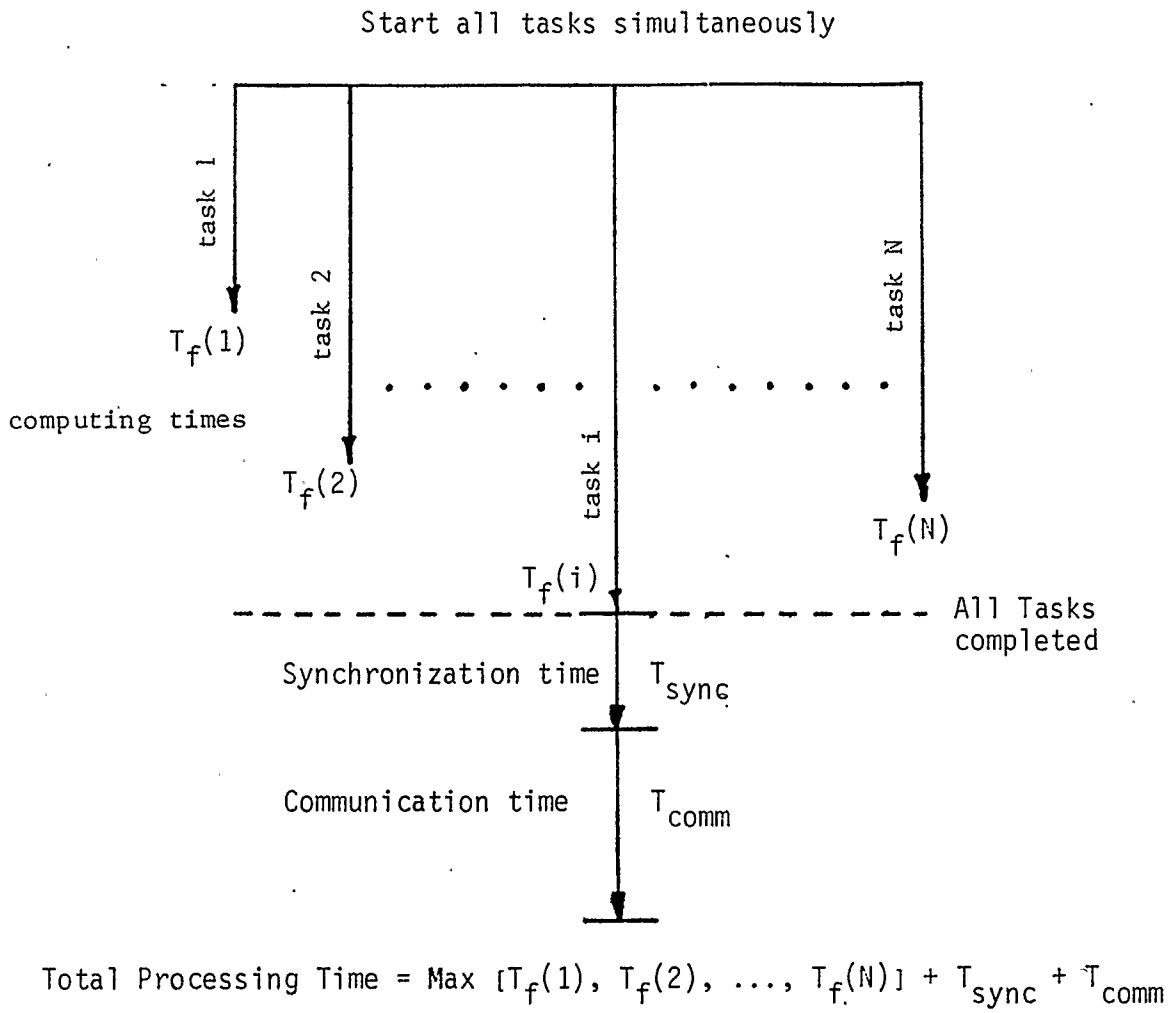


Figure 4.11 Determination of total processing time on SMS-201.

T_{comm} = Total data communication time .

In the autononous phase, the processors execute their own set of instructions as is the case with MIMD computers. However, unlike the true MIMD computers which operate quite asynchronously, all the processors in *SMS* 201 have to finish the processing of their tasks before the communication phase can begin. Also, all processors start their tasks at the same time. These features of global synchronization are very similar to the control of processor arrays which fall under the SIMD architectures. However, in SIMD processor arrays, all processors execute the same instruction in a lock-step manner which is not the case with SMS. Therefore, this model of parallel computation can be classified as a kind of cross between the SIMD and MIMD computer organizations. This combined SIMD/MIMD mode of parallel computation is also supported by Hoshino's "PACS" (Processor Array for Continuum Simulation), later renamed as "PAX" (Processor Array Experiment), parallel processor system developed at the University of Tsukuba, Japan, although processors in this parallel computer are connected in a two-dimensional near-neighbourhood manner [20,51,67-69]. Hoshino calls this mode of parallel processing as quasi-MIMD mode [20] and Wallach calls it ASP (Alternating Sequential/Parallel Processing) [22]. Overall, this model of parallel computation is much closer to MIMD-type operation than SIMD-type operation.

The result of employing the above features of quasi-MIMD operation, distributed communication memories, and data communication by the master processor is that no deadlocks or contentions for shared resources such as the use of bus can occur. Hence, the use of complicated semaphores, etc. in the software to avoid deadlocks/contentions and to ensure the correct execution of the algorithm becomes unnecessary. Moreover, as noted by

Hoshino [20], the process of algorithm design, programming and debugging is easier for the quasi-MIMD operation as compared to the asynchronous true MIMD operation. Another advantage of quasi-MIMD operation is that as a result of global synchronization of all processors at the synchronization points in the parallel algorithm, algorithm complexity results in an analytical form similar to the form for sequential algorithm complexity results can be obtained. Consequently, the performance of the parallel algorithm on quasi-MIMD architecture can be studied without resorting to simulations of the algorithm which is necessary for true MIMD mode of operation due to its asynchronous and non-deterministic nature.

Finally, because of the use of global synchronization by hardware ("anding" of processor interrupts), the synchronization time is very small as compared to process synchronization in asynchronous MIMD computers by software means. For example, in PAX parallel processor, all the processors can be synchronized in less than five floating-point operations time [20].

The constraints of the *SMS*-201 architecture are:

1. The distribution of computational load among the processors should be as equal as possible so that the waiting time of processors (Fig. 4.11) is minimized.
2. The granularity of the tasks on the processors should be sufficiently high so that the overhead of synchronization and data communication is not excessive as compared to actual task computation time.

4.7. On the Number of Processors to be Used

An important decision that has to be made in performing parallel computations on multiple processor systems in a fast and efficient manner concerns the choice of the

number of processors to be used. Ideally, the performance characteristic for an N-processor system should be a speed-up of computations by a factor of N. However, the speed-up obtainable in actual practice is less than N even if the overhead of process synchronization and data communication is negligibly small. This is due to the fact that in actual applications it is rarely possible to completely parallelize all the computations involved in solving a problem. Only a fraction x of the total computations can be parallelized completely and the remaining fraction $(1 - x)$ has to be processed sequentially.

The sequential fraction $(1 - x)$ of total computations determines the overall speed-up that can be obtained by parallel processing even if the parallel computation time for the parallel fraction x is reduced to negligibly small value by using a very large number of processors. This fact was first noted by Amdahl in his 1967 paper [70] and is now known as Amdahl's law. It states that if a computer has two speeds of operation, the slower mode will dominate overall performance even if the faster mode is infinitely fast.

Ware's model of parallel computations [71] also captures the same idea as follows:

Let

$T_1 =$ problem computations time on single processor

$T_N =$ problem computation time on N processors

$x =$ fraction of total computation that can be processed in parallel on N processors

$(1 - x) =$ fraction of total computations to be processed sequentially

Then $T_N = \frac{x\ T_1}{N} + (1 - x)\ T_1$. Therefore, speed-up S is given by

$$S = \frac{T_1}{T_N} = \frac{T_1}{\frac{x\ T_1}{N} + (1 - x)\ T_1}$$

or

$$S = \frac{1}{\frac{x}{N} + (1 - x)} \quad . \tag{4.7}$$

When $N \rightarrow \infty$, the speed-up S is given by

$$S = \frac{1}{(1 - x)} \quad .$$

Therefore, the overall speed-up is limited by the sequential fraction $(1 - x)$ of the total computation. Hence, there is no advantage in using a very large number of processors if the sequential fraction $(1 - x)$ is not very small. Speed-up as given by (4.7) is plotted in Fig. 4.12 as a function of the parallel fraction x for 8, 16, 32 and 64 processors. As can be seen from this plot, for parallel fraction x less than 0.9 or so, the overall speed-up factor obtained by using different numbers of processors is not much different from one another. Hence, a large number of processors should be used only when a very large fraction of total computations can be parallelized.

In our case, the fraction of strictly sequential computation is very small. However, the parallel fraction of the total computations (Chapters 3 and 5) does not have arbitrarily large parallelism in it. Moreover, the amount of parallelism in the algorithms is different at different stages of the algorithm. For an algorithm having negligible sequential part but a fraction x having p_1 parallelism in it and the remaining fraction $(1 - x)$ having p_2 parallelism in it, the Ware's model can be modified as follows:

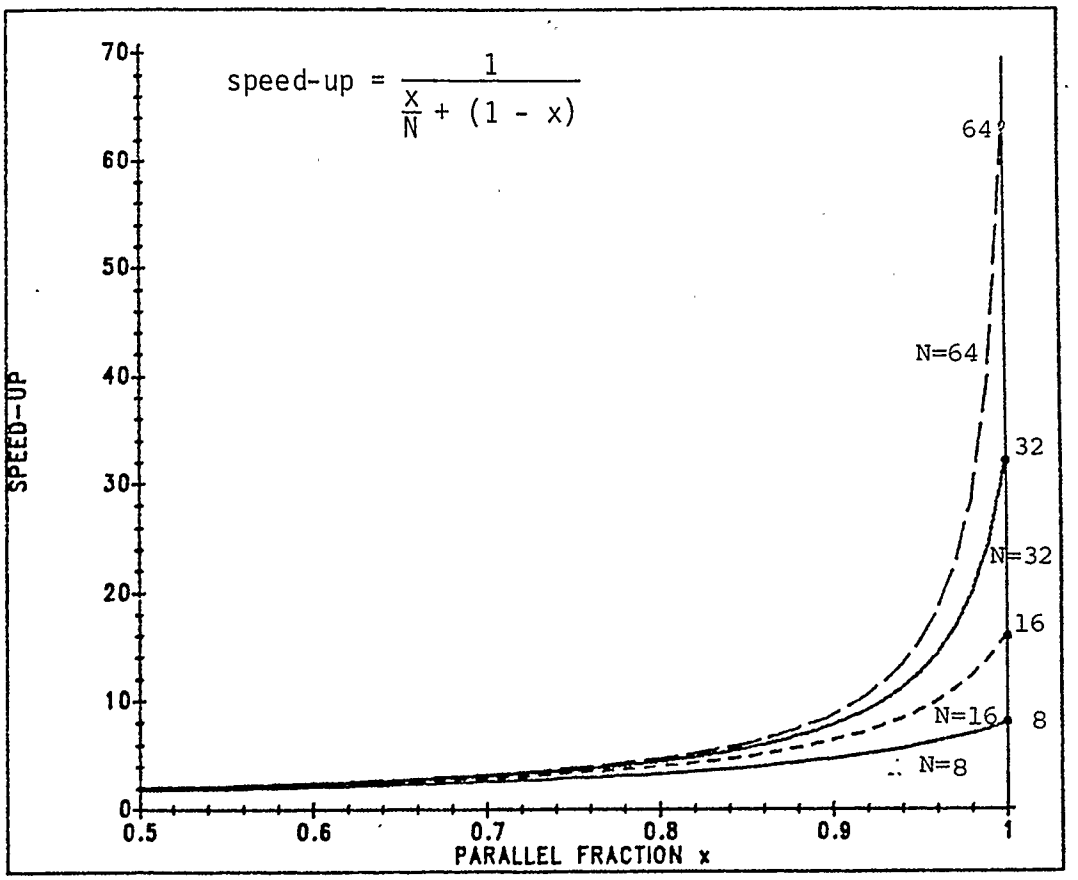


Figure 4.12 Ware's model of parallel computer speed-up for 8, 16, 32 and 64 processors

$$S = \frac{1}{\frac{x}{p_1} + \frac{(1-x)}{p_2}} \quad (4.8)$$

If high computational efficiency, i.e., better hardware utilization is the only criterion, then the number of processors equal to or less than the smaller of the p_1 and p_2 values should be chosen. On the other hand, if higher speed-up is the only criterion, then number of processors equal to the larger of the p_1 and p_2 values should be chosen. In practice, however, it is desirable to choose the number of processors such that a reasonable efficiency (say 40-50 percent at present-day hardware cost) of parallel computation is achieved. If the past decrease in the hardware costs is any indication, the present high cost of 32-bit microprocessors, coprocessors etc. will fall significantly in the near future. Under these assumptions, efficiencies lower than 50% can be tolerated and therefore a larger number of processors can be used to achieve higher values of speed-up. It is also important to note that, in general, the speed-up is not a continuous function of the number of processors. Rather, it varies in a discrete manner (depending on the particular algorithm) with the number of processors used.

From the above considerations, the choice of number of processors equal to the number of chains in the network was found to be a good one. Another good choice was to use a number of processors equal to the number of random links in the network. As shown in Chapter 5, the first choice gives higher efficiency but lower speed-up than the second choice. Details of the actual speed-ups and efficiencies obtained are given in chapters 5 and 6.

CHAPTER 5

ALGORITHM PERFORMANCE ON MULTIPLE PROCESSOR SYSTEMS

5.1. Introduction

This chapter deals with the performance of the Porsching's numerical integration algorithm on multiple processor systems. Two direct parallel methods for solving the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$, which is the dominant computation per time-step in this algorithm for large network problems, are considered. An estimate of parallel computation time for each of these two methods of solving the matrix equation is obtained for three different situations, namely when (i) the number of slave processors is equal to the number of chains s in the network, (ii) number of slave processors is equal to the number of random links q in the network, (iii) number of slave processors is equal to the number of chains s in the network and there is a two-dimensional mesh-connected array of $q \times q$ processors connected to the bus via a two-dimensional memory. An estimate of parallel computation time for performing the rest of the computations is also given. Finally, the speed-ups and efficiencies obtained for these different cases are studied as a function of the number of processors, ratio of one data communication time to one floating-point operation time, ratio of one synchronization time to one floating-point operation time, and the size of the network.

5.2. Parallel Computation of Flow-Rate Increments from the Matrix Equation (3.2)

In this section, two direct parallel methods for solving the matrix equation (3.2) which correspond to the two direct sequential methods of sections 3.2.1.2.1 and 3.2.1.2.2 are discussed.

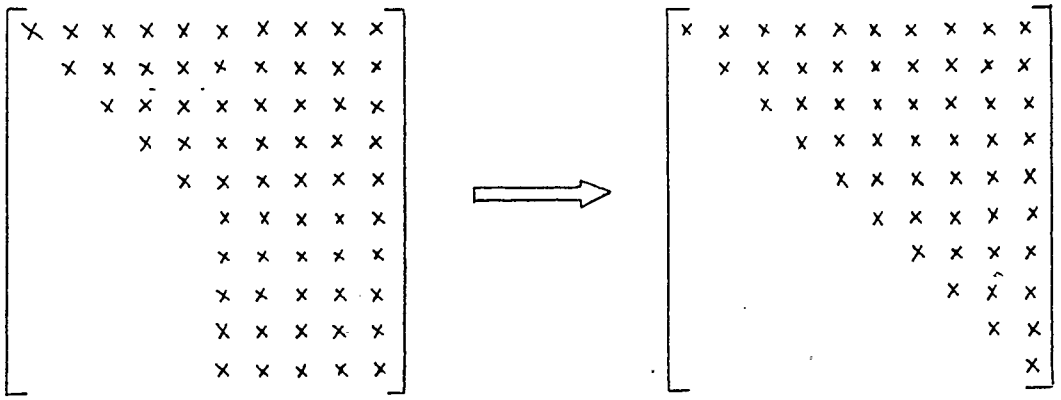
5.2.1. Direct parallel method I (Normal Gaussian elimination plus Gauss-Jordan elimination)

The structural form of the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ is given in (3.2). Parallel solution of this equation by method I involves the following three main steps:

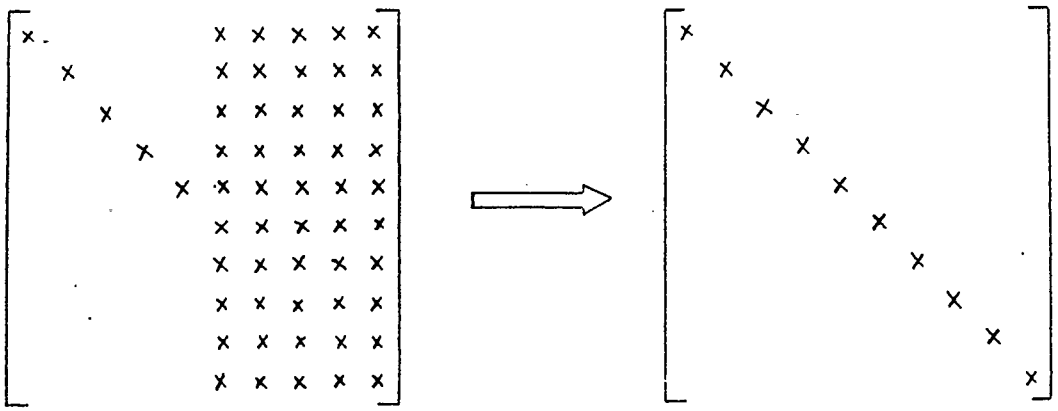
(i) Partial forward elimination of matrix **A** by parallel Gaussian elimination of lower elements of all tridiagonal submatrices $\mathbf{D}_i$ and complete elimination of all row submatrices $\mathbf{B}_i$, $i = 1, 2, \ldots, s$. For this computation, matrices $\mathbf{B}_i$ are partitioned row-wise among the slave processors such that each slave processor is responsible for the elimination of $\left\lceil \frac{q}{N_p} \right\rceil$ rows of each matrix $\mathbf{B}_i$ where N_p is the number of processors and $\lceil\ \rceil$ is the ceiling function. After this computation, matrix equation (3.2) assumes the form shown in (3.4).

(ii) Solution of the sub-vector $\Delta\,\omega_{s+1}$ from (3.6) by parallel Gauss-Jordan elimination [72].

In the Gauss-Jordan method, matrix elements below as well as above the pivot elements are eliminated at each elimination stage as opposed to the Gaussian elimination where elements below the pivot element only are eliminated. This difference in the two algorithms is shown in Fig. 5.1 which shows the partially eliminated matrix $\hat{\mathbf{V}}$ at the kth elimination stage. It is clear from Fig. 5.1(a) that Gaussian elimination is not suitable for



(a) Gaussian Elimination



(b) Gauss-Jordan Elimination

Figure 5.1 Intermediate and final forms of matrix $\hat{V}$ in the forward elimination step for solving the matrix equation.

multiprocessing since the number of elements that can be eliminated concurrently (i.e. elements below the pivot element) decreases steadily as we proceed with the forward elimination process. Moreover, the back-substitution step in the Gaussian elimination is basically a recursive process. Consequently, it is not suitable for multiprocessing. On the other hand, the number of elements to be eliminated at each elimination step in the Gauss-Jordan method remain fixed and large, i.e. equal to the dimension of the matrix minus one throughout the elimination process as shown in Fig. 5.1(b). After elimination, matrix A has only diagonal elements as non-zero elements and therefore the solution vector is recovered by dividing the right-hand side of the matrix equation by the diagonal elements. All these divisions can also be done in parallel. Thus, we observe that the Gauss-Jordan method for solving matrix equations has a very high concurrency in it. However, the Gauss-Jordan method takes about 33% more floating-point operations on a single processor as shown below. Hence, it is generally not used for solving matrix equations on conventional serial computers.

The elimination stage and the pseudo-code for the sequential Gauss-Jordan algorithm for solving the general qth-order matrix equation $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ is given in Fig. 5.2(a) and (b) respectively. In our case, $\mathbf{C} = \hat{\mathbf{V}}$, $\mathbf{x} = \Delta\,\boldsymbol{\omega}_{s+1}$, and $\mathbf{b} = \hat{\boldsymbol{\zeta}}_{s+1}$. The total floating point operation count in the algorithm can be obtained as follows (refer to Fig. 5.2(b)). Operation count for the elimination stage is $\sum_{k=1}^{q} \sum_{i=1}^{q-1} [1 + 2(q - k + 1)]$

$$= q^3 + q^2 - 2q \quad .$$

Operation count for back-substitution is q. Therefore, total computation time

$$T_1(GJ) = (q^3 + q^2 - q)\, t_f \quad . \tag{5.1}$$

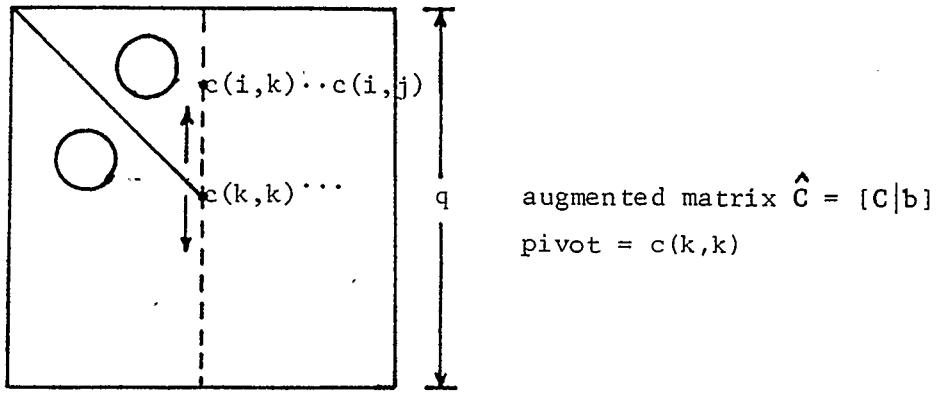


Figure 5.2(a) kth-stage of Sequential Gauss-Jordan elimination for solving $\mathbf{C}\,\mathbf{x} = \mathbf{b}$

---

for $k = 1$ to q do {for each pivot}

  For $i = 1$ to q $(i \neq k)$ do {for each row except pivot row}

    $\text{temp} = \dfrac{c(i, k)}{c(k, k)}$

    For $j = (k + 1)$ to $(q + 1)$ do { for each element of a row}

      $c(i, j) = c(i, j) - \text{temp} \times c(k, j)$ {update element}

    end do {loop j}

  end do {loop i}

end do {loop k}

For $i = 1$ to q do {recover final solution by back-substitution}

  $x(i) = \hat{b}(i)/c(i, i)$

end do

Figure 5.2(b) Pseudo-code for sequential Gauss-Jordan elimination for solving $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ on a single processor.

The total sequential computation time for Gaussian elimination was obtained in Chapter 3 (Eqn. (3.9)). A comparison of the two computation times shows that Gauss-Jordan algorithm is about 33% slower than Gaussian elimination on a single processor.

Partitioning of the parallel Gauss-Jordan algorithm on N_p processors is shown in Fig. 5.3(a) and the pseudo-code of the parallel algorithm is given in Fig. 5.3(b). In the parallel version of the algorithm, a slice of $h_r = \left\lceil \frac{q}{N_p} \right\rceil$ rows of $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ is stored in and assigned to each processor for elimination and back-substitution. The pivoting row is read by the master processor from the slave processor communication memory which holds it and is then broadcast to all the remaining slave processors. After receiving this pivot row, each slave processor performs the Gauss-Jordan elimination on its slice of rows. After performing the elimination step on its slice of rows, all the slave processors synchronize and the whole procedure is repeated again. After elimination, back substitution is also performed in parallel for the assigned variables $x(i)$ to each slave processor. An estimate of computation time for parallel Gauss-Jordan algorithm can be obtained as follows: (refer Fig. 5.3(b)). The slice of rows per processor has been assumed to be an integer in determining the operation count, i.e. $h_r = \left\lceil \frac{q}{N_p} \right\rceil = \frac{q}{N_p}$.

Floating-point operation count for parallel Gauss-Jordan elimination is

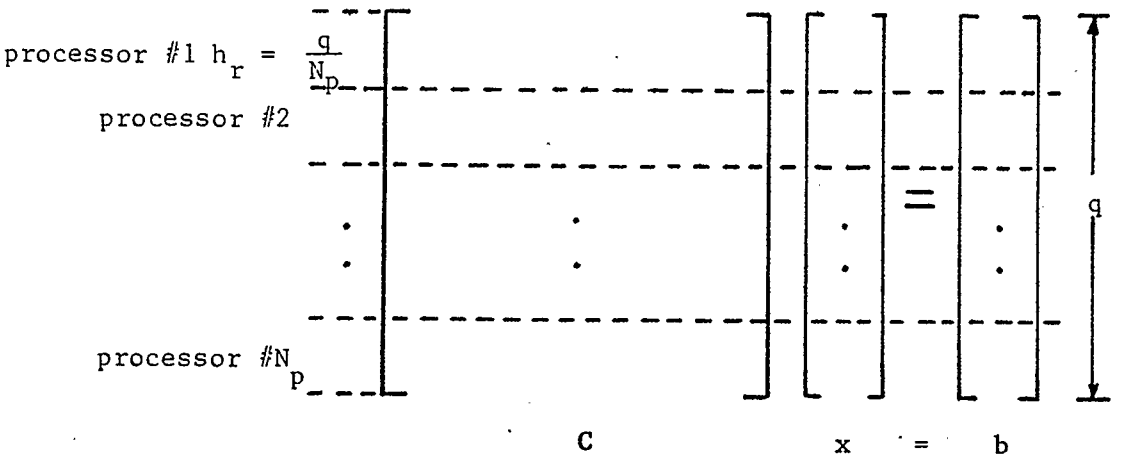


Figure 5.3(a) Problem partitioning in parallel Gauss-Jordan algorithm

PARALLEL ELIMINATION STEP:

for $l = 1$ to Np do {for each processor l}
  for $k = [(l - 1)hr + 1]$ to $hr \cdot l$ do {for each row k of the assigned hr rows to a processor l}
    broadcast row k and its position from processor l to all processors
    call pareliminate (k)
    synchronize all the processors
  end do {loop k}
end do {loop l}

PARALLEL BACK-SUBSTITUTION STEP:

for $l = 1$ to Np pardo {perform the following computations concurrently on Np processors}
  for $i = [(l - 1)hr + 1]$to $hr \cdot l$ do {for each variable $x(i)$ of the assigned hr number of elements to processor l}
    $x(i) = \hat{b}(i)/c(i, i)$
  end do {loop i}
end pardo

Procedure Pareliminate (k):

for $l = 1$ to Np pardo {perform the following computations concurrently on Np processors}
  for $i = [(l - 1)hr + 1]$ to lhr do {for each row i of the assigned hr to a processor l}
    if $i = k$ {do nothing for the pivot row}
      go to E
    end if
    $\text{temp} = \dfrac{c(i, k)}{c(k, k)}$
    for $j = (k + 1)$ to $(q + 1)$ do {for each element of a row}
      $c(i, j) = c(i, j) - \text{temp} \times c(k, j)$ {update the element $c(i, j)$}
E: end do {loop j}
  end do {loop i}
end pardo

Figure 5.3(b) Pseudo-code for parallel Gauss-Jordan elimination for solving $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ on N_p processors.

$$\sum_{l=1}^{N_p} \sum_{k=(l-1)}^{h_r l} [1 + 2(q - k + 1)]$$

$$= \sum_{k=1}^{h_r N_p} h_r [1 + 2(q - k + 1)]$$

$$= h_r [q^2 + 2q] \cdot$$

Floating-point operation count for back-substitution is

$$\sum_{i=(l-1)h_r+1}^{h_r l} 1$$

$$= h_r \cdot$$

Therefore, total parallel time for performing the floating-point operations is $h_r [q^2 + 2q + 1]\, t_f = \frac{q}{N_p} [q^2 + 2q + 1]\, t_f$. If h_r is not an integer, then

$$\text{Total parallel computitional time} = \left\lceil \frac{q}{N_p} \right\rceil [q^2 + 2q + 1]\, t_f \cdot$$

If t_c is the time for one data transfer over the bus, then the total data communication time is

$$\sum_{k=1}^{q} (q + 2 - k)\, t_c$$

$$= \left[\frac{q^2}{2} + \frac{3}{2} q \right] t_c \cdot$$

There are a total of q synchronizations involved in the algorithm. Therefore, if t_s is the time to carry out one synchronization, then the total synchronization time is $q\ t_s$.

The total time to perform parallel Gauss-Jordan elimination is given by the sum of the

above three times. Hence,

Total time to perform parallel Gauss-Jordan elimination on N_p processors,

$$T_{N_p}(GJ) = \left\lceil \frac{q}{N_p} \right\rceil \left[q^2 + 2q + 1\right] t_f + \left[\frac{q^2}{2} + \frac{3}{2} q\right] t_c + q\, t_s \cdot \tag{5.2}$$

The speed-up obtained is given by

$$\text{speed–up } S = \frac{T_1(GJ)}{T_{N_p}(GJ)} = \frac{(q^3 + q^2 - q)\, t_f}{\left\lceil \frac{q}{N_p} \right\rceil (q^2 + 2q + 1)\, t_f + \left[\frac{q^2}{2} + \frac{3}{2} q\right] t_c + q\, t_s} \cdot \tag{5.3}$$

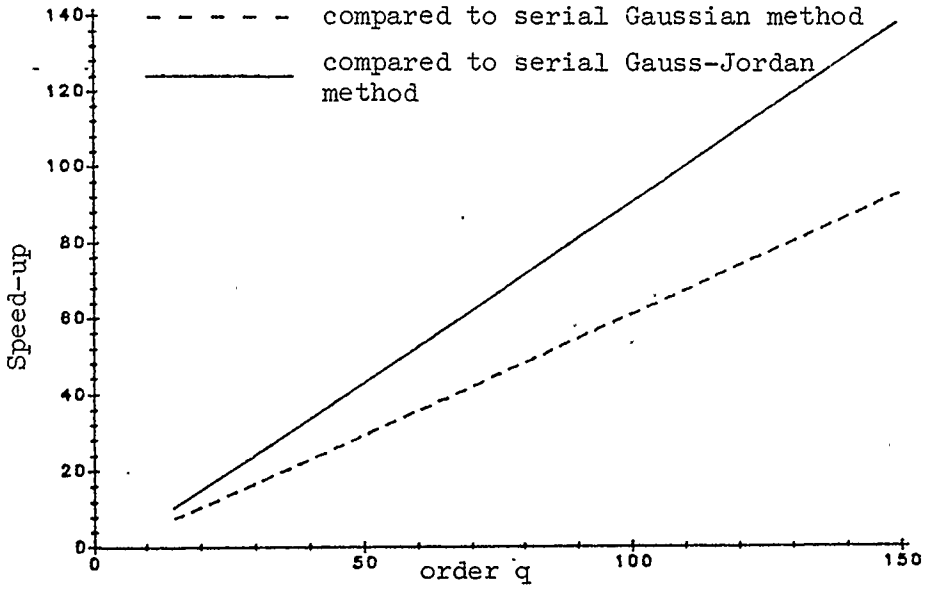
A better idea of the usefulness of the parallel Gauss-Jordan algorithm is obtained when it is compared with the best sequential algorithm, i.e. Gaussian elimination rather than the slower sequential Gauss-Jordan algorithm as done in (5.3). With this definition of speed-up, we have

$$\text{speed–up } S = \frac{T_1(Gauss)}{T_{N_p}(GJ)} = \frac{\left[\frac{2}{3} q^3 + \frac{3}{2} q^2 - \frac{7}{6} q\right] t_f}{\left\lceil \frac{q}{N_p} \right\rceil (q^2 + 2q + 1)\, t_f + \left[\frac{q^2}{2} + \frac{3}{2} q\right] t_c + q\, t_s} \cdot \tag{5.4}$$

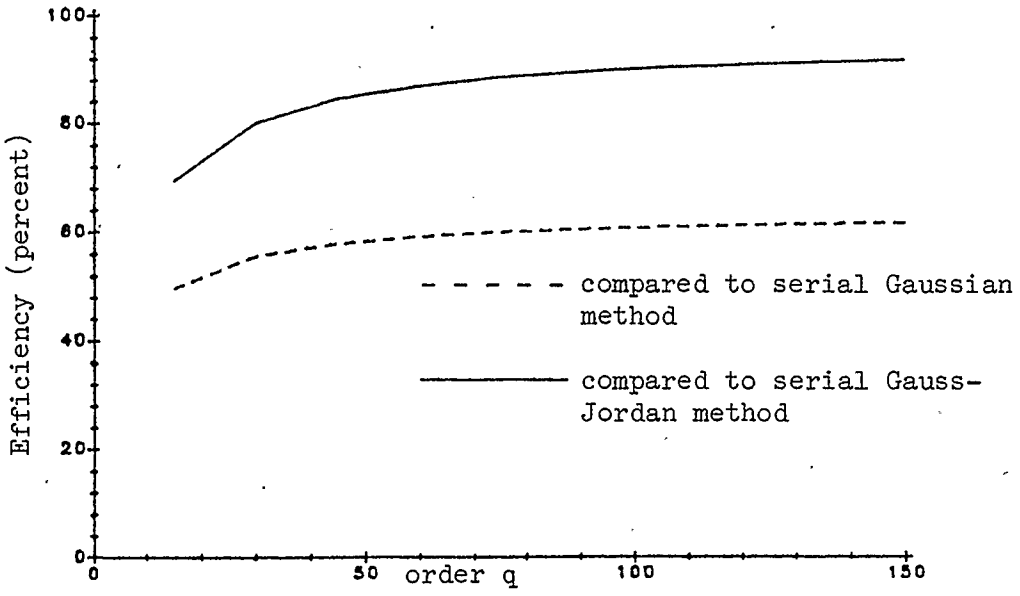
Efficiency of parallel computation is given by

$$\text{Efficiency} = \frac{\text{Actual speed–up}}{\text{Ideal speed–up}} \times 100 = \frac{S}{N_p} \times 100 \cdot \tag{5.5}$$

The speed-up and efficiency as given by (5.3), (5.4) and (5.5) are plotted in Fig. 5.4(a) and (b) respectively for $q = 15$ *to* 150, $t_c = 0.1\, t_f$, and $t_s = 5\, t_f$. As can be seen from these plots, almost a linear speed-up with the number of processors and therefore a constant efficiency is obtained for $q = N_p > 25$. This is so because the data communication in the parallel Gauss-Jordan algorithm consists of a series of data broadcasts for which bus is the



(a) Speed-up versus order q of the equation



(b) Efficiency versus order q of the equation

Figure 5.4 Speed-up and efficiency of parallel processing for solving the matrix equation $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ by Gauss-Jordan method ($t_c = 0.1 \times t_f$, $t_s = 5.0 \times t_f$, $N_p = q$)

best communication network. Also, parallel computation time and data communication time both increase almost at the same rate with the size of the problem. When the order of the matrix equation is small, i.e. when $q < 25$, the efficiency of parallel computation is relatively low due to the significant overhead of processor synchronization as compared to parallel computation time.

(iii) Parallel computation of subvectors $\Delta\ \omega_i$ from (3.7)

Since all the operations for each i are independent of each other, they can be performed concurrently on s processors.

An estimate of parallel computation time for solving the matrix equation $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ is now obtained. Three cases for the number of processors employed are considered: (a) number of processors equal to number of chains s in the network, (b) number of processors equal to number of random links q in the network, (c) number of processors equal to number of chains in the network and a two-dimensional mesh-connected array of $q \times q$ processors connected to the bus via a two dimensional memory.

5.2.1.1. Number of processors equal to number of chains in the network

The pseudo-code of the parallel algorithm for solving $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ on s processors is given in Fig. 5.5. In this method, sub-matrix $\mathbf{X} = [\mathbf{B}_1\ \mathbf{B}_2\ \cdots\ \mathbf{B}_s\ \mathbf{V}\ \zeta_{s+1}]$ of matrix $[\mathbf{A}\ \zeta_{s+1}]$ is partitioned row-wise among the s processors such that a slice $h_r = \left\lceil \frac{q}{s} \right\rceil$ of rows of $\mathbf{X}$ is stored in and eliminated by each processor. The lower elements of $\mathbf{D}_i$ are eliminated on the master processor concurrently with elimination of $\mathbf{X}$ on the slave processors.

{matrix $[\mathbf{B}_1\ B_2 \cdots\ \mathbf{B}_s\ \mathbf{V}\ \zeta_{s+1}]$ has been partitioned row-wise among s processors (see section 5.3.1.1 and pseudo-code of Fig. 5.19)}

For $j = 1$ to $(K - q)$ do {for each pivot row of A}
  broadcast the non-zero elements of the pivot row j of $[\mathbf{A}|\mathbf{z}]$ from master to s slave processors

  par begin {perform the following computations concurrently}
    eleminate the only non-zero element $a_{j+1,\,j}$ of submatrix **D** on the master processor using (3.3)
    for $l = 1$ to s pardo {for each processor l}
      for $i = [(l - 1)hr + 1]$ to lhr do
        if not $(b_{ij} = 0)$ then
          eliminate element b_{ij} and update non-zero elements of row i using (3.3)
        end if
      end do {loop i}
    end pardo
  par end

  synchronize processors

end do {loop j}

{Now the matrix (3.2) assumes the form given in (3.4)}
solve matrix equation (3.6) for $\Delta\boldsymbol{\omega}_{s+1}$ by parallel Gauss-Jordan elimination
{Now $\left\lceil \frac{q}{s} \right\rceil$ elements of $\Delta\boldsymbol{\omega}_{s+1}$ are in each processor}
broadcast solution vector $\Delta\boldsymbol{\omega}_{s+1}$ to the s slave processors
{$\hat{\mathbf{D}}_i$, $\hat{\mathbf{R}}_i$, $\hat{\zeta}_i$, $i = 1, 2, \cdots, s$ have already been broadcast to slaves while eliminating $\mathbf{B}_i$}

for $i = 1$ to s pardo { perform the following computations concurrently on s processors}
  solve $\Delta\boldsymbol{\omega}_i$ from (3.7)
end pardo

Figure 5.5 Pseudo-code for solving $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ on s processors by direct method I

An estimate of parallel computation time can be obtained as follows.

Let

n_{av} = average of non-zero elements per first or last column of all $\mathbf{B}_i$ matrices which is also equal to average degree of end-nodes of chains minus one

d_{av} = average degree of a non-chained node in the network

t_f = one floating-point operation time

t_c = one data communication time

t_s = processor synchronization time

(i) Forward elimination of all $\mathbf{B}_i$ and lower elements of all $\mathbf{D}_i$.

There are an average of n_{av} non-zero elements per first and last column of each $\mathbf{B}_i$. In the worst case, all these non-zero elements in the first column as well as last column of each $\mathbf{B}_i$ can have their indices differing by one. In this case, row-wise partitioning of $[\mathbf{B}_1\ \mathbf{B}_2\ \cdots\ \mathbf{B}_s\ \mathbf{V}\ \zeta_{s+1}]$ would result in the allocation of all the non-zero elements of a column of $\mathbf{B}_i$ to a single slave processor only. As a result, only one of the slave processors and the master processor will have non-zero elements to be eliminated at any elimination step. The rest of the processors will be idle. The only advantage in this case is that, after this computation, matrix $\hat{\mathbf{V}}$ and vector $\hat{\zeta}_{s+1}$ are automatically partitioned row-wise among the s slave processors with $\left\lceil \frac{q}{s} \right\rceil$ rows per processor and therefore, no data transfer is required for the next computation, i.e. the solution of matrix equation $\hat{\mathbf{V}}\ \omega_{s+1} = \hat{\zeta}_{s+1}$. Moreover, this computation is only a small fraction of the total computation for large networks as shown in Fig. 3.6 and therefore, does not degrade the efficiency of parallel computation substantially for large network problems. In this case, the parallel computation

time will be equal to the sequential computation time for eliminating all $\mathbf{B}_i$. Elimination of lower elements of $\mathbf{D}_i$ proceeds concurrently on the master processor and does not affect the parallel computation time since operations involved in this elimination are less than the operation involved in eliminating $\mathbf{B}_i$ (section 3.2.1.2.1(i)).

Assuming 100% overhead for sparse floating-point operations, parallel computation time $T_{sf\,1}$(worst) is

$$\sum_{i=1}^{s} 2 \left[(p_i)\ (2\ n_{av}^2 + 5\ n_{av}) + (8\ n_{av}^2 + 6\ n_{av})\right] t_f$$

or

$$T_{sf\,1}(\text{worst}) = \left[(4\ n_{av}^2 + 10\ n_{av})\ (K - q) + (12\ n_{av}^2 + 2\ n_{av})\ s\right] t_f \quad . \tag{5.6}$$

Assuming 100% overhead for transferring row and column indices of an element, data, communication time $T_{sc\,1}$ is

$$\sum_{i=1}^{s} 2 \left[(p_i - 1)\ (n_{av} + 3) + 2\ n_{av} + 2\right] t_c$$

or

$$T_{sc\,1} = \left[(2\ n_{av} + 6)\ (K - q) + (2\ n_{av} - 2)\ s\right] t_c \quad . \tag{5.7}$$

There are a total of $(K - q)$ synchronizations in this computation as shown in the pseudo-code of Fig. 5.5. Therefore, processor synchronization time is given by

$$T_{ssyn\,1} = (K - q)\ t_s \quad . \tag{5.8}$$

(ii) Solution of $\Delta\ \omega_{s+1}$ from $\hat{\mathbf{V}}\ \Delta\ \omega_{s+1} = \zeta_{s+1}$

A slice of $\left\lceil \frac{q}{s} \right\rceil$ rows of $\left[\hat{\mathbf{V}}\ \zeta_{s+1}\right]$ is in each processor memory after the above elim-

ination step. The matrix equation is now solved by the parallel Gauss-Jordan algorithm as described in the previous section. The parallel computation time is given by (5.2) which consists of processing time, communication time and synchronization time, i.e.

$$T_{sf2} = \left\lceil \frac{q}{s} \right\rceil (q^2 + 2q + 1)\, t_f \tag{5.9}$$

$$T_{sc2} = \left[\frac{q^2}{2} + \frac{3}{2} q \right] t_c \tag{5.10}$$

$$T_{ssyn2} = q\ t_s \tag{5.11}$$

(iii) Solution of $\Delta\ \omega_i$, $i = 1, 2, \ldots, s$ from (3.7)

A sub-vector $\Delta\ \omega_i$ is solved on processor i using (3.7). $\hat{\mathbf{D}}_i$, $\hat{\mathbf{R}}_i$, $\hat{\zeta}_i$ have already been broadcast to processors while eliminating $\mathbf{B}_i$ in step (i). Only elements of vector $\Delta\ \omega_{s+1}$ are to be broadcast to master and slave processors. The computation time for calculating one $\Delta\ \omega_i$ is given by (refer section 3.2.1.2.1 (iii))

Computation time for one $\Delta\ \omega_i$ is $\left[(4\ n_{av} + 3)\, p_i + (4\ n_{av} - 2)\right] t_f$.

For parallel computation, the computation time will be determined by the computation of $\Delta\ \omega_i$ for the longest chain. If p_m is the length of the longest chain, the parallel computation time, T_{sf_3}, is given by

$$T_{sf_3} = \left[(4\ n_{av} + 3)\, p_m + (4\ n_{av} - 2)\right] t_f \tag{5.12}$$

Communication time to broadcast $\Delta\ \omega_{s+1}$, T_{sc_3}, is $q\ t_f$. (5.13)

Total parallel computation time to solve $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ for $\Delta\ \mathbf{W}^{n+1}$ on s processors will be the sum of times given by (5.6) through (5.13). Therefore, total parallel computation time

is given by

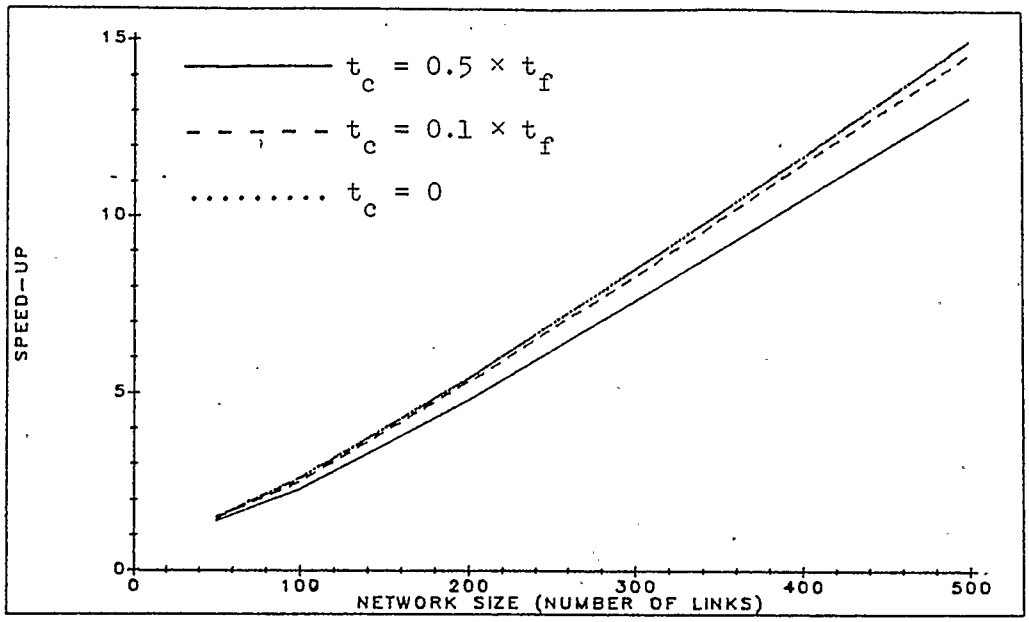
$$T_s(\text{matrix}) \approx \left[\left\lceil\frac{q}{s}\right\rceil (q^2 + 2q) + (4\, n_{av}^2 + 10\, n_{av})\,(K - q) + (12\, n_{av}^2 + 2\, n_{av})s \right.$$

$$\left. + (4\, n_{av} + 3)\, p_m \right] t_f + \left[\frac{q^2}{2} + (2\, n_{av} + 6)\,(K - q) \right. \qquad (5.14)$$

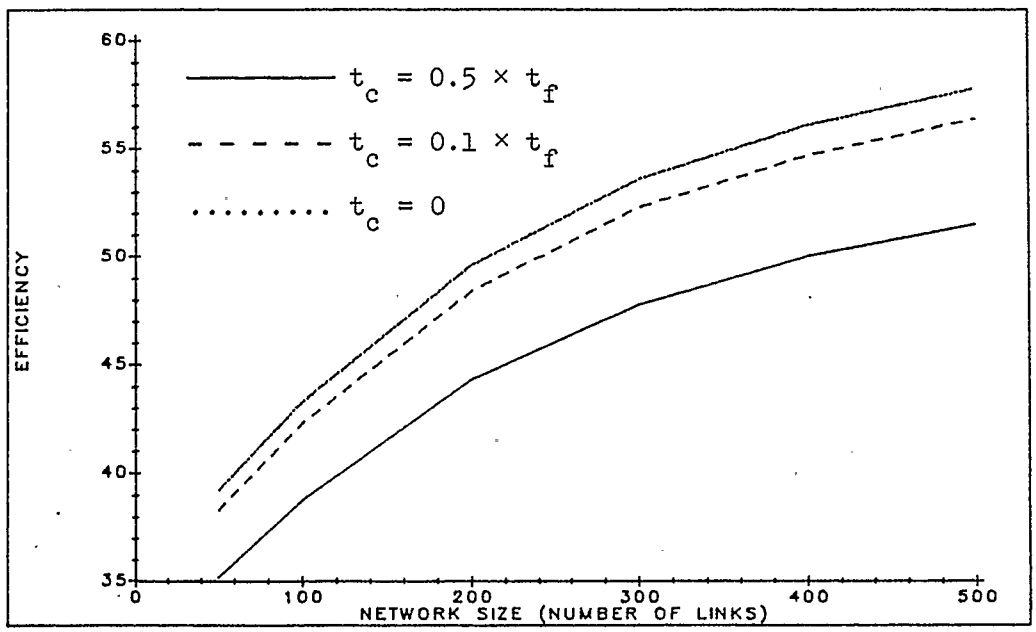
$$\left. + \frac{5}{2}\, q + (2\, n_{av} - 2)s \right] t_c + K\, t_s \quad .$$

The sequential time T_1(matrix) on a single processor is given by (3.11). The speed-up and efficiency of parallel computation versus network size are plotted in Fig. 5.6(a) and (b) for three values of data communication time t_c. Since the number of processors employed is relatively small, the granularity of computation remains high. As a result, data communication overhead in this case remains a relatively small percentage of total processing time and therefore, does not adversely affect the speed-up and efficiency of parallel computation.

It can also be noted that the efficiency is rather low for smaller size networks as compared to efficiency for large networks. This is due to the fact that parallel forward elimination of $\mathbf{D}_i$ and $\mathbf{B}_i$ from the matrix equation (3.2) is very inefficient and this computation forms a significant fraction of total computation time for smaller networks as was shown in Fig. 3.6. Also the worst-case situation was considered in estimating the parallel computation time for this part of computation as given by (5.6). The best case occurs when each non-zero element of a column of $\mathbf{B}_i$ is eliminated concurrently on a separate processor. The computation time in this case will be the computation time T_{sf_1} given by (5.6) divided by n_{av}, i.e.



(a) Speed-up versus network size



(b) efficiency versus network size

Figure 5.6 Speed-up and efficiency of parallel processing for solving $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ on s processors by direct method I.

$$T_{sf_1}(\text{best}) = \left[(4\ n_{av} + 10)\ (K - q) + (12\ n_{av} + 2)s\right] t_f \quad . \qquad (5.15)$$

Actual computation time will lie in-between the two extreme values depending upon the network topology and the labeling employed for the terminating and initiating links to the end nodes of the chains. Finally, the efficiency for smaller size networks is low due to the inclusion of the master processor in counting the total number of processors in the definition of efficiency. This affects the computed efficiency more when the number of slave processors is small.

Another point to be noted is that the efficiency obtained even for the large networks is not very high. This is due to the fact that the sequential Gauss-Jordan elimination method for solving the matrix (3.6) takes about 33% more time than the best sequential method, i.e. Gaussian elimination. The efficiencies have been obtained by dividing the best sequential time with the best parallel execution time, i.e. by using the sequential Gaussian elimination time and the parallel Gauss-Jordan elimination time. For large networks, solution of the matrix equation (3.6) dominates the total matrix solution time (Fig. 3.6). As such, the overall efficiency of parallel computation for the matrix equation (3.2) tends to 66% for the large networks. The actual value for the efficiency for large networks is less than 66% due to overhead of data communication time and processor synchronization time.

5.2.1.2. Number of processors equal to number of random links in the network

Parallel computation time in this case is obtained as follows:

(i) Forward elimination of all $\mathbf{B}_i$ and lower elements of all $\mathbf{D}_i$

In this case, the submatrix $[\mathbf{B}_1\ \mathbf{B}_2\ \cdots\ \mathbf{B}_s\ \mathbf{V}\ \zeta_{s+1}]$ is partitioned row-wise among q slave processors such that a single row is stored in and operated upon by each processor.

The lower elements of $\mathbf{D}_i$ are eliminated concurrently on the master processor as was done for the case of s slave processors.

The parallel computation time in this case is given by

$$T_{qf_1} = \sum_{i=1}^{s} 2\left[(p_i - 1)\,(2\,n_{av} + 5) + (4\,n_{av} + 3)\right] t_f$$

or

$$T_{qf_1} = \left[(4\,n_{av} + 10)\,(K - q) + (4\,n_{av} - 4)\,s\right] t_f \qquad (5.16)$$

Since all the pivot rows are to be broadcast in the elimination process as in the case of s processors, the data communication time remains the same as given by (5.7). Therefore,

$$T_{qc_1} = \left[(2\,n_{av} + 6)\,(K - q) + (2\,n_{av} - 2)\,s\right] t_c \qquad (5.17)$$

The synchronization time is given by

$$T_{qsyn_1} = (K - q)\,t_s \qquad (5.18)$$

(ii) Solution of $\Delta\,\omega_{s+1}$ from $\hat{\mathbf{V}}\,\Delta\,\omega_{s+1} = \zeta_{s+1}$

A single row i of $\left[\hat{\mathbf{V}}\;\zeta_{s+1}\right]$ is in processor i, $i = 1, 2, \ldots, q$ after the previous elimination step. The matrix equation is now solved by the parallel Gauss-Jordan algorithm. Since $N_p = q$ in the present case, $\left\lceil \frac{q}{N_p} \right\rceil$ will be equal to unity in (5.2). Therefore, parallel computation time, communication time and synchronization time will be given by

$$T_{qf_2} = (q^2 + 2q + 1)\,t_f \qquad (5.19)$$

$$T_{qc_2} = \left[\frac{q^2}{2} + \frac{3}{2}\,q\right] t_c \qquad (5.20)$$

$$T_{qsyn_2} = q\ t_s \quad . \tag{5.21}$$

(iii) Solution of $\Delta\ \omega_i$, $i = 1, 2, ..., s$ from (3.7)

A subvector $\Delta\omega_i$ is computed on processor i, $i = 1, 2, ..., s$ as was done for the case of s slave processors. The submatrices $\hat{\mathbf{D}}_i$, $\hat{\mathbf{R}}_i$ and subvectors $\hat{\zeta}_i$ have already been broadcast to slave processors while carrying out the forward elimination of $\mathbf{B}_i$ in part (i). The computation and communication times are, therefore, the same as given by (5.12) and (5.13). Hence,

$$T_{qf_3} = \left[(4\ n_{av} + 3)\ p_m + (4\ n_{av} - 2)\right] t_f \tag{5.22}$$

$$T_{qc_3} = q\ t_c \quad . \tag{5.23}$$

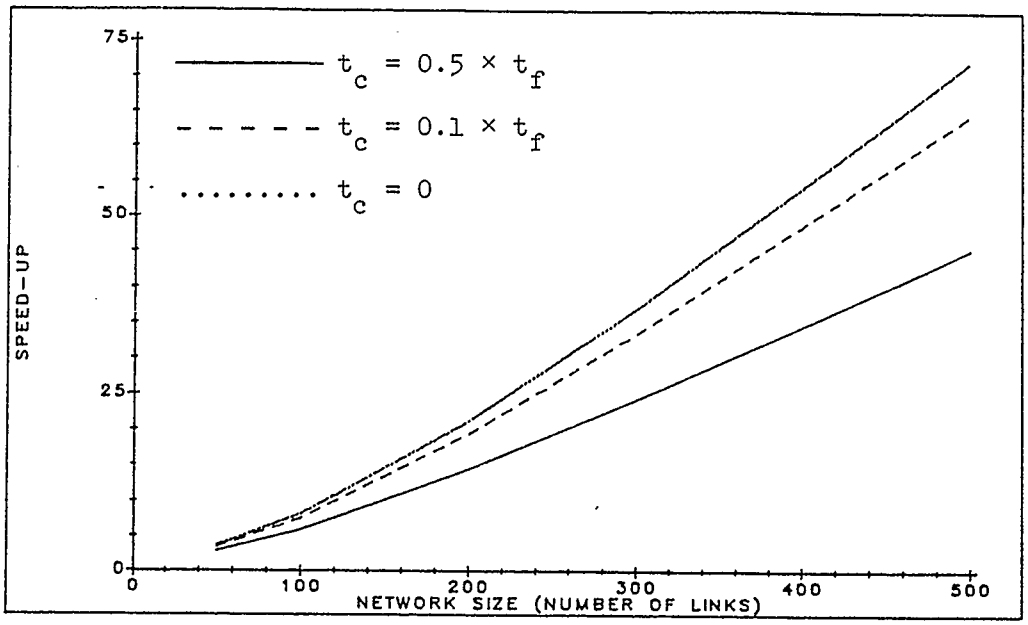
A total of $(q - s)$ processors remain idle for this part of the computation. However, this computation is a small fraction of the total computation (refer Fig. 3.6) for large networks. Hence, it does not significantly affect the efficiency of computation for large networks.

The total computation time will be the sum of the above computation, communication and synchronization times given in (5.16) through (5.23).

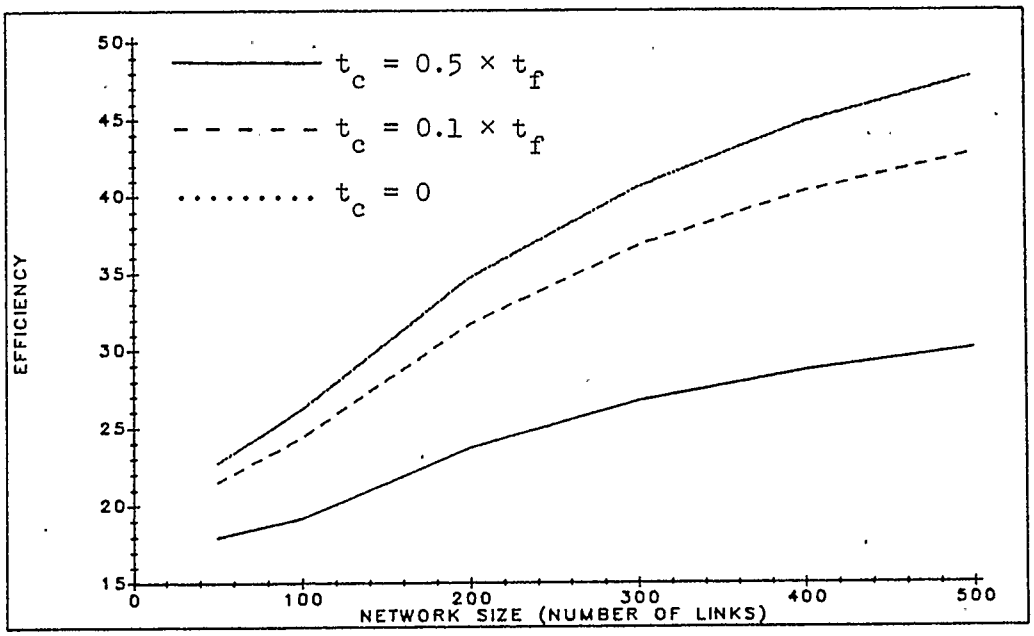
Therefore, total parallel computation time is given by

$$\begin{aligned} T_q(\text{matrix}) \approx & \left[q^2 + (4\ n_{av} + 10)\ (K - q) + 2q + (4\ n_{av} - 4)\ s + (4\ n_{av} + 3)\ p_m\right] t_f \\ & + \left[\frac{q^2}{2} + (2\ n_{av} + 6)\ (K - q) + \frac{5}{2}\ q + (2\ n_{av} - 2)\ s\right] t_c + K\ t_s \quad . \end{aligned} \tag{5.24}$$

The speed-up and efficiency obtained for the example networks are plotted in Fig. 5.7. Even though the communication and synchronization overhead in this case is the same as



(a) Speed-up versus network size



(b) Efficiency versus network size

Figure 5.7 Speed-up and efficiency of parallel computations for solving $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ on q processors by direct method I.

for the case of s slave processors, the efficiency of parallel computation is lower mainly due to smaller granularity of computation. In other words, communication and synchronization overhead becomes a larger fraction of total processing time thus lowering the efficiency of computation. For the same reason, the efficiency is more sensitive to communication time t_c as shown in Fig. 5.7(b). The other reason for lower efficiency, particularly for smaller networks is the inefficient parallel elimination of submatrices $\mathbf{B}_i$ from matrix $\mathbf{A}$ on q processors.

However, if faster execution is the primary computational requirement rather than higher efficiency, then parallel solution on q slave processors should be carried out since, for large networks, it is about four times faster than solution on s processors.

5.2.1.3. Number of processors equal to number of chains in the network plus a two-dimensional mesh-connected array of q x q processors

It was shown in Chapter 3 that, for large networks, most of the computational effort at each integration step is spent in solving the matrix equation $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ (Figs. 3.10(a) and (b)). It was also shown that, for large networks, most of the processing time in solving $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ is taken up by the solution of the q-th order submatrix equation $\hat{\mathbf{V}}\ \Delta\ \omega_{s+1} = \zeta_{s+1}$ (Fig. 3.6). In other words, the total computation time for large networks is dominated by the solution of the q-th order matrix equation $\hat{\mathbf{V}}\ \Delta\ \omega_{s+1} = \zeta_{s+1}$. For real-time applications, it is therefore, important that this matrix equation be solved in a very fast manner using a large number of processors.

On the other hand, it is quite difficult to use a very large number of processors efficiently in a multiprocessing mode with a simple bus interconnection network due to difficulties of problem partitioning among a large number of processors and high data

communication overhead. In order to avoid these difficulties, a number of researchers have proposed the solution of the matrix equation on a two-dimensional mesh-connected array of processors connected in a near-neighbourhood mesh (NNM) fashion [73-77]. Unlike the bus interconnection network where data communication is sequential, data communication in processor arrays is carried out concurrently. In most of these architectures, a combination of pipelining and multiprocessing is employed in the solution of the matrix equation. The parallel matrix solution time for a q-th order matrix equation is on the order of q, i.e. $0(q)$ when run on a $q \times q$ processor array. The processors employed in these architectures are relatively simple and identical with small amount of local memory and the interconnections are local, and short in length. All these characteristics are ideal for their VLSI or WSI (Wafer-scale integration) implementation where a number of these processors can be fabricated on a single chip [78,79]. Thus, the total number of separate chips required for constructing a large-size processor array is reduced. An experimental multi-chip implementation of a systolic array processors using off-the-shelf components is described in [80]. A similar implementation with additional torus connections is also presently being carried out in the Computer Science Department of the University of Calgary under the guidance of Dr. John Cleary.

The processor array is used as an attached processor to the master processor [81] as shown in Fig. 5.8 and linked with the master via buffers and word-to-bit-serial conversion hardware not shown in the figure. The master stores the data in the buffers in a word-serial fashion which is converted to the bit-serial fashion and read by the processor array. After solving the matrix equation, the solution vector is transferred to the stack in a bit-serial fashion and then read by the master in a word serial manner. A Global line, not shown in Fig. 5.8, is connected to all the processors of the array and the master processor

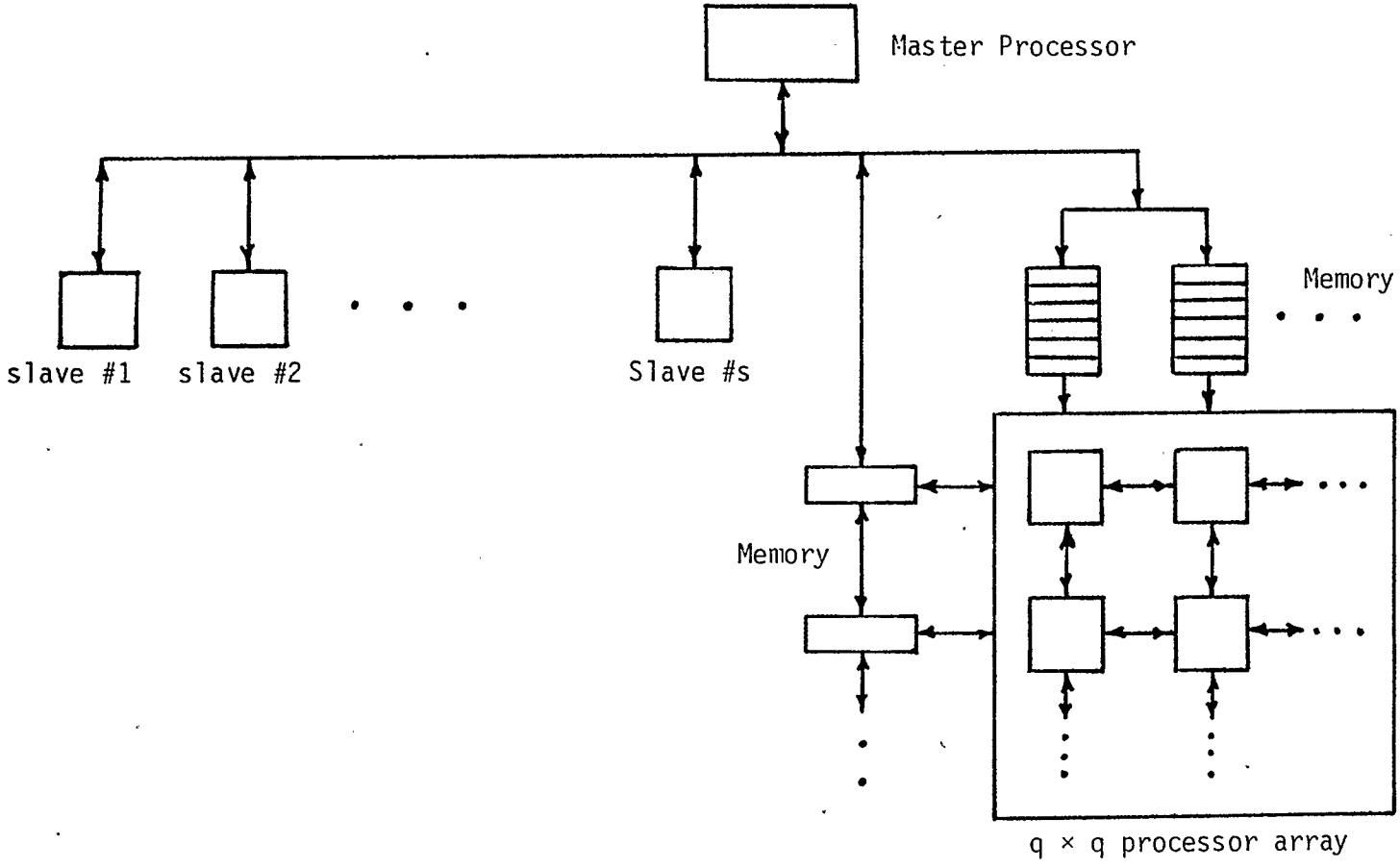


Figure 5.8 Master processor with s slave processors and a two-dimensional mesh-connected array of $q \times q$ processors

for synchronization purposes.

The rest of the computations are performed on s full-fledged slave processors having communication memories, and large local memories for storing their respective programs and large amount of data such as steam tables, etc.

Fawcett's algorithm [76], for solving a general q-th order dense matrix equation $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ on processor array is used in this thesis and is described below.

5.2.1.3.1. Fawcett's algorithm

This algorithm uses **L U** factorization followed by forward and backward substitution using pipelining and multiprocessing on the $q \times q$ processor array (called cellular array in [76]). The main computational step in **L U** decomposition involves the formation of differences between the data array coefficients and inner products of previously computed elements of the triangular factors **L** and **U**, i.e.:

$$u_{ij} = c_{ij} - \sum_{k=1}^{i-1} l_{ik}\, u_{kj} \quad , \quad 1 \le i \le j \le q \tag{5.25}$$

$$l_{ij} = u_{jj}^{-1} \left[c_{ij} - \sum_{k=1}^{j-1} l_{ik}\, u_{kj} \right] \quad , \quad 1 \le j \le i \le q \tag{5.26}$$

where u_{ij} are the elements of upper-triangle matrix **U** and l_{ij} are the elements of lower triangle matrix **L**, i.e.,

$$\mathbf{L} = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & 1 & \\ . & . & & \\ . & . & & \\ . & . & & \\ l_{q1} & l_{q2} & . \ . \ . & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & \cdots & u_{1q} \\ & u_{22} & u_{23} & \cdots & \cdots & u_{2q} \\ & & & & & . \\ & & & & & . \\ & & & & & . \\ & & & & & u_{qq} \end{bmatrix}$$

and $\mathbf{C} = \mathbf{L}\ \mathbf{U}$.

This variation of **L U** decomposition where all diagonal elements l_{ii} of **L** are unity is called the Dolittle's method. The matrix equation to be solved can now be written as

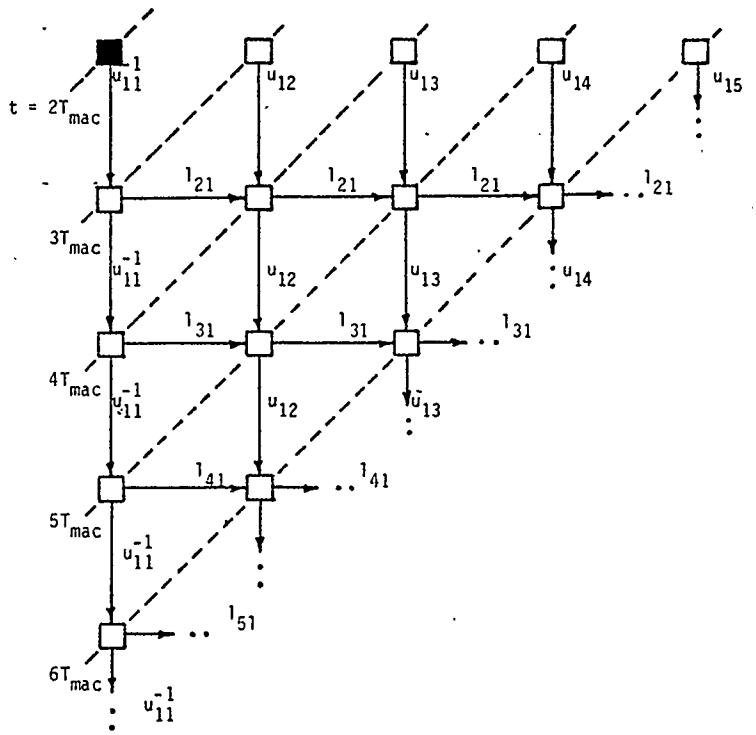
$$\mathbf{L}\ \mathbf{U}\ \mathbf{x} = \mathbf{b}$$

or $$\mathbf{L}\ \mathbf{z} = \mathbf{b}$$

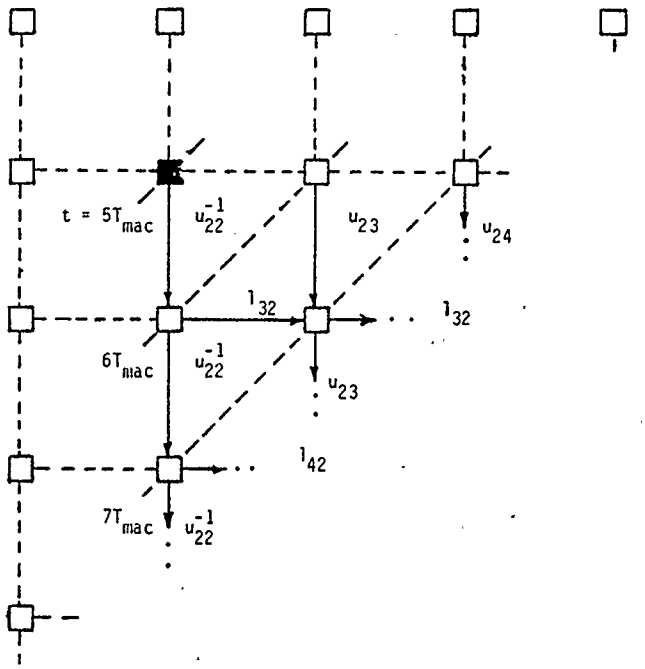
where $$\mathbf{U}\ \mathbf{x} = \mathbf{z}$$

After performing the **L U** factorization, intermediate solution vector **z** is computed by forward substitution given by $\mathbf{L}\ \mathbf{z} = \mathbf{b}$ or $z_i = b_i - \sum_{k=1}^{i-1} l_{ik}\ z_k$, $i = 1, 2, ..., q$. The final solution vector **x** is recovered by the back-substitution $\mathbf{U}\ \mathbf{x} = \mathbf{z}$ or $x_j = u_{jj}^{-1}\ (z_j - \sum_{k=j+1}^{q} u_{jk}\ x_k)$, $j = q, q-1, ..., 1$.

The processor array structure for carrying out these computations is shown in Fig. 5.9. The c_{ij} elements of the coefficient matrix **C** are initially loaded in the array as one

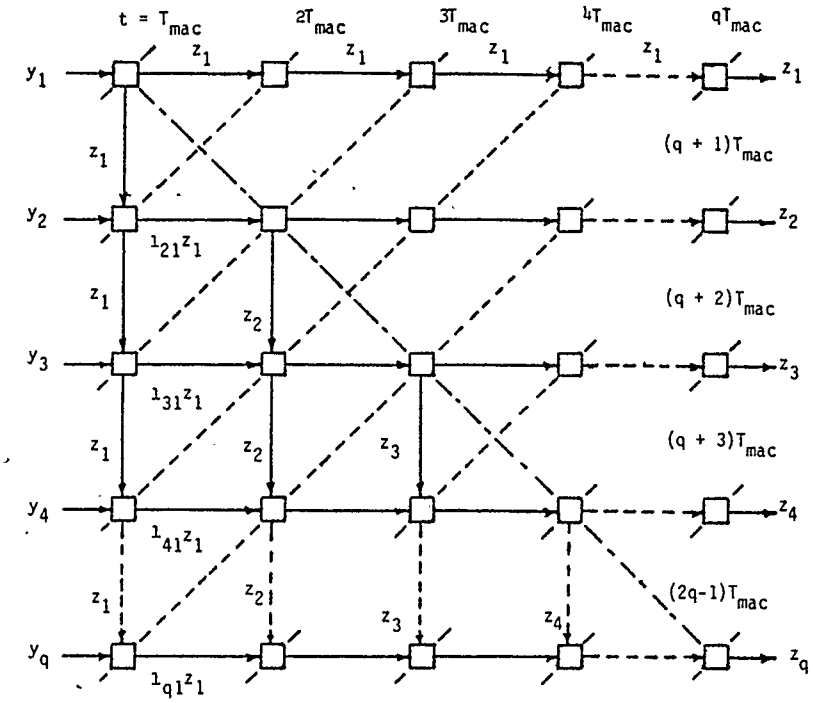


(a) L U decomposition, pass #1

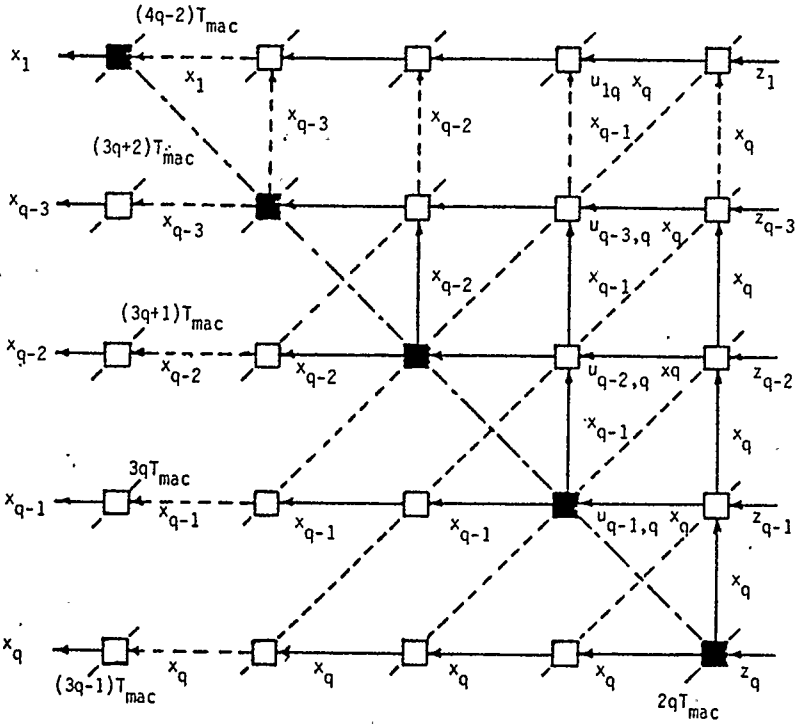


(b) L U decomposition, pass #2

Figure 5.9 Fawcett's algorithm for solving $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ on a mesh-connected array of $q \times q$ processors [76].



(c) Forward substitution



(d) Backward substitution

Figure 5.9

element in each processor. Each processor performs a multiply and accumulate operation of the form sum = sum + data1 * data2 with the data passed to it from processors to its north and west, with the exception of the processors on the main diagonal which perform the reciprocal of u_{ii}.

The product terms in (5.25) and (5.26) are accumulated in a series of passes of computational wave-fronts across the processor array. The first pass originates at diagonal processor (1,1) and travels across the array to the end of the array as shown by skew dotted lines in Fig. 5.9(a). Motion of the wavefront is indicated by the sequence of dotted lines in the figure. In each of the wavefronts, all of the processors are simultaneously busy forming the products required for the **L U** factors. The first pass forms the first product for all the l_{ij} and u_{ij} terms in (5.25) and (5.26). As soon as the first wavefront has crossed processor (2,2), second pass of the wavefront begins at the diagonal processor (2,2) and travels across the array forming the second product terms in (5.25) and (5.26) and so on. The second pass of the wavefront is shown in Fig. 5.9(b). Each pass is pipelined, so that the wavefront associated with each pass follows immediately behind its predecessor. There are a total of $(2q - 1)$ stages of a single wavefront corresponding to $(2q - 1)$ skew dotted lines. If t_{mac} is the time to perform one multiply and accumlate function plus one data communication to the nearest neighbour, then the time for one wavefront to reach the last processor is $(2q - 1)\ t_{mac}$. The rest of the $(q - 1)$ passes of the wavefronts follow behind the first pass in a pipelined fashion. Therefore, time to perform **L U** factorization is

$$(2q - 1)\ t_{mac} + (q - 1)\ t_{mac}$$

$$= (3q - 3)\ t_{mac} \cdot$$

Forward and backward substitution computations are executed on the processor array as shown in Figs. 5.9(c) and (d). For forward substitution, a single pass of the wavefront occurs beginning at processor (1,1) as shown in Fig. 5.9(c). Similarly for back-substitution, a single pass of the wavefront moving in opposite direction occurs beginning at processor (q,q) and ending at processor (1,1) as shown in Fig. 5.9(d). Therefore, the forward substitution time is $(2q-1)\,t_{mac}$ and the backward substitution time is $(2q-1)\,t_{mac}$.

Therefore, total time for the solution of $\mathbf{C}\,\mathbf{x} = \mathbf{b}$ is given by

$$T_{arr}(\text{matrix}) = (3q-3)\,t_{mac} + (2q-1)\,t_{mac} + (2q-1)\,t_{mac}$$

or

$$T_{arr}(\text{matrix}) = (7q-5)\,t_{mac} \quad . \tag{5.27}$$

In general, processor arrays can be operated either in a synchronized, lock-step manner or asynchronously. In the synchronous scheme, there is a global clock network such as an H-tree shown in Fig. 5.10 which distributes the clocking signals over the entire array. The global clock beats out the rythm to which all the processing elements in the array execute their sequential tasks. The clocking rate is determined by the slowest element in the array and by the system clock skew. It has been shown in [82] that the clock skew associated with the H-tree clocking signal distribution scheme is of order q^3, i.e. $0(q^3)$, where q is the size of the array. This result places a severe restriction on the ability to generate a global, synchronous clock signal in a large processor array. Another feature of the synchronous array is its total dedication to implementing a given algorithm. The array is not programmable and, therefore, involves an inflexibility which reduces its scope of applications.

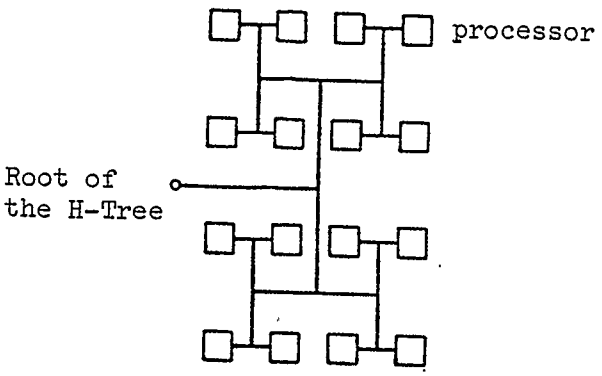


Figure 5.10 Clock distribution in synchronous or systolic arrays using H-tree

An example of synchronous arrays is the systolic array first introduced by H.T Kung and C.E. Leiserson [74].

In contrast, in the asynchronous or self-timed or data flow scheme, there is no global clock, and data communication is by a simple handshaking protocol between each processing element and its immediate neighbours. Whenever the data is available, the transmitting processor informs the receiver of that fact, and the receiver accepts the data whenever it is convenient for it to do so. This suggests that interprocessor communication employ data buffers and "Data Ready/Data Used" flags between adjacent processors as shown in Fig. 5.11 [77].

In summary, it can be said that while the synchronous or systolic scheme may be used for a relatively small size processor array due to reasons of simplicity of design and implementation, asynchronous design may be preferable for large size arrays.

An estimation of parallel computation time for this case is now obtained as follows.

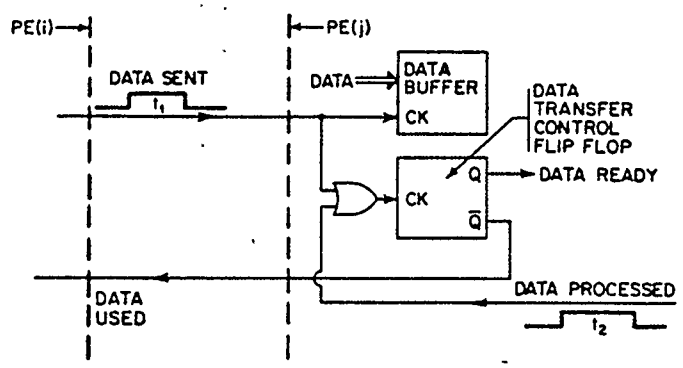


Figure 5.11 Handshaking mechanism in asynchronous processor arrays [77].

(i) Forward Elimination of all B_i and lower elements of all D_i.

This computation is done on s slave processors as described in section 5.2.1.1

Therefore, the computation time, communication time and synchronization time are given by (5.6), (5.7) and (5.8), i.e.

$$T_{sf_1} = [(4\ n_{av}^2 + 10\ n_{av})\ (K - q) + (12\ n_{av}^2 + 2\ n_{av} + s)]\ t_f \tag{5.28}$$

$$T_{sc_1} = [(2\ n_{av} + 6)\ (K - q) + (2\ n_{av} - 2)s\ ]\ t_c \tag{5.29}$$

$$T_{ssyn_1} = (K - q)\ t_s \tag{5.30}$$

(ii) Solution of $\Delta\ \omega_{s+1}$ from $\hat{V}\ \Delta\ \omega_{s+1} = \zeta_{s+1}$

At the end of computations in part (i), matrix $\left[\hat{V} \mid \zeta_{s+1}\right]$ is stored row-wise among s slave processors with a slice of $\left\lceil \frac{q}{s} \right\rceil$ rows in each processor. The master processor now reads these rows of $\left[\hat{V} \mid \zeta_{s+1}\right]$ and writes in the two-dimensional interface memory of the

processor array.

Therefore, data communication time for transferring $\left[\hat{\mathbf{V}} \mid \zeta_{s+1}\right]$ to two-dimensional memory is $q\ (q+1)\ t_c$. The elements of $\hat{\mathbf{V}}$ are now read by the processor array row-wise and in a bit-serial fashion using a series of data shifts. Each processor of the array retains the first matrix element received from the north processor and passes the remaining elements to its south processor in a pipelined fashion till all the data has been read. Let t_c' be the communication time for transferring one data element (or one row) of $\hat{\mathbf{V}}$ to the processor array. Then, data communication time for transferring $\hat{\mathbf{V}}$ to processor array is $2q\ t_c'$.

Therefore, the total data communication time T_{arrc_1} is given by

$$T_{arrc_1} = q(q+1)\ t_c + 2q\ t_c' \tag{5.31}$$

Matrix equation solution time by using **L U** factorization on the array has been estimated previously and is given by (5.27). Therefore, matrix solution time is given by

$$T_{arr}(\text{matrix}) = (7q - 5)\ t_{mac} \tag{5.32}$$

There is one synchronization involved when the processor array completes the matrix solution and informs the master processor so that

$$T_{arrsyn_1} = t_s \tag{5.33}$$

(iii) Solution of $\Delta\ \omega_i$, $i = 1, 2, \ldots, s$ from (3.7) on s processors.

The solution vector $\Delta\ \omega_{s+1}$ computed by the processor array is in the left peripheral memory of the array. It is read by the master processor and broadcast to the s slave processors. Hence, the data communcation time T_{arrc_2} is

$$T_{arrc_2} = q\ t_c \qquad (5.34)$$

Parallel solution time for computing $\Delta\ \omega_i$ on s processors has been estimated in section 5.2.1.1 and is given by (5.12). Hence,

$$T_{sf_3} = [(4\ n_{av} + 3)\ p_m + (14\ n_{av} - 2)]\ t_f \qquad (5.35)$$

where p_m is the number of links in the longest chain in the network.

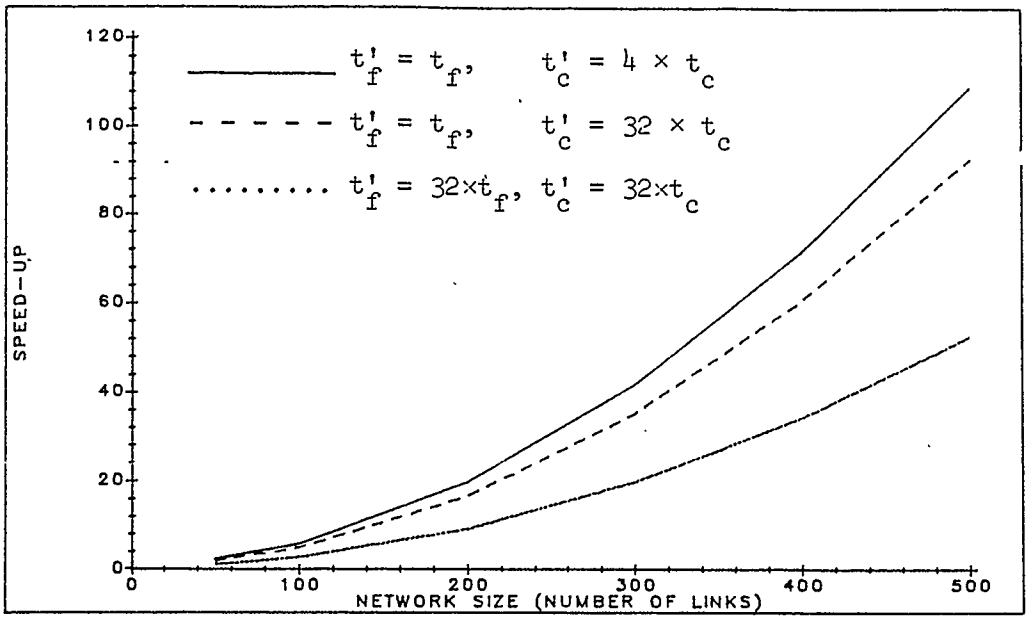
Total parallel computation time to solve $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ will be the sum of the times given by (5.28) through (5.35), i.e.

$$\begin{aligned} T_{s+array}(\text{matrix}) = {} & \left[(4\ n_{av}^2 + 10\ n_{av})\ (K - q) + (12\ n_{av}^2 + 2\ n_{av})\ s + (4\ n_{av} + 3)\ p_m\right] t_f \\ & + (7q - 5)\ t_{mac} + \left[q^2 + (2\ n_{av} + 6)\ (K - q) + 2q\right. \\ & \left. + (2\ n_{av} - 2)s\right]\ t_c + 2q\ t_c' + (K - q + 1)\ t_s \end{aligned} \qquad (5.36)$$

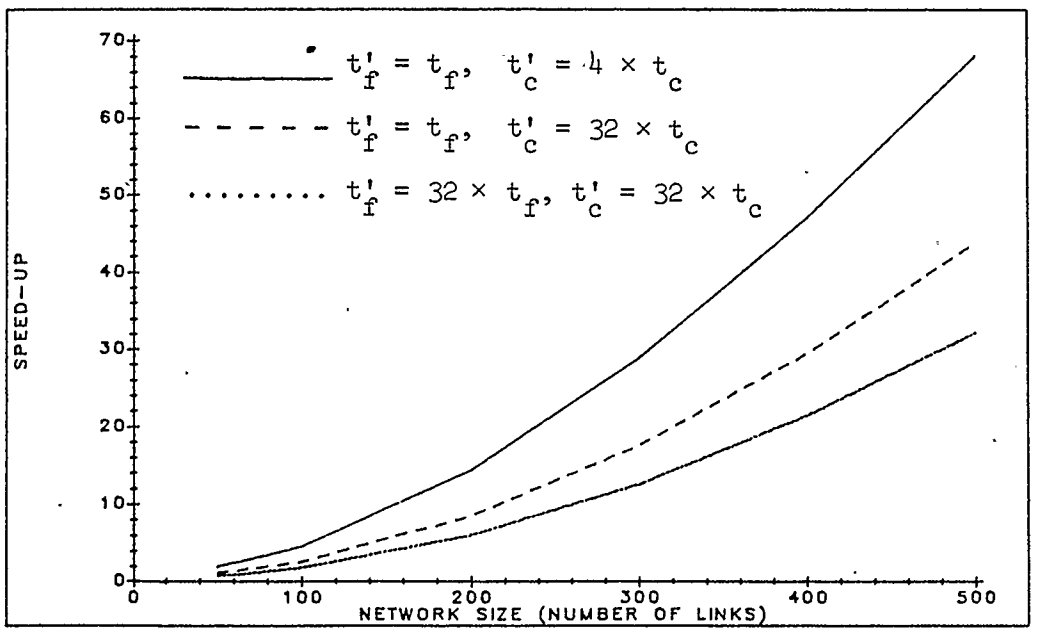
Substituting $t_{mac} = 2\ t_f' + t_c'$ in (5.36) where t_f' is one floating-point operation time on the array, we have

$$\begin{aligned} T_{s+array}(\text{matrix}) = {} & \left[(4\ n_{av}^2 + 10\ n_{av})\ (K - q) + (12\ n_{av}^2 + 2\ n_{av})\ s + (4\ n_{av} + 3)\ p_m\right] t_f \\ & + (14q - 10)\ t_f' + \left[q^2 + (2\ n_{av} + 6)\ (K - q) + 2q\right. \\ & \left. + (2\ n_{av} - 2)s\right]\ t_c + 9q\ t_c' + (K - q + 1)\ t_s \end{aligned} \qquad (5.37)$$

The parallel processing time as given by (5.37) is shown in Fig. 5.12 for a number of network examples and for different values of t_c, t_c' and t_f'. Somewhat better speed-up values are obtained if the best-case time for eliminating $\mathbf{B}_i$ as given by (5.15) instead of worst-case time is used in (5.37). The actual speed-up will be in between the two extreme



(a) $t_c = 0.1 \times t_f$



(b) $t_c = 0.5 \times t_f$

Figure 5.12 Speed-up of parallel computations for solving $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ on s processors and a $q \times q$ processor array by direct method I.

values depending upon the network topology and the link numbering employed.

As can be seen from these plots, the use of a processor array is beneficial only when, (i) the processors in the array have speeds comparable to the speed of slave processors, i.e. ($t_f' = t_f$), (ii) the neighbourhood connections in the array are 8-bit wide, i.e. ($t_c' = 4\ t_c$), (iii) $t_c = 0.1\ t_f$ or so and (iv) the size of the network is large. Otherwise, no special speed advantage is obtained by using the processor array.

5.2.2. Direct parallel method II

The sequential direct method II was described in section 3.2.1.2.2 of Chapter 3 where an expression for its computation time on a single processor was also obtained. This section derives expressions for its computation time on multiple processor systems. Three cases corresponding to three different number of processors being used are considered as was done for direct parallel method I.

5.2.2.1. Number of processors equal to number of chains in the network

The pseudo-code for the parallel algorithm for this case is given in Fig. 5.13. The total computation time can be estimated as follows:

(i) Computation of $\mathbf{c}_i$ from $\mathbf{c}_i = \mathbf{D}_i^{-1}\ \zeta_i$, $i = 1, 2, ..., s$

Each vector $\mathbf{c}_i$ is computed on a separate processor i. Therefore, the total processing time for this computation will be the time to solve the above equation for the longest chain. If p_m is the length of the longest chain, then the time to solve the tridiagonal equation $\mathbf{D}_i\ \mathbf{c}_i = \zeta_i$ for the longest chain is given by (refer section 3.2.1.2.2)

$$T_{sf_1} = (8\ p_m - 7) \quad (5.38)$$

{$\mathbf{D}_i$, $\mathbf{B}_i$, $\mathbf{R}_i$, ζ_i are computed in processor i , $i = 1, 2, \cdots, s$. Therefore no data transfer is required.

Similarly, a slice $\left\lceil \frac{q}{s} \right\rceil$ of rows of V and ζ_{s+1} is computed in each processor i. Therefore no data transfer is required.}

For $i = 1$ to s pardo {perform the following computations concurrently on s slave processors}

$\mathbf{c}_i = \mathbf{D}_i^{-1} \zeta_i$

$\mathbf{E}_i = \mathbf{D}_i^{-1} \mathbf{R}_i$

$\mathbf{h}_i = \mathbf{B}_i \mathbf{c}_i$

$\mathbf{G}_i = \mathbf{B}_i \mathbf{E}_i$

end pardo

Synchronize all processors

Transfer a slice $hr = \left\lceil \frac{q}{s} \right\rceil$ of non-zero rows of $\mathbf{h}_i$ and $\mathbf{G}_i$ from row number $[(l-1)hr + 1]$ to row number $l \cdot hr$ to processor l , $l = 1, 2, \cdots, s$.

for $l = 1$ to s pardo

Compute a slice $hr = \left\lceil \frac{q}{s} \right\rceil$ of rows of matrix Y and vector x from row number $[(l-1)hr + 1]$ to row number $l \cdot hr$ on processor l using equations (3.21) and (3.22)

end pardo

Synchronize all the processors

Compute $\Delta\omega_{s+1}$ from $\mathbf{Y}\Delta\omega_{s+1} = \mathbf{x}$ using parallel Gauss-Jordan elimination.

Synchronize all the processors

broadcast the solution vector $\Delta\omega_{s+1}$ to all the processors

For $i = 1$ to s pardo

$\Delta\omega_i = \mathbf{c}_i - \mathbf{E}_i \Delta\omega_{s+1}$

end pardo

Figure 5.13 Pseudo-code for solving $\mathbf{A} \Delta \mathbf{W}^{n+1} = \mathbf{z}$ on s processors by direct parallel method II.

(ii) Computation of $\mathbf{E}_i$ from $\mathbf{E}_i = \mathbf{D}_i^{-1}\,\mathbf{R}_i$, $i = 1, 2, \ldots, s$

Each $\mathbf{E}_i$ computation involves solving 2 n_{av} tridiagonal systems of equations for each non-zero column in $\mathbf{R}_i$. However, **L U** factorization for $\mathbf{D}_i$ has already been done for computing $\mathbf{c}_i$. Therefore, only forward and backward substitutions as given by (3.23) and (3.24) in Chapter 3 are to be performed for each non-zero column of $\mathbf{R}_i$. The total computation time for computing all $\mathbf{E}_i$ will be determined by the longest chain in the network.

Therefore, total computation time is given by (refer section 3.2.1.2.2)

$$T_{sf_2} = n_{av}\ (5\ p_m - 3)\ t_f \quad . \tag{5.39}$$

(iii) Computation of $\mathbf{h}_i$ from $\mathbf{h}_i = \mathbf{B}_i\ \mathbf{c}_i$, $i = 1, 2, \ldots, s$

Let $n_{\max}$ be the maximum number of non-zero elements in a column of any $\mathbf{B}_i$. Then assuming 100% overhead for sparse computations, total computation time is given by

$$T_{sf_3} = 2(n_{\max} + n_{av})\ t_f \quad . \tag{5.40}$$

(iv) Computation of $\mathbf{G}_i$ from $\mathbf{G}_i = \mathbf{B}_i\ \mathbf{E}_i$

The maximum number of non-zero full columns in any $\mathbf{E}_i$ is $(n_{\max} + n_{av})$. Therefore,

$$T_{sf_4} = 2(n_{\max} + n_{av})\ (n_{\max} + n_{av})\ t_f$$

or
$$T_{sf_4} = 2(n_{\max} + n_{av})^2\ t_f \quad . \tag{5.41}$$

(v) Synchronization of processors

Now the processors have to be synchronized for the data transfer phase. Therefore,

$$T_{s\ sync_1} = t_s \quad . \tag{5.42}$$

(vi) Redistribution of $\mathbf{h}_i$ and $\mathbf{G}_i$, $i = 1, 2, ..., s$ to the slave processors

After step (iv), $\mathbf{h}_i$ and $\mathbf{G}_i$ are in processor i, where $i = 1, 2, ..., s$.

However, they are to be redistributed such that a slice of rows $h_r = \left\lceil \frac{q}{s} \right\rceil$ from row numbers $[(l - 1)\ h_r + 1]$ to $l\ h_r$ is in processor l, $l = 1, 2, ..., s$. Taking 100% communication overhead due to sparse data to be transferred, the total communication time T_{sc_1} is

$$T_{sc_1} = \sum_{i=1}^{s} 2 \left[2\ n_{av} + (2\ n_{av}) \cdot (2\ n_{av}) \right] t_c$$

or

$$T_{sc_1} = (8\ n_{av}^2 + 4\ n_{av})\ s\ t_c \quad . \tag{5.43}$$

(vii) Computation of vector $\mathbf{x}$ from $\mathbf{x} = \zeta_{s+1} - \sum_{i=1}^{s} \mathbf{h}_i$.

A slice of elements of width $\left\lceil \frac{q}{s} \right\rceil$ of vector $\mathbf{x}$ is computed on a separate processor. There are $2\ n_{av}$ non-zero elements in one $\mathbf{h}_i$. The column indices of the non-zero elements in all the $\mathbf{h}_i$ vectors will be different from each other corresponding to those random links in the network whose two ends do not connect the end nodes of two disjointed chains. However, for a random link which connects the end nodes of two disjointed chains, there will be two $\mathbf{h}_i$ vectors with identical non-zero element column indices. Therefore, it takes one subtraction to compute an element of $\mathbf{x}$ for some elements of $\mathbf{x}$ and two subtractions for the other elements of $\mathbf{x}$. This load unbalancing can be minimized by numbering such links with a difference of $\left\lceil \frac{q}{s} \right\rceil$ integers rather than consecutively.

Therefore, taking 1.5 subtractions as the time to compute an element of **x** and taking 100% overhead for sparse computations, the time to compute vector **x** is given by

$$T_{sf_5} = 2 \times \left\lceil \frac{q}{s} \right\rceil \times 1.5\ t_f = 3 \left\lceil \frac{q}{s} \right\rceil t_f \quad . \tag{5.44}$$

(viii) Computation of matrix **Y** from $\mathbf{Y} = \mathbf{V} - \sum_{i=1}^{s} \mathbf{G}_i$.

There are $2\ n_{av}$ non-zero elements in matrix $\mathbf{G}_i$. Here again, the same problem of unequal load distribution in computing elements of **Y** occurs as was the case of computing $\mathbf{h}_i$. Therefore, the parallel computation time for computing **Y** is given by

$$T_{sf_6} = 2 \times 2\ n_{av} \left\lceil \frac{q}{s} \right\rceil \times 1.5\ t_f$$

or

$$T_{sf_6} = 6\ n_{av} \left\lceil \frac{q}{s} \right\rceil t_f \quad . \tag{5.45}$$

(vii) Synchronize all the processors

$$T_{ssync_2} = t_s \quad . \tag{5.46}$$

(vii) Computation of vector $\Delta\ \boldsymbol{\omega}_{s+1}$ from $\mathbf{Y}\ \Delta\ \boldsymbol{\omega}_{s+1} = \mathbf{x}$.

This computation is carried out by parallel Gauss-Jordan elimination as described in section 5.2.1. The computation time is given by (5.2). Therefore,

$$T_{sf_6} = \left\lceil \frac{q}{s} \right\rceil (q^2 + 2q + 1)\ t_f \tag{5.47}$$

$$T_{sc_2} = \left\lceil \frac{q^2}{2} + \frac{3}{2} q \right\rceil t_c \tag{5.48}$$

$$T_{ssync_3} = q\ t_s \quad . \tag{5.49}$$

(viii) Synchronize the processors

$$T_{ssync_4} = t_s \quad . \tag{5.50}$$

(ix) Broadcast solution subvector $\Delta\ \omega_{s+1}$ to all the processors.

$$T_{sc_3} = q\ t_c \quad . \tag{5.51}$$

(x) Computation of $\Delta\ \omega_i$ from $\Delta\ \omega_i = \mathbf{c}_i - \mathbf{E}_i\ \Delta\ \omega_{s+1}$, $i = 1, 2, ..., s$.

Each subvector $\Delta\ \omega_i$ is computed on a separate processor i, $i = 1, 2, ..., s$. There are $2\ n_{av}$ non-zero columns in matrix $\mathbf{E}_i$. If p_m is the length of the longest chain, then, taking 100% overhead for sparse computations,

$$T_{sf_7} = 2(2\ n_{av} + 2\ n_{av})\ p_m\ t_f$$

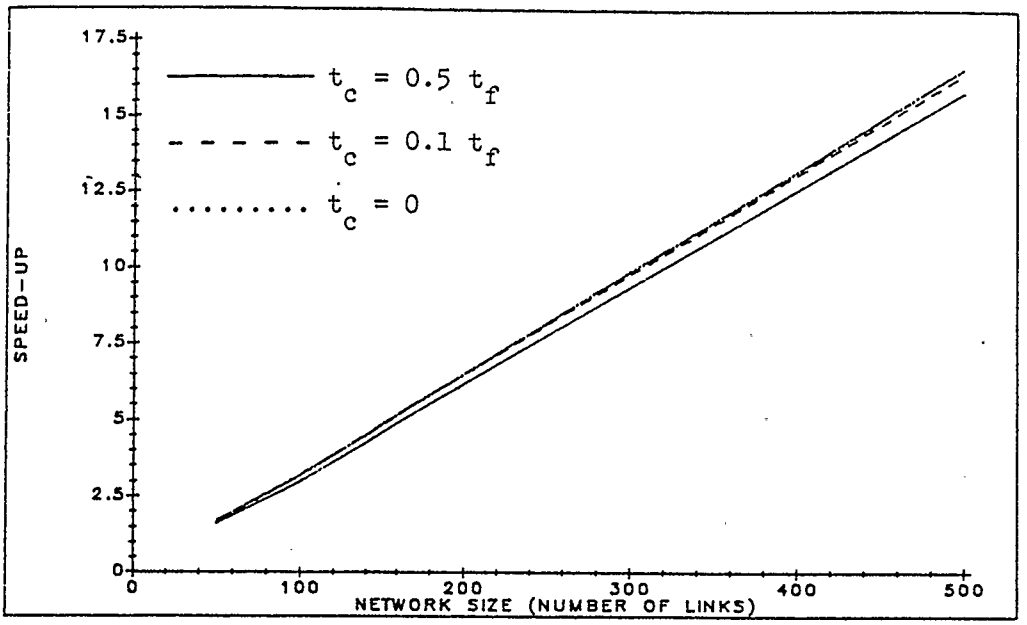
or

$$T_{sf_7} = 8\ n_{av}\ p_m\ t_f \quad . \tag{5.52}$$

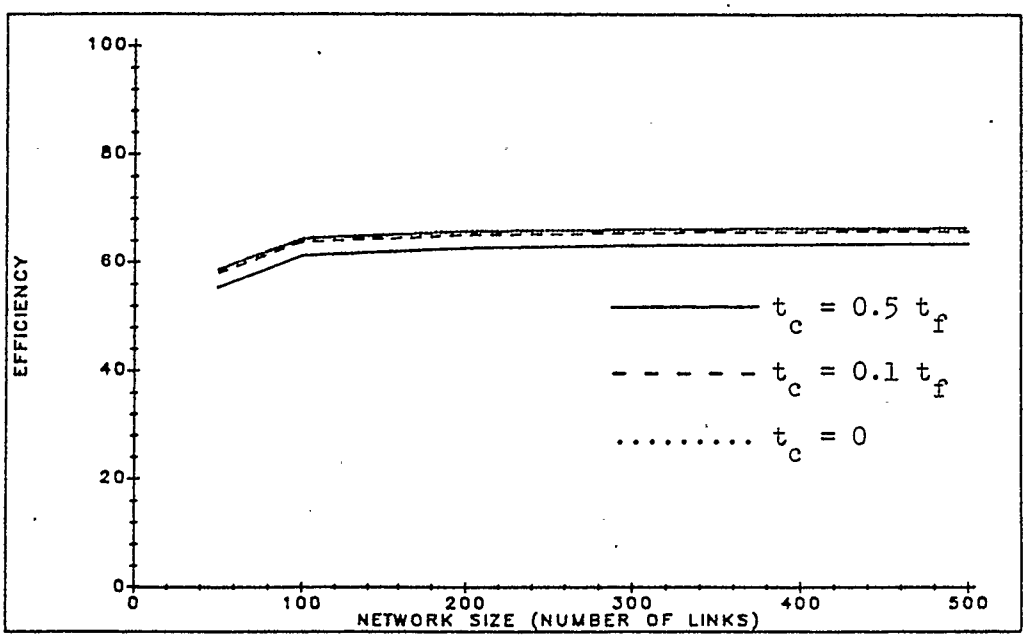
The total parallel processing time is the sum total of times given by (5.38) through (5.51). Hence,

$$\begin{aligned} T_s(\text{matrix}) \approx & \left[\left\lceil \frac{q}{s} \right\rceil (q^2 + 2q + 6\ n_{av}) + (13\ n_{av} + 8)\ p_m + 2(n_{\max} + n_{av})^2 \right] t_f \\ & + \left[\frac{q^2}{2} + \frac{5}{2}\ q + (8\ n_{av}^2 + 4\ n_{av})\ s \right] t_c \\ & + (q + 3)\ t_s \quad . \end{aligned} \tag{5.53}$$

The serial computation time for direct method II on a single processor is given by (3.36). The speed-ups and efficiencies obtained are plotted in Fig. 5.14. As can be seen from these plots, the speed-ups and efficiencies are not very sensitive with respect to the data



(a) Speed-up versus network size



(b) Efficiency versus network size

Figure 5.14 Speed-up and efficiency of parallel computation for solving $\mathbf{A}\ \Delta\ \mathbf{W}^{n+1} = \mathbf{z}$ on s processors by direct parallel method II.

communication time t_c. Also, the speed-ups and efficiencies obtained are higher than the corresponding values for direct parallel method I (Fig. 5.6), particularly for the smaller size networks. Total synchronization time is also less for this parallel method as compared to direct parallel method I.

5.2.2.2. Number of Processors equal to number of random links in the network

Parallel computation on q processors is very similar to the computation on s processors except that the submatrices $\mathbf{D}_i$, $\mathbf{B}_i$, $\mathbf{R}_i$, $\mathbf{V}$ of matrix $\mathbf{A}$ and subvectors ζ_i and ζ_{s+1} of vector $\mathbf{z}$ are to be transferred to the slave processors. $\mathbf{D}_i$, $\mathbf{B}_i$, $\mathbf{R}_i$, ζ_i are transferred to processor i, where $i = 1, 2, ..., s$ and the submatrix $\mathbf{V}$ and subvectors ζ_{s+1} are partitioned row-wise with one row per processor. This is so because the elements of matrix $\mathbf{A}$ and vector $\mathbf{z}$ are partitioned and computed on q processors in such a way that each processor computes an equal number of elements of $\mathbf{A}$ to ensure load balancing among the processors. This partitioning requirement is not the same as for solving the matrix equation. Another difference is that the matrix equation $\mathbf{Y}\,\Delta\,\boldsymbol{\omega}^{n+1} = \mathbf{x}$ is solved on q processors instead of s processors by assigning one row of $\mathbf{Y}$ to one processor.

(i) Transfer of $\mathbf{D}_i$, $\mathbf{B}_i$, $\mathbf{R}_i$, ζ_i and ζ_{s+1} to s processors.

Total number of $a_{k\nu}$ elements in matrix $\mathbf{A}$ where $\nu \neq k$ has been calculated in section 3.2.2 of Chapter 3 as $2K + (2\,d_{av} - 4)\,(q + s)$ where d_{av} is the average degree of a non-chained node in the network. Therefore, total number of $a_{k\nu}$ elements of A including $\nu = k$ is given by

$$3K + (2\,d_{av} - 4)\,(q + s)$$

Since the elements of $\mathbf{A}$ are stored as sparse, the row and column indices of $\mathbf{A}$ also have to

be transferred along with their values. Taking 100% overhead for index transfers,

Total data transfer time for A and z is given by,

$$T_{qc_1} = [2\{3K + (2\ d_{av} - 4)\ (q + s)\} + K]\ t_c$$

or

$$T_{qc_1} = [7K + (4\ d_{av} - 4)\ (q + s)]\ t_c \quad \cdot \tag{5.54}$$

(ii) Compute $\mathbf{c}_i$, $\mathbf{E}_i$, $\mathbf{h}_i$ and $\mathbf{G}_i$ on processor i, where $i = 1, 2, ..., s$.

These computations are done on s of the q processors. Therefore, the computation time for this part is the same as for the case of s processors given in section 5.2.2.1

$$T_{qf_1} = [(5\ n_{av} + 8)\ p_m + 2(n_{\max} + n_{av})^2 + 2(n_{\max} + n_{av}) - 3\ n_{av} - 7]\ t_f \quad \cdot \tag{5.55}$$

(iii) Synchronize all the processors

$$\text{Synchronization time } T_{qsync_1} = t_s \quad \cdot \tag{5.56}$$

(iv) Transfer a row j of $\mathbf{h}_i$ and $\mathbf{G}_i$ to processor j, $i = 1, 2, ..., q$ and $i = 1, 2, ..., s$.

This communication time is the same as given in (5.43) for s processors. Therefore,

$$T_{qc_2} = (8\ n_{av}^2 + 4\ n_{av})\ s\ t_c \quad \cdot \tag{5.57}$$

Now $\mathbf{h}_i$ and $\mathbf{G}_i$ are partitioned row-wise with one row of each $\mathbf{h}_i$, $\mathbf{G}_i$ per processor.

(v) Computation of vector x from $\mathbf{x} = \zeta_{s+1} - \sum_{i=1}^{s} \mathbf{h}_i$

Each element of x is computed on a separate processor so that q elements of x are computed concurrently on q processors. Some elements of x require one subtraction and some require two subtractions as discussed in section 5.2.2.1. Assuming 100% overhead

for sparse computations.

$$\text{parallel computation time } T_{qf_2} = 4\ t_f \quad . \tag{5.58}$$

(vi) Computation of matrix $\mathbf{Y}$ from $\mathbf{Y} = \mathbf{V} - \sum_{i=1}^{s} \mathbf{G}_i$.

$$\text{Parallel computation time } T_{qf_3} = 2 \times 2 \times 2\ n_{av}\ t_f$$

or

$$T_{qf_3} = 8\ n_{av}\ t_f \quad . \tag{5.59}$$

(vii) Synchronize all the processors

$$T_{qsync_2} = t_s \quad . \tag{5.60}$$

(viii) Computation of $\Delta\ \omega_{s+1}$ vector from $\mathbf{Y}\ \Delta\ \omega_{s+1} = \mathbf{x}$ by parallel Gauss-Jordan elimination on q processors.

Parallel processing time for solving a $q \times q$ matrix equation has been derived in section 5.2 and, for q processors, it is given by

$$T_{qf_4} = (q^2 + 2q + 1)\ t_f \tag{5.61}$$

$$T_{qc_3} = \left[\frac{q^2}{2} + \frac{3}{2}\ q\right] t_c \tag{5.62}$$

$$T_{qsync_3} = q\ t_s \quad . \tag{5.63}$$

Now each element of $\Delta\ \omega_{s+1}$ is in a different processor.

(ix) Synchronize all the processors

$$t_{qsync_4} = t_s \quad . \tag{5.64}$$

(x) Broadcast the solution subvector $\Delta\ \omega_{s+1}$ to all the processors

$$T_{qc_4} = q\ t_c \quad . \tag{5.65}$$

(xi) Computation of $\Delta\ \omega_i$ from $\Delta\ \omega_i = \mathbf{c}_i - \Delta\ \omega_{s+1}$, $i = 1, 2, \ldots, s$.

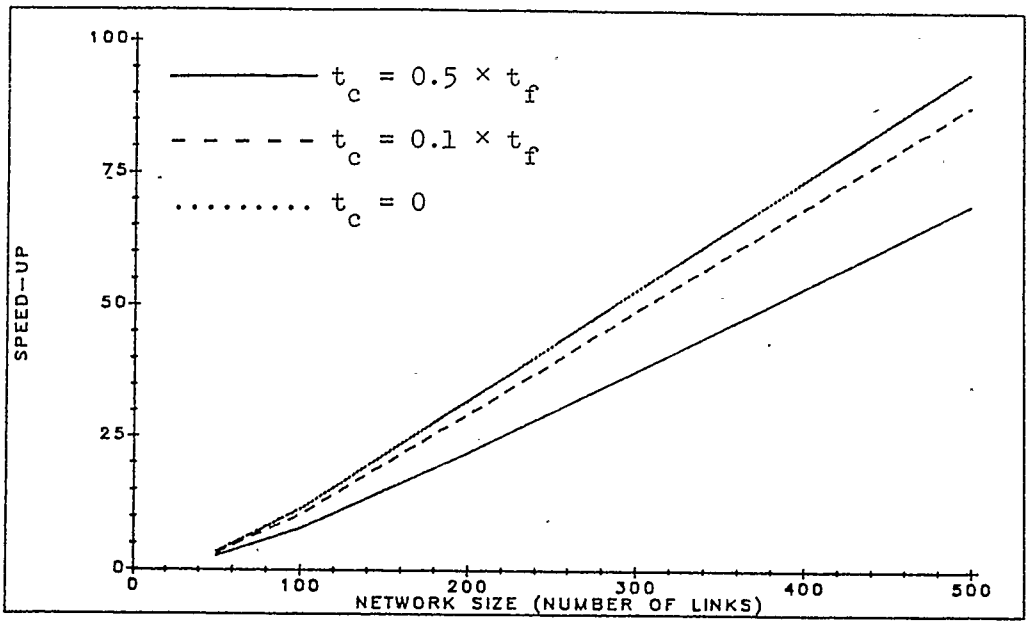
Each $\Delta\ \omega_i$ is computed on a separate processor. Therefore, computation time is the same as given by (5.52). Therefore,

$$T_{qf_5} = 8\ n_{av}\ p_m\ t_f \quad . \tag{5.66}$$

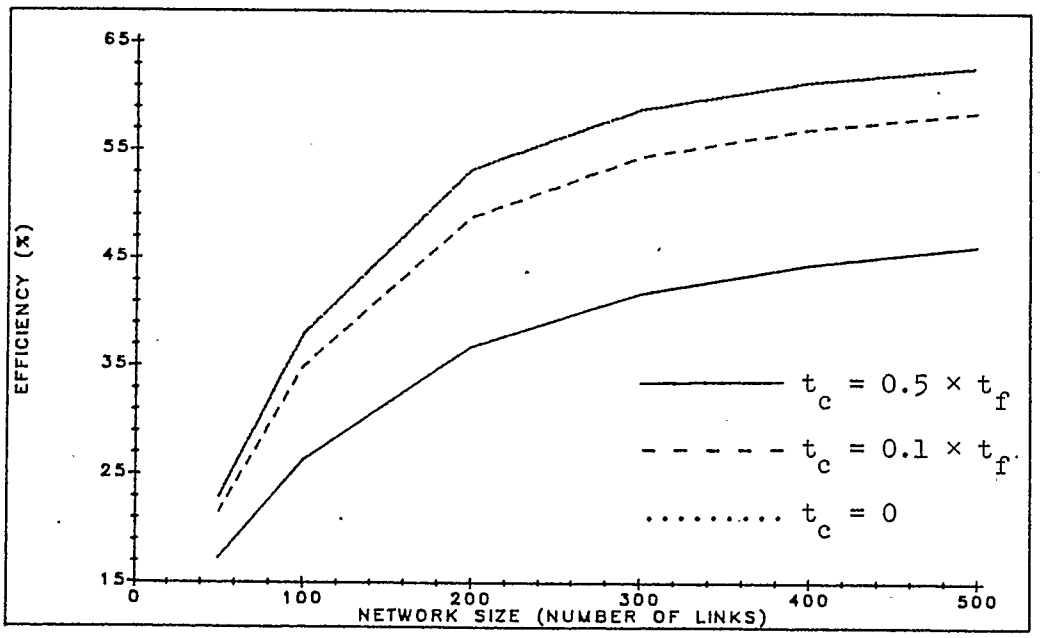
The total processing time is the sum total of times given by (5.54) through (5.66). Therefore,

$$\begin{aligned} T_q(\text{matrix}) \approx & \left[q^2 + (13\ n_{av} + 8)\ p_m + 2q + 2(n_{\max} + n_{av})^2\right] t_f \\ & + \left[\frac{q^2}{2} + 7K + \frac{5}{2}\ q + (4\ d_{av} - 4)\ (q + s) + (8\ n_{av}^2 + 4\ n_{av})s\right] t_c \\ & + (q + 3)\ t_s \quad . \end{aligned} \tag{5.67}$$

The speed-ups and efficiencies obtained by this method are plotted in Fig. 5.15. A comparison with the results of direct parallel method I (Fig. 5.7), shows that these speed-ups and efficiencies are higher than the direct parallel method I for large-size networks but are more or less the same for smaller networks.



(a) Speed-up versus network size



(b) Efficiency versus network size

Figure 5.15 Speed-up and efficiency of parallel computation for solving $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ on q processors by direct method II.

5.2.2.3. Number of processors equal to the number of chains in the network and a two-dimensional mesh-connected array of q x q processors

The pseudo-code of the direct parallel method II for this case is given in Fig. 5.16. The parallel processing time is estimated as follows:

(i) Computation of $\mathbf{c}_i$, $\mathbf{E}_i$, $\mathbf{h}_i$ and $\mathbf{G}_i$ on processor i, $i = 1, 2, ..., s$

Since these computations are done on the s slave processors, the computation time is the same as given by (5.55). Therefore,

$$T_{s+arrf_1} = \left[(5\ n_{av} + 8)\ p_m + 2(n_{\max} + n_{av})^2 + 2(n_{\max} + n_{av}) - 3\ n_{av} - 7\right] t_f \quad (5.68)$$

(ii) Synchronize all the s processors

$$T_{s+arrsync_1} = t_s \quad (5.69)$$

(iii) Transfer vectors ζ_{s+1}, $\mathbf{h}_i$, $i = 1, 2, ..., s$ from s processors to the array's first column of 2-D interface memory.

These vectors are stored in the interface memory as non-zero data element values together with their position in the vector (first the column index of the non-zero element and then its value). Therefore, data communication time is given by

$$T_{s+arrc_1} = \left[2q + \sum_{i=1}^{s} 4\ n_{av}\right] t_c$$

or

$$T_{s+arrc_1} = \left[2q + 4\ n_{av}\ s\right] t_c \quad (5.70)$$

{$\mathbf{D}_i$, $\mathbf{B}_i$, $\mathbf{R}_i$, ζ_i are computed on processor i, $i = i, 2, \cdots, s$. Therefore no data transfer is required}.

for $i = 1$ to s pardo {perform the following computations concurrently on s slave processors}

$\mathbf{c}_i = \mathbf{D}_i^{-1}\, \zeta_i$

$\mathbf{E}_i = \mathbf{D}_i^{-1}\, \mathbf{R}_i$

$\mathbf{h}_i = \mathbf{B}_i\, \mathbf{c}_i$

$\mathbf{G}_i = \mathbf{B}_i\, \mathbf{E}_i$

end pardo

synchronize all the processors

transfer vectors ζ_{s+1} , $\mathbf{h}_i$, $i = 1, 2, \cdots, s$ from s slave processors to the first column of the processor array's 2-D interface memory.

read the ζ_{s+1} and $\mathbf{h}_i$ values from the interface memory into the q processors of the first column of the processor array.

Compute vector $\mathbf{x} = \zeta_{s+1} - \sum_{i=1}^{s} \mathbf{h}_i$ in the array

transfer the vector $\mathbf{x}$ from processor array to the left-hand 1-D interface memory.

Synchronize array with the master processor.

Transfer sparse matrices $\mathbf{V}$, $\mathbf{G}_i$, $i = 1, 2, ..., s$ from slave processors to the 2-D interface memory of the processor array.

read $\mathbf{V}$ and $\mathbf{G}_i$ values in to the processor array.

Compute matrix $\mathbf{Y} = \mathbf{V} - \sum_{i=1}^{s} \mathbf{G}_i$ on the processor array.

Solve the matrix equation $\mathbf{Y}\, \Delta\omega_{s+1} = \mathbf{x}$ for $\Delta\omega_{s+1}$ on the array by the Fawcett's algorithm

synchronize array with the master processor.

read the solution subvector $\Delta\omega_{s+1}$ from the left-hand interface memory and broadcast to the s slave processors.

for $i = 1$ to s pardo {Perform the following computations concurrently on s slave processors}

$\Delta\omega_i = \mathbf{c}_i - \mathbf{E}_i\, \Delta\omega_{s+1}$

end pardo

Figure 5.16 Pseudo-code for solving $\mathbf{A}\, \Delta\, \mathbf{W}^{n+1} = \mathbf{z}$ on s processors plus a $q \times q$ processor array by direct method II.

(iv) Read the ζ_{s+1} and $\mathbf{h}_i$ values from the interface memory to the processors in the first column of the array.

First processor in the first column of the array reads the column index of a non-zero element of the vector and then its value. If the column index is not unity, it passes along the column index and the element value to the processor down below and so on. The same procedure is followed by the lower processors. They read these values from the upper neighbour processor. In this way, the jth non-zero element of any vector $\mathbf{h}_i$ is stored in the jth processor. Therefore,

$$T_{s+arrc_2} = (2q + 2 \times 2\, n_{av}\, s + q)\, t_c'$$

or

$$T_{s+arrc_2} = (3q + 4\, n_{av}\, s)\, t_c' \quad . \tag{5.71}$$

The additional q data transfers in the above equation are for the worst-case when the last element read from the memory is for the bottom-most processor. Here t_c' is the time for one read operation by a processor in the array.

(v) Compute vector **x** in the first column processors of the array

The slowest processor has to perform two subtractions to compute an element of **x** as discussed in section 5.2.2.2. Therefore,

$$T_{s+arrf_2} = 4\, t_f' \tag{5.72}$$

where t_f' is the one floating-point operation time of a processor in the array.

(vi) Transfer **x** to the left-hand interface memory

$$T_{s+arrayc_3} = t_c' \quad . \tag{5.73}$$

(vii) Synchronize array with the master processor

$$T_{s+arraysync_2} = t_s \quad (5.74)$$

(viii) Transfer matrices $\mathbf{V}$ and $\mathbf{G}_i$, $i = 1, 2, ..., s$ from the s slave processors to the array's 2-D interface memory (non-zero column elements of matrices stored in the corresponding columns of the memory along with their row indices.

If d_{av} is the average degree of a non-chained node in the network, then a random link gives rise to $[1 + 2(d_{av} - 1)]$ non-zero elements. Hence, number of non-zero elements in the submatrix $[\mathbf{B}_1\ \mathbf{B}_2\ \cdots\ \mathbf{B}_s\ \mathbf{V}]$ is

$$[2(d_{av} - 1) + 1]\ q$$

$$= (2\ d_{av} - 1)\ q$$

There are n_{av} non-zero elements per first and last column of B_i. Therefore, number of non-zero elements in matrix $\mathbf{V}$ is $[(2\ d_{av} - 1)q - 2\ n_{av}\ s]$.

Since the non-zero elements of $\mathbf{V}$ are stored along with their addresses, data transfer time for storing $\mathbf{V}$ in the interface memory is given by

$$T_{s+arrayc_4} = 2[(2\ d_{av} - 1)q - 2\ n_{av}\ s]\ t_c$$

or

$$T_{s+arrayc_4} = [(4\ d_{av} - 2)q - 4\ n_{av}\ s]\ t_c \quad (5.75)$$

Data transfer time for transferring $\mathbf{G}_i$, $i = 1, 2, ..., s$ to the interface memory,

$$T_{s+arrayc_5} = \sum_{i=1}^{s} 2 \times 2\ n_{av} \times 2\ n_{av}\ t_c$$

$$or \qquad T_{s+arrayc_5} = 8\, n_{av}^2\, s\, t_c \cdot \qquad (5.76)$$

(ix) Read **V** and $\mathbf{G}_i$ values into the array

If $d_{\max}$ is the maximum degree of a non-chained node in the network, then the maximum number of non-zero elements in any column of **V** is approximately equal to $(d_{\max} - 1) + (d_{av} - 1) + 1 = (d_{\max} + d_{av} - 1) \approx (d_{\max} + d_{av})$. Hence, data transfer time for transferring **V** and $\mathbf{G}_i$ submatrices from interface memory to the processor array is given by

$$T_{s+arrayc_6} = 2\left[(d_{\max} + d_{av} + \sum_{i=1}^{s} 2\, n_{av} + q\right] t_c'$$

or

$$T_{s+arrayc_6} = \left[2(d_{\max} + d_{av}) + 4\, n_{av}\, s + 2q\right] t_c' \cdot \qquad (5.77)$$

(x) Compute matrix $\mathbf{Y} = \mathbf{V} - \sum_{i=1}^{s} \mathbf{G}_i$ on the processor array.

The q^2 processors compute the q^2 elements of **Y**. The slowest processor has to perform two subtractions corresponding to two $\mathbf{G}_i$ with identical non-zero element positions which in turn corresponds to a random link connecting the end nodes of two disjointed chains. Therefore,

$$T_{s+arrayf_3} = 4\, t_f' \cdot \qquad (5.78)$$

(xi) Solve the matrix equation $\mathbf{Y}\, \Delta\, \boldsymbol{\omega}_{s+1} = \mathbf{x}$ on the processor array.

The parallel computation time for solving the matrix equation has been obtained in section 5.1.2.3 and is given by (5.27). Hence,

$$T_{s+arrayf_4} = (7q - 5)\, t_{mac}$$

where t_{mac} is the processing time to carry out one multiply and accumulate operation plus one data communication to the nearest neighbour, i.e. $t_{mac} = (2\, t_f' + t_c')$.

Hence, $$T_{s+arrayf_4} = (7q - 5)\,(2\, t_f' + t_c') \quad (5.79)$$

(xii) Synchronize the array with the master processor

$$T_{s+arraysync_3} = t_s \quad (5.80)$$

(xiii) Read the solution vector $\Delta\, \omega_{s+1}$ from the left interface memory and broadcast to the s slave processors. Thus

$$T_{s+arrayc_7} = q\ t_c \quad (5.81)$$

(xiv) Compute $\Delta\, \omega_i$, $i = 1, 2, ..., s$ on s slave processors.

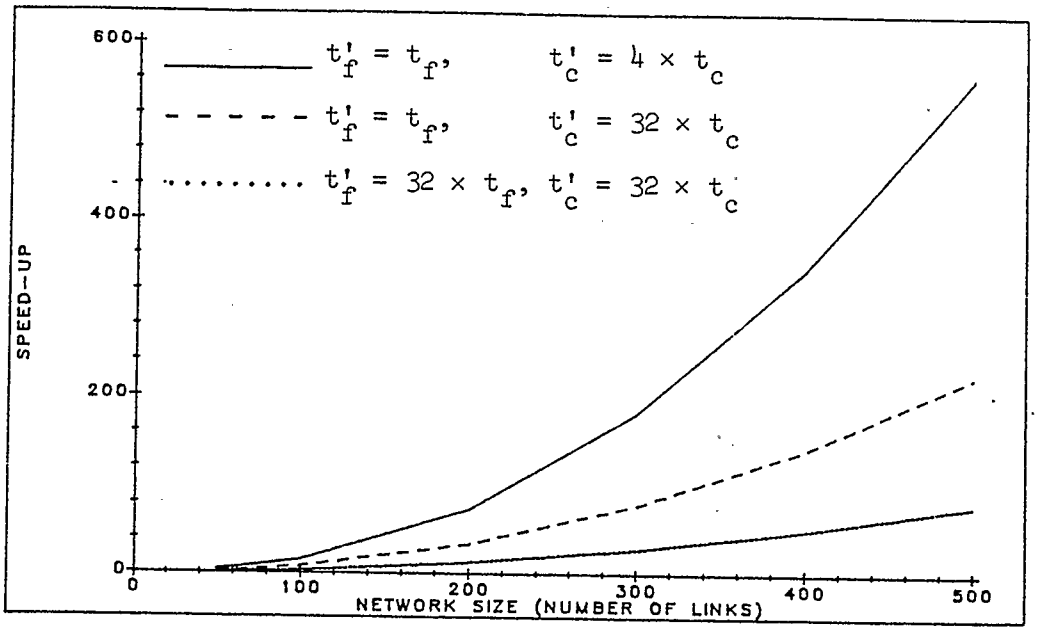
This computation time is the same as given in (5.52). Hence,

$$T_{s+arrayf_5} = 8\ n_{av}\ p_m\ t_f \quad (5.82)$$

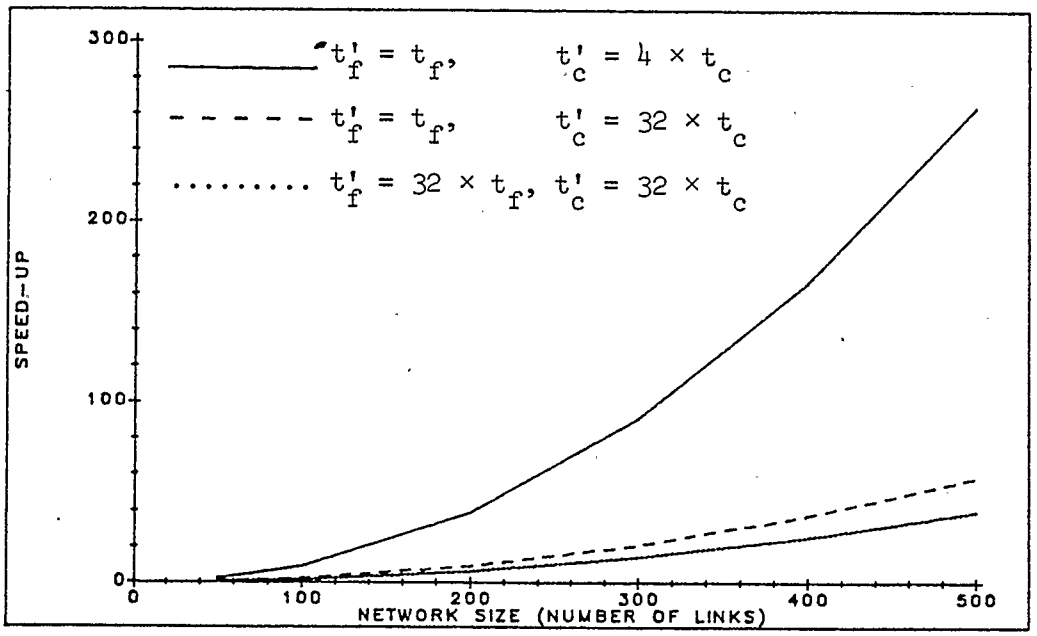
The total processing time is the sum total of above times given by (5.68) through (5.82). Therefore,

$$\begin{aligned} T_{s+array}(\text{matrix}) \approx\ & \left[(13\ n_{av} + 8)\, p_m + 2(n_{\max} + n_{av})^2\right] t_f \\ & + 14q\ t_f' + \left[(4\ d_{av} + 1)q + 8\ n_{av}^2\ s\right] t_c \\ & + (12q + 8\ n_{av}\ s)\ t_c' + 3\ t_s \end{aligned} \quad (5.83)$$

The speed-ups and efficiencies obtained for different values of t_c, t_c', t_f' are plotted in Fig. 5.17. As can be seen, the speed-up values obtained are higher than those obtained



(a) $t_c = 0.1 \times t_f$



(b) $t_c = 0.5 \times t_f$

Figure 5.17 Speed-up of parallel processing for solving $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ on s processors and a $q \times q$ processor array by direct method II.

for direct parallel method I as given by (5.37). Therefore, in this case, direct parallel method II is to be preferred over the direct parallel method I. As was noted for direct parallel method I, processor arrays can be used to advantage only when, (i) $t_f' \approx t_f$, (ii) $t_c' \approx 4\ t_c$, (iii) $t_c \approx 0.1\ t_f$, (iv) large-size network. A summary of computation, communication, and synchronization times for solving the matrix equation by direct methods I and II is given in Tables 5.1 and 5.2 respectively.

A comparison of total processing times for solving the matrix equation by direct method I and direct method II is shown in Fig. 5.18. As can be seen from this plot, the direct method II is somewhat faster than direct method I for the case of one, s, and q processors. It is significantly faster (a factor of about 2 to 5 depending upon the network size) for the case of s processors and a $q \times q$ processor array.

5.3. Parallel Processing of the Remaining Computations

In this section, as estimation of parallel processing time for the remaining computations is obtained. Two cases for the number of processors used are considered: (1) number of processors equal to s, (2) number of processors equal to q. For each case, the processing time will be slightly different depending on whether the matrix equation $\mathbf{A}\ \Delta\ \boldsymbol{\omega} = \mathbf{z}$ is being solved by direct parallel method I or direct parallel method II. This is due to the different partitioning of the matrix **A** elements for the two parallel methods and hence different data communication times involved.

5.3.1. Number of processors equal to s

In this case, the computations are partitioned in such a way that computations for the links and nodes of a chain are done on a separate processor. Thus the s chains are parti-

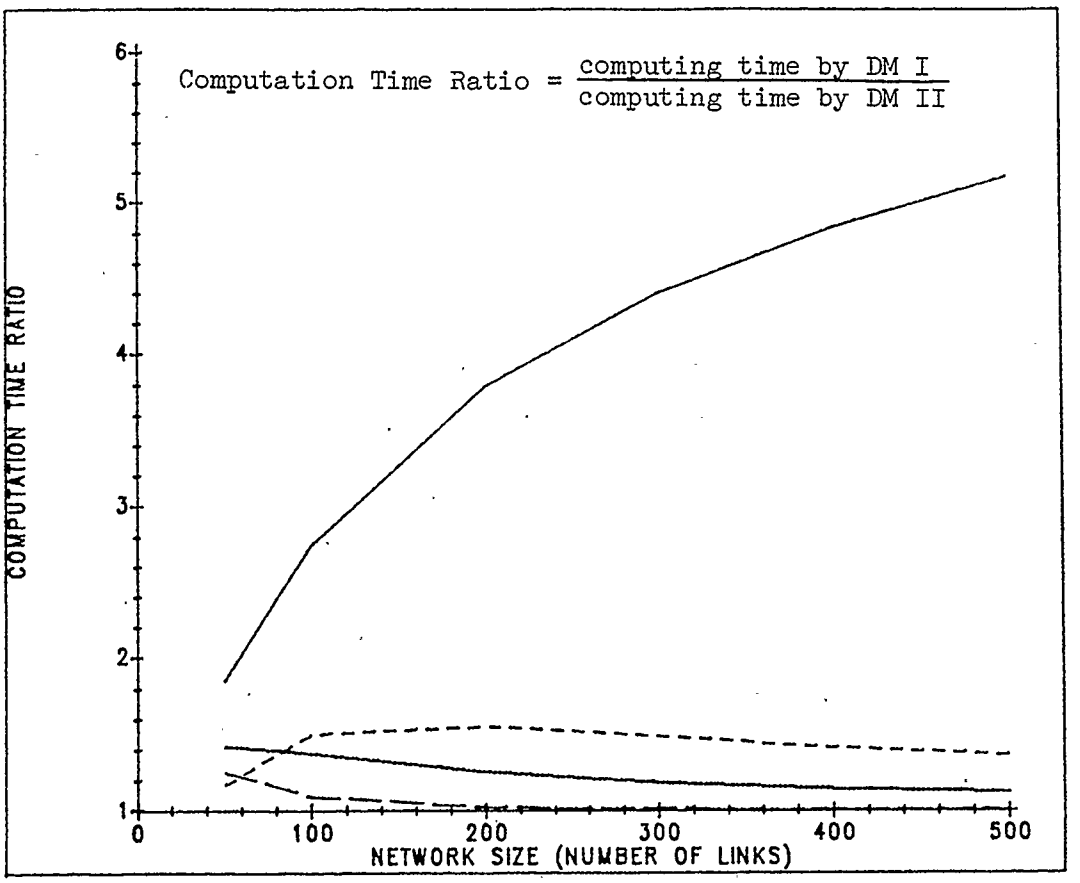


Figure 5.18 A comparison of computation times for solving $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ by direct method I and direct method II ($t_c = 0.1 \times t_f$).

tioned on s processors. This reduces data communication time. The computations for the random links and nodes are divided equally among the s slave processors.

5.3.1.1. Matrix equation being solved by direct parallel method I

The pseudo-code for parallel processing is given in Fig. 5.19. The computation time is obtained as follows:

1. computation of ω_j^{n+1}, $j = 1, 2, ..., s+1$.

If p_m is the length of the longest chain in the network, then

$$T_{sf_1} = \left[p_m + \left\lceil \frac{q}{s} \right\rceil\right] t_f \quad . \tag{5.84}$$

2. Computation of ΔU_i, $i = 1, 2, ..., N$ from (2.55)

If d_{av} is the average degree of a non-chained node of degree greater than two including the two end nodes of a chain, then computation time for the chained nodes is $[5(p_m - 1) + 2(d_{av} + 1)]\, t_f$.

Now, total number of non-chained nodes N_{nc} is $N - \sum_{i=1}^{s} (p_i + 1)$ or $N_{nc} = (N - K + q - s)$.

Computation time for the non–chained nodes is $(2\, d_{av} + 1) \left\lceil \frac{N_{nc}}{s} \right\rceil t_f$.

Therefore, total computation time for all ΔU_i is

$$\left[5(p_m - 1) + 2(2\, d_{av} + 1) + (2\, d_{av} + 1) \left\lceil \frac{N_{nc}}{s} \right\rceil \right] t_f$$

{Each flow subvector $\Delta\omega_j^{n+1}$ is already computed and stored in processor j , $j = 1, 2, ..., s$ and the $\Delta\omega_{s+1}^{n+1}$ subvector is partitioned row-wise among s processors (see section 5.2.1.1 and pseudocode of Fig. 5.5)}

for $j = 1$ to s pardo {compute subsector ω_j^{n+1} on processor j}

$$\omega_j^{n+1} = \omega_j^n + \Delta\omega_j^{n+1}$$

end pardo

Compute q elements of flow subvector ω_{s+1}^{n+1} on s processors

Compute ΔU_i^{n+1} , ΔM_i^{n+1} , U_i^{n+1} , M_i^{n+1}, u_i, v_i, P_i, H_i, $\frac{\partial P_i}{\partial U_i}$, $\frac{\partial P_i}{\partial M_i}$ for the chained nodes (each chain on a separate processor) and for the random nodes (equally divided among s slave processors)

synchronize the processors

partially broadcast flow rates ω_{s+1}^{n+1} for the random links only to processors requiring it.

partially broadcast v_i, P_i, H_i, $\frac{\partial P_i}{\partial U_i}$, $\frac{\partial P_i}{\partial M_i}$ for the random nodes only

Compute f_{K+i}, f_{K+N+i} first for the chained links (one chain per processor) and then for the random links (equally divided among s processors)

Compute f_k, $\frac{\partial f_k}{\partial U_i}$, $\frac{\partial f_k}{\partial U_j}$, $\frac{\partial f_k}{\partial M_i}$, $\frac{\partial f_k}{\partial M_j}$ for the chained links and random links

synchronize the processors

Figure 5.19 Pseudo-code for the parallel processing of the remaining computations on s slave processors (matrix equation to be solved by direct parallel method I)

partially broadcast f_{K+i}, f_{K+N+i} for the random nodes only

for $j = 1$ to s pardo {compute concurrently on s processors}

compute ζ_j

compute $\mathbf{D}_j$

compute $\mathbf{B}_j$

compute $\mathbf{R}_j$

compute a slice j of $\left\lceil \frac{q}{s} \right\rceil$ rows of matrix $\mathbf{V}$ and subvector ζ_{s+1} on processor j

end pardo

synchronize the processors

Transfer ζ_j, $\mathbf{D}_j$, $\mathbf{R}_j$, $j = 1, 2, ..., s$ to the master

store a slice of rows of width $\left\lceil \frac{q}{s} \right\rceil$ of $[\mathbf{B}_1\ \mathbf{B}_2\ \cdots\ \mathbf{B}_s]$ in each processor

{Now all $\mathbf{D}_i, \mathbf{R}_i, \zeta_i$, $i = 1, 2, \cdots, s$ are stored in the master and the matrix $[\mathbf{B}_1\ \mathbf{B}_2\ \cdots\ \mathbf{B}_s\ \ \mathbf{V}\ \zeta_{s+1}]$ is partitioned row-wise equally among s slave processors.}

Figure 5.19 Continued

or

$$T_{sf_2} \approx \left[5\, p_m + 4\, d_{av} + (2\, d_{av} + 1)\left\lceil \frac{N_{nc}}{s} \right\rceil\right] t_f \quad . \tag{5.85}$$

3. Computation of ΔM_i, $i = 1, 2, ..., N$ from (2.56).

It is similar to the computation of ΔU_i. Therefore,

$$T_{sf_3} = \left[2(p_m - 1) + 2\, d_{av} + d_{av} \left\lceil \frac{N_{nc}}{s} \right\rceil\right] t_f$$

or

$$T_{sf_3} \approx \left[2\, p_m + d_{av} \left\{\left\lceil \frac{N_{nc}}{s} \right\rceil + 2\right\}\right] t_f \quad . \tag{5.86}$$

4. Computation of U_i^{n+1}, M_i^{n+1}, u_i, v_i, $i = 1, 2, ..., N$.

$$u_i = V_i/M_i$$

$$v_i = \text{Volume (i)}/M_i$$

This time is given as

$$T_{sf_4} \approx 4\left[p_m + \left\lceil \frac{N_{nc}}{s} \right\rceil\right] t_f \quad . \tag{5.87}$$

5. Computation of P_i, $i = 1, 2, ..., N$.

If $T_p\ t_f$ is the time to compute one value of P_i, then

$$T_{sf_5} = \left[(p_m + 1) + \left\lceil \frac{N_{nc}}{s} \right\rceil\right] T_p\ t_f \quad . \tag{5.88}$$

6. Computation of H_i, $i = 1, 2, ..., N$.

$H_i = u_i + P_i\ v_i$

Therefore,

$$T_{sf_6} = 2\left[p_m + 1 + \left\lceil \frac{N_{nc}}{s} \right\rceil\right] t_f \quad . \tag{5.89}$$

7. Computation of $\frac{\partial P_i}{\partial U_i}$, $\frac{\partial P_i}{\partial M_i}$, $i = 1, 2, ..., N$.

Let

$T_{dpu}\ t_f =$ time to compute one value of $\frac{\partial P_i}{\partial U_i}$

$T_{dpm}\ t_f =$ time to compute one value of $\frac{\partial P_i}{\partial M_i}$

Then,

$$T_{sf_7} = \left[p_m + 1 + \left\lceil \frac{N_{nc}}{s} \right\rceil\right] \left(T_{dpu} + T_{dpm}\right) t_f \quad . \tag{5.90}$$

8. Synchronize processors.

The synchronization time T_{ssync_1} is given by

$$T_{ssync_1} = t_s \quad . \tag{5.91}$$

9. Partially broadcast flow rates ω_{s+1}^{n+1} for the random links only and v_i, P_i, H_i, $\frac{\partial P_i}{\partial U_i}$, $\frac{\partial P_i}{\partial M_i}$ values for the random nodes only.

The data communication time T_{sc_1} is given by

$$T_{sc_1} = \Big[(q + 2s) + 5(2s) + 5(N - K + q - s)\Big]\, t_c$$

or

$$T_{sc_1} = \Big[5N - 5K + 6q + 7s\Big]\, t_c \quad (5.92)$$

10. Computation of f_{K+i} from (2.4), $i = 1, 2, \ldots, N$.

Computation time for chained nodes is $[4(p_m - 2) + 2(d_{av} + d_{av})]\, t_f$.

Computation time for other nodes is $\left\lceil \frac{N_{nc}}{s} \right\rceil (d_{av} + d_{av})\, t_f$.

Therefore,

$$T_{sf_8} = \left[4\, p_m + 4\, d_{av} - 8 + 2\, d_{av} \left\lceil \frac{N_{nc}}{s} \right\rceil\right] t_f \quad (5.93)$$

11. Computation of f_{K+N+i}, $i = 1, 2, \ldots, N$ from (2.5).

$$T_{sf_9} = \left[p_m + 2\, d_{av} - 4 + (d_{av} - 1) \left\lceil \frac{N_{nc}}{s} \right\rceil\right] t_f \quad (5.94)$$

12. Computation of f_k, $k = 1, 2, \ldots, K$ from (2.2).

$$T_{sf_{10}} = \left[p_m + \left\lceil \frac{q}{s} \right\rceil\right] t_f \quad (5.95)$$

13. Computation of $\frac{\partial f_k}{\partial U_i}$, $\frac{\partial f_k}{\partial U_j}$, $k = 1, 2, \ldots, K$ from (2.2).

$$T_{sf_{11}} = 2\left[p_m + \left\lceil \frac{q}{s} \right\rceil\right] t_f \quad (5.95)$$

14. Computation of $\frac{\partial f_k}{\partial M_i}$, $\frac{\partial f_k}{\partial M_j}$, $k = 1, 2, ..., K$ from (2.2).

$$T_{sf_{12}} = 8 \left[p_m + \left\lceil \frac{q}{s} \right\rceil \right] t_f \quad (5.96)$$

15. Synchronize all the processors.

$$T_{ssync_2} = t_s \quad (5.98)$$

16. Partially broadcast f_{K+i}, f_{K+N+i} for the random nodes only.

$$T_{sc_2} = 2(N - K + q + s)\, t_c \quad (5.99)$$

17. Computation of elements of subvectors ζ_i, $i = 1, 2, ..., s+1$ of $\mathbf{z}$ from (2.57).

$$T_{sf_{13}} = 10 \left[p_m + \left\lceil \frac{q}{s} \right\rceil \right] t_f \quad (5.100)$$

18. Computation of $\mathbf{D}_i$, $\mathbf{B}_i$, $\mathbf{R}_i$, $i = 1, 2, ..., s$ from (2.58) and (2.59).

$$T_{sf_{14}} = (17\, p_m + 12\, d_{av} - 18)\, t_f \quad (5.101)$$

19. Computation of elements of matrix $\mathbf{V}$.

Calculation of matrix $\mathbf{V}$ is partitioned row-wise among s processors.

$$T_{sf_{15}} = \left[11 \left\lceil \frac{q}{s} \right\rceil + 3 \times 2\, (d_{av} - 1) \left\lceil \frac{q}{s} \right\rceil \right] t_f$$

or

$$T_{sf_{15}} = \left\lceil \frac{q}{s} \right\rceil (6\, d_{av} + 5)\, t_f \quad (5.102)$$

20. Synchronize all the processors.

$$T_{ssync_3} = t_s \quad (5.103)$$

21. Transfer ζ_j, $\mathbf{D}_j$, $\mathbf{R}_j$, $j = 1, 2, ..., s$ to the master.

$$T_{sc_3} = (K - q) + \sum_{i=1}^{s} (3\, p_i - 2) + 2\, n_{av}\, s$$

$$T_{sc_3} = [4(K - q) + (2\, n_{av} - 2)\, s]\, t_c \quad (5.104)$$

22. Partition matrix $[\mathbf{B}_1\ \mathbf{B}_2\ \cdots\ \mathbf{B}_s]$ row-wise among s processors.

Matrices $\mathbf{B}_i$ are sparsely stored. Therefore, taking 100% overhead for communication of addresses of matrix elements,

$$T_{sc_4} = 4\, n_{av}\, s \quad (5.105)$$

Now the matrix $\mathbf{A}$ and vector $\mathbf{z}$ are partitioned among the master and s slave processors as required by direct parallel method I for solving the matrix equation (refer section 5.2.1.1). Total processing time for the rest of the computations will be the sum of times given by (5.84) through (5.105). Therefore,

$$T_s(\text{rest}) = \left[63\, p_m + (34 + 6\, d_{av}) \left\lceil \frac{q}{s} \right\rceil + (6\, d_{av} + 4) \left\lceil \frac{N_{nc}}{s} \right\rceil + 24\, d_{av} - 28 \right] t_f$$

$$+ \left[(p_m + 1) + \left\lceil \frac{N_{nc}}{s} \right\rceil \right] (T_p + T_{dpu} + T_{dpm})\, t_f$$

$$+ \left[7\, N - 3K + 4q + (6\, n_{av} - 7)s \right] t_c + 3\, t_s$$

Taking $T_p = T_{dpu} = T_{dpm} = 25$,

$$T_s(\text{rest}) = \left[138\, p_m + (6\, d_{av} + 34) \left\lceil \frac{q}{s} \right\rceil + (6\, d_{av} + 79) \left\lceil \frac{N - K + q - s}{s} \right\rceil \right. \\ \left. + 24\, d_{av} + 47 \right] t_f + \left[7N - 3K + 4q + (6\, d_{av} - 7)\, s \right] t_c + 3t_s \quad (5.106)$$

The resulting speed-ups are plotted in Fig. 5.20(a) as a function of network size.

5.3.1.2. Matrix equation to be solved by direct parallel method II

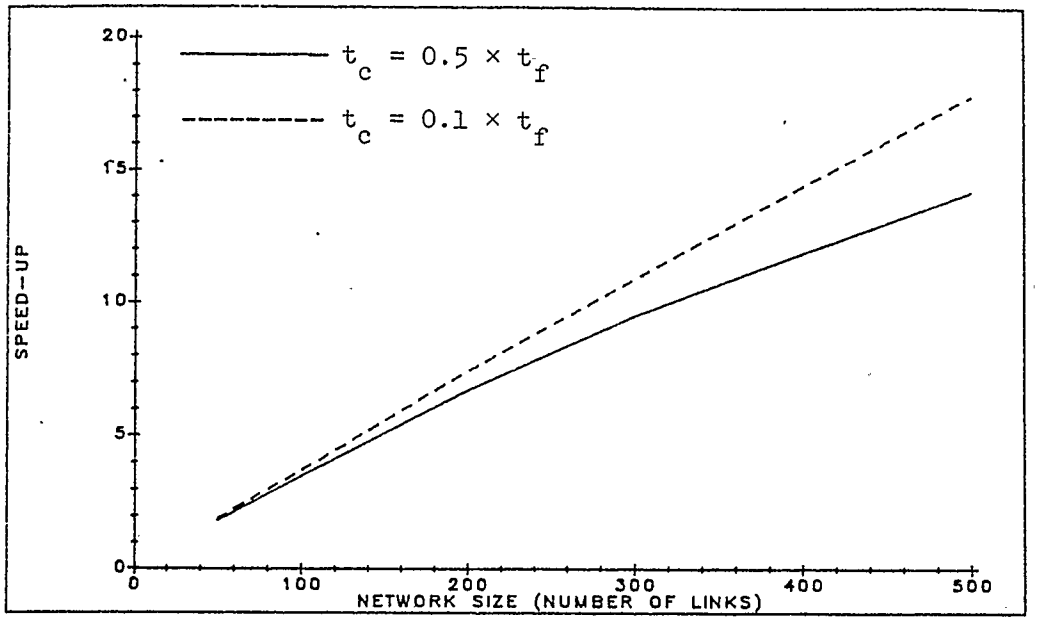
The pseudo-code for this case is the same as given in Fig. 5.18 for direct parallel method I except that ζ_j, $\mathbf{D}_j$, $\mathbf{R}_j$, $\mathbf{B}_j$ are not to be transferred as in the case of direct method I. Hence, the parallel processing time in this case is the same as given by (5.106) minus the communication times given by (5.104) and (5.105). Hence,

$$T_s(\text{rest}) = \left[138\, p_m + (6\, d_{av} + 34) \left\lceil \frac{q}{s} \right\rceil \right. \\ \left. + (6\, d_{av} + 79) \left\lceil \frac{N - K + q - s}{s} \right\rceil + 24\, d_{av} + 47 \right] t_f \\ + [7N - 7K + 8q]\, t_c + 3t_s \quad (5.106)$$

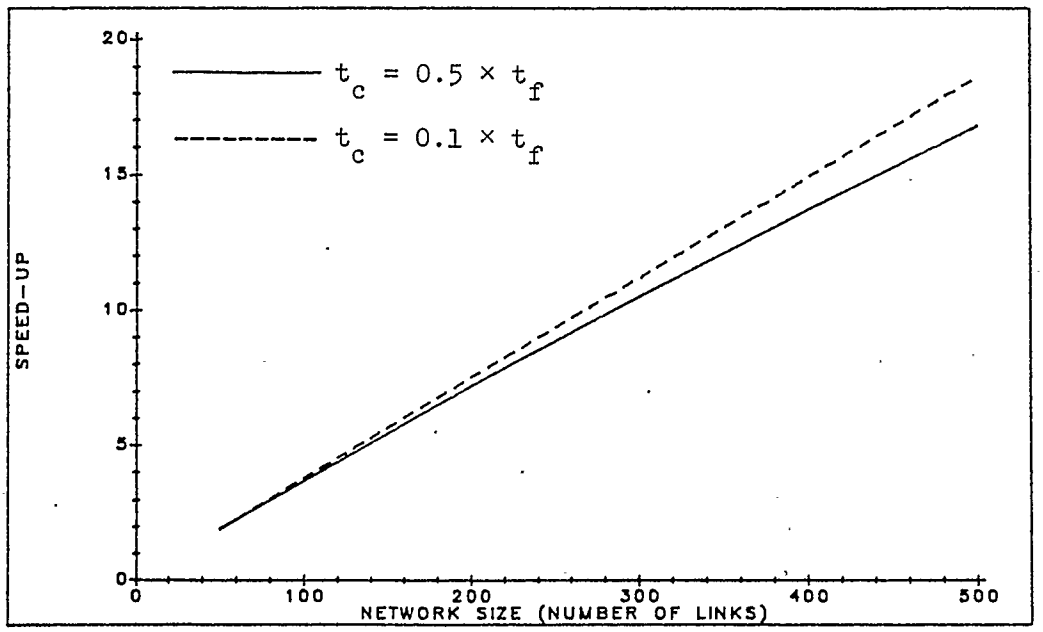
The speed-ups obtained in this case are somewhat higher than the previous case and are plotted in Fig. 5.20(b)

5.3.2. Number of processors equal to number of random links q in the network

In this case, all the computations are equally divided among q processors unlike the case of s processors where the computations are partitioned as one chain per processor for the chained links and nodes and the remaining computations are then equally divided among s processors. The parallel processing time for this case can be obtained similar to the case of s processors and is given by



(a) matrix equation solution by direct method I



(b) matrix solution by direct method II

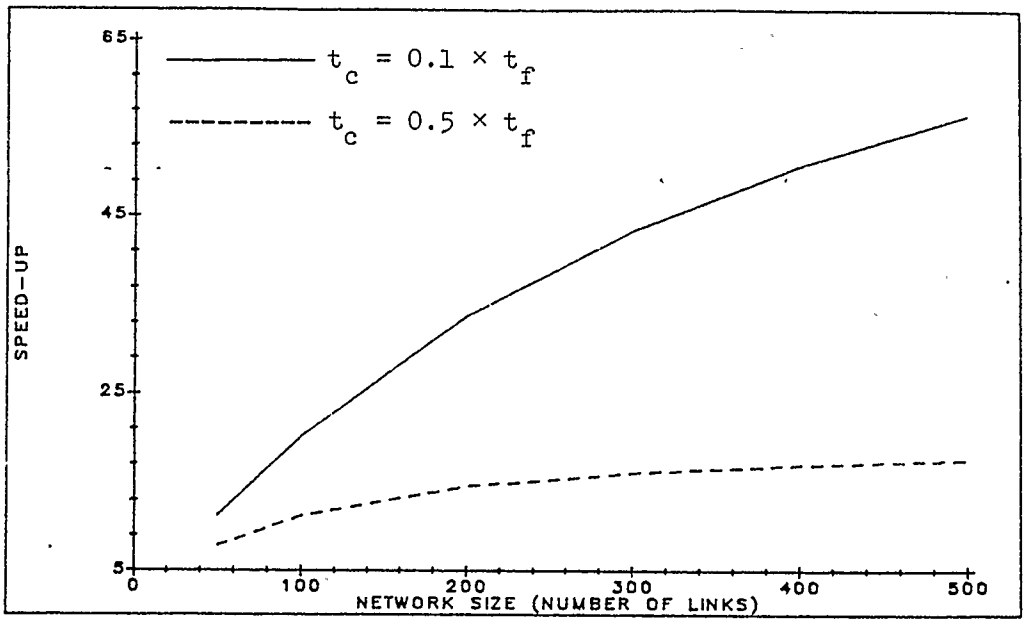
Figure 5.20 Speed-up of parallel processing for the rest of the computations on s processors

$$T_q(\text{rest}) = \left[40 \left\lceil \frac{k}{q} \right\rceil + (T_p + T_{dpu} + T_{dpm} + 6) \left\lceil \frac{N}{q} \right\rceil \right.$$
$$+ 13 \left\lceil \frac{k - q - s}{q} \right\rceil + (6\, d_{\max} + 1) \left\lceil \frac{N - K + q + s}{q} \right\rceil$$
$$\left. + 3 \left\lceil \frac{\{2K + (2\, d_{av} - 4)\,(q + s)\}}{q} \right\rceil \right] t_f \qquad (5.107)$$
$$+ [7N + 6K + (2\, d_{av} - 4)\,(q + s)]\, t_c + 3t_s$$

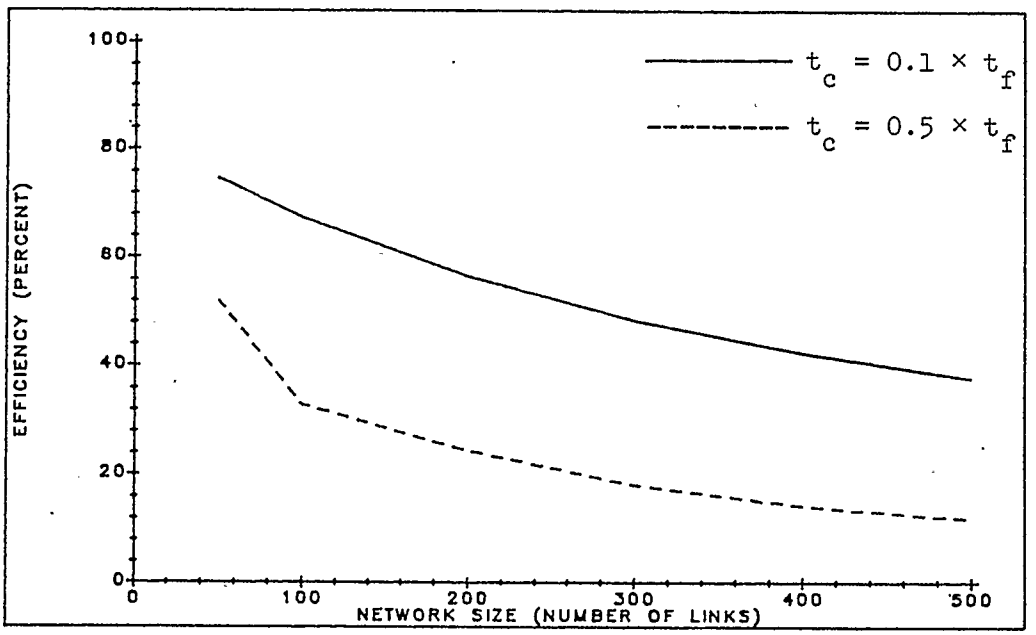
where $d_{\max}$ is the maximum degree of a node in the network. Taking $T_p = T_{dpu} = T_{dpm} = 25$, we have

$$T_q(\text{rest}) = \left[40 \left\lceil \frac{k}{q} \right\rceil + 81 \left\lceil \frac{N}{q} \right\rceil + 13 \left\lceil \frac{K - q - s}{q} \right\rceil \right.$$
$$+ (6\, d_{\max} + 1) \left\lceil \frac{N - K + q + s}{q} \right\rceil$$
$$\left. + 3 \left\lceil \frac{\{2K + (2\, d_{av} - 4)\,(q + s)\}}{q} \right\rceil \right] t_f \qquad (5.108)$$
$$+ [7N + 6K + (2\, d_{av} - 4)\,(q + s)]\, t_c + 3t_s$$

This value of parallel computation time is the same irrespective of whether the matrix equation is solved by direct parallel method I or direct parallel method II since all the computed values of elements of **A** and **z** are to be communicated in the two cases. The resulting speed-ups and efficiencies for a number of network examples are plotted in Fig. 5.21. The efficiency falls off with the increase in the problem size and the number of processors used. The fall in efficiency is rather drastic for the case when $t_c = 0.5\, t_f$, i.e. for a slow bus with respect to the processors. However, as the problem size increases, the rest of the



(a) Speed-up versus network size



(b) efficiency versus network size

Figure 5.21 Speed-up and efficiency of parallel processing for performing the rest of the computations on q processors.

computaions form a small fraction of the total computations as shown in Fig. 3.10(a) and (b). Therefore, the overall efficiency of parallel computation remains reasonable high as is shown in the next section. A summary of processing times for the rest of the computations is given in Table 5.3

5.4. Parallel Processing Performance for Total Computations

In the preceding sections, the parallel processing performance for the solution of matrix equation was considered separately from the rest of the computations. In this section, the overall performance of total parallel computations is considered. Total parallel time is given by the sum of the parallel times for the matrix equation computations and the rest of the computations as given in Tables 5.1, 5.2 and 5.3. The speed-ups and efficiencies of total parallel computation are plotted in Fig. 5.22 through 5.27.

The smallest permissible values of integration time-step for real-time computations are given in Tables 5.4(a) and (b) and also plotted in Fig. 5.28(a) and (b).

From these plots, it can be seen that, for the case of a single processor, the minimum step size for real-time computations increases very rapidly with the size of the problem. These large values of step size cannot be used from the point of view of integration accuracy requirements. On the other hand, the step size requirements for real-time computing increase slowly when parallel processing is employed. For networks not too large in size (say, with upto 300 links), number of processors equal to the number of chains in the network can be employed to give a reasonably small integration step size. For large networks, however, number of processors equal to the number of random links q in the network should be employed in order to limit the step size to small values. Number of processors equal to s plus a $q \times q$ processor array can also be employed for large-size

Table 5.1

Summary of computation, communication, and synchronization times for solving $\mathbf{A}\,\Delta\,\mathbf{W}^{n+1} = \mathbf{z}$ by direct method I.

Number of Processors	Computing Time	Communication Time	Synchronization Time
1	$[\frac{2}{3}q^3+\frac{3}{2}q^2+(4n_{av}^2+18n_{av}+13)(K-q)+(12n_{av}^2+2n_{av}-12)s-\frac{7}{6}q]t_f$	-	-
Number of Chains s	$[\lceil\frac{q}{s}\rceil(q^2+2q)+(4n_{av}^2+10n_{av})(K-q)+(12n_{av}^2+2n_{av})s+(4n_{av}+3)p_m]t_f$	$[\frac{q^2}{2}+(2n_{av}+6)(K-q)+2.5q+(2n_{av}-2)s]t_c$	Kt_s
Number of random links q	$q^2+2q+(4n_{av}+10)(K-q)+(4n_{av}-4)s+(4n_{av}+3)p_m]t_f$	$\frac{q^2}{2}+(2n_{av}+6)(K-q)+2.5q+(2n_{av}-2)s]t_c$	Kt_s
Number of Chains s and a $q\times q$ processor array	$[(4n_{av}^2+10n_{av})(K-q)+(12n_{av}^2+2n_{av})s+(4n_{av}+3)p_m]t_f+(14q-10)t_f'$	$[q^2+(2n_{av}+6)(K-q)+2q+(2n_{av}-2)s]t_c+9qt_c'$	$(K-q+1)t_s$

Table 5.2

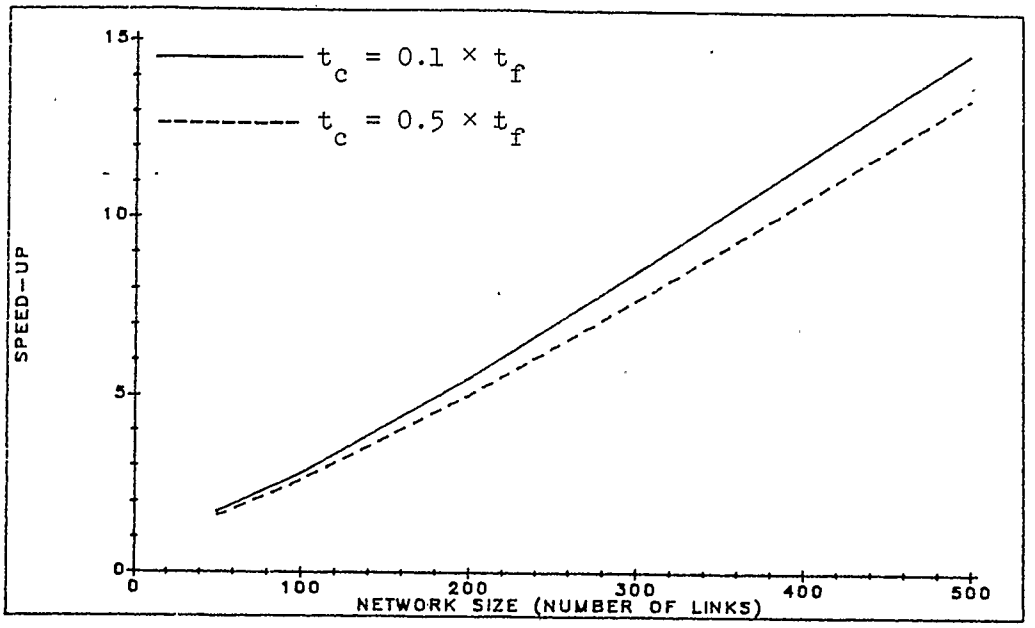
Summary of computation, communication, and synchronization times for solving $\mathbf{A}\,\Delta\,W^{n+1} = \mathbf{z}$ by direct method II.

Number of Processors	Computation Time	Communication Time	Synchronization Time
1	$[\frac{2}{3}q^3+\frac{3}{2}q^2+(13n_{av}+8)(K-q)+(16n_{av}^2+5n_{av}-7)s-\frac{7}{6}q]t_f$	-	-
Number of Chains s	$[\left\lceil\frac{q}{s}\right\rceil(q^2+2q+6n_{av})+(13n_{av}+8)p_m+2(n_{\max}+n_{av})^2]t_f$	$[\frac{q^2}{2}+\frac{5}{2}q+(8n_{av}^2+4n_{av})s]t_c$	$(q+3)t_s$
Number of random links q	$q^2+2q+(13n_{av}+8)p_m+2(n_{\max}+n_{av})^2]t_f$	$\frac{q^2}{2}+\frac{5}{2}q+7K+(4d_{av}-4)(q+s)+(8n_{av}^2+4n_{av})s]t_c$	$(q+3)t_s$
Number of Chains s and a $q\times q$ processor array	$[(13n_{av}+8)p_m+2(n_{\max}+n_{av})^2]t_f+14qt_f'$	$[(4d_{av}+1)q+8n_{av}^2s]t_c+12q+8n_{av}s)t_c'$	$3t_s$

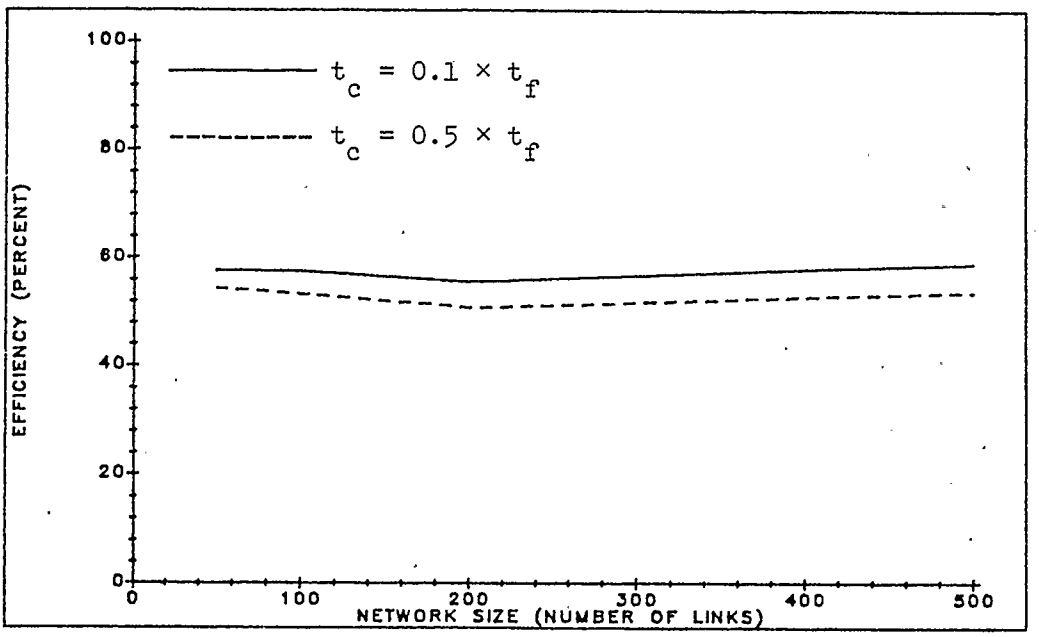
Table 5.3

Summary of computation, communication and synchronization times for performing the rest of the computations.

Number of Processors	Computing Time	Communication Time	Synchronization Time
1	$[(57-6d_{av})K+(T_p+T_{dpu}+T_{dpm}+6d_{av}+7)N+(12d_{av}-24)(q+s)]t_f$	-	-
Number of Chains s	$[63p_m+\left\lceil\frac{q}{s}\right\rceil(6d_{av}+34)+\left\lceil\frac{N_{nc}}{s}\right\rceil(6d_{av}+4)+(p_m+\left\lceil\frac{N_{nc}}{s}\right\rceil+1)(T_p+T_{dpu}+T_{dpm})+24d_{av}-28]t_f$	$[7N-3K+4q+(6n_{av}-7)s]t_c$	$3t_s$
Number of random links q	$[40\left\lceil\frac{K}{q}\right\rceil+(T_p+T_{dpu}+T_{dpm})\left\lceil\frac{N}{q}\right\rceil+13\left\lceil\frac{K-q-s}{q}\right\rceil+(6d_{av}+1)\left\lceil\frac{N-K+q+s}{q}\right\rceil+3\left\lceil\frac{\{2K+(2d_{av}-4)(q+s)\}}{q}\right\rceil]t_f$	$[7N+6K+(2d_{av}-4)(q+s)t_c$	$3t_s$

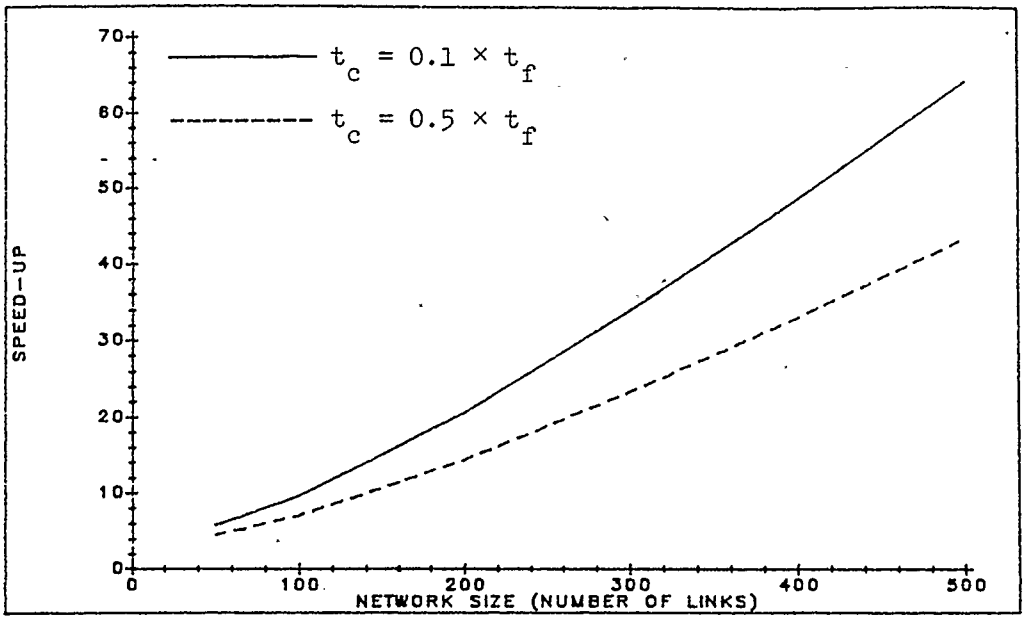


(a) Speed-up versus network size

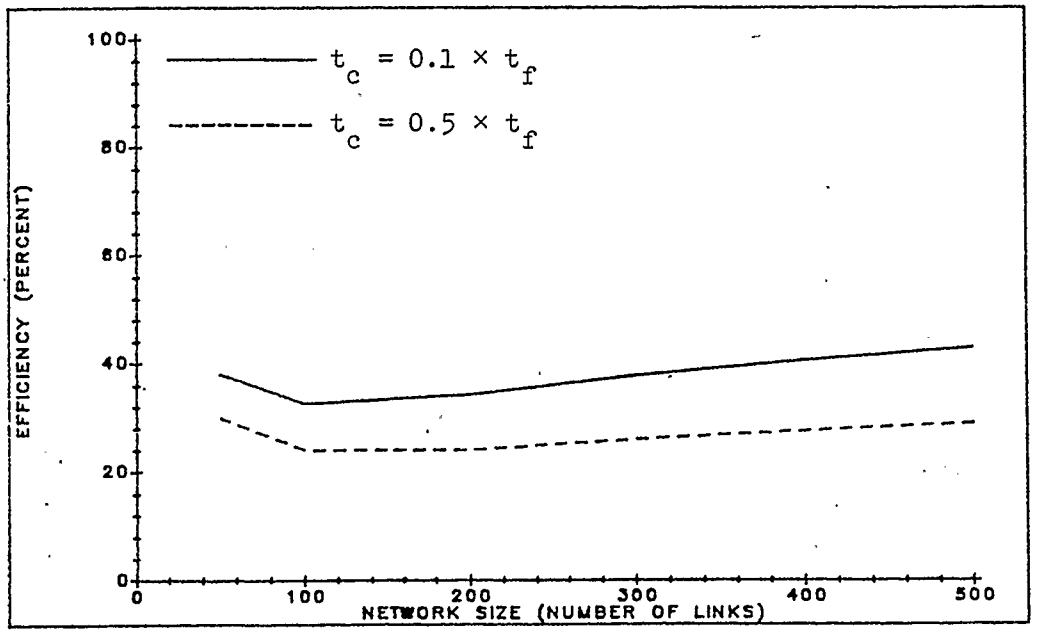


(b) Efficiency versus network size

Figure 5.22 Speed-up and efficiency of parallel processing for total computations on s slave processors. Matrix solution is by direct method I



(a) Speed-up versus network size



(b) Efficiency versus network size

Figure 5.23 Speed-up and efficiency of parallel processing for total computations on q processors. Matrix equation solution is by direct method I

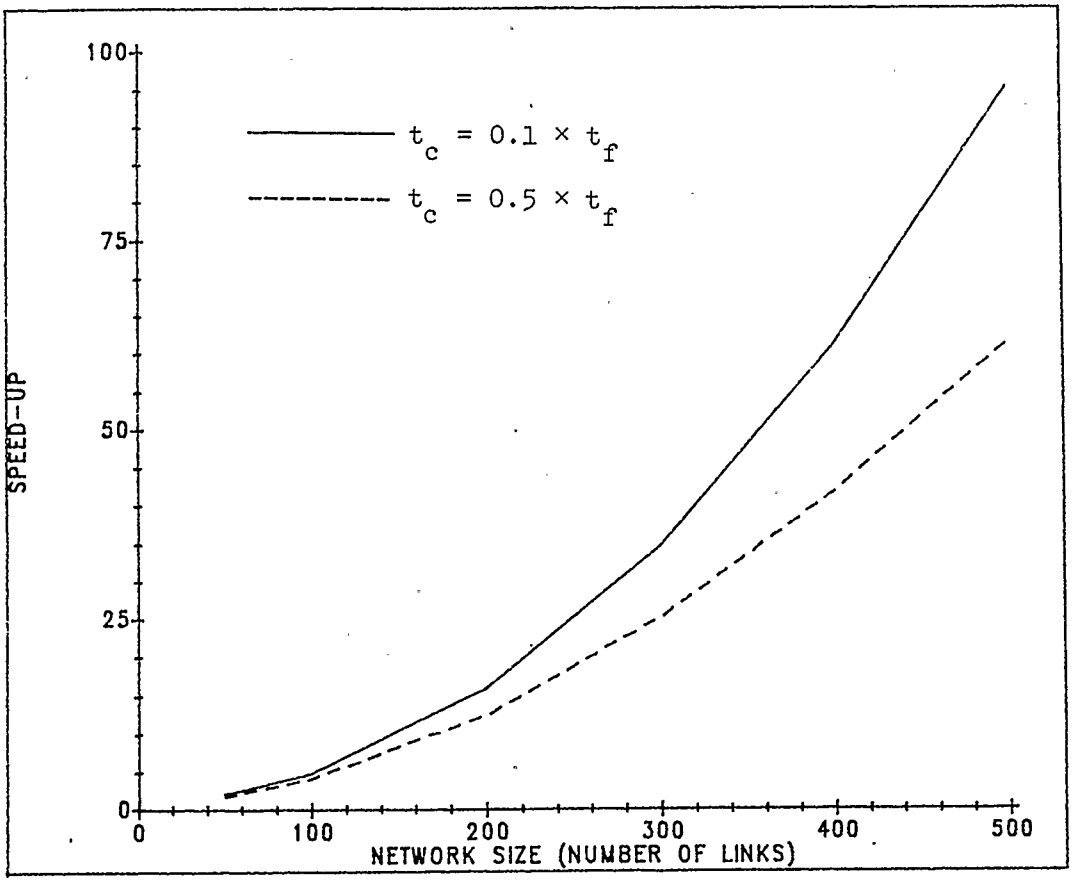
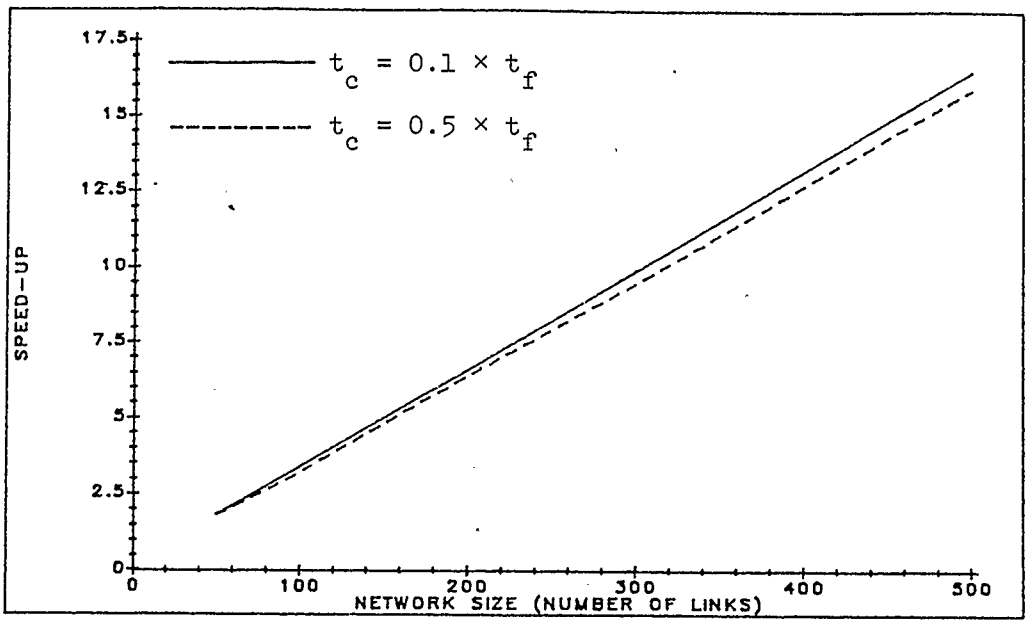
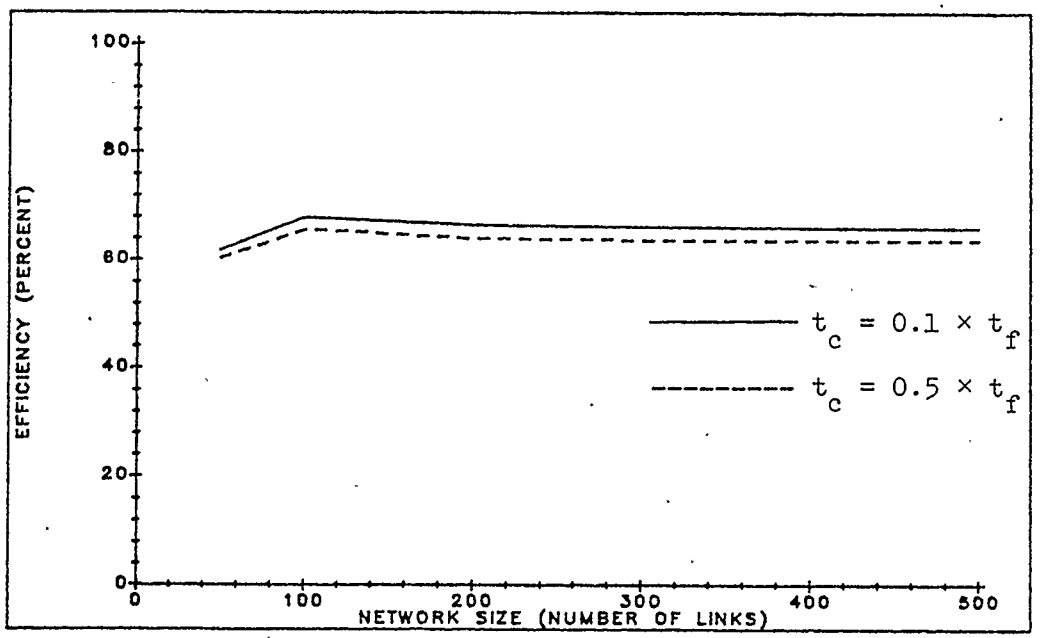


Figure 5.24 Speed-up of parallel processing for total computations on s slave processors and a $q \times q$ processor array. Matrix solution is by direct method I ($t_f' = t_f$, $t_c' = 4 \times t_c$)

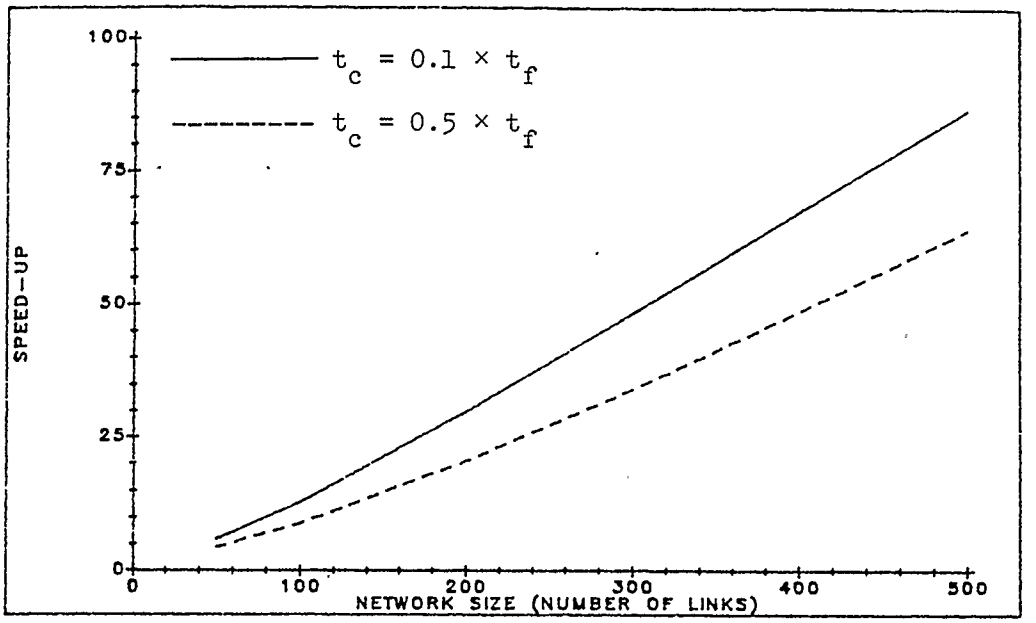


(a) Speed-up versus network size

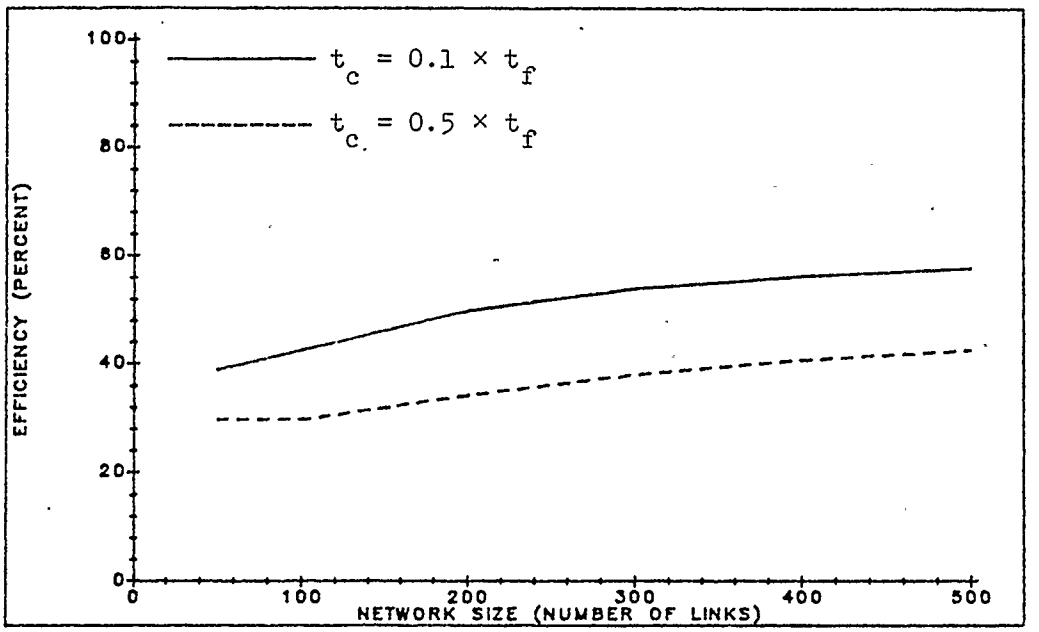


(b) Efficiency versus network size

Figure 5.25 Speed-up and efficiency of parallel processing of total computations on *s* slave processors. Matrix solution is by direct method II



(a) Speed-up versus network size



(b) Efficiency versus network size

Figure 5.26 Speed-up and efficiency of parallel processing of total computations on q slave processors. Matrix solution is by direct method II

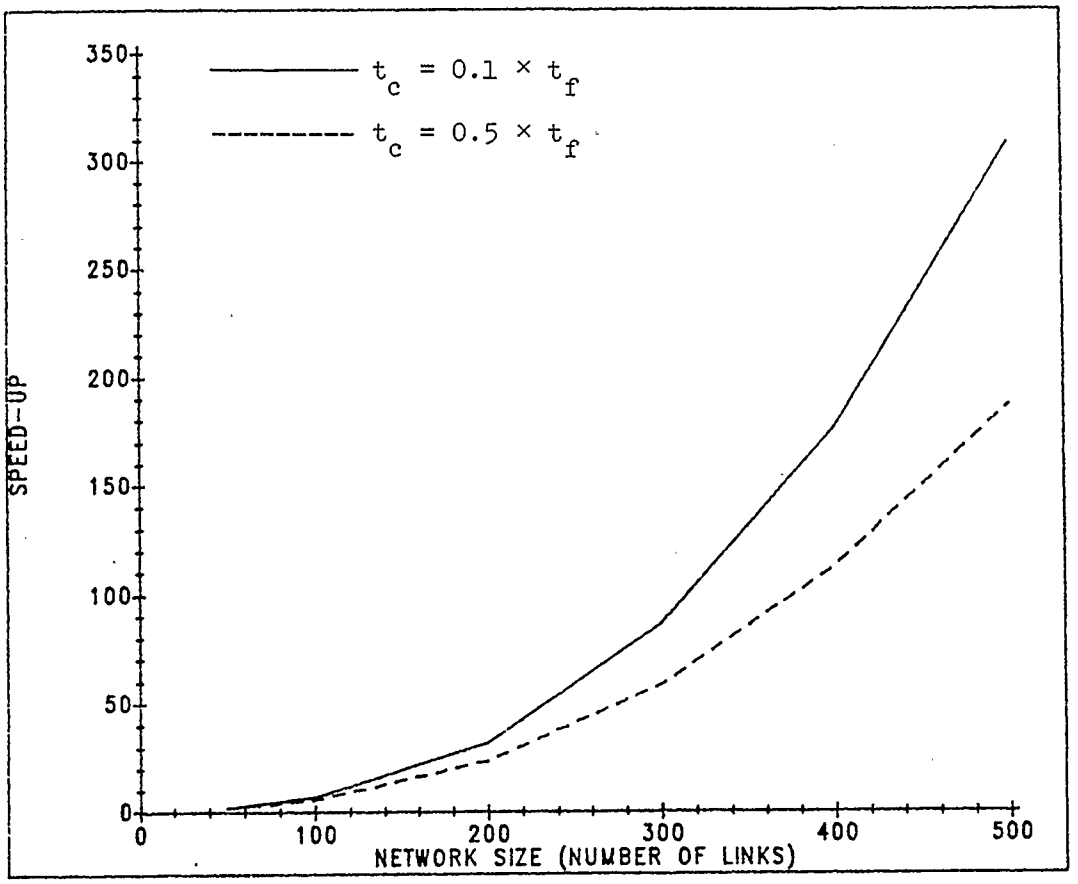


Figure 5.27 Speed-up of total computations on s slave processors and a $q \times q$ processor array with $t_f' = t_f$, $t_c' = 4 \times t_c$. Matrix equation solution is by direct method II

Table 5.4

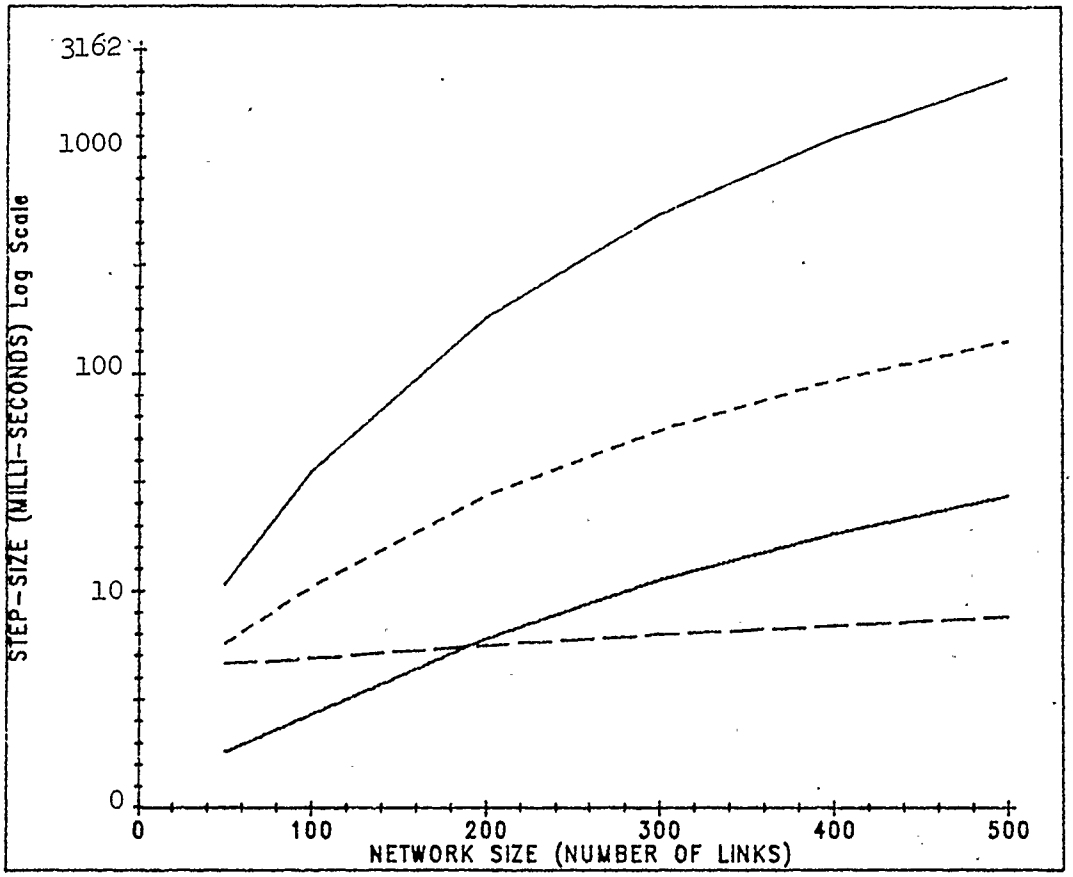
Step-size values for real-time computations using processors with $t_f = 1$ μsec. Matrix solution is by direct method II. ($n_{av} = 2.0$, $n_{max} = 6$, $d_{av} = 3.0$, $p_m = 20$, $t_s = 5.0\ t_f$, $t_f' = t_f$, $t_c' = 4\ t_c$)

Network example	Step-size for real-time computations (milliseconds)			
	using one processor	using s processors	using q processors	using s processors and a $q \times q$ array
K=50, q=15, s=3, N=45	10.6	5.7	1.8	4.6
K=100, q=30, s=5, N=90	35.3	10.4	2.7	4.9
K=200, q=60, s=10, N=180	181.4	27.3	6.0	5.6
K=300, q=90, s=15, N=270	546.1	55.0	11.2	6.3
K=400, q=120, s=20, N=360	1237.0	93.7	18.3	6.9
K=500, q=150, s=25, N=450	2363.0	143.2	27.3	7.6

(a) $t_c = 0.1\ t_f$

Network example	Step-size for real-time computations (milliseconds)			
	using one processor	using s processors	using q processors	using s processors and a $q \times q$ array
K=50, q=15, s=3, N=45	10.6	5.8	2.3	5.1
K=100, q=30, s=5, N=90	35.3	10.7	3.9	5.9
K=200, q=60, s=10, N=180	181.4	28.3	8.8	7.6
K=300, q=90, s=15, N=270	546.1	57.2	15.9	9.3
K=400, q=120, s=20, N=360	1237.0	97.3	25.3	10.9
K=500, q=150, s=25, N=450	2363.0	148.6	36.9	12.6

(b) $t_c = 0.5\ t_f$



(a) $t_c = 0.1 \times t_f$

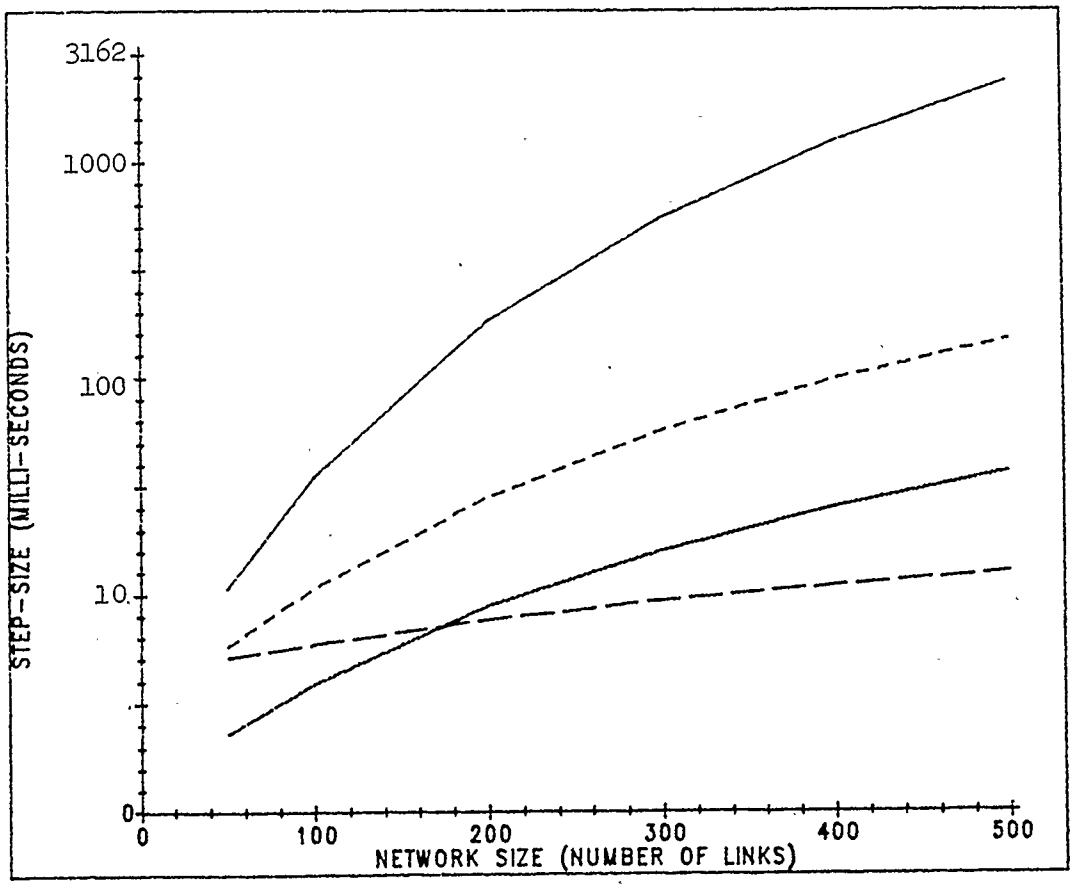
———— Number of processors = 1

--------- Number of processors = s

..... Number of processors = q

— — — — Number of processors= s plus a q × q processor array

Figure 5.28 Permissible step-size versus network-size for real-time computations (t_f = 1 micro-second, $t_f' = t_f$, $t_c' = 4 \times t_c$)



(b) $t_c \doteq 0.5 \times t_f$

———— Number of processors = 1

---------- Number of processors = s

..... Number of processors = q

— — — — Number of processors = s plus q × q processor array

Figure 5.28 Continued

Table 5.5

Effect of synchronization time t_s on the permissible step-size for real-time computations. Matrix solution is by direct method II ($t_f = 1$ μsec, $t'_f = t_f$, $t'_c = 4 \times t_c$, $t_c = 0.1 \times t_f$)

Network example	Step-size for real-time computations (milliseconds)					
	Processors = s		Processors = q		Processors = s and $q \times q$ array	
	$t_s = 5 \times t_f$	$t_s = 50 \times t_f$	$t_s = 5 \times t_f$	$t_s = 50 \times t_f$	$t_s = 5 \times t_f$	$t_s = 50 \times t_f$
K=50, q=15 s=3, N=45	5.7	6.6	1.8	2.7	4.6	4.8
K=100, q=30 s=5, N=90	10.4	12.0	2.7	4.3	4.9	5.1
K=200, q=60 s=10, N=180	27.3	30.2	6.0	8.9	5.6	5.8
K=300, q=90 s=15, N=270	55.0	59.3	11.2	15.5	6.3	6.5
K=400, q=120 s=20, N=360	93.7	99.3	18.3	23.9	6.9	7.1
K=500, q=150 s=25, N=450	143.2	150.2	27.3	34.3	7.6	7.8

networks but this scheme is at present not very cost-effective because of large hardware requirement which is not utilized very efficiently for this problem. However, with improvements in wafer-scale integration technology, even this solution may prove to be quite attractive as a result of integration of a number of processors of the processor array on a single chip.

It can also be observed from Figs. 5.22 through 5.28 that the speed-ups and efficiencies of parallel computation are not severely affected when the communication time t_c is increased from $0.1 \times t_f$ to $0.5 \times t_f$. This is particularly true when the number of processors employed is equal to the number of chains s in the network. Hence, the low-cost time-shared bus as the inter-connection network is quite adequate for this problem.

Finally, the effect of variation of synchronization time t_s from $5 \times t_f$ to $50 \times t_f$ on the permissible time step size for real-time computations is shown in Table 5.5. As can be seen from this table, the time-step size is not very sensitive to the synchronization time in this range.

In summary, it can be stated that this parallel algorithm is quite robust with respect to variations in data communication time t_c and processor synchronization time t_s within their considered range of values.

CHAPTER 6

PERFORMANCE OF A PARALLEL ITERATIVE METHOD

6.1. Introduction

In Chapter 5, two direct parallel methods for solving the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$ were described. In this chapter, the performance of an iterative parallel method for solving this equation is studied. Simulation results for two network examples are also given.

6.2. Jacobi's Iterative Method

Jacobi's iterative method for computing the solution vector $\mathbf{x}$ from the Nth order matrix equation $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ consists of performing the following iterations repeatedly until the convergence is achieved, i.e. the error between two successive iterations is less than a predetermined value ε.

$$x_i^{(m+1)} = \sum_{j=1, j=i}^{N} \left[\frac{a_{ij}}{a_{ii}}\right] x_j^{(m)} + \left[\frac{b_i}{a_{ii}}\right], \quad 1 \le i \le N, \quad m \ge 0 \tag{6.1}$$

until error(i) $= \left|x_i^{(m+1)} - x_i^{(m)}\right| \le \varepsilon$, $1 \le i \le N$, $m \ge 0$ where ε is an arbitrarily small quantity. Sometimes the following criterion of Euclidean norm of total error is also used for determining convergence for the iterative process.

$$\text{total error} = \left[\sum_{i=1}^{N} (x_i^{(m+1)} - x_i^{(m)})^2\right]^{1/2} \le \varepsilon \tag{6.2}$$

In this iterative method, all the element values of vector $\mathbf{x}$ at the iteration m, i.e. $\mathbf{x}^{(m)}$ are used in iteration $m + 1$. As a result, computations in (6.1) for different values of i can be

carried out concurrently as they are independent of each other.

A sufficient condition for the convergence of the above iterative method is that the matrix **A** is strictly diagonally dominant [83], i.e.

$$\left|a_{ii}\right| > \sum_{j=1, j\neq i}^{N} \left|a_{ij}\right|, \; 1 \leq i \leq N \tag{6.3}$$

In our case, all the off-diagonal elements a_{kv} of matrix **A** given by (2.54) are of the form $h^2 \times (\;)$ and all the diagonal elements a_{kk} given by (2.53) are of the form $[1 - h \times (\;) - h^2 \times (\;)]$. Therefore, the matrix **A** in our case can be made diagonally dominant by choosing the time-step size h sufficiently small such that magnitudes of diagonal elements are large as compared to magnitudes of off-diagonal elements.

These iterative methods are generally not used when the matrix **A** is full as the complexity of computation per iteration in this case is of the order of N^2, i.e. $0(N^2)$. However, these methods become attractive as compared to direct methods when the matrix is sparse, large and has random structure. In the case of sparse matrices, the number of non-zero elements in each row does not grow with the size of the matrix and hence, the computational complexity grows as $0(N)$.

The pseudo-code for the Jacobi's iterative method of solving $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ is given in Fig. 6.1

6.3. Algorithm Performance on a Single Processor

Floating-point operation count per iteration for the Jacobi's iterative method on the serial computer can be obtained as follows:

Step 1: for $i = 1$ to N do {start the iterative process by assuming zero initial values for the vector x}

$x_{old}(i) = b(i)/a(i, i)$

end {do}

Step 2: for $i = 1$ to N do {compute new iterate values of vector x}

$$x_{new}(i) = \frac{1}{a(i, i)} \left[b(i) - \sum_{j=1, j \neq i}^{N} a(i, j)\ x_{old}(j) \right]$$

end {do}

Step 3: for $i = 1$ to N do

error = absolute$[x_{new}(i) - x_{old}(i)]$

if error > ϵ then {replace old x with new x and repeat the iterations}

for $i = 1$ to N do

$x_{old}(i) = x_{new}(i)$

end {do}

go to Step 2

else continue

end {do}

Step 4: Output the solution vector $\mathbf{x}_{new}$

Figure 6.1 Pseudo-code for solving the matrix equation $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ by Jacobi's iterative method on a single processor.

(i) Initialization of $\Delta W(i)$ to $z(i)/a(i,i)$.

Computation time $T_a = K\ t_f$

(ii) Computation time per iteration for the chained links.

Except for the two end links of a chain, the new value of the flow increment $\Delta W_{new}(i)$ is given by

$$\Delta W_{new}(i) = \frac{z(i) - a(i,i-1)\,\Delta W_{old}(i-1) - a(i,i+1)\,\Delta W_{old}(i+1)}{a(i,i)} \quad (6.4)$$

The total number of floating-point operation in (6.4) is 5 and there are $(K - q - 2s)$ such links. Therefore, the total operation count for such links is $5(K - q - 2s)$.

If n_{av} is the average number of non-zero elements in the first or last row of $\mathbf{R}_i$, then taking 100% overhead for sparse computations, the operation count for the end links is $2 \times 2s\,[2\ n_{av} + 3]$.

Hence, total computation time for chained links is

$$T_b = [5(K - q - 2s) + (8\ n_{av} + 12)\ s]\ t_f \quad (6.5)$$

(iii) Computation time per iteration for the random links.

If d_{av} is the average degree of an end-node of a random link, operation count for updating ΔW_i for one random link is $4(d_{av} - 1) + 1$.

Total operation count for q random links is $(4\ d_{av} - 3)\ q$. Since these are sparse computations, taking 100% overhead for index manipulation, the total computation time for the random links T_c is

$$T_c = (8\ d_{av} - 6)\ q\ t_f \quad (6.6)$$

(iv) Computation time (worst case) for computing error of convergence

At worst, the convergence error for K elements of vector $\Delta \mathbf{W}$ will be computed before detecting non-convergence. At best, convergence error for the first element of $\Delta \mathbf{W}$ is computed before detecting non-convergence. Taking an average value of $\frac{K}{2}$ computation for error before detecting convergence, the computation time T_d is given by

$$T_d = \frac{K}{2} t_f \quad . \tag{6.7}$$

Total computation time per time-step will be sum of T_a, T_b, T_c and T_d. Therefore, total computation time per time-step is given by

$$K + [5.5K + (8\ d_{av} - 11)\ q + (8\ n_{av} + 2)\ s]\ t_f \quad . \tag{6.8}$$

If I is the total number of iterations required to achieve convergence, total computation time of Jacobi's method is given by

$$T_1(\text{matrix}) = \left[K + \{5.5K + (8\ d_{av} - 11)\ q + (8\ n_{av} + 2)\ s\}I\right] t_f \quad . \tag{6.9}$$

Neglecting the initialization time $K\ t_f$ at step (i), we have

$$T_1(\text{matrix}) \approx \left[5.5K + (8\ d_{av} - 11)\ q + (8\ n_{av} + 2)\ s\right] I\ t_f \quad . \tag{6.10}$$

Total computation time to solve the matrix equation by direct method II has been obtained in Chapter 3 and is given by (3.36). Therefore, for the Jacobi's method to be more efficient than the direct method II, computing time given by (6.10) should be less than the time given by (3.36). This value of the number of iterations I for the iterative method to be faster is plotted in Fig. 6.2 as a function of the network size. From this figure, it is clear that this method is to be used only for very large-size networks having large number

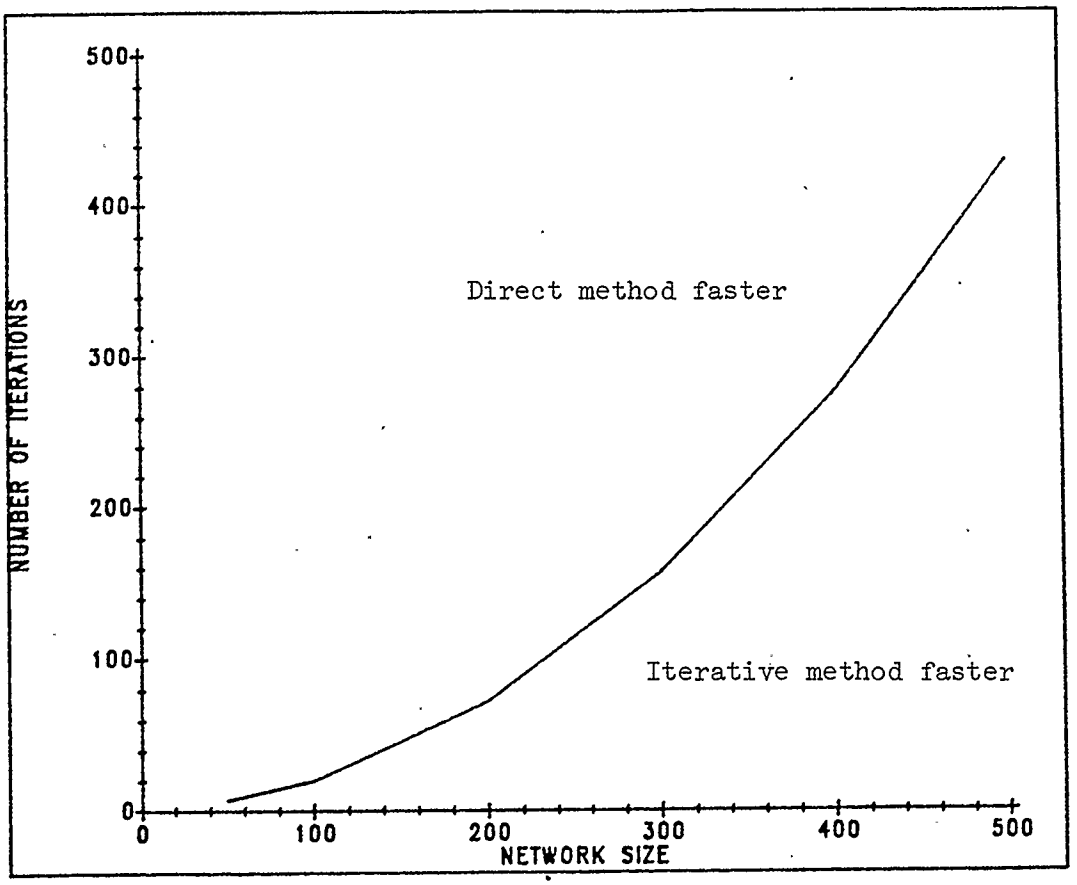


Figure 6.2 Maximum number of iterations for Jacobi's method to be faster than the direct elimination method for solving $A \Delta W = z$.

of random links. Also, the method is to be used where the iterations can be terminated early by using relatively large value of iteration error ε. An example could be training simulators for the operators where a 5-10 percent error in the simulation results may not be noticeable to the operator.

6.4. Algorithm Performance on Multiple Processors

6.4.1. Parallel Jacobi's algorithm

In the parallel version of Jacobi's algorithm [72,84], the iterative computations given by (6.1) are divided among N_p processors such that each processor updates $l = \frac{N}{N_p}$ variables, i.e. processor 1 updates variables x_1 through x_l, processor 2 updates variables x_{l+1} through x_{2l} etc. After this, the variables are checked for their convergence on their respective processors. If convergence has not occurred for one of the variables assigned to a processor, then that processor sets the flag equal to 1, sends a synchronization signal to the master and stops computations. Once all processors have finished computing and have sent their synchronization signals to master, the master processor reads the flags from the communication memories of slave processors and checks to see if any of the flags is unity, i.e. if the solution has not converged. If this is the case, the master reads the computed values of variables $x(i)$ from communication memories and broadcasts on the bus for processors requiring these values. After this, the whole procedure is repeated again till the convergence has occurred.

The pseudo-code of the parallel Jacobi's method for solving $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ is given in Fig. 6.3.

{matrix equation **A x = b** is partitioned row-wise among N_p processors with $l = \frac{N}{N_p}$ rows of **A**, **x** and **b** per processor}

Step 1: for $n = 1$ to N_p pardo {initialize all $x(i)$ elements of vector **x** concurrently on N_p processors}
  for $i = [(n-1)l + 1]$ to nl do
  $x_{old}(i) = \frac{b(i)}{a(i,i)}$
  end do
end pardo

Step 2: Synchronize all the processors

Step 3: Transfer $x_{old}(i)$ values on the bus

Step 4: for $n = 1$ to N_p pardo {concurrently initialize flag $(n) = 0$ on processor n and compute new values of variables $x(i)$}
  flag$(n) = 0$
  for $i = [(n-1)l+1]$ to nl do {up-date a slice l of $x(i)$ variables on each processor}

$$x_{new}(i) = \frac{1}{a(i,i)}\left[b(i) - \sum_{j=1, j\neq i}^{N} a(i,j)\, x_{old}(i)\right]$$

  end do
  for $i = [(n-1)l + 1]$ to nl do
    error = absolute $[x_{new}(i) - x_{old}(i)]$
    if error $> \epsilon$ then
      set flag$(n) = 1$
      for $j = [(n-1)l+1]$ to nl do {replace old $x(j)$ with new $x(j)$ values}
        $x_{old}(j) = x_{new}(j)$
      end do
      stop computing and send synchronization signal
    else continue
  end do
  for $i = [(n-1)l+1]$ to nl do {replace old values of $x(i)$ with new values}
    $x_{old}(i) = x_{new}(i)$
  end do
  send synchronization signal from processor(n)
end pardo (n)

Figure 6.3 Pseudo-code for solving the matrix equation **A x = b** by parallel Jacobi's method on N_p processors

Step 5: read flags from slave processors to master processor

Step 6: for $i = 1$ to Np do {check the flags on the master processor.
Repeat the iterative procedure if convergence not occurred}
if $flag\ (i) = 1$ then
go to Step 3
else continue
end do
output the solution vector $\mathbf{x}_{new}$.

Figure 6.3 Continued

6.4.2. Estimation of parallel computation time

An estimation of parallel computing time for the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$ will now be obtained. Since the computations in (6.1) are independent of each other, any arbitrary number of processors up to a maximum of K processors can be employed to perform parallel processing for solving $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$. However, analytical expressions for parallel computing time for the general case of N_p arbitrary number of processors cannot be obtained due to the dependence of communication time on the problem partitioning among N_p processors. Therefore, estimation of parallel computing time for the special case of number of processors equal to number of chains s in the network will be obtained. This choice of number of processors is also attractive from the performance point of view since the $\Delta\,\mathbf{W}_i$ variables associated with a chain are computed on the same processor thus minimizing communication overhead. Also, the granularity of computations remains relatively coarse so that the effect of communication and synchronization overhead on the computing efficiency is reduced. Since the chains in the network may be of different length, the computing load for different chains will be different. The load balancing can be achieved by suitably partitioning the iterations for the remaining q random links in the network among s processors such that the total load per processor for performing iterations for the chained links and random links is roughly the same for all the processors. However, for the sake of simplicity, it is assumed in the following analysis that iterations for the q random links are equally divided among s processors. This may result in somewhat poor speed-ups particularly for the case of a network having very long and very short chains. Estimation of parallel computing time is now obtained.

(i) Computation time to initialize $\Delta W_{old}(i)$.

If p_m is the length of the longest chain, then time to initialize $x_{old}(i)$ is given by

$$T_{sf_1} = \left[p_m + \left\lceil \frac{q}{s} \right\rceil\right] t_f \cdot$$

(ii) Time to synchronize processors.

$$T_{synch_1} = t_s \cdot$$

(iii) Transfer $\Delta W_{old}(i)$ values on the bus.

$\Delta W_{old}(i)$ for the random links and ends links of chains are only to be transferred. Therefore,

$$T_{sc_1} = [q + 2s]\, t_c \cdot$$

(iv) Time to compute $\Delta W_{new}(i)$ using $\Delta W_{old}(i)$.

If p_m is the length of the longest chain, then

$$T_{sf_2} = \left[2 \times 2(2\, n_{av} + 3) + 5(p_m - 2) + \left\lceil \frac{q}{s} \right\rceil (8\, d_{av} - 6)\right] t_f$$

or

$$T_{sf_2} = \left[5\, p_m + 8\, n_{av} + \left\lceil \frac{q}{s} \right\rceil (8\, d_{av} - 6) + 2\right] t_f \cdot$$

(v) Time to compute convergence error. The worst case time T_{sf_3} is given by

$$T_{sf_3} = \left[p_m + \left\lceil \frac{q}{s} \right\rceil\right] t_f \cdot$$

(vi) Time to synchronize the processors.

$$T_{s\ snych_2} = t_s \cdot$$

(vii) Time to read $flag(i)$, $i = 1, 2, ..., s$ from communication memories to master processor.

$$T_{sc_2} = s\ t_c$$

If I is the total number of iterations required for convergence, then total processing time will be sum of time in step (i) and I multiplied by sum of the times given in steps (ii) through (vii). Hence,

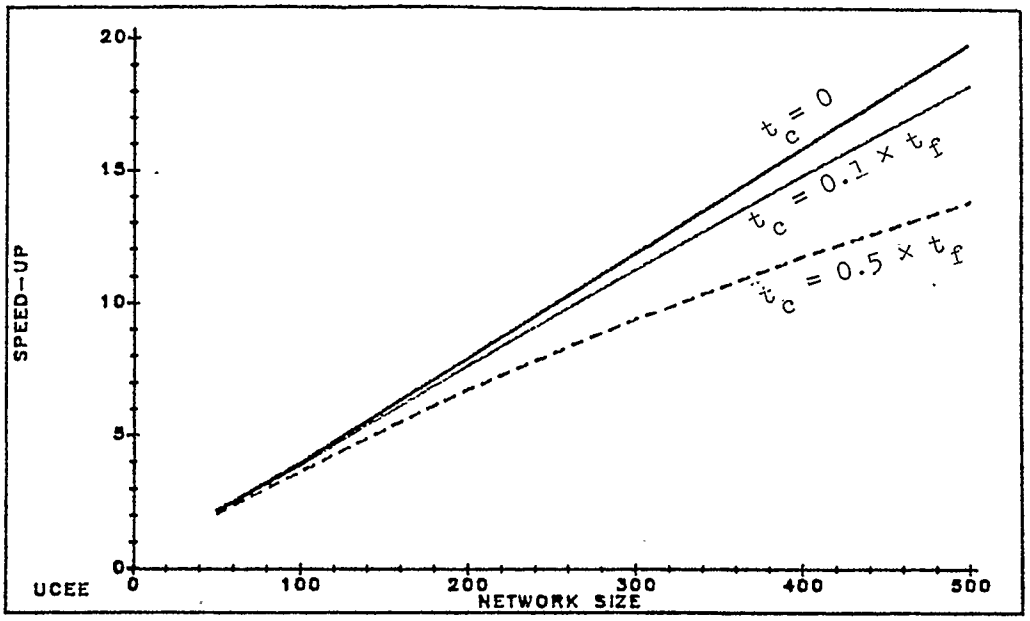
$$T_s(\text{matrix}) = \left[p_m + \left\lceil \frac{q}{s} \right\rceil\right] t_f + \left[\left\{6\, p_m + 8\, n_{av} + \left\lceil \frac{q}{s} \right\rceil (8\, d_{av} - 5) + 2\right\} t_f + (q + 3s)\, t_c + 2\, t_s\right] I \quad (6.11)$$

Neglecting initialization time of $\left[p_m + \left\lceil \frac{q}{s} \right\rceil\right] t_f$ at step (i), we have

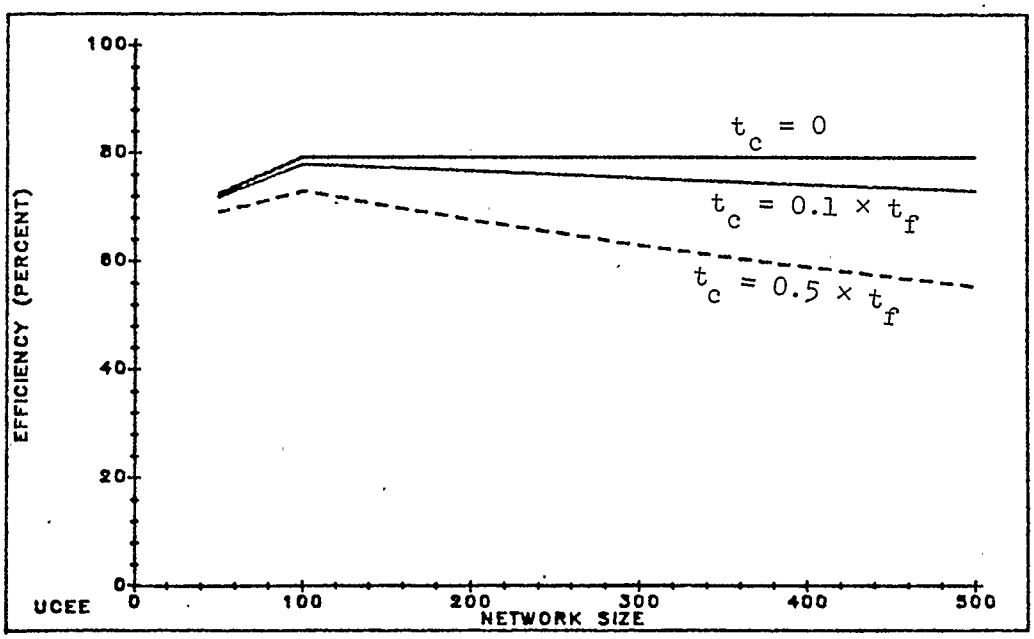
$$T_s(\text{matrix}) \approx \left[\left\{6\, p_m + 8\, n_{av} + \left\lceil \frac{q}{s} \right\rceil (8\, d_{av} - 5)\right\} t_f + (q + 3s)\, t_c + 2\, t_s\right] I \quad (6.12)$$

The speed-up of parallel processing will be the ratio of T_1(matrix) given by (6.10) and T_s(matrix) given by (6.12). Since number of iterations I cancels out on the numerator and denominator side, the speed-up is independent of the number of iterations I.

Effect of communication time t_c, synchronization time t_s, and the length p_m of the longest chain on the speed-up and efficiency of parallel processing is shown in Figs. 6.4, 6.5 and 6.6 respectively. Although the speed-ups and efficiencies fall with an increase in the value of these parameters (particularly for larger-size networks), this degradation in performance is not very severe. This is due to the reasons of relatively coarse computa-

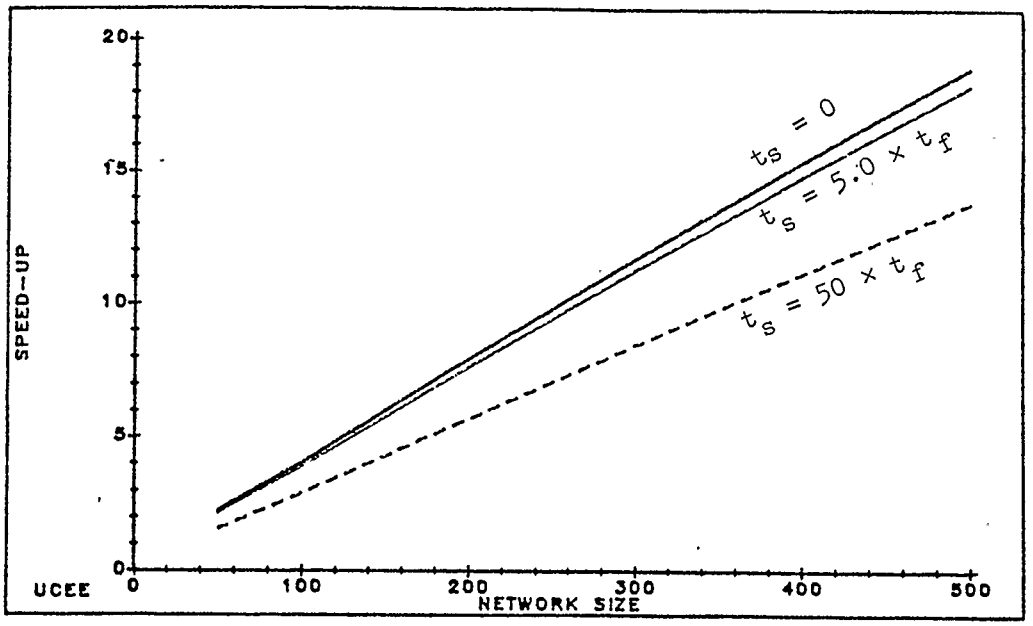


(a) Speed-up versus network size

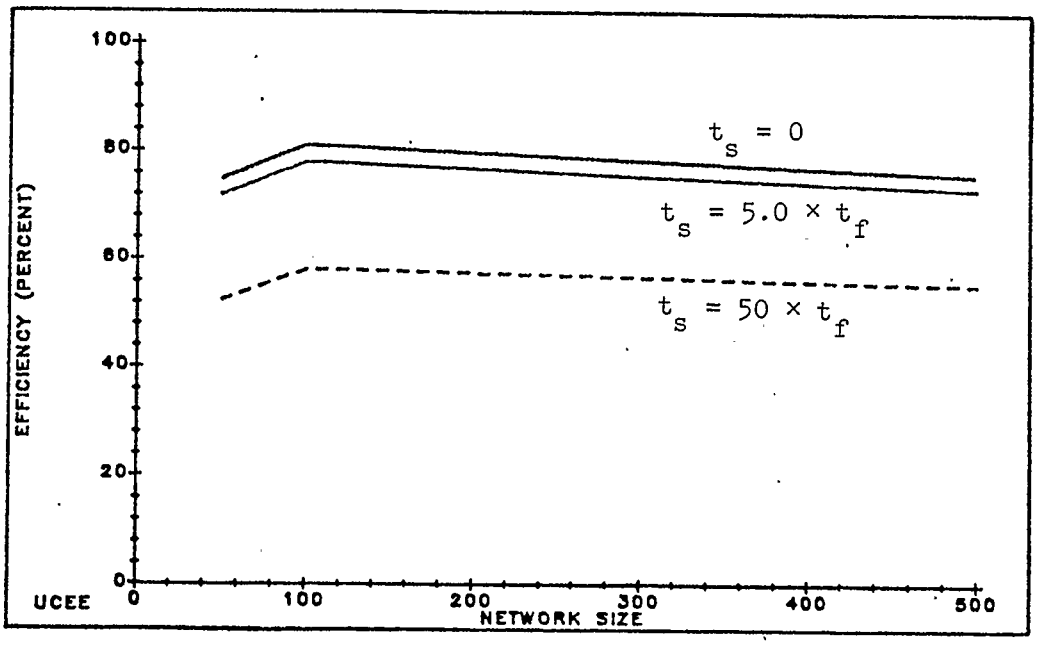


(b) Efficiency versus network size

Figure 6.4 Effect of communication time t_c on the performance of Jacobi's iterative method for solving $\mathbf{A}\ \Delta\ \mathbf{W} = \mathbf{z}$ on s processors ($t_s = 5.0 \times t_f$, $p_m = 20$).

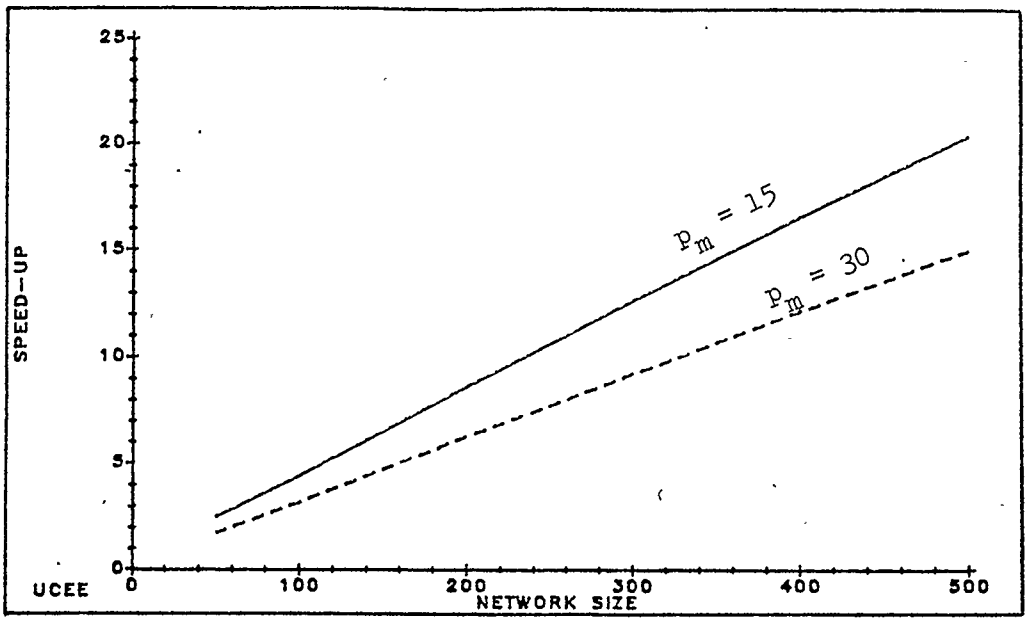


(a) Speed-up versus network size

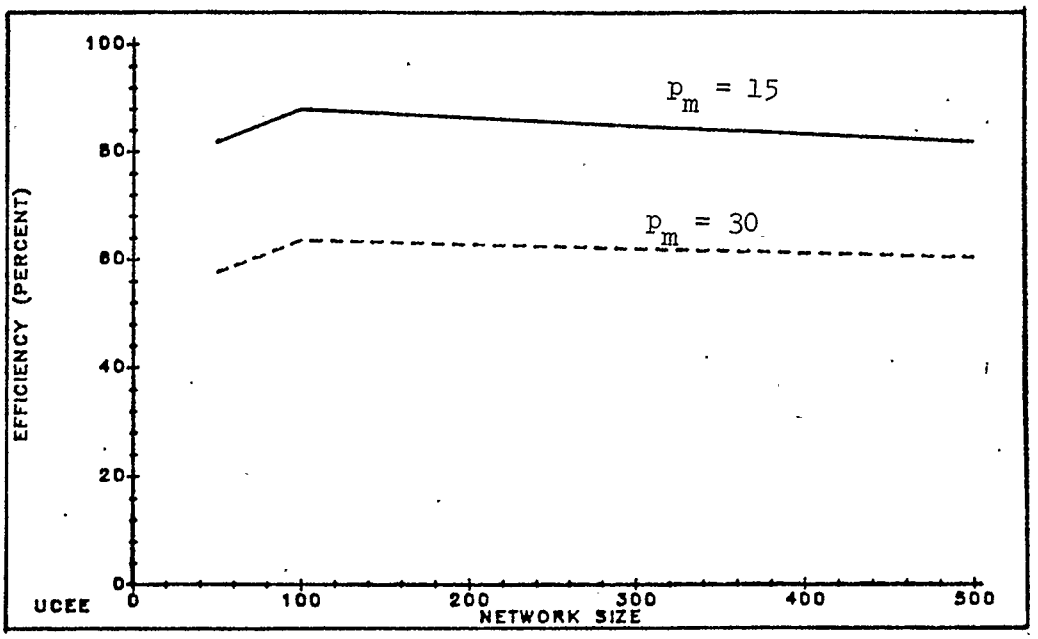


(b) Efficiency versus network size

Figure 6.5 Effect of synchronization time t_s on the performance of Jacobi's iterative method for solving $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$ on s processors ($t_c = 0.1 \times t_f$, $p_m = 20$).



(a) Speed-up versus network size



(b) Efficiency versus network size

Figure 6.6 Effect of the length p_m of the longest chain in the network on the performance of Jacobi's iterative method on s processors ($t_c = 0.1 \times t_f$, $t_s = 5.0 \times t_f$)

tional granularity and minimization of shared data values to be communicated among the processors due to this particular partitioning of iterations among s processors. Moreover, the dependence of the algorithm on the longest chain length p_m can be minimized by suitably dividing the iterations for the q random links among s processors instead of their equal division as was done for the purpose of this analysis.

6.4.3. Simulation results

In the previous section, the speed-up and efficiency of parallel processing using s processors has been studied by deriving analytical expressions for the computation times.

Since the iterations in (6.1) are decoupled from each other, the number of processors larger than s and upto a maximum of K processors can be employed to perform these parallel computations. However, the resulting efficiencies of parallel computation will be much smaller due to the fact that the communication and synchronization overhead now will form a larger fraction of the total processing time. This is due to the smaller granularity of computation and increase in the shared data to be communicated among the processors. In order to study this degradation in algorithm performance with the increase in the number of processors, simulations for performing these parallel computations were carried out for an 8-node network example and a 64-node network example as described below.

6.4.3.1. An 8-node network example

Figure 6.7 shows the schematic diagram of a tank filled with water at high temperature and pressure and its 8-node and 8-link representation [6]. Parameters and initial conditions associated with the nodel model are given in Table 6.1 and are the same as given in [6] except that they have been converted to *MKS* system of units. The transient is initiated

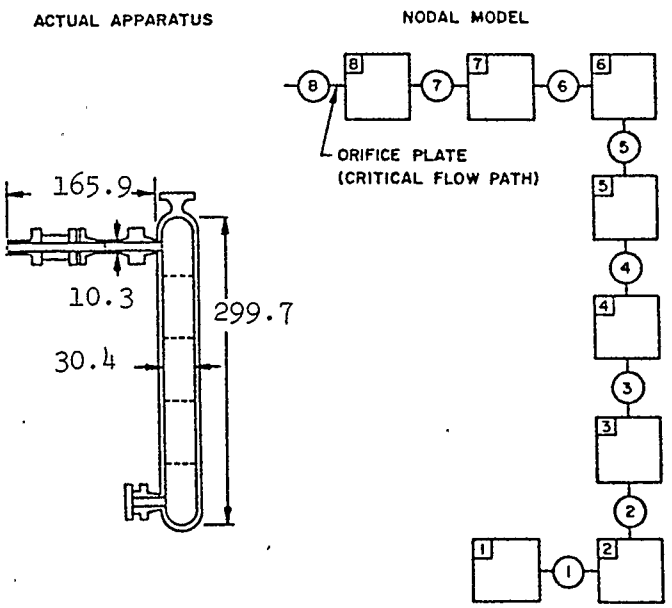


Figure 6.7 Schematic of pressurized tank and control volume representation. (dimensions are in centimetres)

Table 6.1

Parameters and initial conditions for the 8-node network of Fig. 6.7

Node Number	Volume (meter^3)	Initial enthalpy (K Joule/Kg)	Initial pressure (Kg/cm^2)
1	2.83×10^{-3}	1181.7	164.30
2	4.24×10^{-2}	1181.7	164.28
3	4.24×10^{-2}	1181.7	164.23
4	4.24×10^{-2}	1181.7	164.19
5	4.24×10^{-2}	1181.7	164.13
6	4.24×10^{-2}	1181.7	164.09
7	4.46×10^{-3}	1181.7	164.11
8	4.46×10^{-3}	1181.7	164.11

(a) Nodal Data

a

Table 6.1. Continued

Link Number	L/A (meter^{-1})	Initial Flow (Kg/Sec)	Flow Area (meter^2)	$\frac{(fr)L}{D}$
1	63.97	0	8.36×10^{-3}	0.06
2	8.00	0	7.29×10^{-2}	0.04
3	8.00	0	7.29×10^{-2}	0.04
4	8.00	0	7.29×10^{-2}	0.04
5	8.00	0	7.29×10^{-2}	0.04
6	32.15	0	8.36×10^{-3}	0.06
7	63.97	0	8.36×10^{-3}	0.015
8	0.0	0	3.14×10^{-6}	0.0

(b) Link Data

by opening the 2mm diameter disc in link 8 at time $t = 0$. For this network, the matrix A in equation $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$ assumes the form of a tridiagonal matrix. Although the direct method of solving the matrix equation will be faster for this example due to tridiagonal matrix structure and its smaller size, Jacobi's iterative method has been used to solve this problem to study the performance of this parallel algorithm.

In order to compute the values of different variables such as pressure and its derivatives with respect to other variables, steam flow data from steam tables [85] was stored in the form of polynomials fitted to the steam table data. The following times for the arithmetic operations which roughly correspond to INTEL 8086 processor with 8087 co-processor [86] were used in the simulations:

floating-point addition/subtraction time =	20 μsec
floating-point multiplication time =	20 μsec
floating-point divide time =	40 μsec
absolute value computation time =	2 μsec
integer decrement/increment time =	0.4 μsec

A time-step of 100 milliseconds was chosen for the integration of differential equations and simulations for 10 seconds of real-time were carried out. In the solution of $\Delta\,\mathbf{W}$ from the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$ by Jacobi's iterative method, iterations were terminated when $\left|\Delta W_{new}(i) - \Delta W_{old}(i)\right|$ was less than or equal to 10^{-3} for all the flow rate increments. In order to study the effect on speed-up and efficiency, the number of processors was increased from one to eight, the communication time t_c was set equal to $t_c = 0$, $t_c = 0.1 \times t_f$ and $t_c = 0.5 \times t_f$, the synchronization time t_s was set equal to $t_s = 0$,

$t_s = 5 \times t_f$ and $t_s = 50 \times t_f$.

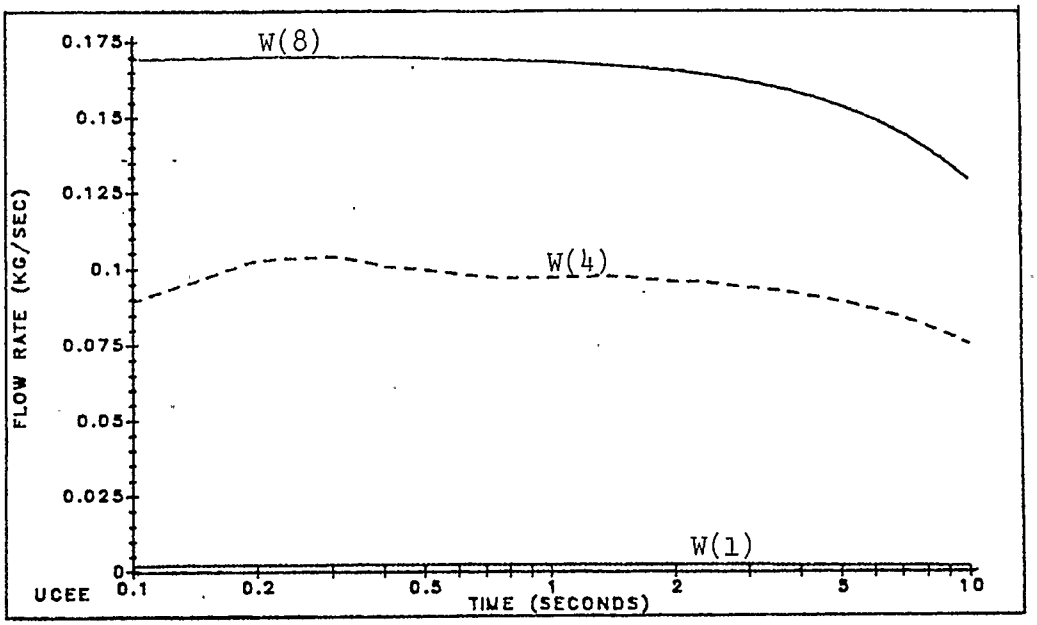
The results obtained from the simulations are plotted in Figs. 6.8 through 6.10. Fig. 6.8 shows the variation of flow rates $W(1)$, $W(4)$ and $W(8)$ in links 1, 4 and 8 respectively and pressure $P(8)$ of node 8.

Figure 6.9 shows the normalized computation time for solving the problem when the number of processors is varied from one to eight. Figure 6.9(a) depicts the effect of communication time t_c. Parallel computation time for this example is not very sensitive to the communication time. This is due to the fact that only a few data values are to be communicated among the processors because of the smaller size of the problem. Figure 6.9(b) shows the effect of synchronization time t_s on the parallel processing time which is significant due to the smaller granularity of parallel computation of this algorithm.

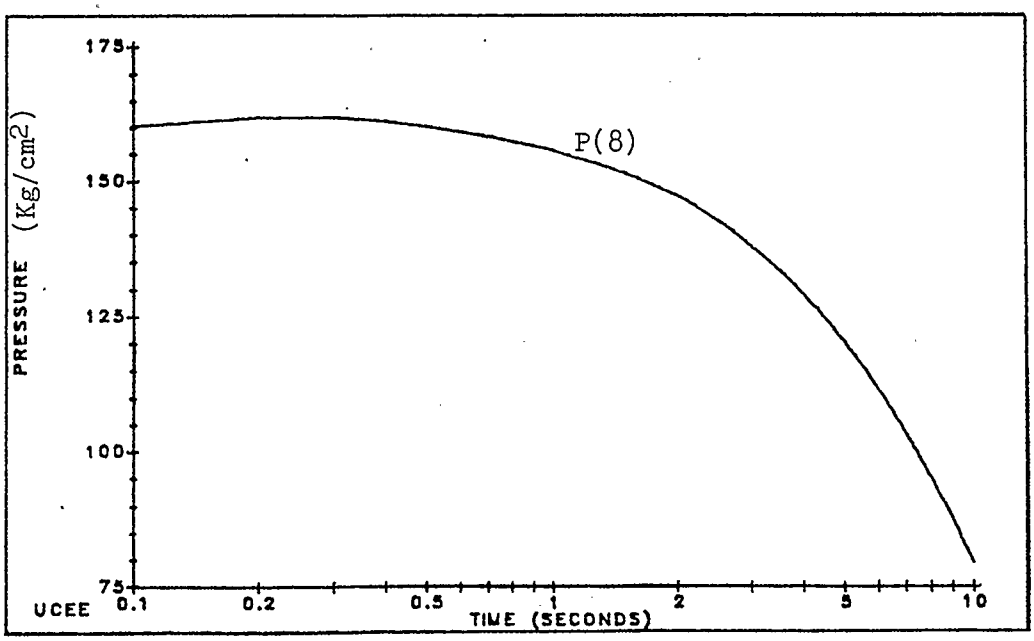
Figure 6.10 shows the speed-ups and efficiencies obtained when the number of processors is increased from one to eight. The efficiency goes down as the number of processors is increased. This is due to the fact that communication and synchronization overhead form a larger fraction of the total processing time as the number of processors is increased. Also, the plots are not smooth due to the unequal distribution of computing load on the processors for certain choices of number of processors.

6.4.3.2. A 64-node network example

In order to study the effect of problem size on the performance of the algorithm, the number of nodes in the network of Fig. 6.7 representing the pressurized water tank was increased from 5 nodes to 61 nodes so that the total number of nodes in the network was 64 instead of 8. The simulations were repeated for this larger network by varying the communication time t_c, synchronization time t_s, and the number of processors employed.

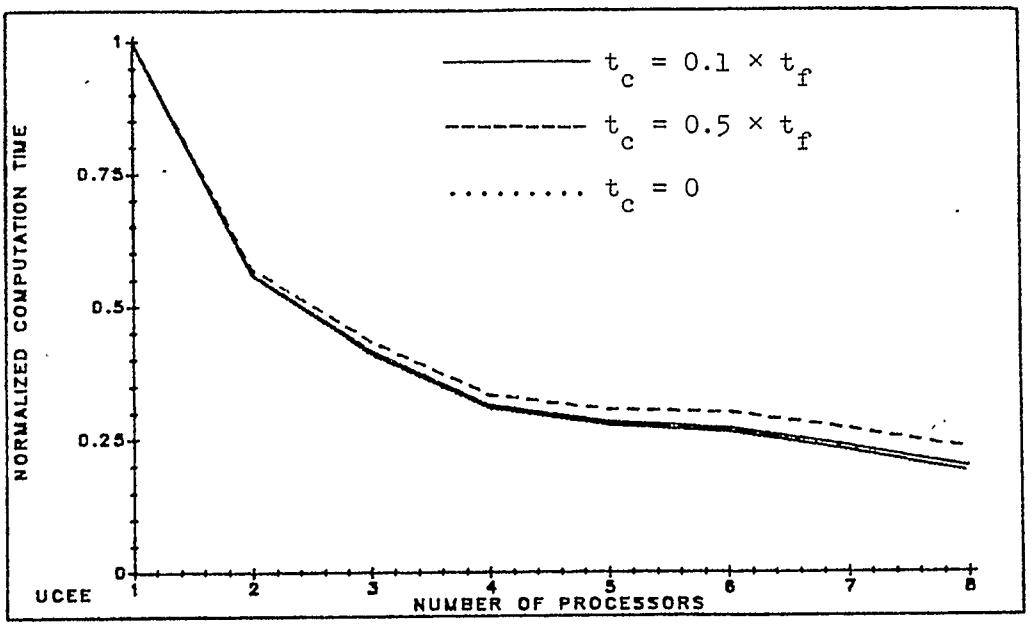


(a) Flow rate versus Time

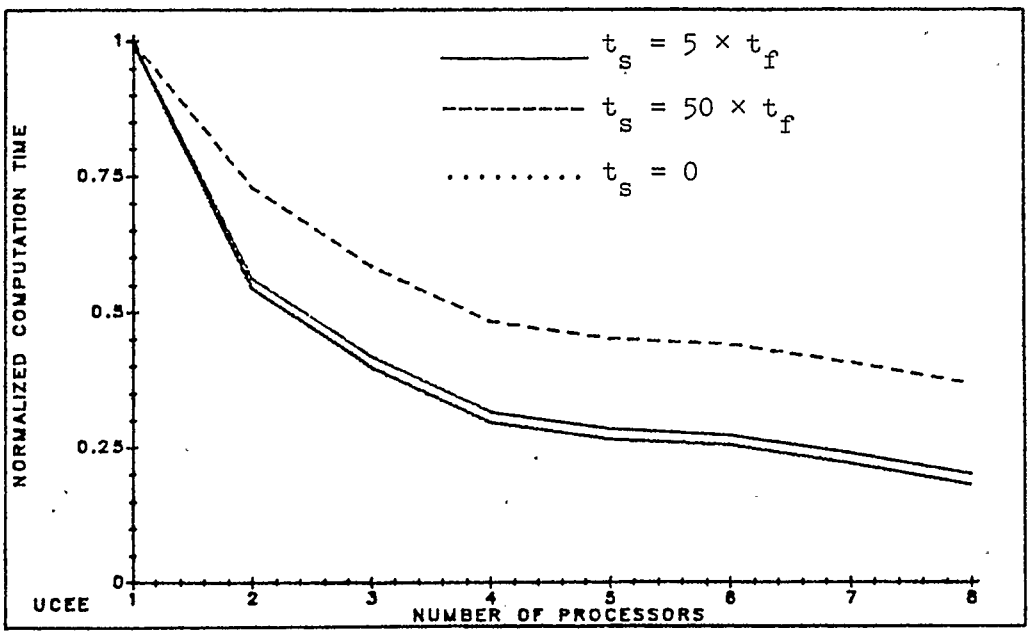


(b) Pressure versus time

Figure 6.8 Variation of link flow rates $W(1)$, $W(4)$ and $W(8)$ and node pressure $P(8)$ with time for the 8-node network example (leak diameter 2mm)

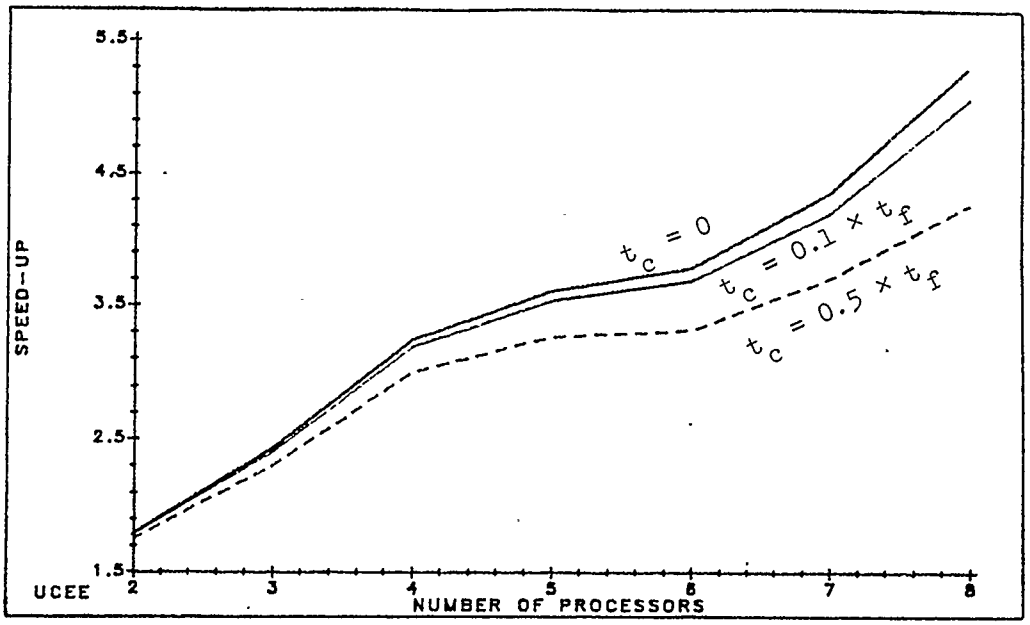


(a) Effect of Communication time t_c ($t_s = 5 \times t_f$)

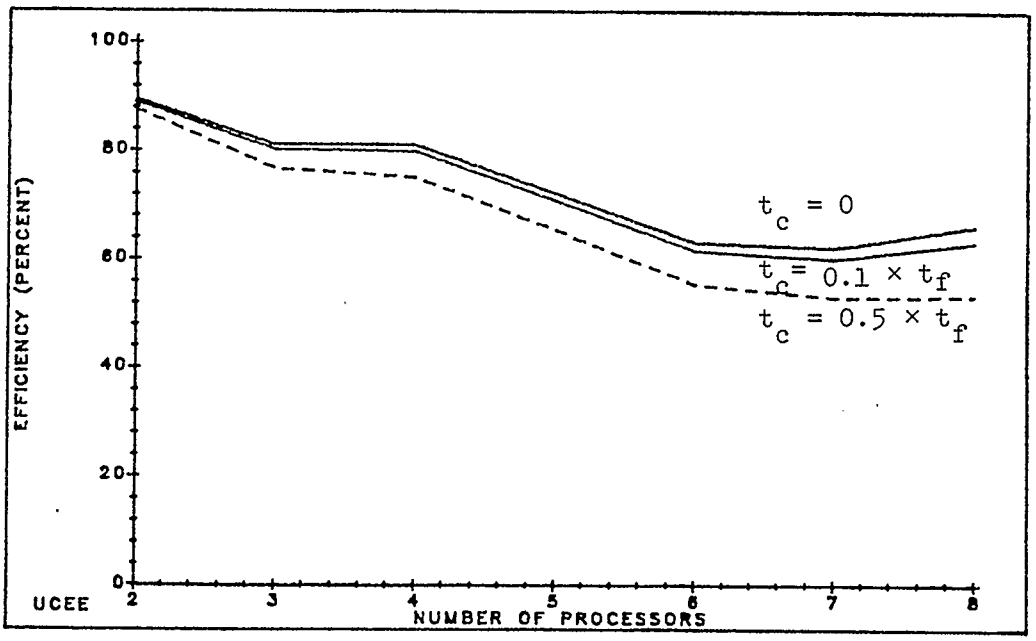


(b) Effect of synchronization time t_s ($t_c = 0.1 \times t_f$)

Figure 6.9 Normalized parallel computation time versus number of processors for the 8-node network (matrix solution by Jacobi's iterative method)



(a) Speed-up versus number of processors

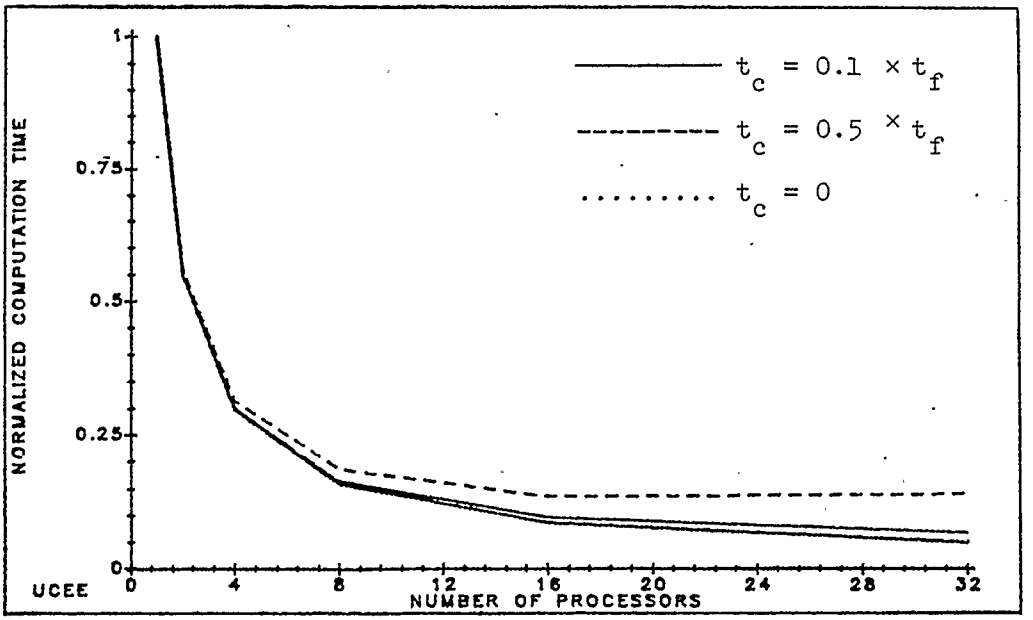


(b) Efficiency versus number of processors

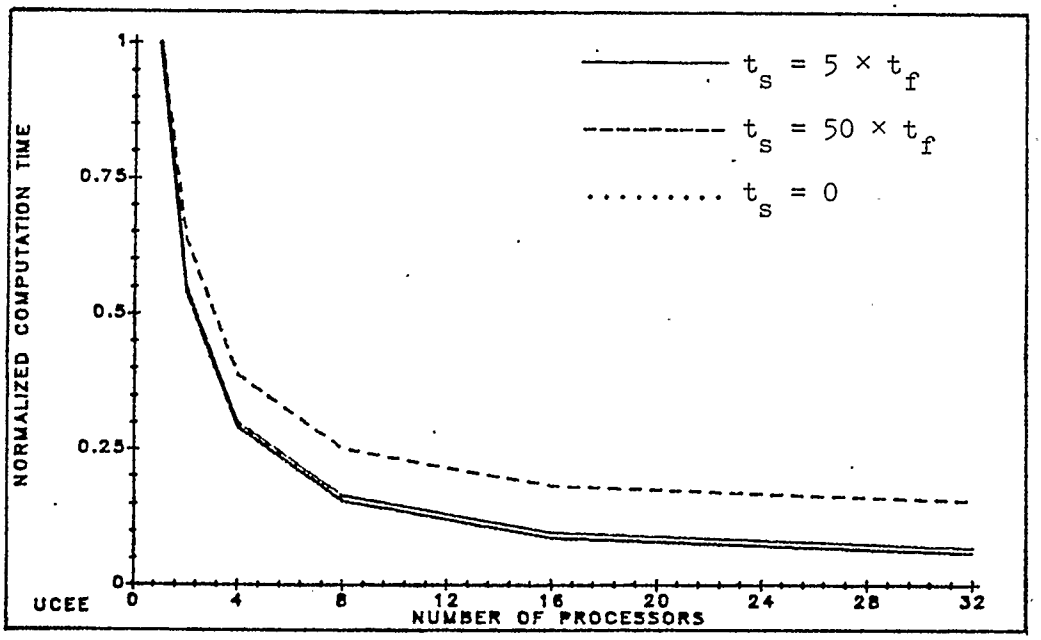
Figure 6.10 Speed-up and efficiency of parallel processing for the 8-node network example. Matrix solution is by Jacobi's iterative method ($t_s = 5 \times t_f$)

The simulation results obtained are plotted in Fig. 6.11 through 6.13. As shown in Fig. 6.12, the effect of communication time t_c on the performance of the algorithm is now more serious than the 8-node example. This is due to the fact that larger number of data values are to be communicated among the processors when the number of processors is increased from 2 to 32. Effect of communication and synchronization overhead on computing efficiency can also be seen from Fig. 6.13. When the number of processors is small, synchronization and communication overhead do not affect the efficiency very much. However, when the number of processors approaches the size of the matrix A, synchronization and communication overhead become very significant with respect to computing time and the efficiency of parallel processing becomes low. This situation is worse for larger-size networks (say, 500 link network) due to the larger data communication requirements. Hence, although number of processors equal to the size of the matrix A can be employed to perform the Jacobi's iterations, it is not a good choice from the hardware utilization point of view. The choice of number of processors equal to the number of chains s in the network ensures that the synchronization and communication overhead do not form a significant fraction of the computing time and thus resulting in good computing efficiency as was shown in section 6.4.2. (Figs. 6.4 through 6.6).

The synchronization overhead in the Jacobi's iterative method can be avoided by using a shared memory-based architecture in which the processors read the shared variables from the shared memory and write the updated shared variables of iteration (6.1) to the shared memory. The processors do not wait for other processors to finish the updating of their assigned variables. Such algorithms which do not require any synchronization for their operation on multiple processor systems are termed asynchronous iterative algorithms by Kung in reference [87]. However, since the communication patterns become non-

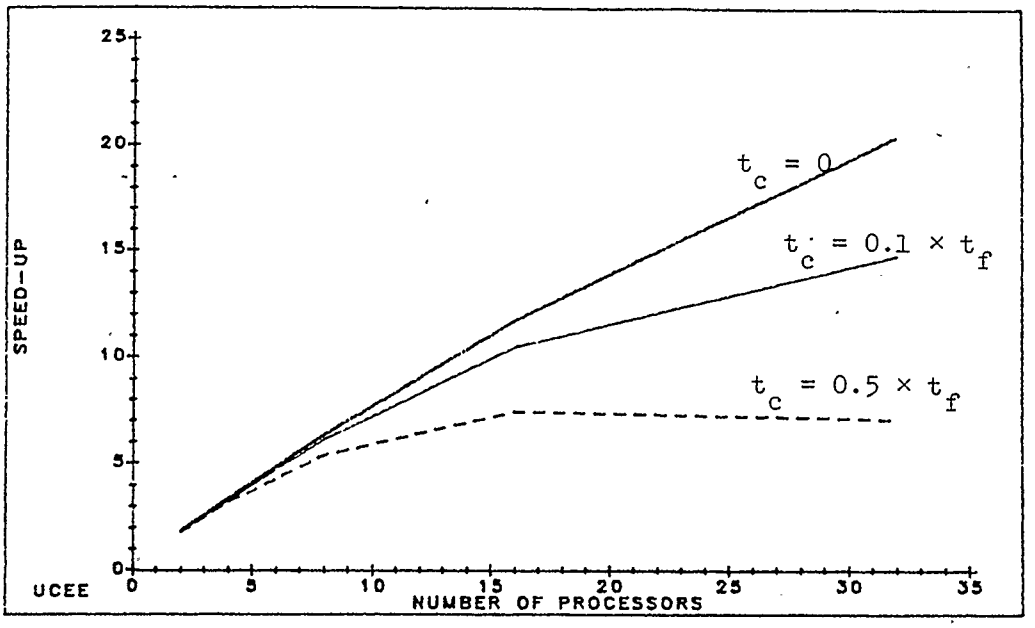


(a) Effect of communication time t_c ($t_s = 5 \times t_f$)

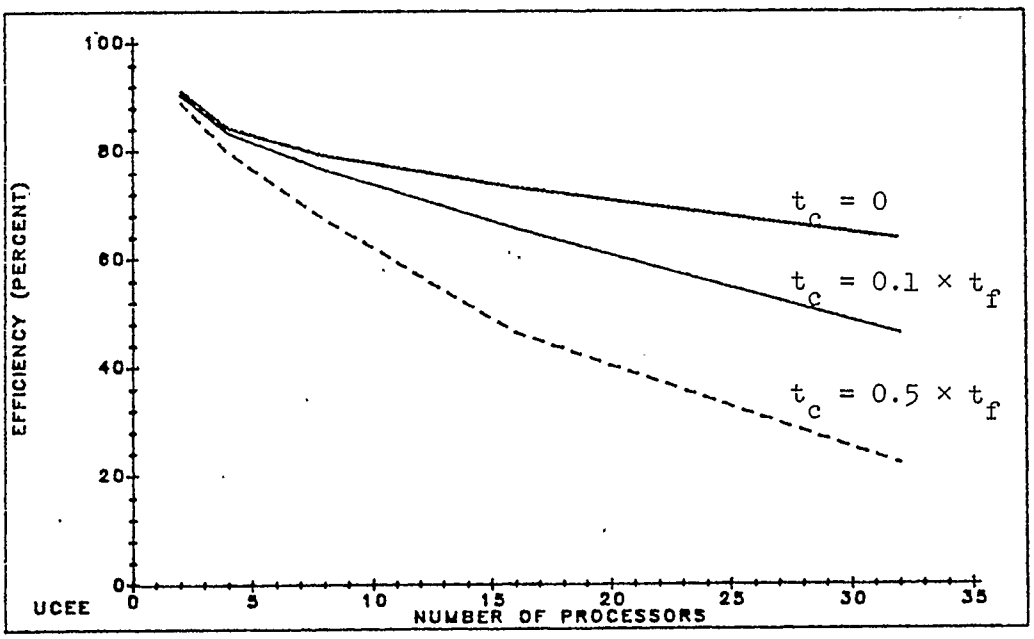


(b) Effect of synchronization time t_s ($t_c = 0.1 \times t_f$)

Figure 6.11 Normalized parallel computation time versus number of processors for the 64-node example. (Matrix solution by Jacobi's iterative method)

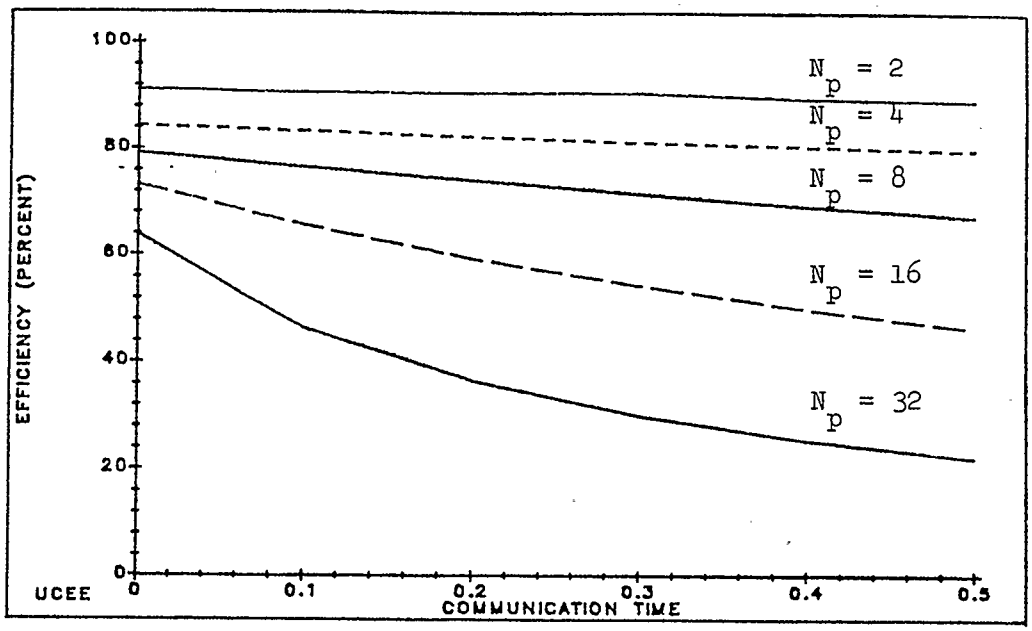


(a) Speed-up versus number of processors

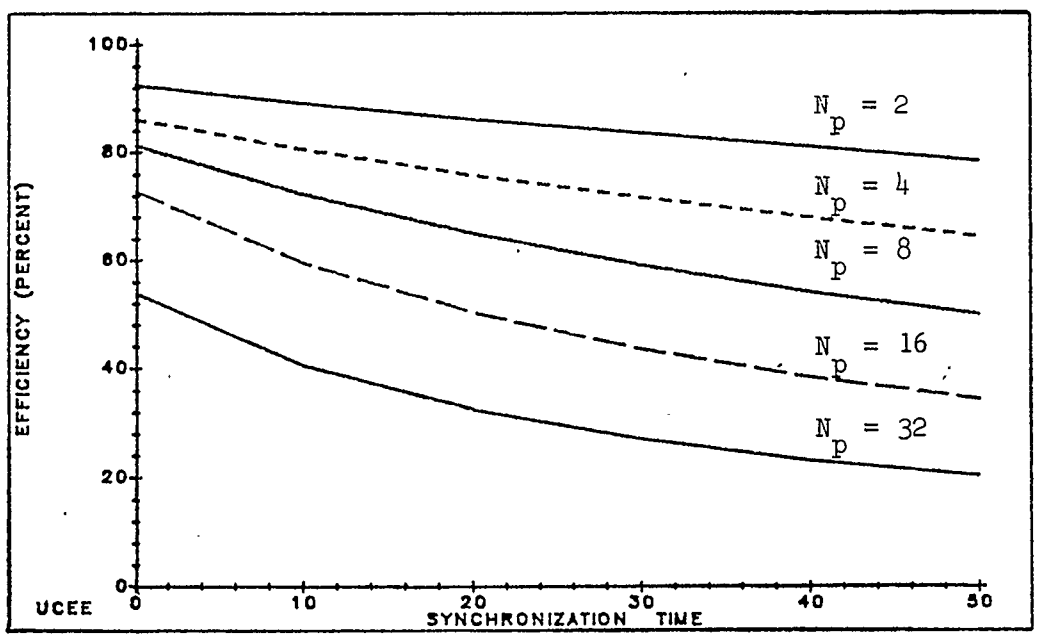


(b) Efficiency versus number of processors

Figure 6.12 Speed-up and efficiency of parallel processing for the 64-node example. Matrix solution is by Jacobi's iterative method ($t_s = 5 \times t_f$)



(a) Effect of communication time t_c on efficiency ($t_s = 5 \times t_f$)



(b) Effect of synchronization time on efficiency ($t_c = 0.1 \times t_f$)

Figure 6.13 Effect of communication and synchronization times on the efficiency of parallel computation for the 64-node network. Matrix solution is by Jacobi's iterative method using N_p processors.

deterministic in the case of asynchronous iterative algorithm, a more powerful communication network than the simple time-shared bus is required in order to minimize delays due to contentions for the same communication paths. Moreover, since the iterations generated by an asynchronous iterative algorithm do not satisfy any recurrence relations such as (6.1) for the Jacobi's algorithm, it is difficult to obtain conditions for convergence such as given in (6.3) for the case of Jacobi's algorithm [87]. Hence, iterations for certain problems may not converge at all.

CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER WORK

7.1. Conclusions

In this thesis, parallelization of Porsching's numerical integration algorithm for integrating thermo-hydraulic network differential equations which govern the dynamic behaviour of these networks, has been studied. Parallel solution of these differential equations is necessary in order to satisfy the real-time requirements for applications such as the development of operator training simulators and interactive computing requirements of design and development workstations. The following conclusions can be drawn from this study:

1. For large networks, the solution of the matrix equation $\mathbf{A}\,\Delta\,\mathbf{W} = \mathbf{z}$ at each integration time-step is the major computational effort in the serial Porsching's algorithm. The remaining computations form a small fraction of the total computing time. Therefore, this equation needs to be solved in a very fast and efficient manner. Two direct parallel methods and one iterative parallel method for solving this equation have been described in this thesis. The iterative method is to be preferred over direct methods only for simulating very large networks.

2. Even though the remaining computations form a small fraction of the total computing time, it is still very important to parallelize these computations if good overall parallel processing performance is required. This is due to the fact that these remaining computations begin to dominate the total computing time if the matrix solution only is

speeded up by using a large number of processors. This is a direct consequence of Amdahl's law.

3. The quasi-MIMD mode of parallel computation employing global synchronization and simple time-shared bus with broadcast facility as the inter-connection network and distributed communication memories is a good model of parallel computation for this problem as can be seen from the speed-up and efficiency results given in Chapters 5 and 6. This solution has become all the more attractive with the recent availability of fast 32-bit microprocessors with on-chip floating-point computations facility in hardware.

4. Partitioning of the Porsching's algorithm on the number of processors equal to the number of chains in the network is a good natural choice if high computing efficiency or hardware utilization is the primary requirement. This high efficiency results from localization of much of the data among the respective processors and from higher granularity of computation.

5. Number of processors equal to the number of random links in the network is also a good choice if high speed-up rather than high efficiency is the main consideration.

6. Even higher speed-ups can be achieved by employing a mesh-connected two-dimensional array of processors connected as a peripheral device for solving the matrix equation. However, the resulting overall speed-up is not at all proportional to the additional very large, although simpler, number of processors employed. This is again due to the fact that the remaining computations in the algorithm, which now form a significant fraction of the total processing time, are not speeded up on the array. Also the amount of the total data communication in the problem solution

increases substantially. This parallel solution, therefore, is not very attractive at present but may be employed after the development of wafer-scale integration technology where a number of processors can be integrated on a single chip.

7. Potential for real-time performance does exist. The smallest step-size that can be used in the integration algorithm for real-time computations using one MFLOP microprocessors is given in Chapter 5.

7.2. Recommendations for Further Work

Recommendations for further work in this area are given below:

1. The study of parallel algorithms reported in this thesis is only the first step in the actual implementation of these ideas on a working parallel machine. The verification of these ideas can be carried out only if a parallel computer based on the model of computation described in this thesis is actually built and the parallel software development tools such as a suitable parallel programming language for expressing these parallel algorithms, its compiler, debugger etc. are made available.

2. It will be interesting to see how much improvement in the performance of these parallel algorithms can be achieved if a more powerful interconnection network such as a hypercube or an Omega network is employed for interconnecting the processors. Since the communication patterns now would be asynchronous in nature, accurate analytical results cannot be obtained in this case. Therefore, simulations of actual network examples will be necessary in order to study the performance of the algorithms on processors connected by concurrent networks. Upper bounds on the performance of these parallel algorithms are given in Chapter 5. These have been obtained by assuming zero communication time in the total processing time.

3. In the parallel iterative method for solving the matrix equation $\mathbf{A}\ \Delta\ \mathbf{W} = \mathbf{z}$ described in Chapter 6, Jacobi's parallel method which generally converges more slowly than the Gauss-Siedel iterative method and also requires global synchronization at each pass of the iterative process has been used. A parallel iterative process called the "chaotic relaxation" method [87] which is similar to the Jacobi's method but which does not wait for results from other processors (and therefore, does not require any synchronization) up to a number of iterations can be tried. This can be done efficiently only on the shared memory machine with concurrent communication network in order to minimize contention caused by random communication patterns.

4. Porsching's integration algorithm is of the implicit type and, therefore, permits a larger step-size in the integration process but requires the solution of a matrix equation at each time-step. Another approach for dealing with this type of stiff problem having widely separated time-constants is the partitioning approach suggested by Palusinski et. al. [88]. This involves partitioning the system into a slow and a fast subsystem and then solving these resulting systems in parallel by explicit integration methods. Smaller step sizes can be used for the fast sub-system and larger time-step sizes can be used for the slower system. However, the authors report that the problem of partitioning the system without actually determining the eigenvalues of the linearized system is not yet solved in general. They also report that large errors may be introduced due to the use of interpolated values for the slow components in the fast components. If these and related problems can be solved, then this approach appears to be quite attractive specially from the point of view of parallel processing.

REFERENCES

[1] Maslo Ronald M., "A Recommended Alternative for Economic Simulation of Accident Dynamics for Light-Water Reactor Systems" in *Simulation Methods for Nuclear Power Systems,* EPRI, WS-81-212, May 1981.

[2] Resecky R.J. and Heck F.M., "An Integrated Approach Toward Simulator Utilization", in *Simulation Methods for Nuclear Power Systems,* EPRI, WS-81-212, May 1981.

[3] Kelber Charles N., "Advances in Safety Technology and Safety", in *Simulation Methods for Nuclear Power Systems,* EPRI, WS-81-212, May 1981.

[4] Steven Alain P., "Improvement of Training Simulator Capability in Light of TMI-2", in *Simulation Methods for Nuclear Power Systems*", EPRI, WS-81-212, May 1981.

[5] Cheng H.S., et al., "Boiling Water Reactor Plant Analyzer Development at Brookhaven National Laboratory", *Nuclear Science and Engineering,* 92, 1986, pp. 144-156.

[6] Porsching T.A., et al., "Stable Numerical Integration of Conservation Equations for Hydraulic Networks", *Nuclear Science and Engineering,* 43, 1971, pp. 218-225.

[7] Fischetti Mark A., "March of the Memories", *The Institute (Supplement to IEEE Spectrum),* Volume II, No. 6, June 1987, pp. 9.

[8] Mead C.A. and Conway L.A., "*Introduction to VLSI Systems*", Addison-Wesley, USA, 1980.

[9] Milutinovic Veljko, et al., "Architecture/Compiler Synergism in GaAs Computer Systems", *Computer,* Vol. 20, No. 5, May 1987, pp. 72-93.

[10] Milutinovic Veljko, "GaAs Microprocessor Technology", *Computer,* Vol. 19, No. 10, October 1986, pp. 10-13.

[11] Arrathoon R., "Digital Optical Computing: Possibilities and Pitfalls", *SPIE, Vol. 564, Real Time Signal Processing VIII,* 1985, pp. 108-118.

[12] Hillis Daniel W., "*The Connection Machine*", The MIT Press, Cambridge, MA, 1985.

[13] Barnes G.H., et al., "The ILLIAC IV Computer", *IEEE Transactions on Computers,* Vol. C-17, No. 8, August 1968, pp.746-757.

[14] Bucher Ingrid Y., "The Computational Speed of Supercomputers", *The Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems,* August 1983, pp. 151-165.

[15] Hockney R.W. and Jesshope C.R., "*Parallel Computers: Architecture, Programming and Algorithms*", Adam Hilger Ltd., Bristol, England, 1981.

[16] Misako Ishiguro and Harada Hiroo, "Vectorization of the Light Water Reactor Transient Analysis Code RELAP5", *Nuclear Science and Engineering*, 92, 1986, pp. 126-135.

[17] Newton A.R. and Sangiovanni-Vincentelli A.L., "Computer-Aided Design for VLSI Circuits", *Computer*, Vol. 19, No. 4, April 1986, pp. 38-60.

[18] This CPU Does Floating-Point Faster Than Any Two-Chip Set", *Electronics*, November 27, 1986, pp. 51-55.

[19] Gear S.W., "*Numerical Initial Value Problems in Ordinary Differential Equations*", Prentice-Hall, Inc., New Jersey, 1971.

[20] Hoshino T., "An Invitation to the World of PAX", *Computer*, Vol. 19, No. 5, May 1986, pp. 68-79.

[21] Kober R., et al., "*SMS 101 - A Structured Multi-microprocessor System with Deadlock-Free Operation Scheme*", Euromicro 1976, North-Holland Publishing Co., Amsterdam, pp. 56-64.

[22] Wallach Y., "Alternating Sequential/Parallel Processing", *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS-100, No. 11, November 1981, pp. 4397-4401.

[23] Tuazon J., et al., "CALTECH/JPL Mark II Hypercube Concurrent Processor", *Proceedings of IEEE International Conference on Parallel Processing*, St. Charles, Illinois, August 1985, pp. 666-673.

[24] Seitz Charles, "The Cosmic Cube", *Communications of the ACM*, Vol. 28, No. 1, January 1985, pp. 22-33.

[25] Gottlieb Allan, et al., "The NYU Ultra-Computer-Designing an MIMD Shared Memory Parallel Computer", *IEEE Transactions on Computers*, Vol. C-32, No. 2, February 1983, pp. 175-189.

[26] Chua Leon O. and Lin Pen-Min, "*Computer Aided Analysis of Electronic Circuits: Algorithms & Computational Techniques*, Prentice-Hall, Inc., 1975.

[27] Kuo Benjamin C., "*Automatic Control Systems*", Prentice-Hall, Inc., 1975, pp. 118-122.

[28] Porsching T.A., et al., "FLASH-4: A Fully Implicit Fortran IV Program for the Digital Simulation of Transients in a Reactor Plant", WAPD-TM-840, *Bettis Atomic Power Laboratory*, 1969.

[29] Staff Report, "RELAP 4/MOD 5, A Computer Program for Transient Thermal-Hydraulic Analysis of Nuclear Reactors and Related Systems - User's Manual", *RELAP 4/MOD 5 Description,* ANC-NUREG-1335, Vol. 1, Idaho National Engineering Laboratory, 1976.

[30] Staff Report, "RETRAN, A Program for One-Dimensional Transient Thermal-Hydraulic Analysis of Complex Fluid Systems", *Compter Code Manual,* EPRICCM-5, Vol. 1, Electric Power Research Institute, Palo Alto, CA, 1978.

[31] Staff Report, "RETRAN-02, A Program for One-Dimensional Transient Thermal-Hydraulic Analysis of Complex Fluid Systems", *Equations and Numerics,* NP-1850-CCM, Vol. 1, EG & G Idaho Inc., 1981.

[32] Lin Cherng-Shing and Kastenberg William E., "A Practical Nonhomogeneous Two-Phase Flow Model for Light Water Reactor System Codes", *Nuclear Science and Engineering:* 86, 1984, pp. 388-400.

[33] Sandberg I.W. and Shichman H., "Numerical Integration of Systems of Stiff Nonlinear Differential Equations", *The Bell System Technical Journal,* April 1968, pp. 511-527.

[34] Duff Iain S., "A survey of Sparse Matrix Research", *Proceedings of the IEEE,* Vol. 65, No. 4, April 1977, pp. 500-535.

[35] Burden Richard L., Faires J.D. and Reynolds A.C., "*Numerical Analysis*", Prindle, Weber and Schimidt, Boston, MA, 1981.

[36] Stark Peter A., "*Introduction to Numerical Methods*", The Macmillan Company, London, 1970, pp. 170-172.

[37] Skears J. and Toong T., "*SOPHT Programmers Manual*", Ontario-Hydro, Toronto, May 1975.

[38] Flynn M.J., "Very High-Speed Computing Systems", *Proceedings IEEE, Vol. 54, 1966, pp. 1901-1909.*

[39] Flynn M.J., "Some Computer Organizations and Their Effectiveness", *IEEE Trans. on Computers,* C-21, No. 9, Sept. 1972, pp. 948-960.

[40] Feng T.Y., "Some Characteristics of Associative/Parallel Processing", *Proc. 1972 Sagamore Computer Conference,* Syracuse University, 1972, pp. 5-16.

[41] Shore J.E., "Second Thoughts on Parallel Processing", *Computers and Electrical Engineering 1, 1973, pp. 95-109.*

[42] Handler W., "The Impact of Classification Schemes on Computer Architecture", *Proc. 1977 International Conference on Parallel Processing,* pp. 7-15.

[43] Barnes, G.H., et.al., "The ILLIAC IV Computer", *IEEE Trans. on Computers,* Aug. 1968, pp. 746-757.

[44] Bouknight W.J., et al., "The ILLIAC IV System", *Proc. IEEE,* Vol. 60, No. 4, April 1972, pp. 369-388.

[45] Reddaway S.F., "The DAP Approach", *Infotech State of the Art Report: Super Computers,* Vol. 2, 1979

[46] Batcher K.E., "Design of a Massively Parallel Processor", *IEEE Trans. on Computers;* C-29, Sept. 1980, pp. 836-840.

[47] Wulf W.A. and Bell C.G., "C.mmp - a multi-mini-processor", *AFIPS Conference Proceedings,* Vol. 41, Part II, FJCC 1972, pp. 765-777.

[48] Fuller S.H., et al., "Multi-microprocessors: An Overview and Working Example", *Proceedings of the IEEE,* Vol. 66, No. 2, February 1978, pp. 216-228.

[49] Kober R. and Kuznia C., "SMS 201 - A Powerful Parallel Processor with 128 Microprocessors", *Euromicro Journal 5,* 1979, pp. 48-52.

[50] Smith B.J., "Architecture and Application of the HEP Multiprocessor Computer System", *Proc. of Real-Time Signal Processing IV,* SPIE, August 1981, pp. 241-248.

[51] Hoshino T., et al., "PACS: A Parallel Microprocessor Array for Scienctific Calculations", *ACM Transactions on Computer Systems,* Vol. 1, No. 3, August 1983, pp. 195-221.

[52] Gajski D., et al., "Cedar", *Proc. of Compcon,* Spring 1984, pp. 306-309.

[53] Crowther W., et al., "Performance Measurements on a 128-node Butterfly Parallel Processor", *Proc. IEEE International Conference on Parallel Processing,* August 1985, pp. 531-540.

[54] Pfister G.F., et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. IEEE International Conference on Parallel Processing,* August 1985, pp. 764-771.

[55] Dennis J.B., "Data Flow Supercomputers", *Computer,* November 1980, pp. 48-56.

[56] Bucher I.Y., "The Computational Speed of Supercomputers", *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems,* August 1983, pp. 151-165.

[57] Feng T.Y., "A Survey of Interconnection Networks", *Computer*, December 1981, pp. 12-27.

[58] Bokhari S.H., "On the Mapping Problem", *IEEE Trans. on Computers*, Vol. C-30, No. 3, March 1981, pp. 207-214.

[59] Stone Harold S., "Parallel Processing with the Perfect Shuffle", *IEEE Trans. on Computers*, Vol. C-20, No. 2, February 1971, pp. 153-161.

[60] Lawrie D.H., "Access and Allignment of Data in an Array Processor", *IEEE Trans. on Computers*, C-24, No. 12, December 1975, pp. 1145-1155.

[61] Kruskal C. and Snir M., "The Performance of Multi-stage Interconnection Networks for Multiprocessors", *IEEE Trans. on Computers*, Vol. C-32, December 1983, pp. 1091-1098.

[62] Pfister G.F. and Norton V.A., ""Hot Spot" Contention and Combining in Multistage Interconnection Networks", *Proc. IEEE International Conference on Parallel Processing*, August 1985, pp. 790-795.

[63] Kober R., Kopp H. and Kuznia Ch., "SMS 101 - A Structured Multimicroprocessor System with Deadlock-Free Operation Scheme", *Euromicro Newsletter*, 2, 1976, pp. 56-64.

[64] Kober R., "The Multiprocessor System SMS 201 - Combining 128 Microprocessors to a Powerful Computer", *Digest of Papers, COMPCON*, Fall 1977, pp. 225-229.

[65] Kopp H., "Numerical Weather Forecast with the Multi-Microprocessor System SMS-201", *Proc. IMACS-GI Symposium on Parallel Computers and Parallel Mathematics*, Munich, March 1977, pp. 265-268.

[66] Kober R. and Kuznia Ch., "SMS - A Multiprocessor Architecture for High Speed Numerical Calculations", *Proc. IEEE International Conference on Parallel Processing*, 1978, pp. 18-24.

[67] Hoshino T., et al., "Parallel Processing for Scientific Applications", *Proc. International Symposium on Applied Mathematics and Information Sciences*, Kyoto University, March 1982, pp. 7-17 to 7-26.

[68] Hoshino T., et al., "Highly Parallel Processor Array "PAX" for Wide Scientific Applications", *Proceedings IEEE International Conference on Parallel Processing*, August 1983, pp. 95-105.

[69] Hoshino T., et al., "Super Freedom Simulator PAX", *VLSI Engineering: Beyond Software Engineering*, Editied by Kunii T., 1984, pp. 39-51.

[70] Amdahl G.M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *Proc. AFIP*, Vol. 30, Thompson, Washington D.C., 1967, pp. 483-485.

[71] Ware W., "The Ultimate Computer", *IEEE Spectrum*, March 1972, pp. 84-91.

[72] Nagel K., "Solving Linear Equations with the SMS-201 Parallel Processor", *Euromicro Journal 5*, 1979, pp. 53-54.

[73] Kant Rajani M. and Kimura T., "Decentralized Parallel Algorithms for Matrix Computation", *The 5th Annual Symposium on Computer Architecture*, 1978, pp. 96-100.

[74] Kung H.T. and Leiserson C.E., "Systolic Arrays (for VLSI)", *Sparse Matrix Proc. 1978*, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

[75] Horowitz Ellis, "VLSI Archtectures for Matrix Computations", *Proc. IEEE International Conference on Parallel Processing*, 1979, pp. 124-127.

[76] Fawcett James W., "Solutions of Large Sets of Linear Equations Using Cellular Arrays", *Proc. 24th Midwest Symposium on Circuits and Systems*, Albuquerque, New Mexico, 1981, pp 527-531.

[77] Kung S.Y., et al., "Wavefront Array Processor: Languages, Architectures and Applications", *IEEE Trans. on Computers*, Vol. C-31, No. 11, November 1982, pp. 1054-1066.

[78] Mead C.A. and Conway L., "*Introduction to VLSI Systems*", Addison-Wesley, MA, 1980.

[79] Seitz, Charles L., "Concurrent VLSI Architectures", *IEEE Trans. on Computers*, Vol. C-33, No. 12, December 1984, pp. 1247-1265.

[80] Symanski J.J., "Implementation of Matrix Operations on the Two-dimensional Systolic Array Testbed", *Proc. SPIE, Vol. 298, Real-Time Signal Processing VI, Society of Photo-Optical Instrumentation Engineers*, 1983, pp. 136-142.

[81] Kung S.Y., "VLSI Array Processors", *IEEE Acoustics, Speech and Signal Processing Magazine*, July 1985, pp. 4-22.

[82] Kung S.Y. and Gal-Ezer R.J., "Synchronous versus asynchronous Computation in Very Large Scale Integrated (VLSI) Array Processors", *SPIE*, Vol. 341, Real-Time Signal Processing V, 1982, pp. 53-65.

[83] Varga Richard S., "*Matrix Iterative Analysis*", Prentice-Hall Inc., 1972.

[84] Conrad Victor and Wallach Y., "Iterative Solution of Linear Equations on a Parallel Processor System", *IEEE Trans. on Computer*, Vol. C-26, No. 2, September 1977, pp. 838-

847.

[85] "*1967 Steam Tables*", Edward Arnold (Publishers) Limited, London, 1967.

[86] "*iAPX 86,88 Users Manual*", Intel Corporation, Santa Clara, August 1981.

[87] Kung H.T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", in *Algorithms and Complexity*, edited by Traub J.F., Academic Press, Inc., New York, 1976, pp. 153-200.

[88] Palusinski O.A., et al., "Numerical Integration Techniques using Model Partitioning", *Simulation Methods for Nuclear Power Systems*, EPRI, WS-81-212, May 1981, Section 4, pp. 55-68.