1994

A New Method for Efficient Parallel Solution of Large Linear Systems on a SIMD Processor.

Okon Hanson Akpan
Louisiana State University and Agricultural & Mechanical College

A NEW METHOD FOR EFFICIENT PARALLEL SOLUTION OF LARGE LINEAR SYSTEMS ON A SIMD PROCESSOR

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agriculture and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Okon Hanson Akpan
B.S., Maryville College, 1976
M.S., University of Tennessee, 1980
M.S., University of Southwest Louisiana, 1988
December 1994

To Jesus, the Christ. .

Acknowledgements

I am very grateful to many individuals whose support, comments, and sometimes complaints provided the needed impetus for the completion of this study. I wish to thank, in particular, my major adviser, Dr. Bush Jones whose self-less devotion and indefatigable support provided inspirational guidance throughout this work. My gratitude also goes to my minor adviser, Dr. El-Amawy who provided valuable reference materials that helped place this work on the right track from its inception, and also proferred a helping hand of support throughout its duration. I am very much grateful to Dr. S. Iyengar who gave the much needed moral and financial support without which the completion of this study would have been seriously delayed. For their excellent advise and especially their timely response to the several and at times irrepressible requests and demands, I thank Drs. SQ Zheng, D. Kraft and C. Delzell. Their patient support, selfless dedication, and devotion to details not only energized but also sustained every conceivable effort which made the successful and timely completion of this study possible. My appreciation goes to my wife, Victoria whose personal sacrifices which most of the times included many additional hours of work to eke out the family resources and thus help me to focus my entire concentration on this study. I am also appreciative of her effort in helping with the task of editing, typesetting, and, finally, of putting this dissertation together in a presentable and acceptable volume. My gratitude also goes to my children, although some were too young to understand what I was going through (in fact, would not care had they understood), neverthelesss did contribute in their little ways to maintain sanity and to thus make this work and its timely completion possible.

Table of Contents

Dedication ii

Acknowledgements iii

List of Tables vii

List of Figures viii

Symbols x

Abstract xvi

CHAPTER

1 Exploitation of Parallelism: Literature Survey 1
- 1.1 RATIONALE FOR PARALLELISM 1
- 1.2 HARDWARE PARALLELISM 4
  - 1.2.1 The Genesis of High-Performance Computers 4
    - 1.2.1.1 The Early Computers 5
    - 1.2.1.2 The Role of Technology and Innovative Design Philosophies 7
    - 1.2.1.3 A Breed of Modern High-Performance Machines 8
    - 1.2.1.4 The Advent of Array Processors 10
    - 1.2.1.5 The Future of High-Performance Computers 10
  - 1.2.2 Models of Computations and their Classifications 11
    - 1.2.2.1 Flynn's Taxonomy 12
    - 1.2.2.1 Other Taxonomies 13
- 1.3 SOFTWARE PARALLELISM 14
  - 1.3.1 Desirable Features of a Parallel Algorithm 14
  - 1.3.2 Definition of a Parallel Algorithm 16
  - 1.3.3 Common Methods for Creating Parallel Algorithms 17
  - 1.3.4 Parallelism in Numerical Algorithms 19
    - 1.3.4.1 Matrix/Vector Algorithms 20
    - 1.3.4.2 Linear Systems 21

1.4 THE PURPOSE OF THIS DISSERTATION 24

1.4.1 The Statement of Purpose.. 24

1.4.2 The Choice of Problem... 25

1.4.3 Rationale for the Choice of this Solution Approach.. 26

1.4.4 The Major Contributions by this Study.................... 27

1.5 CHAPTER SUMMARY 28

2 The Model of Computation 31

2.1 OVERVIEW OF MPP SYSTEM 32

2.1.1 Rationale for MPP Development.............................. 32

2.1.2 MPP: Its Technology and Architecture.................... 33

2.2 THE MasPar MODEL **MP-X** 34

2.2.1 The **MP-X** Hardware.. 34

2.2.2 The **MP-X** Softwares 39

2.2.3 The MasPar Programming Language (MPL) 39

2.3 CHAPTER SUMMARY 43

3 Mathematical Formulation of the Model Problem 46

3.1 ELLIPTICITY 47

3.1.1 General Characteristics of Ellipticity 47

3.1.2 Two-Dimensional Elliptic Systems............................ 48

3.1.3 Finite Difference Approximation of Elliptic Equations 50

3.2 THE MODEL PROBLEM 53

3.2.1 Properties of the Model Problem 55

3.3 CHAPTER SUMMARY 57

4 Algorithmic Tools Development 59

4.1 A BRIEF OVERVIEW OF THE MODEL PROBLEM STRUCTURE 61

4.2 DIRECT BENCHMARKS 63

4.2.1 Gaussian Elimination Method (GE) and **LU** Factorization 63

4.2.1.1 **GE & LU** Factorization for Dense Linear Systems 64

4.2.1.2 **GE & LU** Factorization for Banded Linear Systems 70

4.2.2 Orthogonal Factorization for Banded Systems 75

4.2.2.1 Householder Factorization for Dense Linear Systems 77

4.2.2.2 Householder Factorization for Banded Linear Systems 79
4.3 ITERATIVE BENCHMARKS 80
4.3.1 Symmetric Over-relaxation Method (SOR) 81
4.3.1.1 The **SOR** Method for Dense Linear Systems.. 81
4.3.1.2 The **SOR** Method for Banded Linear Systems.. 85
4.3.2 Conjugate Gradient Method (CG). 87
4.3.2.1 The **CG** Method for Dense Linear Systems 89
4.4 THE NEW TECHNIQUE 90
4.4.1 Recursive Doubling for Linear Recursions 92
4.4.2 Cyclic or Odd-Even Reduction 95
4.5 CHAPTER SUMMARY 98

5 Implementations and Experimental Results 101
5.1 DATA STRUCTURES 102
5.2 THE GRID CONSTRUCTION 105
5.3 THE EXPERIMENTS 108
5.3.1 The System Size Determination and Usage............ 108
5.3.2 Exact Solution 109
5.3.3 Experimental Approach 110
5.4 EXPERIMENTAL RESULTS 111
5.4.1 Results and Observations 112
5.4.2 Conclusion 125

Bibliography 126

Vita 136

List of Tables

4.1 Operation Count of SerGE Algorithm 67

4.2 Operation Count of SerGET Algorithm 72

5.1 Errors of Results when $n = 305,\ N = 93,025$ 114

5.2 Errors of Results when $n = 200,\ N = 40,000$ 115

5.3 Errors of Results when $n = 100,\ N = 10,000$ 116

5.4 Errors of Results when $n = 40,\ N = 1,600$ 117

List of Figures

2.1 MP-X Hardware Subsystems 34

2.2 MP-X Communication Buses 37

3.1 Mesh Region to Approximate the Domain of the Model Problem 51

4.1 Gauss Elimination for Dense Linear Systems (SerGE) 66

4.2 Parallel GE and LU Factorization for Dense Linear Systems (ParGE)... 69

4.3 Gauss Elimination for Tridiagonal Systems (SerGET) 71

4.4 Householder Reduction for Dense Linear Systems 79

4.5. Serial Jacobi Method for Dense Linear Systems (SerJAC) 83

4.6 Parallel Jacobi Method for Dense Linear Systems (ParJAC) 83

4.7 Conjugate Gradient Method for Dense Linear Systems (SerCG). 88

4.8 Recursive Doubling Technique for Linear Recursive Problems .. 94

4.9 Cyclic (Odd/Even) Reduction for the New Technique 99

4.10 The New Algorithm Computing a Tridiagonal System at Various Reduction Levels 100

5.1 Structure of Matrix A Resulting from Model Problem, $\beta = n$ 104

5.2 The Basic Data Structures and their Allocation 106

5.3 The Mesh Region and the Mesh Point Coordinates 107

5.4 The Exact Solution of the Model Problem 113

5.5 Values of U at the First 400 Grid Points when $n = 305$, $N = 93,025$ 114

5.6 Values of U at the First 400 Grid Points when $n = 200$, $N = 40,000$ 115

5.7 Values of U at the First 400 Grid Points when $n = 100$, $N = 10,000$ 116

5.8 Values of U at the First 200 Grid Points when $n = 40$, $N = 1,600$ 117

5.9 CPU Times of the Algorithms 118

5.10 Megaflop Rates of the Algorithms 119

Symbols

In the following symbol lists, the context in which some of the symbols are first used are indicated as the equation number the symbols are used. The equation number in enclosed in parentheses following explanation of the symbols' meaning. Many symbols have multiple meanings, and, in that case, we believe the context of their use will clear any ambiquity.

Arabic Letters

$\mathbf{A}$	Coefficient matrix of a system of linear equations (1.1)
$\mathcal{A}$	Coefficient matrix resulting from second order, n-dimensional PDE
A_{ij}	Coefficient of a partial differential of a second order, n-dimensional PDE when the differentiation is taken with respect to vectors $x_i,\ x_j \in \mathbb{R}^n$ (3.1)
$\acute{\mathcal{A}}$, $\acute{\mathcal{B}}$, $\acute{\mathcal{C}}$	Coefficients of second order, 2-dimensional elliptic equation when differentiations are taken with respect to x, y, and xy respectively (3.7)
$\mathbf{D}$	Diagonal matrix triangular matrix in $\mathbf{A} = \mathbf{U^TDU}$ (4.6)
$d_1, \ldots, d_5$	Diagonal vectors (5.1)
$\mathcal{F}$, f	Forcing function, the right hand sides of second order, n-dimensional elliptic equation (3.6, 3.7)
k, h	Grid widths in $x-$ and $y-$ coordinate directions
$\mathbf{k}$	Right hand side (vector) of linear system of equations

$\mathbf{L}$	Lower triangular matrix in $\mathbf{A} = \mathbf{LU}$ (4.5)
l	Reduction level (4.54)
m, n	Number of partitions of mesh region in $x-$ and $y-$ directions
m, N	Number of linear equations resulting from approximating of elliptic equation in rectangular domain; $N = n^2$
$\mathbb{N}^n$	$n-$dimensional space (3.1)
p	Diagonal matrix triangular matrix in $\mathbf{A} = \mathbf{U^T D U}$(4.6)
P_i	ith permutation matrix
$P_{i\ j}$	Product of permutation matrices P_i through P_j
$I\!R^n$	$n-$dimensional space of <u>real</u> numbers (3.1)
$\mathbf{Q}$	Orthogonal matrix $\mathbf{A} = \mathbf{QR}$ (4.14)
$\mathbf{r}$	Residual vector
u	Dependent variable of second order, n-dimensional elliptic equation (3.1)
U	An approximated value of u (3.14)
$\mathbf{U},\mathbf{R}$	Upper triangular matrix in $\mathbf{A} = \mathbf{LU}$ (4.5),and in $\mathbf{A}$=$\mathbf{LU}$ (4.14)
$\mathbf{w}$	Orthogonal factorization factor (4.15)
$\mathbf{x}$, $\mathbf{y},\mathbf{z}$	Solution vectors (4.7,4.8)
$\mathbf{Z}$	Matrix such that $\mathbf{Z} = \mathbf{I} - \mathbf{D}^{-1}$ (4.23)

Greek Letters

α, β	Scalar values in Conjugage gradient algorithm, SerCG()
β	Semi-bandwidth
Ω	A bounded domain
Γ	Boundary of domain Ω
∇	Gradient $\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n} p$
ω	Relaxation factor of SOR, SSOR, etc. $1 \leq \omega \leq 2$
$\rho(A)$	Spectral radius of A
Ψ_1, Ψ_2	Sets of coordinate points in $x-$ and $y-$ directions

Superscripts

A^{T}	Transpose of AInverse of A
$\mathbf{x}^{(l)}$	ith iterate, (l) being iteration level

Subscripts

$\mathbf{a}_{ij}$	i j^{th} element of matrix **A**
$\mathbf{v}_i$	i^{th} element of vector **v**
$\mathbf{P}_i$	i^{th} permutation matrix

Abbreviations

a, ja, ia	substructures of the "a, ja, ia" format of data
ACU	Arithmetic control unit, a control unit of MPP

ARU	Array unit, a parallel component of MPP
blk	A sequence of blocks (MPL) of plural variables seperated in memory by a fixed address increment
CG	Conjugate gradient algorithm
DPU	Data parallel unit of MPP performs all parallel processing
$det(\mathbf{A})$	Determinant of **A** ()
FE	Front end, a host computer of MPP
ixproc	PE's index in $x-$coordinate direction, $0 \leq ixproc \leq nxproc - 1$
iyproc	PE's index in $y-$coordinate direction, $0 \leq iyproc \leq nyproc - 1$
JOR	Jacobi over-relaxation method
mat	A 2-dimensional array (MPL) of arbitrary size
MPL	Maspar programming language, a MULTRIX C based language for programming MPP
MPPE	Maspar programming environment of MPP
MPP	Massively parallel processor of the MasPar corporation, Goodyear Aerospace.
MP-X	Versions of MPP where X=1 or 2. That is, MP-X $\Rightarrow$ MP-1 or MP-2
(sign)	$\pm$
nproc	Number of PEs
nxproc	Number of PEs in $x-$direction of PE-array
nyproc	Number of PEs in $y-$direction of PE-array

lnproc	nproc in bit representation
lnxproc	nxproc in bit representation
lnyproc	nyproc in bit representation
ParJAC	Parallel implementation of Jacobi iterative method
ParCR	Parallel implementation of cyclic reduction technique
ParCG	Parallel implementation of Conjugate gradient algorithm
ParGET	Parallel implementation of Gaussian elimination technique modified for tridiagonal systems
ParRD	Parallel implementation of recursive doubling technique
PE	Processing element, a processor of MPP
PMEMSZ	Size (usually in KB) of the local PE' memory. For MP-1 (model 1208B), PMEMSZ=64KB
plural	A storage class which, when used as a part of a type attribute, causes the datum with the attribute to be allocated on local memories of all processors. In other word, the datum effectively becomes a vector
RISC	Reduced instruction set computer. The PE array of MPP is a RISC processor
ROW$_i$	A set of indices of a row of matrix A allocated on or brodcast to PE$_i$
SerJAC	Serial implementation of Jacobi iterative method
SerCG	Serial implementation of Conjugate gradient algorithm
SerGET	Serial implementation of Gaussian elimination technique modified for tridiagonal systems
SOR	Symmetric over-relaxation method

SSOR	Symmetric over-relaxation method
LSOR	Line symmetric over-relaxation method
singular	A storage class which which causes the datum to be allocated on ACU, thus making it be an elementary datum such as integer, character, etc.
vex	A 1-dimensional array in x—major (vector) orientation
vey	A 1-dimensional array in y—major (vector) orientation

Abstract

This dissertation proposes a new technique for efficient parallel solution of very large linear systems of equations on a SIMD processor. The model problem used to investigate both the efficiency and applicability of the technique was of a regular structure with semi-bandwidth β, and resulted from approximation of a second order, two-dimensional elliptic equation on a regular domain under the Dirichlet and periodic boundary conditions. With only slight modifications, chiefly to properly account for the mathematical effects of varying bandwidths, the technique can be extended to encompass solution of any regular, banded systems. The computational model used was the MasPar MP-X (model 1208B), a massively parallel processor hostnamed hurricane and housed in the Concurrent Computing Laboratory of the Physics/Astronomy department, Louisiana State University.

The maximum bandwidth which caused the problem's size to fit the *nyproc* $\times$ *nxproc* machine array exactly, was determined. This as well as smaller sizes were used in four experiments to evaluate the efficiency of the new technique. Four benchmark algorithms, two direct — Gauss elimination (GE), Orthogonal factorization — and two iterative — symmetric over-relaxation (SOR) ($\omega = 2$), the conjugate gradient method (CG) — were used to test the efficiency of the new approach based upon three evaluation metrics — deviations of results of computations, measured as average absolute errors, from the exact solution, the cpu times, and the mega flop rates of executions. All the benchmarks, except the GE, were implemented in parallel.

In all evaluation categories, the new approach outperformed the benchmarks and very much so when $N \gg p$, p being the number of processors and N the problem size. At the maximum system's size, the new method was about 2.19 more accurate, and about 1.7 times faster than the benchmarks. But when the system size was a lot smaller than the machine's size, the new approach's performance deteriorated precipitously, and, in fact, in this circumstance, its performance was worse than that of GE, the serial code. Hence, this technique is recommended for solution of linear systems with regular structures on array processors when the problem's size is large in relation to the processor's size.

Chapter 1

Exploitation of Parallelism: Literature Survey

The more extensive a man's knowledge of what has been done, the greater will be his power of knowing what to do.
Benjamin Disreali.

We live in a World which requires concurrency actions...
- K.J. Thuber.

1.1 RATIONALE FOR PARALLELISM

A battery of satellites in outer space are collecting data at the rate of 10^{10} bits per second. The data represent information on the earth's weather, pollution, agriculture, and natural resources. In order for this information to be used in a timely fashion, it needs to be processed at a speed of at least 10^{13} operations per second.

Back on earth, a team of surgeons wish to view, on a special display, a reconstructed three-dimensional image of a patient's body in preparation for surgery. They need to be able to rotate the image at will, obtain a cross-sectional view of an organ, observe it in living detail, and then perform a simulated surgery while watching its effect, and without touching the patient. A minimum processing speed of 10^{15} operations per second would make this approach feasible.

The preceding two examples are not from science fiction. They are representative of actual and commonplace scientific applications where tremendously fast computers are needed to process vast amount of data or to perform a large number of computations in real-time. Other such applications include aircraft testing, development of new drugs, oil exploration, modeling fusion, reactors, economic planning, cryptoanalysis, managing large databases, astronomy, biomedical analysis, real-time speech recognition, robotics, and the solution of large systems of partial differential equations arising from numerical simulations in disciplines as diverse as seismology, aerodynamics, and atomic, nuclear, and plasma physics. No computer exists today that can deliver the processing speeds required by these applications. Even the so-called supercomputers peak at a few billion operations per second.

Over the past forty years or so, dramatic increases in computing speeds were achieved largely due to the use of inherently fast electronic components by the computer manufacturers. As we went from relays to vacuum tubes to transistors, and from small to medium to large to very large and presently to ultra large scale integrations, we witness — often in amazement — the growth in size and range of the computational problems that could be solved.

Unfortunately, it is evident that this trend will soon come to an end. The limiting factor is a simple law of physics that gives the speed of light in vacuum. This speed is 3^8 meters per second. Now, assume that an electronic device can perform 10^{12} operations per second. Then it takes longer for a signal to travel between two such devices. Why then not put the two devices together? Again, physics tells us that the reduction of distances between electronic devices reaches a point beyond which they begin to interact, thus reducing not only speed but also their reliability.

It appears, therefore, that the only way around this problem is to use parallelism. The idea here is that if several operations can be performed simultaneously, then the time taken by a computation can be significantly reduced. This is fairly an intuitive notion, and the one to which we are accustomed in organized society. There are a number of approaches for achieving parallel computation.

This chapter traces the evolution of parallelism in computing systems, and discusses the highlights of such evolution that directly impact very large scientific computing problems such as the ones mentioned above. Because there has been mainly a two-prong development in parallelism, namely, along the hardware and the software fronts, the survey focuses specifically on the efforts at the hardware and software designs which, either singly or in concert, support parallelism in modern computers. §1.2 discusses parallel hardware features of modern computers emphasing their innovative architectural designs and the influence of means[1]technologies on such designs. The three most popular classification (taxonomies) schemes for computers which are based upon the various architectural aspects of the computers, and the efforts to address the deficiencies of such schemes to adequately describe and classify some modern parallel computers, are all given in §1.2. §1.3 treats the software parallelism emphasizing differences between parallel and serial algorithms, common methodologies and essential design steps for creating parallel algorithms, the characteristic features of parallel algorithms, and, lastly, the parallel algorithms ranging from the previously proposed to the most recent ones that are available for efficient solution of a broad spectrum of scientific problems. Based upon the information gained from the studies covered in this literature survey, an effort will be undertaken to design efficient algorithms for the type of problem that this dissertation sets out to solve in accordance with the purpose of this study as succinctly explained in §1.4. The chapter summary is given in §1.5.

[1]In computer architecture, the word technology refers to the electronic, physicochemical, or mechanical for the entire range of implementation techniques for processors and other components of computers.

1.2 HARDWARE PARALLELISM

1.2.1 The Genesis of High-Performance Computers

Spurred by the advances in technologies, the past four and a half decades have seen a plethora of hardware designs which have helped the computer industry to experience generations of unprecedented growth and development which have been physically marked by a rapid change of computer building blocks from relays and vacuum tubes (1940 - 1950s) to the present ultra-scaled integrated circuits (ICs) [HOCK 88, GOLUB 93, DUNCAN 90, STONE 91].

As a result of these technological advances and improved designs, computer has undergone a remarkable metamorphosis from the slow uniprocessors of the 1950s to today's high performance machines which include supercomputers whose peak performance rates are thousands order of magnitude over those of earlier computers [HWANG 84, STONE 80, HOCK 88, HAYNES 82]. The requirements of engineers and scientists for ever more powerful digital computers have been the main driving force in the development of digital computers. The attainment of high computing speeds has been one of the most, if not the most, challenging requirement. The computer industry's response to these challenges has been tremendous, and the result is the remarkable evolution in computers in just three short decades (1960s to 1980s) as exemplified by an existence of commercial and experimental high-performance computing systems produced in these decades [HELLER 78, GOLUB 93, MIRAN 71, STONE 90, HWANG 89]. The quest for even more powerful systems for sundry scientific applications such as are mentioned above continues unabated into the 1990s.

1.2.1.1 The Early Computers

The early computers were not powerful because they were designed with much less advanced technology compared to what we have today. The first electronic computer was ENIAC. It was designed by the Moore School Group of University of Pensylvania in 1942 to 1946. ENIAC had no memory and it was a stored-program computer. That is, a program had to be manually loaded through external pluggable cables. ENIAC, like all other early computers, especially those of the late 1930s through 1940s (e.g., the first electronic computer built in 1939 by Atanosoff [MACK 87] to solve matrix equations), was very crude and had a very limited computational capability that is not even a match to that of some of today's hand calculators. For example, the computer by Atanosoff could solve matrices of order up to only 29 whereas some calculators can compute matrices of a lot higher orders than 29.

Very soon after the ENIAC design was frozen in 1944, von Neumann[2] Became associated with the Moore School Group as a consultant, and the design of EDVAC[3], which was already underway, proceeded full steam. The basic design of EDVAC, as outlined in the very widely publicized momerandum authored by von Neumann [VON 45], consisted of a memory "organ" whose "different parts" could perform different functions (e.g., holding input data, immediate results, instructions, tables). Thus with EDVAC was born the stored-program concept with a homogeneous multifunctional memory.

The later variation of the stored-program concept became the underlying principle for design of many computers from 1950s until today. The architectural designs of many familiar systems ranging from the early uniprocessors (e.g., IBM 2900)

[2]A Hungarian born mathematician (1903 - 1957).
[3]EDVAC was the immediate successor of ENIAC.

of the 1950s to the contemporary systems (e.g., IBM System/370, and the 4300 family [SIEWIO 82], the DEC VAX series, namely, VAX 11/750, 11/780, and 8600 [DIGIT 81a, DIGIT 81b, DIGIT 85], and the Intel 8086 [MORSE 78], etc.), are all based upon this design principle. This principle is known as the von Neumann architecture even though von Neumann became associated with the EDVAC design team only after the design had already begun. A storm of controversy over this monopolistic claim has been raging since the 1940s mainly because no other members of the design team, not even Eckert and Mauchly who were the chief architects of both ENIAC and EDVAC, received any credit[4] for the ENIAC design. For detailed treatment of the stored-program concept, how the concept came to be associated with only von Neumann, and the controversy that has been ensuing as the result of that association, see the publications by Wilkes [WILKES 68], Metropolis and Whorlton [METRO 80], Randell [RAND 75], and Goldstine [GOLD 72] (Part 2). For the von Neumann architecture, its characteristic features and deficiencies (e.g., the notorious bottleneck due to mainly the monolithic memory access), as well as the modern architectural means[5] of overcoming these deficiencies, see the texts by Stone [STONE 90], and Dasgupta [DASGUP 89b]. Our only concern here is to emphasize the fact that ENIAC, EDVAC, and other early computers were very slow, awkward, and with a capability to compute only the most elementary numerical problems the like of which is nothing compared with the type of problems mentioned in the beginning of this chapter.

[4]The von Neumann's 1945 memorandum contained neither references to nor acknowledgement of other members of the EDVAC design team.

[5]In fact, the existence of a number supercomputers are attributable to overcoming these deficiencies.

1.2.1.2 The Role of Technology and Innovative Design Philosophies

The two most obvious factors responsible for the crude and rudimentary nature of the early computers were that the components of these computers were constructed from slow and bulky vacuum tubes and relays as switching devices, and the fact that the computer design then was at its infancy and, therefore, lacked the touch of sophistication of today's design. The needed touch of sophistication and the influence that touch would have over the computer designs in the following decades, and the culmination of those designs in today's high-performance computers, awaited the invention of a better technology.

That invention was a fast solid-state binary switching device known as transistor. Transistor was invented in 1947 by three American physicisists — John Bardeen, Walter H. Bratain, and William B. Shockley — at the Bell Telephone Laboratories. Transistors are the building blocks of integrated circuits (ICs), the basic components of modern computers. The first discrete transistors were used in the digital computers about 1959. Since then, there has been dramatic improvement in the computational power of digital computers. Such improvement has been proceeding hand in hand with advances in mostly the silicon-based technologies evidenced in the reduction of chip's feature size and IC miniaturization[6] which results in significant increases in IC integration levels[7]. These and similar technological advances have led to decrease in cost of hardware components (e.g., memory) while improving their performance. More than anything else, it is the improvement in the hardware technology which has been responsible for transforming the slow uniprocessors of the earlier decades to the high-performance machines of the 1980s to 1990s. For full discussion of computer technologies and relevant

[6]Examples: Bipolar Transistors (TTl, ECL, I^2L) and Field-effect Transistors (NMOS, PMOS, HMOS, CMOS - CMOS-1, CMOS-2, CMOS-3, CMOS-4), GaAs MESFETs).

[7]The number of logic gates or components per IC.

topics, see chapter 1 of the book by Dasgupta [DASGUP 89a], and the paper by Hollingsworth [HOLL 93].

1.2.1.3 A Breed of Modern High-Performance Machines

The above transformation has also been accelerated by revolutionary design, which often occured in tandem with improved technology. The commonnest design technique has been to incorporate some parallel features into the modern computer. Since the 1960s, literally hundreds of highly parallel structures have been proposed, and many have been built and put into operation by the 1970s. The Illiac IV was operational at NASA's Ames Research Center in 1972 [BOUK 72] ; the first Texas Instrument Inc. Advanced Scientific Computer (TI-ASC) was used in Europe in 1972 [GOLUB 93]; the first Control Data Corp. Star-100 was delivered to Lawrence Livermore National Laboratory in 1974; the first Cray Research Inc. Cray-1 was put to service at Los Alamos National Laboratory in 1976.

The machines mentioned immediately above not only were the pioneers in innovative designs which have endowed these machines with an unprecedented power, but were also the fore-runners of even more powerful computing systems yet to come. Therefore, while their fames were still undimmed, they soon gave way to other generations of more powerful computers which culminate in today's supercomputing systems. Thus by 1980's, a substantial number of high-speed processors of the 1960s and 1970s either ceased to be operational or played less significant computing role while, in the same decades, the advance of the processing speeds and the improvement in the cost/performance ratio — an important design metric — continued unabated. The overall result were the introduction of new and more radical[8] architectural design philosophies such as evidenced in the reduced

[8]Vis á vis Von Neumann architecture.

instruction-set computers (RISC), a widespread commercialization of multiprocessing, and the launching of the initial releases of massively parallel processors. Thus Illiac-IV ceased operation in 1981; TI-ASC is no longer in production since 1980; since 1976, the Star-100 has evolved into the CDC Cyber 203 (no longer in production) and also into the Cyber 205 which signaled the CDC's entry in the supercomputing field; the Cray-1 (pipelined uniprocessor) has evolved into Cray-1S which has considerably more memory capacity than the original Cray-1, Cray-XMP/4 (4-processor, 128 MWord supercomputer with peak performance rate of 840 MFLOPS), Cray-2 (256 Mword, 4-processor reconfigurable supercomputer with 2 GFLOPS peak performance), Cray-3 (16-processor, 2 GWord supercomputer with 16 GFLOPS peak performance rate). Other superperformance computers produced in the 1980s include Eta-10, Fujitsu VP-200, Hitachi S-810, IBM 3090/400/VF, and a hoste of others (see chapter 2 of [HWANG 89], also note the influence of technologies on the design of these systems). Thus, by 1980s, the high-speed processors of the 1960s to 1970s have evolved in the supercomputers of the 1980s through 1990s, otherwise, totally new breed of high-performance machines were born.

Other computers of some historical interest, although their primary purpose was not for numerical computation, include Goodyear Corporation's STARAN [GOODYR 74, GILM 71, RUDOL 72, BATCH 74], and the C.mmp system at Carnegie-Mellon University [WULF 72]. Also of some historical interest, although it was not commercialized, is Burroughs Corporation's Scientific Processor (CSP) [KUCK 82].

1.2.1.4 The Advent of Array Processors

The Illiac IV had only 64 processors. Other computers consisting of a large (or potentially large) number of processors include Denelcor HEP and the International Computers Ltd. Data Array Processor (ICL DAP), both of which are offered commercially, and a number of one of a kind systems in the various stages of completion [GOLUB 93, HOCK 65]. These include the Finite Element Machine at NASA's Langley Research Center; MIDAS at the Lawrence Berkeley Laboratory; Cosmic Cube at the California Institute of Technology; TRAC at the University of Texas; CM* at Carnegie-Mellon University; ZMOB at the University of Maryland; Pringle at the University of Washington and Purdue university; and the Massively Parallel Processor (MPP) at NASA's Goddard Space Flight Center. Only a few (e.g., MPP, ICL DAP) are designed primarily for numerical computation while the others are for research purposes.

1.2.1.5 The Future of High-Performance Computers

Thus without a doubt, the 1970s to 1980s saw a quantuum leap in computer designs which embody parallel features for high performance capabilities. The need for ever faster computers continues into the 1990s. Every new application seems to push the existing computers to their limit. So far, the computer manufacturers have kept up with the demand admirably well. But will the trend continue beyond the 1990s? We think it will but that, as previously surmised, the trend will be tempered somewhat, if not completely checked, in a not-too-distant future, unless we seek other avenues[9] to find other means and techniques of improving upon the future high-performance computing systems. These means include new technologies, more radical or exotic designs, and software parallelisms.

[9]Other than silicon-based technologies.

Experts in computer-related traditions — hardware and software designers, technology inventors, etc. — are upbeat in expectation, otherwise, are near-unanimous in their agreement concerning the future of high-performance computations (see Stone et al. [STONE 90, STONE 91, DASGUP 89a, DASGUP 89b, DUNCAN 90, HENSY 91, HAYNES 82, BATCH 74]). The above listed alternatives as well as other means for improving the the performance of the future systems must be aggressively pursued because, as noted in §1.1, the present largely silicon-based technologies have a number of limitations — the pin-limitation, speed-limitation of metal interconnection, chip's feature size limitation, integration limitation, and the package density limitation — which are not likely to be overcome in the foreseeable future. Since the insatiable taste of scientists and engineers for high-performance machines with more and more capabilities is not likely to be slaked with any wait for the limitations of the present technologies to be overcome, a clarion call is sounded for immediate pursuit of these other means. Fortunately, according to the literature, the call has been answered and, as a result, other means and techniques (new technologies, pipelining, cache and parallel memories, RISC ideas, various topologies of processors in the multi-processor systems, efficient methodologies for design of parallel softwares, etc.) are already involved in the design of tomorrow's supercomputers. If this effort continues, and we are hopeful that it will continue, then we are more than likely to be blessed with future generations of supercomputers which will be capable of performing mind-boggling computational feats hardly envisioned today.

1.2.2 Models of Computations and their Classifications

As noted in §1.1.1, the decades of the 1960s through 1990s have witnessed the introduction of a wide variety of new computer architectures for parallel processing that complement and extend the major approaches to parallel computing developed in the 1960s and 1970s. The recent proliferation of parallel processing technologies has included new parallel hardware architectures (systolic and hypercube), interconnecting technologies (multistage switching topologies). The sheer diversity of the field poses a substantial obstacle to comprehend what kinds of parallel architectures exist and how their relationship to one another defines an orderly schema. We examine below the Flynn's taxonomy which is the early and still the most popular attempt at classifying parallel architectures and then give brief summary of the more recent classification schemes proposed to reddress the inadequacy of Flynn's taxonomic scheme in order to unambiguously include all parallel processors, both the old and the modern, into proper taxa.

1.2.2.1 Flynn's Taxonomy

Flynn's taxonomy [FLYNN 66, FLYNN 72] classifies architectures based on the presence of single or multiple streams of instructions and data into the following categories:

- SISD (single instruction, single data stream) — defines serial computers with parallelism in neither data nor instruction stream.
- SIMD (single instruction, multiple data stream) — involves multiple processors under the control of one control processor simultaneously executing in lockstep the same instruction on different data. Thus a SIMD architecture incorporates parallelism in data stream only.
- MISD (multiple instruction, single data stream) — involves multiple processors applying different instructions to a single data stream. MISD exhibits parallelism only in the instruction stream.

- MIMD (multiple instruction, multiple data stream) — has multiple processors applying different instructions to their respective data streams thus exhibiting parallelism in both the instruction and the data streams.

1.2.2.2 Other Taxonomies

Although the Flynn's taxonomy provides a useful shorthand for characterizing computer architectures, it is insufficient for classifying various modern computers. For example, pipelined vector processors merit inclusion as parallel architectures, since they exhibit substantial concurrent arithmetic execution and can manipulate hundreds of vector elements in parallel. However, they can not be regarded as any of the Flynn's taxa because of the difficulty to accomodate them into the taxonomy as these computers lack processor that execute the same instruction in SIMD lockstep and fail to possess the autonomy of the MIMD category. Because of the deficiency of the Flynn's taxonomy, attempts have been made to extend the Flynn's taxonomy to accomodate modern parallel computers. There are other taxonomies which are distinctively different from the Flynn's mainly because they are based upon criteria different from those of the Flynn's taxonomy. Among the best known of these taxonomies are those by Händler [HÃND], Giloi [GILOI 81, GILOI 83], Hwang and Briggs [HWANG 84, HWANG 87], and Duncan [DUNCAN 90]. But these taxonomies generally lack the simplicity inherent in the Flynn's, and, consequently, they are not as widely used as the Flynn's. We give below a brief summary of the distinctive features of these taxonomies here and refer the interested readers to the works of Mayr [MAYR 69], Hockney [HOCK 87] , Ruse [RUSS 73] , Sokal and Sneath [SOKAL 63], Dasgupta [DASGUP 89b], and Skillcorn [SKILL 88], for in-depth treatments of these taxonomic schemes.

1.3 SOFTWARE PARALLELISM

The treatment so far exposes the existence of parallelism at the hardware level as a direct result of the advances in technologies and the improvement of architectural designs of the modern computers. One may be tempted to conclude that in order to realize more parallelism and, therefore, achieve greater computational power from a computer, one has to use faster technologies (circuits) or use more advanced architectures. As noted in §1.1.1, the improvement in these areas have been immense. However, as also noted in that section and by Dasgupta [DASGUP 89a], the design of circuits have some limitations which naturally militate against much additional gains in parallelism realizable from the hardware. Therefore, any significant additional improvement must be sought elsewhere such as the software arena. There are a number of efficient parallel algorithms proposed for solution of practical and experimental scientific problems. These are well documented in the literature. Among the early excellent surveys on such algorithms are those of Miranker [MIRAN 71], Traub [TRAUB 74], Voigt [VOIGT 77], and Heller [HELLER 78]. The more recent such surveys include that by Ortega [ORTEGA 85].

1.3.1 Desirable Features of a Parallel Algorithm

The challenge posed by software parallelism is to devise algorithms and arrange computations to match the features of the algorithms to match those of the underlying hardware in order to maximize parallelism. As noted by Ortega and Voigt [ORTEGA 85], some of the best known serial (sequential) algorithms turn out to be unsatisfactory and need some modifications or even be discarded while, on the other hand, many older algorithms which had been thought to be less than optimal on serial machines have had rejuvenation because of their inherent parallel properties. Also, the current researches in parallel computation indicate that we can not directly apply our knowledge of serial computation to parallel computation because efficient sequential algorithms do not necessarily lead to efficient parallel algorithms. Moreover, there exist several problems perculiar to parallelism that must be overcome in order to perform efficient parallel computations. Stone [STONE 73] and Mikloško [MIKLOŠ 84] cite these problems as being:

1. Efficient serial algorithms are not necessarily efficient for parallel computers. Moreover, inefficient serial algorithms may lead to efficient parallel algorithms.

2. Data in parallel computers must be arranged in memory for efficient parallel computation. In some cases, the data must be rearranged during the execution of the algorithm. The corresponding problem is nonexistent for serial or sequential computers.

3. The numerical stability, speed of convergence, cost, and the complexity analysis of serial and parallel algorithms can be different.

4. Serial algorithms can have severe constraints that appear to be inherent, but most, if not all, can actually be removable.

What all this means is that

1. an efficient parallel algorithm is not a trivial extension of an efficient serial counterpart, but that it may be necessary to identify and then eliminate the essential constraints that are perculiar to the serial algorithm in order to create the parallel algorithm,

2. each of different parallel architectures (SIMD, MIMD, etc.) requires a change in the way a programmer formulates the algorithm and develops the underlying code. The programmer must establish a sensitivity to the underlying

hardware architecture he/she is working with to a degree unprecedented in serial computers. An efficient and effective parallel algorithm exploits the strengths of the underlying hardware while de-emphasing its weaknesses,

3. suitable data structures (static and/or dynamic) and appropriate mapping of the data structures onto a parallel run-time environment, also, the interprocessor communication, are all indispensable for efficient parallel programming. Because this mapping of data is usually more complicated than solving the problem itself, there have been some successful efforts in automating the process. Geuder, et al. [GEUDER 93], Levesque [LEVES 90], for example, have proposed and designed a software dubbed GRIDS which has been used by Lõhner [LÕHNER 93] to provide the user an interface for automatic mapping of his/her program structures to parallel run-time components.

1.3.2 Definition of a Parallel Algorithm

At any stage within an algorithm, the parallelism of the algorithm is defined as the number of arithmetic operations that are independent and, therefore, can be performed concurrently (simultaneously). On a pipeline computer, the data for operations are commonly defined as vectors, and a vector operation as a vector instruction. The parallelism is then the same as the vector length. On array processors, the data for each operation are allocated on different processors (commonly known as processing elements or PEs), and operations on all PEs are performed at the same time, but with different data, but under the control of the master control processor. The parallelism is then the number of data elements operated upon simultaneously. In MIMD computers, the number of processors involved in asynchronous computation of a given problem is the parallelism.

Natural parallelism is present mainly in the algorithms of linear algebra and in the algorithms for the numerical computation of partial differential equations. It is also present in algorithms that are based on the iterative computation of the same operator over different data, e.g. identical computations dependent on many parameters, method of successive approximations, some iterative methods for solving systems of linear equations, Monte Carlo methods, numerical integration, and in complex algorithms which consist of a large number of almost independent operators, e.g. iterative computation of Dirichlet's problems by the difference methods, the finite difference methods.

There are some parameters which are often used for accessing the performance of parallel algorithms. These include the running time, T, speedup, S, cost or work, complexity, number of processors, n. The definition of these and other relevant metrics are given with respect to various underlying parallel hardwares by Lambiotte [LAMBIO 75], Ortega [ORTEGA 85, ORTEGA 88, GOLUB 93], Heller [HELLER 78], Leighton [LEIGH 92], Aki [AKI 89], Voigt [VOIGT 77], and Eager et al. [EAGER 89].

1.3.3 Common Methods for Creating Parallel Algorithms

There are a number of procedures for efficient design of serial algorithms, and most of these naturally carry over to the design of parallel algorithms. Most parallel algorithm constructions follow one or more of the following approaches:

1. method[10]Restructuring of a numerically equivalent parallel algorithm from a given serial algorithm.

2. Divide-and-Conquer which recursively splits a given problem into subproblems that are solved in parallel.

[10]Also called reordering by some authors.

3. Vectorization of the internally serial algorithm in which a given direct serial algorithm is converted to an iterative method which rapidly converges to the solution under certain conditions.

4. Asynchronous parallel implementation of a serial, strictly synchronized algorithm.

We note that methods 1 and 2 have also been used to render serial computations effective; but methods 3 and 4 are new and are only suitable for parallel computations. Method 1, for example, has been used for enhancement of parallelism of arithmetic expressions in mathematical formulae by exploiting associative, commutative, and distributive laws of computation [MURA 71, KUCK 77]. Restructuring is also often applied to reorder the computation domain or the sequence of operations in order to increase the fraction of the computation that can be executed in parallel. For example, the order in which mesh points on a grid are numbered can increase or decrease parallelism when solving elliptic equation. Method 2 is usually applied to realize parallelism in numerical computation of initial value problems of, say, ordinary differential equations. The algorithms described by Nievergelt and Pease [NIEVER 64, PEASE 74] consist of multiple parallel applications of a numerical method for solving many initial value problems. The divide-and-conquer method (see the excellent treatment of this topic by Grit and McGraw [GRIT 88] for MIMD architecture) is widely used for computation on both SIMD and MIMD computers. A good example of algorithm which lends itself to the application of divide-and-conquer principle is the inner product computation $\sum a_i b_i$, where the product $a_i b_i$ can be computed by processor p_i. This might involve sending a_{i+1} and b_{i+1} to p_i for i odd. The sum operation is now "divided" among $p/2$ processors with p_i doing the addtion $a_i b_i + a_{i+1} b_{i+1}$ for i odd. The idea is repeated $\log n$ times until the sum is "conquered" in processor p_i. Mehod 3, vectorization method, is usually applied to accelerate vector solution of numerical problems and are particularly effective for executiing on SIMD vector processors. Method 4 is often used for designing parallel algorithms, and its application is demonstrated by Miranker and Chazan [MIRAN 71, CHAZAN 69] using the so-called chaotic or asynchronous implementation of parallel solution of linear system arranged as $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$.

1.3.4 Parallelism in Numerical Algorithms

Enormous research efforts have been expended in exposing and exploiting software parallelism. Efficient parallel algorithms have been created by reformulating the existing serial algorithms and by creating new ones for solution of problems from areas of sciences as diverse as fluid dynamics, transonic flows, reservoir simulations, structural analyses, mining, and weather prediction, and computer simulations. There have been remarkable progress in designing software for solving problems in all facets of scientific applications due to several favorable factors che chief of which being the advancement in and availability of parallel computing technology (hardware and software). In the area of computer simulation, for instance, much has been achieved. Such a success is exemplified by such ambitous attempt at developing a very large-scale Numerical Nuclear Plant (NNP) by the Reactor Simulation and Control Laboratory (RSCL) team at the Argonne National Laboratory (ANL) [TENT 94]. NNP will simulate the detail response of nuclear plants to a number of specified transients. Below, we give a brief summary of the parallel algorithms which are commonly utilized in solving systems resulting from many scientific applications.

1.3.4.1 Matrix/Vector Algorithms

Since the invention of the first electronic computer by Atanasoff to solve matrix equations of order 29 in 1939, researchers in many scientific and engineering disciplines have found their problems invariably reduced to solving systems of simulataneous equations that simulate and predict physical behaviors. Hardly is there any area of scientific applications in which are matrix and vector manipulations are not involved in one form or the other in developing algorithms for solutions of the problems. Little wonder, therefore, that a good number of these algorithms include a variety of data structures for memory allocation of matrices and vectors in order to facilitate processor-processor data routing necessary for efficient manipulation of matrices and vectors. Such include algorithms for matrix-matrix multiplication, matrix-vector multiplication, matrix transpose and inverse, eigenvalues, spectral problems, etc.(see text books with excellent treatments of these and other relevant topics by Aki [AKI 89] and Modi [MODI 88]) Aki and Modi have given these algorithms for computation on different kinds of parallel architectures of various topologies — MIMD and SIMD (tori, and tree-, mesh-, shuffle-, cube-, and ring-connected). Leighton [LEIGH 92] has extended the treatment to include other architectures such as systolic machines. Hockney and Jesshope [HOCK 88] have given a parallel matrix-matrix multiplication for execution on mesh-connected SIMD (ECL DAP) architecture and concluded that the maximum parallelism of $O(n)$ is realizable with outer-product algorithms, that the outer-product method is, therefore, suitable for array processors, and that the maximum parallelism is possible if the dimension $n \times n$ of the matrices match the processor array. When such a match is not possible, they have proposed a combination of techniques with which parallelism as much as n^3 results. Heller [HELLER 78] recommends inner-product matrix-matrix multiplication algorithms for vector computers on the ground that the inner-product is generally included among the hardware operations of many vector computers, and that the use of such built-in hardware operations will not only make the algorithm efficient and robust but will also save the start-up time.

1.3.4.2 Linear Systems

Parallel solutions of arbitrary linear systems, expressed in a vector-matrix form as

$$\mathbf{Ax} = \mathbf{b}, \tag{1.1}$$

are of very practical importance in all branches of science where they have been extensively used. Therefore, much research work has been directed to finding solutions of solving linear systems. Linear systems are the most heavily investigated area as far as parallel computations are concerned, and a good many parallel solutions have been developed. Lambiotte [LAMBIO 75] covers many parallel solution approaches with respect to the CDC STAR. Ortega [ORTEGA 88] has given a report of a survey in which an extensive overview of parallel algorithms commonly used for solving linear systems in many areas of sciences is included, and, in the text book [ORTEGA 85], he has treated a large number of parallel algorithms for solution of linear systems on mainly SIMD and vector architectures. Hockney and Jeshoppe [HOCK 88] have given similar solutions but with respect to only array processors (ECL DAP). Hockney [HOCK 81] has developed a number of efficient algorithms for solution of spectral problems resulting from simulation of particles. Parallel solution for the linear systems include such typical topics as the various elimination and factorization techniques, Fast Fourier Transforms, FFT, which are commonly employed in developing serial algorithms, as well as topics such as cyclic reduction, recursive doubling which are presently used exclusively as parallel solution techniques. Any parallel solution developed for a general dense linear system must, of necessity, be modified before it can used to solve a sparse sparse systems. The nature and extrent of the modification, of course, depend on the given sparse system. That is, whether it is banded (e.g., triangular, tridiagonal, pentadiagonal, etc.) or nonbanded (e.g., sparse systems with nonregular structures such as a skewed-Hermitian system or those with almost regular structures such as a Toeplitz system, etc.). For example, cyclic reduction and recursive doubling methods mentioned above are generally used for developing parallel algorithms for symmetric positive definite linear systems. For other types of systems, these methods must be modified or different methods are developed for their solution. Sun [SUN 93a] has proposed an efficient simple parallel prefix (SPP) algorithm for solution of almost Toeplitz system on array processor[11] and on Cray-2 supercomputer, and another algorithm [SUN 93b] , the parallel diagonal dominant (PDD) algorithm for symmetric and skewed-symmetric Toeplitz tridiagonal systems, the efficiency of which algorithm was tested on an Intel/860 multiprocessor. Garvey [GARVEY 93] has proposed a new method for the solution of the eigenvalue problem for matrices having skewed-symmetric (or skewed-Hermitian) component of low rank, and shows that it is faster than the established methods of comparable accuracy for the general unsymmetric $n \times n$ matrix provided the rank of the skewed-symmetric component is less than $\frac{n}{7.3}$.

Most of the techniques for solving linear systems are direct rather than iterative. Some iterative methods such as Jacobi display some parallelism in that evaluation of new iterates can be carried out with simultaneous substitutions of the old ones thus giving a maximum parallelism of O(n). But Jacobi technique is not recommended for parallel computation because of its notoreity for a slow convergence rate. But the Jacobi method has been a foundation of other iterative techniques with accetable convergence rates. These include successive over-relaxation (SOR)

[11]MasPar MPP.

method and its variants such as successive line over-relaxation (SLOR), symmetric successive over-relaxation (SSOR), the Chebychev SOR and block SOR. Other iterative techniques with good convergence rates are the alternating direction implicit (ADI), preconditioned conjugate gradient (PCG) methods.

Treatments of utilization of these iterative approaches for parallel solution of linear systems are given by Ortega [ORTEGA 85], Lambiotte [LAMBIO 75], Baucomb [BAUCOM 88]. Lambiotte carries out investigations for parallel solution of very large linear systems with repeated right hand sides. In particular, he has developed efficient parallel solution of large linear systems on the CDC 100 (vector processor) using both direct and iterative techniques under different storage schemes. Beacomb has also investigated the parallel solution of similar systems (that is, very large linear systems arising from self-adjoint elliptic systems) using a MIMD computing model — the Alliant FX/8 multiprocessors. She applied the preconditioned conjugate gradient method and its several variants such as reduced[12] system conjugate gradient (RSCG), diagonal-scaled reduced conjugate gradient (DRSCG), incomplete Choleski reduced system conjugate gradient (ICRSCG). Other investigators who have devised and used parallel algorithms to solve similar systems include Hockney [HOCK 65],hockney88) who designs an efficient direct method which uses a block cyclic reduction and FFT in a rectangular mesh domain to solve a Poisson equation with periodic and Dirichlet boundary conditions on the IBM 7090 computer. Swarzttrauber [SWARZ 89], Sweet [SWEET 73], Schumann and Sweet [SCHUM 76], Buzbee, et al. [BUZBEE 70], have extended the Hockney's solution approach in a variety of ways to encompass solution of elliptic equation with periodic, Dirichlet and Neumann boundary conditions in either regular or irregular domains. Recently, similar studies on sparse systems have been carried out by Anderson [ANDER 89], Golub [GOLUB 93], and Sweet [SWEET 93].

[12]Reductions with red-black, Cuthill-McKee reordering.

1.4 THE PURPOSE OF THIS DISSERTATION

Armed with the knowledge gained from this literature survey of the various studies of both the hardware and the software parallelism, we now demonstrate how this knowledge will be exploited in our quest for efficient softwares to solve the kind of problem that this dissertation sets forth to solve. The description of the problem and the ways and means of solving it are the major topics of treatment in the following section.

1.4.1 The Statement of Purpose

This dissertation has a three-fold purpose which is summarized in the following steps:

Step 1: Develop an efficient parallel algorithm based upon the cyclic reduction and recursive doubling techniques to solve large linear system such as given in eq.(1.1). The model of this problem will be developed in chapter 3. The iteration steps of the cyclic reduction component of this method will be carried out to the maximum reduction level of $\lceil \log_2 N \rceil - 1$, where N is the size of the system. The recursive doubling step will solve the system resulting from that reduction process using the LU factorization technique and the removal by parallel means any recursions which occur during the factorization.

Step 2: Develop parallel algorithms based upon the following direct approaches:

- Direct methods — Gauss Elimination (GE), LU and Orthogonal factorizations,

- Iterative methods — Symmetric Over-relaxation (SOR), Conjugate Gradient(CG) techniques.

Step 3 : Solve the model problem using the parallel algorithm of steps 1 and 2 on an array processor, the MasPas MP-X whose architecture and software specifications are described in chapter 2, and use the solutions from step 2 methods as the benchmarks to evaluate the efficiency of step 1 solution. The computational model for the execution of the above proposed algorithms is MasPar MP-X model 1208B, and its architectural and software features will be treated in chapter 2. The characteristics of the system to be solved and the relevance of those characteristcs in developing the mathematical model of the problem will be extensively covered in chapter 3. The development of the parallel algorithms suggested in Steps 1 and 2 will be covered in detail in chapter 4. Such coverage will, of course include design of appropriate data structures for efficient data routings and other intra- and inter-processor communications necessary for the efficient implementation of the algorithms. Steps 3 treatment will be elaborated in chapter 5. The results of and the conclusion on the investigations proposed in step 3 will be given in chapter 5. Finally, chapter 6 will propose potential future research work based upon this study.

1.4.2 The Choice of Problem

The linear system of equations, such as depicted in eq. (1.1), is the problem chosen for solution in accordance with the statement of the purpose given above. This kind of system occurs very frequently and often as a result of some mathematical modeling of certain problems in every scientific discipline. But in the context of this study, we will confine the linear systems to those resulting from elliptic problems in regular domain and with Dirichlet boundary conditions. Moreover, the linear system is assumed to be very large, highly sparse, and with regular structures. Examples of such systems include banded linear systems with semi-bandwidth of, say, β. The most widely studied members of the family of these problems include tridiagonal, pentadiagonal, and Toeplitz systems. These are the type of systems solved by Lambiotte [LAMBIO 75], Baucomb [BAUCOM 88] and many other investigators on various parallel machines.

1.4.3 Rationale for the Choice of this Solution Approach

Our choice of cyclic reduction methodology for the parallel solution of the linear system is based upon the following reasons:

1. Cyclic reduction and its variants has been one of the most preferred methods for solving general linear systems (dense & sparse), and it is particularly efficient for solving linear systems with certain properties such as sparsity and regularity of structures, the type serve as the model for this study. Cyclic reduction, when used in combination with other techniques, has been heavily applied for solving these types of systems.

2. Although it was originally developed (see brief disscussion below) with no parallelism in mind, cyclic reduction has considerable inherent parallelism.

3. Cyclic reduction is also known for its superior numerical stability.

Cyclic reduction was originally proposed for the general tridiagonal systems by G. Golub and developed by R. Hockney for special block tridiagonal systems [HOCK 65]. In that work, Hockney used the method of cyclic reduction in combination with a Fourier analysis to solve a linear system resulting from a model of a particle-charge problem on a rectangular mesh region of 48×48 on the IBM 7090. This method was known as FACR. Since then, this approach and its several variants[13] have been successfully and efficiently applied by a number of in-

[13]Variants, given as FACR(l), are based on reduction to level l.

vestigators for the solution of linear systems resulting from elliptic systems (see [SWEET 73, SWARZ 89, SCHUM 76, HOCK 70, GOLUB 93, SWEET 93]).

The technique of cyclic reduction has been traditionally used in conjunction with the serial Gauss Elimination (GE) technique for solution of the reduced system which results from the iterative reduction process using the cyclic reduction. Searching through the past and the current literature, no one has proposed any study nor designed an algorithm which involves the two techniques for solving a big linear system of equations. Given this parallel feature of the recursive doubling and our suspicion that this technique may be more efficient in terms of speed than the serial GE method, we propose a methodology comprising of these two powerful solution tools to solve the type of linear problems to be modeled for this study, and to have the problems solved on an array processor. We call this new methodology. The efficiency of this technique will be tested against the performances of some of the classical techniques (see above for their listing) for solving linear systems. Most of the algorithms involved will be implemented in parallel.

1.4.4 The Major Contributions by this Study

Although, as indicated above, the cyclic reduction has been widely applied for parallel solution of systems of linear equations, there is no report in the literature on major studies undertaken to specifically compare the computational merits of the classical methodologies traditionally applied for solution of large linear systems, such as to be used for investigation in this study, when such methodologies are used in conjucntion with cyclic reduction technique to solve the linear systems. Moreover, the large number of studies that involve the application of cyclic reduction often use one or so of these classical methodologies to develop serial algorithms used in the final step to solve linear systems resulting from reduction of the original system with cyclic reduction. The best example to support this observation is afforded by the work of Temperton [TEMP 80]. Temperton [TEMP 80] designed some versions of FACR(l) program called PSOLVE which incorporated the serial Gaussian elimination to solve Poisson equation in rectangular domain. PSOLVE which was run on both IBM 360/195 (optimal reduction level: $l = 2$), and on Cray-1 (optimnal reduction level: $l = 2$ in scalar mode, $l = 2$ in vector mode) turns out to be the fastest parallel program on these machines. The uniqueness of this dissertation lies in the provision of a unifying treatment incorporating the cyclic reduction and recursive doubling techniques for the purpose of solving large sparse linear systems on a SIMD machine, and also in comparing performances of these direct methods in terms of some of the standard measures of effectiveness of any algorithm. These measures will be given in chapter 5. Equally important is the fact that most of the algorithms involved in the development of the parallel program will be parallel.

1.5 CHAPTER SUMMARY

This chapter has traced in a somewhat chronological fashion the progress of the computer which, only a few decades ago was a mere slow uniprocessor with a very restricted computing capability, to become, through unbroken series of technological and architectural advancements, a high-performance supercomputer of today. Technology has been the major contributor to these advancements. But in the light of the fact of many limitations inherent in the technology and the unlikelihood of these limitations to be overcome in the foreseeable future, this chapter has also echoed the concern and the sentiment of many an expert over reliance of technology alone on the computer design. Other factors must be considered in such a design before any significant improvement in the computational capability of computer can be possible. Fortunately, these factors have already become a commonplace fixture of the modern computer design. The most notable of these factors are a) innovative design philosophies exemplified by CISC, RISC philosophies, b) improved architectures mostly through embodiment of parallel features (e.g., pipelining, multiplicity of CPS's functions, etc.) into computer's subsystems.

The advancements have resulted in many breeds of high-performance computers the proper classification of which has defied the simple taxonomy afforded by the Flynn's schema. This problem has necessitated both the formulation of other taxonomies and the improvement of the Flynn's to include these multi-faceted computers.

In order to realize the computational potential of a modern high-performance computer to the fullest, any program which runs on the computer must be designed to exploit the parallel features of the underlying hardware of the computer. That means that the program itself must have parallel features which can be easily mapped to those of the hardware. Many research efforts have been expended on both the hardware and software designs to make such an exploitation possible.

The chapter concluded by summarizing the objective of this dissertation, stating the problem (large, sparse and, linear) and the type of machine (a SIMD) to solve the problem. The proposed solution is to be constructed according to the steps suggested in the experience gained in design of parallel programs as reported in the literature.

The literature survey given in this chapter has demonstrated the progress that has been made and continues to be made in both the hardware and software fronts. The literature is very rich in records of such progress. Among the excellent books covering topics on the computer technologies, architecture, and harware design are those by Dasgupta [DASGUP 89a, DASGUP 89b], Hwang [HWANG 84, HWANG 87, HWANG 89], and Stone [STONE 80, STONE 90, STONE 91]. Among the articles which dwell on the same topics are those by Bell [BELL 85], Bouknight [BOUK 72], Dongara [DONGA 86], Duncan [DUNCAN 90]. In the area of parallel softwares and the parallel algorithm design methodologies, there are a number of excellent text books among which are those by Leighton [LEIGH 92], Aki [AKI 89], Boisvert [BOIS 72], Hockney [HOCK 81, HOCK 87, HOCK 88] Sweet [SWEET 93], Golub [GOLUB 93], and Dongara [DONGA 86]. Among the best articles treating the various topics in the arena of parallel software designs are those by Geuder [GEUDER 93], Daddy [DADDY 92], Hockney [HOCK 65, HOCK 81], Fox [FOX 86], Gušev [GUŠEV 91], [[GUŠEV 92a], [GUŠEV 92b]], and Voigt [VOIGT 77].

Chapter 2
The Model of Computation

The life so short, the craft so long to learn.
Hippocrates.

A chain is only as strong as its weakest link.
Proverb.

As indicated in §1.4, the computational model chosen for all experiments in this study is the MasPar Model 1208B (MP-1)[14], one of the members in the family of the MasPar SIMD processors. The 1208B, with its 8192 processing elements (PEs), is actually one half the size of MP-1. This chapter attempts to expose the pertinent features of this array processor in sufficient depth and breath as will be necessary to speed up our understanding of this machine. Such an understanding will not only lead to our greater appreciation of its computational capabilities but will also guide and facilitate our effort in mapping the parallel features of any algorithm that is intended for execution on the machine to those of the machine's underlying hardware. Such is a desirable parallel software design approach that was suggested in §1.2.

The 1208B is a model of massively parallel processor (MPP). The newer version of MP-1 is MP-2. Because, as indicated in the opening statement, we are going to use the 1208B, an MP-1, for all our study, our treatment is focused on MP-1 and MP-2. Because MP-1 and MP-2 are basically the same — the difference between the two processors is in the number of PEs of which MP-2 has more — the following treatment applies for both processors. Any distinction in their description will be indicated where such a distinction is necessary or deemed expedient for our speedy understanding of these massively-parallel processors. In addition, the description of

[14]Housed in the Concurrent Computing Laboratory (CCL) of the Physics/Astronomy Department, Louisiana State University, Baton Rouge Campus.

the MasPar MP-1 and MP-2 (or simply MP-X) will dwell mainly upon those hardware/software features of the MasPar the mastery of whose working characteristics is indispensable for efficient programming of the system.

In §3.1, we give a very brief description of MPP's hardware. We then broaden the hardware description, in §2.2, to encompass those of the MP-X stressing the conformity of their hardware design to those of the MPP. Also, in §2.2, the software components of the MP-X will be described in sufficient detail as to be useful for our immediate application for program development. The chapter summary is given in §2.3.

2.1 OVERVIEW OF MPP SYSTEM

2.1.1 Rationale for MPP Development

The MPP is a large-scale single-instruction, multiple-data stream (SIMD) processor developed by Goodyear Aerospace Corporation under contract to NASA's Goddard Space Flight Center [BATCH 82, GILM 83]. Its delivery to NASA-Goddard in May 1983 culminated four and one-half years of design and development effort by Goodyear Aerospace, the achievement which was preceded by an eight-year NASA program aimed at developing high-speed image processing systems [SCHAE 77]. The rationale for the MPP design derived from the data processing requirements associated with NASA's satellite programs, and the consequent need for a processor with high-speed data processing capability. Such a requirement was the major objective of NASA's NEEDS program which was launched in 1977. The NEEDS Program was aimed at having a processor with capacity of processing data at the rate of as high as 10^{10} per second. Computers at that time were capable of peak performance of about 10^{8} operations per second, and it was very doubtful that any serial processors could ever achieve the processing rates that ultimately would be required. For satellite data processing, it was agreed that a parallel processor (i.e., SIMD) was particularly appropriate since the algorithms employed in image processing are well suited to parallel processing. So, the government's "Request for Proposals" was issued in January, 1978. Of the four contractors which responded to the solicitation, Goodyear won the contract to build an MPP with the data processing requirements. With a total development cost of the then $6.7 millions, the MPP was delivered to Goddard on May 2, 1983.

2.1.2 MPP: Its Technology and Architecture

The MPP uses a custom-designed HCMOS VLSI multi-processor IC chip as the basic building block to form the array of bit serial PEs. Each custom IC contains 8 PEs configured in a (logical) 2-by-4 sub-array. The principal MPP printed board type contains 192 PEs in a 16-by-12 array using 24 VLSI chips. Eleven such boards make up an array slice of 16-by-132 PEs. The custom IC was fabricated in an HCMOS technology using 5-micron design rules. The design was implemented using 8000 transistors and required a chip size of 235-by-131 mils.

The basic hardware components of MPP are the array unit (ARU) which processes arrays of data, the arithmetic control unit (ACU) that control the operations of ARU, the staging memory that buffers and permutes data, and the front end computers that control the flow of data. The ARU contains a 128×128 array of bit-serial PEs, each having 1024 bits of local memory. Two-by-four sub-arrays of PEs are packaged in a custom VLSI HCOS chip. The staging memory is a large multi-dimensional access memory. Its primary purpose is to perform a "corner-turning" operation that converts data stored in conventional format to the bit-plane format required for the ARU processing. The ARU is responsible for all parallel computation. Most of non-parallel processing is carried out in the front end.

2.2 THE MasPar MODEL MP-X

2.2.1 The MP-X Hardware

The MP-X is one of the models of the MPP. It is a massively parallel processor (MPP) which also is a fine-grained machine with a data parallel processing unit. Massively parallel means that the machine has at least 1,000 processors or processing elements; data parallel means that the parallel data processors are acting in unison on a single instruction at a time but on different data; fine grained means that these processors are much smaller and simpler than general purpose central processing units (CPS) of conventional computers. Figure 2.1 below shows the block diagram of the hardware elements of MP-X. These are the front end, the data parallel unit, and the input/out subsystems.

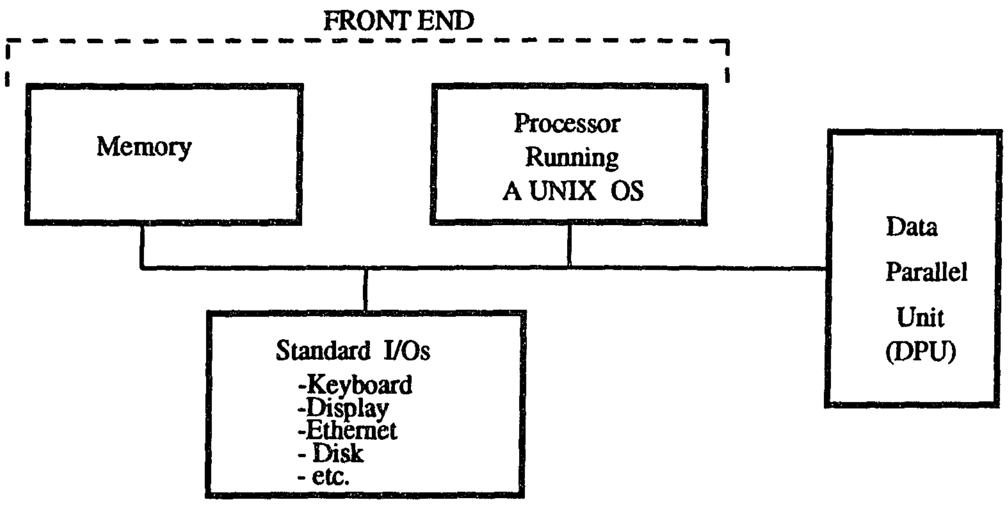

Figure 2.1 MP-X Hardware Subsystems

The Front End (FE) subsystem is the processor that runs the ULTRIX[15] on graphics workstations with windowing capability using the standard I/O devices. The workstation provides the user with the keyboard and display, a network environment, a multi-user operating system, a file system, a programming environment, and an access to the DPU.

On the MP-1, hardware arithmetic occurs on 4-bit data. Instructions that operate on 8, 16, 32, and 64-bit integers and on 32 and 64-bit IEEE floating-point numbers are implemented in microcode. All operations occur on data in registers; only load and store instructions reference memory. Peak performance of a full 16K MP-1 is 550 Mflops (1208B: 325 Mflpos) on IEEE 64-bit floating-point data. The MP-2 uses 32-bit hardware integer arithmetic with microcode for higher precision and floating-point operations. Peak performance goes up to nearly $2,000$ Mflops in the MP-2.

The Data Parallel Unit (DPU) subsystem is that part of the MasPar MPP that performs all the parallel processing. DPU contains the following subsystems:

1) The Array Control Unit (ACU) which is a 32-bit, custom integer RISC MasPar subsystem that stores the program, instruction fetching and decoding logic, and is used for scalars, loop counters, etc. It includes its own private memory.

2) The PE Array , a two-dimensional mesh of processors. Each PE may communicate directly with 8 nearest neighbors. The PE in the array is a RISC processor with 64K bytes of local memory. All PEs execute the same instruction broadcast by the ACU. The machine clock rate is 12.5 MHZ. The PE at the edges of the mesh are connected by wrap-around channels to those at the opposite edge, thus making the array a two-dimensional torus[16]. The two directions in the PE array are referred to as x and y axes of the machine. The machine's dimensions are referred to as nxproc and

[15]The DEC's implementation of the UNIX operating system.
[16]Other connections – cylindrical, spiral,etc – are also possible.

nyproc and the number of PEs as nproc = $nyproc \times nxproc$. Thus, every PE has an x coordinate ixproc and a y coordinate iyproc, and also a serial number iproc $= ixproc + nxproc \times iyproc$.

The set of all PE memories is called parallel memory or PMEM. It is the main data memory of the machine. Ordinarily, PMEM addresses are included in the instruction, so that all PEs address the same element of their memory. By a memory layer we mean the set of locations at one PMEM address, but across the whole array pf PEs. Each PE includes 16 64-bit registers. PEs communicate with each other through various communication primitives which must be explicitly invoked in the case of MPL. Most of the communication Primitives which support various ACU-PE, PE-PE, and FE-ACU communications are shown in Figure 2.2. The ACU-PE communication is automatically supported when a program runs on ACU as opposed to FE. That is, any program written in MPFortran[17], or MPL — the two high level MasPar languages — automatically support ACU-PE communication. Certain commands, especially those involving slow operations (e.g., I/O calls) are automatically routed to be executed on ACU even if the module containing them are intended for execution on PE array.

There is a constant communication between the ACU and PE during an execution of a parallel programs (i.e., the one written using the high level languages mentioned before). After all it is the ACU which decodes instructions and broadcasts them to the PE array for execution (see Flynn's taxonomy in §1.2.2(A), and SIMD processor arrays in §1.2.2(B)(B.1)). Such a program can effect explicitly the ACU-PE communication with functions such as connected(), rfetch(), rsend(), xfetch(), and xsend().

[17]FORTRAN 90 – a variant of FORTRAN 77 with parallel features.

The PE-PE communication is effected by a variety of means the most used of which being the nearest-neighbor and the router primitives.

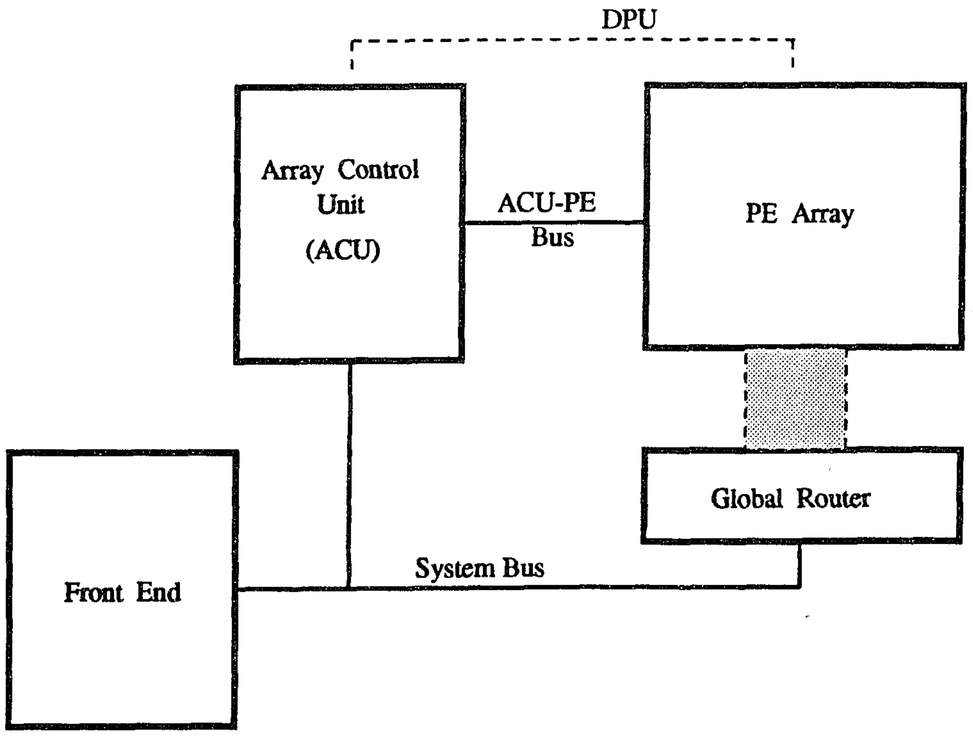

Figure 2.2 MP-X Communication Buses

Nearest neighbor communications among the PEs are commonly effected with XNET construct. This construct enables a PE to communicate with other PEs at some specified distance in any of the eight directions: north (xnetN),north-east (xnetNE), north-west (xnetNW), south (xnetS), south-east (xnetSE), south-west (xnetSW), west (xnetW), and east (xnetE). An access which permits copy and pipe is also possible. Copy and pipe communications are differentiated with the letter "c" and "p" in the xnet primitives. Thus for a PE to undergo a near neighbor communication with intention of copying from the neighbor's local memory, the xnet primitives xnetcN, xnetcNE, xnetcNW, xnetcS, xnetcSE, xnetcSW, xnetcW and xnetcE are used. Comunication in arbitrary patterns is often carried out through a hardware global router.

This is commonly effected with the router constructs. These communications are expressed in the instruction set as synchronous, register-to-register operations. This allows interprocessor communication with essentially zero latency and very high bandwidth — 200 Gbits per second for nearest neighbor communication, for example.

A program running on the front end machine can communicate with those running on the ACU using any of the FE-ACU communication primitives which are either the blocking or nonblocking. Among these are the functions callRequest(), copyIn(), copyOut(), blockIn(), blockOut(), checkReply(), awaitReply(), etc.

The Front Input/Output (I/O) subsystem offer a wide range of I/O performance ranging from standard workstation speeds to a 230 Megabyte/second 64-bit channel (MPIOC), and then up to a maximum of 1.5 gigabytes/second with special user-designed hardware. The MP-X high-speed I/O subsystem utilizes the same innovative technology that provides global communications within the PE array. It performs massively-parallel I/O with performance that scales the configuration size. It provides support for high-performance disk arrays with multiple gigabytes of storage. It is designed to support frame-buffer graphics systems, and such external interfaces as FDDI and HPPI. The subsystem's open interface specification allows customer access to the MP-X's high-performance I/O for interfacing of custom I/O devices that operate at up to 1.5 gigabytes/second.

Presently, the 1208B[18] at Louisiana State University provides a high-resolution by means of interfacing frame-buffer through MPIOC at the rate of up to 115 Mbytes/second. This permits a high resolution animation to be performed at rates approaching normal video rates — 30 frames/second.

[18]Hostnamed Hurricane.

2.2.2 The MP-X Softwares

The MP-X softwares fall into two main categories: languages and programming environment. MP-X provides a very rich programming environment called the MasPar Programming Environment or MPPE. No further mention of MPPE will be made in the rest of this discussion. So, we refer any intersted reader to the MasPar MPPE User Guide [MPPE] for detailed treatment of this topic.

Presently, the two languages supported by MP-X are the MasPar Programming Language or MPL, and the MasPar FORTRAN. MPL is the MasPar's level programming language which provides access to the DPU. Two versions of MPL are the ANSI C-based, the new version of MPL, and the K&K C-based which is the older version of MPL. MasPar's FORTRAN is the MasPar implementation of FORTRAN 90. It is, therefore, based upon FORTRAN 77 with FORTRAN 90 array extensions and intrinsic data set. As algorithms in this study will be written mostly in MPL, we devote the remaining of this section to discussing the features of MPL which will be beneficial for implementing algorithms to be developed for this study.

2.2.3 The MasPar Programming Language (MPL)

In MPL, one must use explicit statements in order to effect a ACE-PE communication. If one's program runs on the front end, such program must make explicit calls to DPU for parallel execution. It is highly advisable that only the parts of program which exhibit parallel characterists be committed to DPU for execution while the rest of the parts should run on FE[19].

Variables in MPL, variables have a storage class, either singular or plural. Singular variables are the traditional C variables of any type. On the other hand, if a variable,

[19]FE routine is contained in a file with extension .c (C's), .f (FORTRAN's), etc., while DPU routine must be in a file with .m as extension.

say, x is declared via plural double, then there instance of x on every PE, so that x is implicitly an array. The operation $x *= 2$ causes every PE to multiply its x by 2. This is the only way to express parallelism in MPL. That is, to involve many PEs (preferably, all the nproc PEs for maximum parallelism) as possible usually carried out with use of plural variables.

The integer constants nxproc, nyproc, and nproc (which define the machine size) are predefined in MPL, as are plural integer constants iyproc, ixproc, and iproc (which define PE's coordinates). Other useful predefined integer constants for bit-wise operations, are lnproc, lnxproc, and lnyproc, which are the number of bits to represent nproc, ixproc and iyproc respectively. Equations immediately below demonstrate the way these variables are usually employed expediently in a typical DPU program:

$$
\begin{aligned}
&int\ \ n0,\ \ n1,\ \ n2; \\
n0 &= n \gg lnproc; \quad /*\ n0 \leftarrow n \div nproc */ \\
n2 &= n - (n0 \ll lnproc; \quad /*\ n \leftarrow n\ mod\ nproc */ \\
n1 &= n \ll lnproc; \quad /*\ n1 \leftarrow n \times nproc */.
\end{aligned}
$$

Data Structures in MPL require special allocation on PE memories. Since matrix and vector data structures may involve more than nproc elements, several memory locations on each PE, organized as plural array, are required for their storage. The MPl programmer must code the loops over these arrays required to implement simple matrix and vector operations such as the addition of two vectors. This process is called virtualization, and it is required to create a programming model in which there is one virtual processor per matrix or vector element.

To simplify the coding of matrix and vector operations, MasPar provides a library (the MPLABB library) of simple operations on matrices and vectors of arbitrary sizes. The virtualization looping is hidden within the software layer. The types of data objects supported are:

- vex which is a one-dimensional array with x-major orientation. That is, the ith element of a vex resides on PE ($imodnproc$). In a vex u, the x coordinate of the PE holding u(i) varies most rapidly as i increases. If the vector length is greater than nproc, then multiple memory layers are used for its storage. If its length is n, then $nb = (n + nproc - 1) \div nproc$ layers are declared as (plural double u[nb]), and element i is stored in memory layer $i \div nproc$.

- vey is a one-dimensional array with y-major orientation. That is, its ith element is on the PE with coordinates $nproc = (i\ modnyproc)$ and $ixproc = (i \div nyproc)$. In a vey u, the y coordinate of the PE holding u(i) varies most rapidly as i increases.

- mat is a two-dimensional array of arbitrary shape. In an $ny \times nx$ mat A, element $A(i,j)$ is stored on PE having coordinates $ixproc \equiv j\ mod\ nxproc$ and $iyproc \equiv imod\ nyproc$. The mapping to PEs are referred to in the literature as the cyclic or torus wrap mapping, and in the MasPar's jargon as the cut and stack mapping. In this mapping, the matrix is viewed as a matrix of blocks each of which is $nyproc \times nxproc$. The number of blocks in the y direction is $nby = \lceil (ny \div nyproc) \rceil$ and in the x direction is $nbx = \lceil (nx \div nxproc) \rceil$. The blocks are then stored in memeory as plural array a in x-major orientation. Thus, the data declaration for such a mat would be $plural\ double\ a[nby * nbx]$, and the scalar element $A(i,j)$ resides in the block at coordinates $yblks = i \div nyproc$ and $xblk = j \div nxproc$, which is contained in the plural array element $a[(yblk * nbx) + xblk]$.

- blk is a sequence of nblks plural variables seperated in memory by a fixed address increment. Thus,

  $\{$*plural double* $a[10],\ b[10];$

  $\quad p\_blkadd(5, a, 1, b+1, 2);$

  $\}$

  is equivalent to

  $\{$*plural double* $a[10],\ b[10];$

  $\quad$ *int* $i;$

  $\quad for(i=0;\ i<5;\ i^{++})\ b[i+2*i]\ +*\ a[i];$

  $\}$

  That is, the data in all PEs and at memeory locations $a, a+1, \ldots, a+4$ are added to the data at memory locations $b+1, b+3, \ldots, b+9$.

The typical MPLABB routines are:

- p_veynrm2 computes the Euclidean norm of a vector stored as a vey.
- p_mattovey moves data from a column of a mat into a vey.
- p_matfromvey which broadcasts data from a vey into several consecutive columns of a mat.
- p_matsumtovey sums the rows of a mat, placing the sums into a vey.
- p_blkaxpy adds a scalar multiple of a sequence of memory layers to a second sequence of memory layers.

2.3 CHAPTER SUMMARY

The MPP system has demonstrated the feasibility of the concept of massive parallelism. The enormous processing power of the MPP provides support envisioned by NASA in the 1970s for an ultra-high speed processing of satellite imagery. The speed and flexibility of the MPP system — in particular, the data-reformatting capability — supports its effective application to a wide variety of computational problems. The MPP's system software suite provides an extensive and powerful set of capabilities to support convenient development and execution of application software.

The MPP is a SIMD architecture that was designed by Goodyear Aerospace in the 1970's, and was delivered to NASA in 1983. NASA needed a processor with sufficient computational punch as to be capable of performance close to the rate at which satellite data could be processed and transmitted to a wide variety of earth-bound users. Such performance rate is measured in terms of billions of operations per second. As no processor of the 1970s were capable of such rate of operations, the NASA folk agreed that a processor with parallelism capability, and a SIMD in particular, could meet such a performance requirement. Hence, a call for the design of the MPP.

MPP cosists of mainly four hardware subcomponents — the array unit or ARU, the array control unit or ACU, a unique staging memory, and the front end computers. The functionalities of these subsystems are briefly described in §2.1. The MP-X has all these basic hardware components, although they may be differently dubbed. In the MP-X's jargon, MPP's ARU is called DPU while the names of the other subsystems are retained. Also, these four hardware subsystems serve the same functions in MP-X as in the MPP.

Because this study applies a model of MPP called MasPar 1208B, which, with only 128 × 64 PEs, is one half the size of MPP, we devote most of this chapter to describing both the hardware and the software features of MP-1 together with those of its newer cousin, the MP-2. In this report, the MP-1 and MP-2 are collectively called MP-X. Presently, MP-X can be programmed with two high-level programming languages — MasPar FORTRAN which is FORTRAN 90 (FORTRAN 77 with parallel features) and MasPar programming language or MPL. MPL is lower level than MasPar FORTRAN, therefore, a detailed knowledge of the innards of MP-X's hardware is required for efficient programming with MPL than with MasPar FORTRAN. Most of the programming in this study are carried out using MPL.

MPL has two data classes — the singular and the plural types. In writing a program in MPL, it is important that the programmer uses data of correct class. A singular variable has one instance on ACU's memory while a plural variable has nproc instances. That is, an instance on the same location in all PE memories. The "same location in all PE memories" constitutes a layer. MPL has important data structures — vex, vey, matand blk. To simplify coding with these data structures, MPL has a very rich collection of implicit functions. Without giving the syntax of any particular function, this chapter (see §2.2) has provided the meanings and examples of usage of a few typical of these functions.

We conclude this summary by referring the interested reader to the following MasPar publications from which most of the discussion in this chapter is obtained:

1. [MPPSYS] for for MP-X's system overview — hardware and software.
2. [MPML] for detailed meaning, usage syntax, and a complete listing of the functions mentioned above.
3. [MPPE] for MasPar's environment.
4. [MPPPLS] for the MasPar Programming Language or MPL. This publication not

only give a detailed treatment of MPL, it also contains very good examples on usage of the pertinent structures of the language for writing efficient parallel programs to execute on PDU.

5. [MPCMDS] for the MasPar's commands.

Chapter 3

Mathematical Formulation of the Model Problem

The laws of mathematics and logic are true simply by virtue of our conceptual scheme.

W.V. Quinn.

Every man takes the limits of his own field of vision for the limits of the world.

Schopenhauer.

The main focus of this dissertation is to utilize the the established mathematical and algorithmic tools to develop an efficient novice parallel algorithm for solving linear system of equations such as resulting from discretization of elliptic equations or from any mathematical models provided the system of equations exhibits certain properties. Exposition and discussion of these properties are the thrust of this chapter. We are particularly interested in second order elliptic equations since most of the scientific and engineering systems and the concommitant researches are overwhelmingly concentrated on application of second order partial differential equations (PDEs). This is particularly so in such areas as wave propagation, heat conduction, elasticity, vibrations, boundary layer theories, etc. Works and research activities involving higher order (rarely exceeding fourth order) are also carried out, but their theoretical developments and practical applications are neither as prolific nor are as well established as those involving second order PDEs. In this study, therefore, we are interested in second order elliptic equations as the basis for the model system specification for our study.

In §2.1, the general characterization of ellipticity is given within the scope of this study, in order to keep this study focused on its objective and not to clutter it with correct but unnecessary pedagogical detail. The generalized treatment of elliptic systems given in §2.1 is tailored to the treatment of more specific systems, namely the two-dimensional elliptic problems, in §2.2. We believe that such focused treatment will be just adequate to furnish solid framework from which any specific elliptic systems of interest, such as to be used in this study, can be developed. In §3.2, the model system to be used for all investigations in this study is developed from the mathematical tools generated in §3.1. The properties of the model system will also be discussed as the knowledge of such properties can greatly simplify a formulation of the solution strategy for the system. Also given in §3.2 is the justification for our choice of the approximation technique needed to develop the model problem. Finally, §3.3 gives the summary of this chapter.

3.1 ELLIPTICITY

3.1.1 General Characteristics of Ellipticity

Let $\Omega \subseteq \mathbb{R}^n$ be a bounded, simply connected domain whose boundary is $\Gamma = \delta\Omega$. Let also $u = u(\mathrm{x})$ be some function of n-dimensional vector $\mathrm{x} \in \Omega$. Then the second order, n-dimensional PDE

$$\sum_{i=1}^{n}\sum_{j=1}^{n} \mathcal{A}_{ij}(\mathrm{x}) \frac{\partial^2 u}{\partial x_i \partial x_j} = \mathcal{F}(\mathrm{x}, u, \nabla u) \tag{3.1}$$

in which $\nabla u = grad(u) = (\frac{\partial u}{x_1}, \dots, \frac{\partial u}{\partial x_n})$, is an n-dimentional, second order elliptic equation if the matrix $\mathcal{A}$ given as

$$\mathcal{A}(\mathrm{x}) = \|\mathcal{A}_{ij}(\mathrm{x})\| \tag{3.2}$$

is either positive definite or negative definite identically, $\forall \mathrm{x} \in \Omega$. In eq. (3.1), we assume that the term

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \neq 0,$$

thus implying that at least one of the derivatives is present. Other pertinent properties of the above elliptic equation are summarized below:

1. If, as in eq. (3.1), the elliptic equation is linear in the highest derivative (which here is second order), then the equation is said to be semi-linear.

2. If the coefficients a_{ij} depend also on u and ∇u, or the forcing function $f(\mathrm{x}, u, \nabla u)$ is nonlinear, then the equation is quasi-linear.

3. If the coefficients a_{ij} and or the function f depend only on x or ∇u, then the equation is said to be linear.

4. Because in eq. (3.1) $\frac{\partial^2}{\partial x_1 \partial x_2} = \frac{\partial^2}{\partial x_2 \partial x_1}$ elliptic equations are inherently symmetric.

3.1.2 Two-Dimensional Elliptic Systems

If in eq.(3.1), one sets $n = 2$, and $x = x_1$, $y = x_2$, $\mathcal{A}' = \mathcal{A}_{11}, \mathcal{B}' = \mathcal{A}_{12}$ (also $\mathcal{B}' = \mathcal{A}_{21}$, by the virtue of the system's symmetry), $\mathcal{C}' = \mathcal{A}_{22}$, then a then eq.(3.1) becomes

$$\begin{aligned}
&\frac{\partial}{\partial x}(\mathcal{A}(x,y,u,\nabla u)\frac{\partial u}{\partial x}) + \frac{\partial}{\partial x}(\mathcal{B}'(x,y,u,\nabla u)\frac{\partial u}{\partial y}) \\
&+ \frac{\partial}{\partial y}(\mathcal{B}'(x,y,u,\nabla u)\frac{\partial u}{\partial x})\frac{\partial}{\partial y}(\mathcal{C}(x,y,u,\nabla u)\frac{\partial u}{\partial y}) = \mathcal{F}(x,y,u,\nabla u). \qquad (3.3)
\end{aligned}$$

Letting all the coefficients and the forcing function $\mathcal{F}$ in eq.(3.3) to be functions of only $x\, and\, y$, and setting the coefficient d to zero, and, finally, setting $a = \mathcal{A}', b = 2\mathcal{B}', c = \mathcal{C}'$, eq. (3.3) takes on the more familiar form:

$$\frac{\partial}{\partial x}(a(x,y)\frac{\partial u}{\partial x}) + \frac{\partial}{\partial x}(b(x,y)\frac{\partial u}{\partial y}) + \frac{\partial}{\partial y}(c(x,y)\frac{\partial u}{\partial y}) + du(x,y) = f(x,y) \quad (3.4)$$

where the term $du(x,y)$ comes from the forcing function $f(x,y,u,\nabla u)$ of eq. (3.3). For eq. (3.4), the domain Ω is a two-dimensional space which, together with its boundary Γ, is given as

$$\Omega \subseteq \mathbb{R}^2,$$
$$and \qquad \Gamma = \delta\Omega. \tag{3.5}$$

Because f and the coeffients of eq. (3.4) depend only on x and y, the equation becomes a two-dimensional, second order PDE which is ELLIPTIC equation if the condition expressed in eq. (3.2) holds. The condition holds if, in eq. (3.4),

1. $c \neq 0$,
2. (sign)a = (sign)c, and
3. $4ac > b^2$.

Important class of systems whose modeling and solution strategies involve elliptic equation includes steady-state problems. Examples of such include steady-state heat and diffusion problems. Other practical problems (beside elliptic systems, that is), which deservedly have been for decades the subject of intensive scientific inquiry and and sundry applications, can be derived from eq. (3.4). The most known and widely applied of these are:

1. **PARABOLIC** problems if $4ac = b^2$ among which are the unsteady-state problems such as transient heat and diffusion problems.
2. **HYPERBOLIC** problems if $4ac < b^2$ included among which are the wave equations,the transport phenomena problems (e.g., diffusion of matter, neutron diffusion, and radiation transfer), wave mechanics, gas dynamics, supersonic flows.

3.1.3 Finite Difference Approximation of Elliptic Equations

The commonest method of solving of elliptic equation such as given in eq. (3.4) is a numerical approximation in which both the differential operators and the problem domain are replaced by some mathematical approximations, the most widely used of which are the finite difference, finite element, or the method of lines [RICE 83, BIRK 83]. In this study, we will adopt the finite (central) difference method to solve eq. (3.4) in a rectangular domain $(a_1, b_1) \times (a_2, b_2)$ with Dirichlet boundary conditions. With only a slight modification, the method extends naturally to rectangular domains with periodic boundary conditions. The finite difference approximation takes three steps outlined below:

- Step 1: Let Ω be a rectangular domain defined as $\Omega = (a_1, b_1) \times (a_2, b2)$, and let $\Gamma = \delta\Omega$ be its boundary. Let the interval intervals $a_1 \le x \le a_2,\ b_1 \le y \le b_2$ that define Ω in x and y directions be subdivided into into n and m equal parts respectively so that $h = \frac{a_2 - a_1}{n}$ and $k = \frac{b_2 - b_1}{m}$. Then the rectangular domain Ω is approximated and covered with $h \times k$ meshes as shown in Figure 3.1 below. Thus,

$$\Omega = \{((a_1 + ih),\ (b_1 + jk))|\ 1 \le i \le n,\ 1 \le j \le m\},$$

$$\Gamma = \{((a_1 + ih),\ b_1),\ ((a_1 + ih),\ b_2),\ (a_1, (b_1 + jk)),\ (a_2,\ (b_1 + jk)) \\ |0 \le i \le n, 0 \le j \le m\}.$$

The boundary conditions are given on the four sides of the rectangular mesh region as follow. Let Ψ_1, and Ψ_2 be the sets of coordinate positions in the x and y directions respectively. That is,

$$\begin{aligned} \Psi_1 &= \{a_1 + ih \mid 0 \le i \le n\}, \\ \Psi_2 &= \{b_1 + jh \mid 0 \le j \le m\}. \end{aligned} \tag{3.6}$$

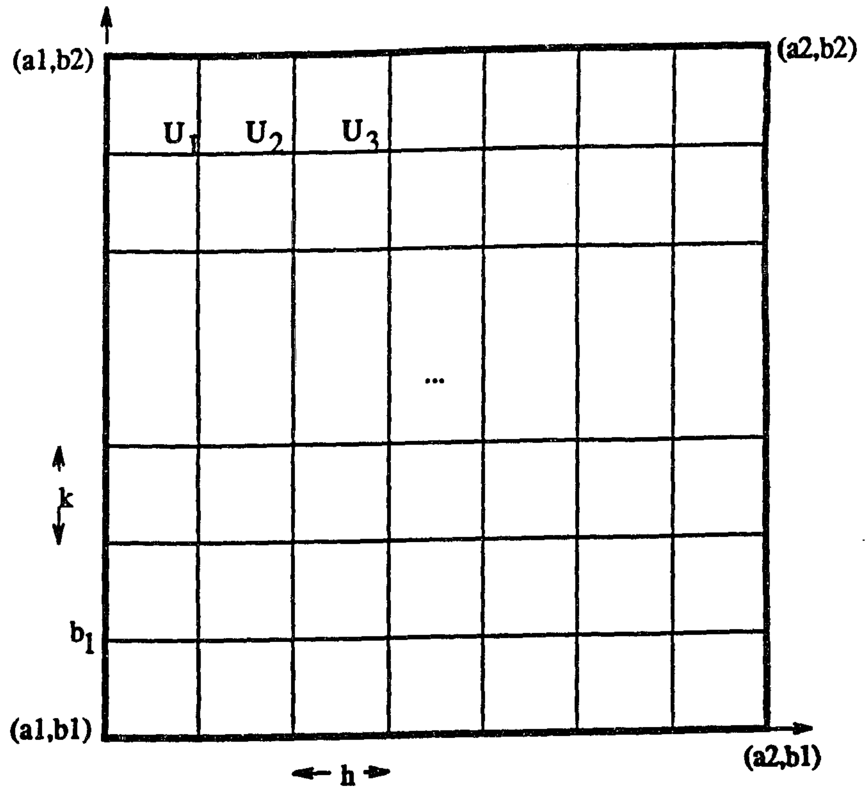

Figure 3.1: Mesh Region to Approximate the Domain of the Model Problem

Finally, let $U(x,y) \approx u(xy)$, $\forall (x,y) \in \Omega \cup \Gamma$.

Then the boundary conditions on the 4 boundaries of Ω may be defined as:

$$\begin{aligned}
U(x,y) &= g_1(x,y),\ x \in \Psi_1,\ y = b_1, \\
U(x,y) &= g_2(x,y),\ x \in \Psi_1,\ y = b_2, \\
U(x,y) &= g_3(x,y),\ x \in \Psi_2,\ y = a_1, \\
U(x,y) &= g_4(x,y),\ x \in \Psi_2,\ y = a_2, \qquad (3.7)
\end{aligned}$$

where the functions g_1, g_2, g_3, , g_4 are the 4 boundary functions and which are assumed to be continuous on the respective boundaries.

- Step 2: Replace the differential operators in eq. (3.4) with the following difference operators:

$$\begin{aligned}
\frac{\partial}{\partial x}(a(x,y)\frac{\partial u}{\partial x} &\approx \frac{1}{h^2}(a(x+\frac{h}{2},y)\{u(x+h,y)-u(x,y)\} \qquad (3.8) \\
&\quad -a(x-\frac{h}{2},y)\{u(x,y)-u(x-h,y)\}) \\
\frac{\partial}{\partial y}(b(x,y)\frac{\partial u}{\partial y} &\approx \frac{1}{k^2}(b(x,y+\frac{k}{2})\{u(x,y+k)-u(x,y)\} \\
&\quad -b(x,y-\frac{k}{2})\{u(x,y)-u(x,y-k)\}) \qquad (3.9)
\end{aligned}$$

Using the above approximations for differential operators in eq. (3.4), we obtain

$$\begin{aligned}
U(i,j) \;=\;& a[((a_1+h(i+\frac{1}{2})),(b_1+jk)]\{U((a_1+h(1+i)),(b_1+jk)) \\
& -U((a_1+ih),(b_1+jk))\} - a[(a_1+h(i-\frac{1}{2})),(b_1+jk)] \\
& \{U((a_1+ih),(b_1+jk)) - U((a_1+h(i-1)),(b_1+jk))\} \\
& +\frac{h^2}{k^2}[b((a_1+ih),(b_1+k(j+\frac{1}{2})]\{U((a_1+ih),(b_1+k(1+j))) \\
& -U((a_1+ih),(b_1+jk))\} - b[(a_1+ih),(b_1+k(j-\frac{1}{2}))] \\
& \{U((a_1+ih),(b_1+jk)) - U((a_1+ih),(b_1+k(j-1)))\}
\end{aligned}$$

$$
\begin{aligned}
&+h^2c[(a_1+ih),(b_1+jk)]U((a_1+ih),(b_1+jk)) \\
&= h^2f((a_1+ih),(b_1+jk)), \qquad (3.10)\\
&\forall((a_1+ih),(b_1+jk)) \in \Omega, (1 \le i \le n, and 1 \le j \le m).
\end{aligned}
$$

- Step 3: Derive a system of linear equations by applying eq. (3.10) to each mesh point $((a_1+ih),(b_1+jk))$, where $((a_1+ih),(b_1+jk)) \in \Omega$, also, by using the boundary conditions specified in eq. (3.7). This will result in a system

$$\mathbf{Ax} = \mathbf{k} \qquad (3.11)$$

of linear equations where $\mathbf{A}$ is an $N \times M$ ($N = n^2$, and $M = m^2$).

The original second order 2-dimensional elliptic equation given in eq.(3.4) has been successfully reduced to its approximated form in eq.(3.10) by the finite difference technique. Through a set of assumptions, we will see very shortly how eq.(3.10) is further simplified into a less cumbersome form to yield our model problem.

3.2 THE MODEL PROBLEM

We make the following assumptions concerning eq.(3.10) in order to get the model problem:

Assumption 1: The model problem is a self-adjoint, second order elliptic equation. This means that, in deriving the model system, we start with eq. (3.4) which we have already showed to be second order, 2-dimensional elliptic equation. Self-adjoint means that all derivatives in eq.(3.4) must be in the form $\frac{\partial}{\partial x_i}(\mathcal{A}(\mathrm{x})\frac{\partial u}{\partial x_j})$, and, therefore, the second term (which is not of that form) must be dropped from eq. (3.4). Also, since the particular elliptic equation to be used are the Poisson or Laplace equations, the last term on the right hand side of eq. (3.4) must be dropped too.

The resulting equation, after changing the coefficient c which remains in eq. (3.4) to b, are, $\forall(x,y) \in \Omega \subseteq \mathbb{N}^2$,

$$PoissonEquation : \mathcal{L}(u) = \frac{\partial}{\partial x}(a(x,y)\frac{\partial u}{\partial x}) + \frac{\partial}{\partial y}(b(x,y)\frac{\partial u}{\partial y}) = f(x,y) \quad (3.12)$$

$$LaplaceEquation : \mathcal{L}(u) = \frac{\partial}{\partial x}(a(x,y)\frac{\partial u}{\partial x}) + \frac{\partial}{\partial y}(b(x,y)\frac{\partial u}{\partial y}) = 0. \quad (3.13)$$

Assumption 2: The model problem is defined and approximated on a rectagular domain $\Omega = (a_1, b_1) \times (a_2, b_2)$ where $a_1 = b_1 = 0$, and $a_2 = b_2 = 1$.

Assumption 3: The model problem is a boundary-value problem with Dirichlet boundary conditions such that the boundaries functions in eq.(3.7 are $g_1 = g_2 = g_3 = g_4 = 0$. That is the model problem has zero boundary conditions on all the four boundaries.

Assumption 4: The grids have same width. That is, $h = k$.
In this work, we will be using k as the grid width.

Applying the above assumption 1 on eqs.(3.12), and (3.13), the equations become

$$\mathcal{L}(u) = -\frac{\partial^2 u(x,y)}{\partial x^2} - \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \quad (3.14)$$

$$\mathcal{L}(u) = -\frac{\partial^2 u(x,y)}{\partial x^2} - \frac{\partial^2 u(x,y)}{\partial y^2} = 0, \quad (3.15)$$

with the Dirichlet boundary conditions which, by assumption 3, are

$$u(i,j) = 0, \ where \ (i,j) \in \Psi_1 \bigcup \Psi_2. \quad (3.16)$$

Applying assumptions 1, 2, and 4 and replacing u of eq.(3.15) with its approximated form U of eq.(3.10), eq.(3.15) becomes

$$4U_{ij} - U_{i+1\,j} - U_{i-1,\,j} - U_{i\,j+1} - Ui\ j-1 = k^2 f(i,j), \quad (3.17)$$

where $k^2 f(i,j) = 0$ in the case Laplace equation. When eq.(3.17) is written for all the grid points in the defined rectangular domain, an $N \times N$ ($N = n^2$) system of linear equations given in eq.(3.11) results. This system has a set of very interesting properties soon to be given in §3.2.1 below. The solution of the linear system (eq.(3.11)) can be simplified by exploiting such properties in the formulation of the solution technique.

Eqs.(3.14) and (3.15) and their approximated form given in eq.(3.17) together with the 4 assumptions given above constitute the model problem for this study. The approximation (eq.(3.17)) will be used to generate the linear system of equations (eq.(3.11) that is to be solved using the techniques mentioned in chapter 1. The model problem is, therefore, a second order, two-dimentional elliptic equation with Dirichlet boundary condition on a square domain $(0,1) \times (0,1)$ where the problem is approximated using a central finite difference approximation technique.

3.2.1 Properties of the Model Problem

There is a more practical and much quicker way of obtaining eq.(3.17). This is through the use of a 5-star stencil of the finite difference approximation. By using the five-point (or five-star) stencil of the finite difference directly on each grid point, we obtain an approximated Finite Element[20]solution $U(i,j)$, for $u_{i,j}$ $\forall(i,j) \in \Gamma \cup \Omega$ thus obtaining eq. (3.17). Numbering the unknowns in the natural row-by-row ordering [BAUCOM 88, ORTEGA 85, GOLUB 93], we get the $N \times N$ ($N = n^2$) block tridiagonal system of linear equation given in eq.(3.11) where

$$\mathbf{A} = \begin{pmatrix} T & -I & \\ & T & -I \\ & \cdots & \\ & & -I \\ & -I & T \end{pmatrix}$$

$\mathbf{x} = u$, and $\mathbf{b}$ holds the values resulting from the boundary conditions and the

[20]This solution can also be obtained using the Raleigh-Ritz-Galerkin approximations with linear right-triangular elements.

values of $h^2 f_{ij}$ as well. The matrix T is an $n \times n$ matrix given as

$$\mathbf{T} = \begin{pmatrix} 4 & -1 & \\ -1 & 4 & -1 \\ & \cdots & \\ & & -1 \\ & -1 & 4 \end{pmatrix}$$

and I is the identity matrix. The error associated with the above approximation is $\mathrm{O}(n^2)$. A slightly different linear system results from using a nine-point[21] finite difference operator to approximate the differential operator:

$$\begin{aligned} -8U_{ij} &+ U_{i-1\,j-1} + U_{i-1,\,j+1} + U_{i\,j-1} + Ui+1\,j-1 \\ &+ U_{i+1\,j} + U_{i+1\,j+1} + U_{i\,j+1} = k^2 f(i,j). \end{aligned}$$

The corresponding matrix is again an $N \times N$ block tridiagonal system given in eq. 3.11) but in which the matrix $\mathbf{A}$ is given as

$$\mathbf{A} = \begin{pmatrix} B & -C & \\ -C & B & -C \\ & \cdots & \\ & & -C \\ & -C & B \end{pmatrix}$$

where B and C are given as

$$\mathbf{B} = \begin{pmatrix} 8 & -1 & \\ -1 & 8 & -1 \\ & \cdots & \\ & & -1 \\ & -1 & 8 \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} 1 & 1 & \\ 1 & 1 & 1 \\ & \cdots & \\ & & 1 \\ & 1 & 1 \end{pmatrix}.$$

The error associated with the finite difference nine-point discretization is $\mathrm{O}(n^6)$.

The system of linear equations depicted in eq. (3.11) has a number of properties some of which have already been pointed out. It is these properties that we hope to

[21]It can also be derived using Raleigh-Ritz-Galerkin with a basis of tensor products of piece-wise linear functions.

exploit in devising the novice parallel algorithm for solving the linear systems. We summarize hereunder some of these most pertinent properties:

(a). **A** is symmetric positive definite. Symmetric means that $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$, and positive definite means that $\forall \mathbf{z} \neq \mathbf{0}, \mathbf{z}^T\mathbf{A}\mathbf{z} > \mathbf{0}$.

The reasons for these properties of **A** have already been given in §2.1.

(b). **A** (resulting from either five- or nine-point discretization) is banded with semibandwidth of $\beta = 1$, which implies that **A** is tridiagonal.

Certain consequences are immediately evidenced from the above properties:

1. Being definite positive (property (a)), the system is guaranteed to have only nonzero eigenvalues, and, consequently, to be nonsigular (has unique solution).

2. Because the system is tridiagonal (property (b)), it is highly sparse with at most five nonzero values for any row of **A**. This means that most of elements of **A** are zeros. In fact, applying the five-point finite difference approximation to the model problem on a mesh $n \times n = 128 \times 128$ leads to about $16,000$ equations each with at most five variables as has been already pointed out. The complete matrix (**A**) has 256 million entries, with about $80,000$ ($\approx 0.03\%$) being nonzero [HOCK 81]. Any algorithm for solving the system of eq. (3.11) must be capable of storing only the nonzero elements and also to avoid creating further nonzeros during its execution.

3.3 CHAPTER SUMMARY

In this chapter, we have explicated the properties of elliptic systems and demonstrated the relevance of the properties to the specification of the model problem to be used in this study. We have also shown how these properties, namely symmetric and positive definite, carry over to be the most significant attributes of the model problem — the Poisson or Laplace problem in a rectangular domain with either the Dirichlet or periodic boundary condition — defined from the elliptic system. It is these properties that must be exploited by any algorithm that must solve the problem in an efficient manner.

A number of classical discretization approaches for approximating the resulting model problem were also discussed. The methods include the finite difference, the finite element methods and the method of line. Both the five- and nine-point finite difference discretization were applied to approximate the model problem in a rectangular mesh region of unit dimension $(0,1) \times (0,1)$ with the meshwidth of k in both directions, and also having the boundary conditions specified on the four sides of the mesh.

As the result of discretization, a system of equations $\mathbf{Ax} = \mathbf{k}$ results. This system has a number of properties such as sparsity and symmetric positive definiteness. The former property poses the challenge of having a suitable data structures that will not only hold the nonzero elements but will also prevent nonzero elements which did not exist before from being created, during of execution; the later property guarantees the existence and uniqueness of solution of the model problem. The methods proposed in chapter 1 will be used in solving the resulting linear system of equation on an array processor with p processing elements, and it is assumed that such a system will be very large, that is, $N \gg p$ (N, the size of the system).

Chapter 4
Algorithmic Tools Development

"Where shall I begin, please, your majesty ? " he asked.
"Begin at the beginning," the king said, gravely,
"and go on till you come to the end: then stop."
Lewis Carrol (Alice in Wonderland).

And let ours also learn to maintain good works for necessary uses, that they be not unfruitful.
The Holy Bible (KJV), Titus 3:14.

The intent of this chapter is to develop a background for the design of a new algorithm for efficient solution of the model system which was developed in chapter 3, and also to designate and describe some benchmark algorithms that will be used for the purpose of measuring the efficiency of the algorithm. The design of the new algorithm for parallel solution of the model problem on an array processor, here the MP-X, and the measure of its efficiency, are the main thrust of this dissertation as it was pointed out in §1.4. This efficiency measure will be carried out with some benchmarks that will be developed from the classical techniques which have been traditionally applied for the solution of linear systems of equations. As it was noted in chapter 1 (see §1.3), these classical methods are predominantly either direct or iterative, and, while they are also typically serial in nature, a good majority of them exhibit intrinsic parallel features. It is these parallel features that will be exploited, as much as such exploitation is feasible, to design efficient parallel benchmarks. Effort at parallelizing the benchmarks will be abandoned if the resulting parallel algorithms are found to be less efficient than their serial equivalents.

The model problem to be solved using the new algorithm was developed in chapter 3 by utilizing the finite difference or finite element approximation of elliptic equation in a regular domain. As we witnessed, the very nature of the mathematical formulations essential for derivation of the problem imparts to it the ubiquitous property of sparcity and banded-ness among the several important properties peculiar to this and similar systems (see §3.4.3 for a listing of such properties). These properties will also be exploited to the fullest in the design of both the new algorithm and the benchmarks. Also, every design will be in accordance with the recommended steps of parallel algorithm design set forth and suggested in §1.2.

The algorithms developed in this chapter are specifically intended for solution of our model problem, which, as might be recalled, is symmetric positive definite, and banded with semibandwidth of $\beta = n$. With slight modifications, mainly to account for the mathematical effect of the bandwidth, β, the algorithms can be easily generalized to solve any symmetric-banded systems. All algorithmic developments will follow these two steps:

(a) A review of computational steps that are incorporated in a given algorithm when such algorithm is intended for solution of general dense linear systems. These steps will be used without much elaboration since they are found in any good textbooks for numerical methods and analysis. For detailed treatment of these and other methodologies traditionally used for solving general linear systems, we refer the interested reader to but one of such books — the excellent text by W.H. Press[22] [PRESS 86] which serves as single-volume source that treats most of the methodologies for linear system solution used in this study, and, also, that casts those methodologies in actual subprogram forms ready for immediate inclusion in any user-written programs

(b). Design of an equivalent but a parallel algorithm, if possible, along the treat-

[22]Numerical Recipes which come in several versions each of which based on a major programming languague – FORTRAN, C, Pascal, etc.

ments suggestive in those steps, so that the resulting parallel algorithm can efficiently solve the model problem. Efficiency here is measured in terms of the storage requirement, data routings, inter-processor communication, and the overall time for computing the problem. The design of any parallel algorithm will follow the general guidelines given in §1.2.

A brief overview of the salient features of the model problem, especially such as are of immediate relevance to algorithmic development, is made in §4.1. Section §4.2 treats the development of some benchmark algorithms based upon direct methods for solving linear systems — dense and sparse. These developments will follow the guidelines above listed guidelines. As much as possible, the parallel implementations of these algorithms will be used as the benchmarks. When they cannot be easily parallelized or if an attempt at parallelizing them will result in codes so complicated as to render the resulting codes' execution less efficient than that of their serial ancestors, then only the serial codes will be used. Any difficulties arising in realizing efficient parallel implementation of the benchmarks will be duly explained in the context of their treatment. In §4.3, the treatment of §4.2 will be repeated but for designing parallel codes based on iterative methods. The detailed development of the new methodology for solution of banded linear systems will be given in §4.4.

4.1 A BRIEF OVERVIEW OF THE MODEL PROBLEM STRUCTURE

The model system that was developed in chapter 3, can be presented in in matrix-vector form of:

$$\mathbf{A}\mathbf{x} = \mathbf{k}, \tag{4.1}$$

where $\mathbf{A}$ is an $N \times N$ matrix, $\mathbf{x}$ and $\mathbf{k}$ $N \times 1$ vectors. If there are m right hand sides (this case $\mathbf{b}$ may be viewed as $N \times (m+1)$ matrix), we assume that the procedures to be described hereunder may be safely repeated, where necessary, for each of them. However, if m is large, say $\mathcal{O}(\mathrm{N})$, then only some of the procedures described may be appropriate.

The system being also block-tridiagonal, it be equally represent in block-tridiagonal form as:

$$\begin{pmatrix} b_1 & \mathbf{c}_1 & \mathbf{0} & & \mathbf{0} \\ \mathbf{a}_2 & \mathbf{b}_2 & \mathbf{c}_2 & & \\ \mathbf{0} & & & & \\ & & \cdots & & \\ & & & & \mathbf{0} \\ & & \mathbf{a}_{N-1} & \mathbf{b}_{N-1} & \mathbf{c}_{N-1} \\ \mathbf{0} & & \mathbf{0} & \mathbf{a}_N & \mathbf{b}_N \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \cdots \\ \mathbf{x}_{N-2} \\ \mathbf{x}_{N-1} \\ \mathbf{x}_N \end{pmatrix} = \begin{pmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \mathbf{k}_3 \\ \cdots \\ \mathbf{k}_{N-2} \\ \mathbf{k}_{n-1} \\ \mathbf{k}_N \end{pmatrix}. \tag{4.2}$$

Note that the above system is $N \times N$ where $N = n^2$, that each element (i.e., $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, or $\mathbf{0}$) of matrix $\mathbf{A}$ is a $n \times k$ tridiagonal block matrix, that the elements of vectors $\mathbf{x}$ and $\mathbf{n}$ are all $n-$dimensional block vectors, and, finally, that the system of eq.(4.2) is itself tridiagonal, and, as such, has $\beta = 1$. Because the system resulted from discretization of elliptic equation over a rectangular region, eq.(4.2) has the perculiar form of:

$$\begin{pmatrix} \mathbf{b}_1 & \mathbf{I} & \mathbf{0} & & \mathbf{0} \\ \mathbf{I} & \mathbf{b}_2 & \mathbf{I} & & \\ \mathbf{0} & & & & \\ & & \cdots & & \\ & & & & \mathbf{0} \\ & & \mathbf{I} & \mathbf{b}_{N-1} & \mathbf{I} \\ \mathbf{0} & & \mathbf{0} & \mathbf{I} & \mathbf{b}_N \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \cdots \\ \mathbf{x}_{N-2} \\ \mathbf{x}_{N-1} \\ \mathbf{x}_N \end{pmatrix} = \begin{pmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \mathbf{k}_3 \\ \cdots \\ \mathbf{k}_{N-2} \\ \mathbf{k}_{N-1} \\ \mathbf{k}_N \end{pmatrix} \tag{4.3}$$

where $\mathbf{x}$ and $\mathbf{k}$ are n-dimensional vectors, $\mathbf{I}$ a $n \times n$ identity matrix, and $\mathbf{b}$ a $n \times n$ block-tridiagonal matrix having the form:

$$\begin{pmatrix} -4 & 1 & 0 & & 0 \\ 1 & -4 & 1 & & \\ 0 & & & & \\ & & \cdots & & \\ & & & & 0 \\ & & 1 & -4 & 1 \\ 0 & & 0 & 1 & -4 \end{pmatrix}. \tag{4.4}$$

We will be using the presentation of eq.(4.2) to demonstrate the theoretical development of all the algorithms needed while the block presentation of eq.(4.3) will be used in actual implementation.

4.2 DIRECT BENCHMARKS

The direct methods chosen as benchmarks are the **Gaussian Elimination** method, the LU, and the Householder factorizations. These are treated in the following subsections.

4.2.1 Gaussian Elimination Method (GE) and LU Factorization

Both the GE and the LU factorization depend upon decomposition of the matrix **A** of eq.(4.1) according to formulation:

$$\mathbf{A} = LU \tag{4.5}$$

where **U** and **L** are respectively the upper and lower triangular matrices of **A**, and the factorization is possible if **A** is nonsingular. Equivalently, **A** can be factored using formulation

$$\mathbf{A} = \mathbf{U}^{\mathbf{T}}\mathbf{D}\mathbf{U} \tag{4.6}$$

where **A** and **U** are as defined above and **D** a positive diagonal matrix.

Once the factorization has been carried out to obtain **U** and **L**, the solution vector **x** is obtained, with the following operations

$$\mathbf{Ly} = \mathbf{k}, \quad \mathbf{Ux} = \mathbf{y}, \tag{4.7}$$

which are called the forward and back substitutions respectively. In case of the factorization given in eq. (4.6), the equivalent computation for solution vector **x** is

$$\mathbf{U^T y} = \mathbf{k}, \ \mathbf{Dz} = y, \ \mathbf{Ux} = \mathbf{z} \tag{4.8}$$

There are three basic steps incorporated in the above solutions:

(a) **Factorization Step.** This step proceeds in accordance with eq. (4.2) above. That is, the factorization of **A** results in an upper and a lower triangular matrices **U** and **L** respectively, with **L** obtained using $\mathbf{L} = \mathbf{P}_1\mathbf{L}_1 \ldots \mathbf{P}_n\mathbf{L}_n$, where $\mathbf{P}_i$ is the permutation matrix which interchanges the ith and jth row ($i \leq j$) of **A** on which it operates, and $\mathbf{L}_i$ is the unit lower triangular elementary matrix which, at the ith step of reduction, performs the usual row operations to zero out the ith column.

(b) **Forward Substitution Step.** The forward substitution gives $\mathbf{L}y = \mathbf{x}$ using the procedure $\mathbf{P}_1\mathbf{L}_1\mathbf{P}_2\mathbf{L}_2 \ldots \mathbf{P}_n\mathbf{y} = \mathbf{k}$.

(c) **Back Substitution Step.** This step yields $\mathbf{Uy} = \mathbf{x}$.

4.2.1.1 GE & LU Factorization for Dense Linear Systems

(1). Serial GE (LU) for Dense Systems

We digress to give a discussion of the above factorization steps when the system depicted in eq.(4.1) is a dense linear system. The discussion is intended as a review of the above steps for the general linear systems as we believe such a review will help in the development of the equivalent solution technique for the solution of the model problem which is block-tridiagonal system of linear equations.

The usual serial solution afforded by both the GE method and the LU factorization for for general dense linear systems is given in the algorithm provided in figure 4.1. The algorithm, dubbed SerGE for Serial Gaussian Elimination and LU methods, is strictly serial and incorporates the three solution steps outlined above. The same algorithm is used for both GE and the LU decomposition. Finally, the arithmetic operation counts involved in SerGE is given in table 4.1. From the results of these treatments, we make the following very important observations about the solution of linear systems as a whole using the above solution steps:

(a). The factorization step (step 1.2.3) subtracts the multiples of jth row of **A** from the succeeding rows which are then updated. It is this updating operation which constitutes the bulk of the work in the algorithm.

b). The number of steps involved in SerGE is cubic given exactly as

$$Operation\ Count \;=\; \frac{2}{3}N^3 + N^2 - \frac{16}{3}N \;\rightarrow\; \mathrm{O}(N^3).$$

The dominant contributor to this count is the factorization step which alone contributes quadratic count, $\mathrm{O}(N^2)$.

c). It is the above cubic operation count which makes SerGE not so efficient for solution of large linear systems.

d). A further degradation in speed is caused by the pivoting operation (step 1.2.2) sometimes needed to ensure the numerical stability of the algorithm.

```
ALGORITHM  SerGE()
   input: (1) A real array
          (2) x, b real vector

   output: (1) U, L real array
           (2) x real vector

   BEGIN
          step 1: comment Factorization Step
                step 1.1: Define $\mathbf{A}^{(0)} \equiv \mathbf{A}$
                step 1.2: for $k \leftarrow 1, N-1$ do

                      step 1.2.1: Search for pilot from column i of $\mathbf{A}^{(i-1)}$
                                  Find $\max_{i \geq k} \; a_{ik}^{(k-1)}$

                      step 1.2.2: Interchange rows k and i of $\mathbf{A}^{(k-1)}$
                                  for j $\leftarrow k+1, i+2, \ldots, N$ do
                                     $u_{kj} = a_{ij}^{(k-1)}$
                                     $a_{ij}^{(k-1)} = a_{kj}^{(k-1)}$

                      step 1.2.3: Modify jth row of $\mathbf{A}^{(k-1)}$ $for\ j > k$
                                  for i $\leftarrow k+1, i+2, \ldots, N$ do
                                     $l_{ik} = a_{ik}^{(k-1)}/u_{kk}$
                                     for $j \leftarrow k+1, i+2, \ldots, N$ do
                                        $a_{ij}^{(k)} = a_{j}^{(k-1)} - l_{ik}u_{kj}$

          step 2: comment: Forward Substitution Step
                step 2.1: Define $\mathbf{y}^{(0)} \equiv \mathbf{b}$
                step 2.2: for $i \leftarrow 1, 2, \ldots, N-1$ do
                      step 2.2.1: Perform interchange
                      step 2.2.2: Modify jth component of $y^{(i-1)} for\ j > i$
                      step 2.2.3: for $j \leftarrow i+1, i+2, \ldots, N$ do
                                     $y_j^{(i)} = y_j^{(i-1)} - l_{ji}$

          step 3: comment: Backward Substitution Step
                step 3.1: Compute each component $x_i$ $of$ $\mathbf{U}\mathbf{x} = \mathbf{y}$
                step 3.2: for $i \leftarrow N, N-1, \ldots, 1$ do
                      $x_i = (y_i - \sum_{k=n}^{i+1} u_{ik}x_k)/u_{ii}$
   END                                                                □
```

Figure 4.1: Gauss Elimination for Dense Linear Systems (SerGE)

Table 4.1: Operation Count of SerGE Algorithm

Step	*Substep*	*Operation Count*	
		$\times/\div$	+/-
Factor-ization	1.2.3	$(\times)\ \sum_{i=1}^{N}(N-i)^2 = \frac{1}{3}N^3\frac{1}{2}N^2+\frac{1}{6}N$ $(\div)\ \sum_{i=1}^{N}(N-i) = \frac{1}{2}(N^2-N)$	$\sum_{i=1}^{N}(N-i)^2$ $= \frac{1}{3}N^3-\frac{1}{2}N^2+\frac{1}{6}N$
Forward Subst.	2.2.3		$\sum_{i=1}^{N}(N-i)$ $= \frac{1}{2}(N^2-N)$
Back Subst.	3.2	$(\times)\ \sum_{i=N}^{1}(i-N+2) = \frac{1}{2}(5N-N^2)$ $(\div)\ \sum_{i=N}^{1}(i-N+2) = \frac{1}{2}(5N-N^2)$	$\sum_{i=N}^{1} 1 = N$

(2). Parallel GE (LU) for Dense Systems

Although the GE (or LU) method has been used in most applications as serial algorithm because of its very sequential nature, it does possess some degree of parallelism. We demonstrate hereunder one approach at paralleling the factorization step (step 1.2.3) of the SerGE. According to this step, a total of $n = 1, \ldots, n-1$ iterative stages will be needed to complete the factorization. The back substitution step is mostly serial and the parallelizing effort will not involve this step.

According to the factorization step of SerGE, a total of $k = 1, \ldots, N-1$ iterative stages will be needed to complete the factorization. At Nth stage of factorizing, all elements below a_{kk} in column k are effectively reduced to zeros. We show how the computation can be carried out using an array processor with p processors, but under the following assumptions:

<u>Assumption 1: $N = p$</u>

Row i of **A** is allocated on the local memory of processor p_i. This is known as straight-storage and it is shown in figure 4.2a below. At the first stage (i.e., $k = 1$) of reduction, processor P_1 broadcasts row$_1$ to processors $P_2, \ldots, P_N$, and the computation (see step 1.2.3)

$$l_{i1} = a_{i1}/a_{11}, \quad a_{ij} = a_{ij} - l_{i1}a_{1j}, \; j = 2, \ldots, N \tag{4.9}$$

are carried out by processors $P_2, \ldots, P_n$ in parallel. At stage $k = 1$, the process is repeated during which processor P_2 broadcasts row_2 to processors $P_3, \ldots, P_n$, and these processors compute

$$l_{i2} = a_{i2}/a_{22}, \quad a_{ij} = a_{ij} - l_{i2}a_{2j}, \; j = 3, \ldots, N. \tag{4.10}$$

This process of reduction process is repeated another $n - 2$ times to complete factorization of **A** into **L** and **U**. In actual implementation, **L** and **U** overwrite the original matrix **A**.

Although this process is easy to implement on any array processor, it, however, has 2 major drawbacks:

1). There is a considerable data communication at each stage.

2). The number of active processors decrease by one at a completion of each stage. The problem of existence of idle processors is known as load balance problem, and it is a serious drawback that, with judicious design of data structures, should be minimized if not altogether eliminated.

Assumption 2: $N \gg p$

Load balance difficulty may be mitigated by making judicious storage of **A**. Suppose p> n. Then the storage of the rows of **A** can be interleaved among the processors. Assuming that $n = kp$, rows 1, $p + 1$, $p + 2, \ldots$ are stored on processor 1, rows 2, $p + 2$, $2p + 2, \ldots$ on processor 2, and so on. Again, row 1 will be sent from processor P_1 to the other processors and then the computation of eq.(4.9) and eq.(4.10) will be done in blocks of k sets of operations in each processor. As before, processors will become idle as the computation proceeds, but this storage greatly alleviates the problem of processors becoming idle. This

```
ALGORITHM  ParGE()
  BEGIN
        step 1: broadcast(row 1)
        step 2:
          for $k \leftarrow 1, N-1$ do
            step 2.1: if $k \notin$ ROW$_k$ then
                        receive(row k)

            2.2: forall rows $(i > k)$ and $(k \in$ ROW$_k)$ do
                        $l_{ik} = a_{ik}/a_{kk}$
                        for $j \leftarrow k+1, N$ do
                          $a_{ij} = l_{il} - a_{ik}a_{kj}$
                        if $(i = k)$ and $(i \neq N)$ then
                          broadcast(row 1)
  END                                                      □
```

Figure 4.2: Parallel GE and LU Factorization for Dense Linear Systems (ParGE)

type of storage is known as wrapped-interleaved storage [ORTEGA 88]. We now give an algorithm based upon the wrapped-interleaved storage scheme. We call the algorithm ParGE for Parallel Gauss Elimination (and LU) method. In this algorithm, ROW$_i$ is a set of indices of rows of **A** allocated to a processor P$_i$. Thus, for processor P$_1$, ROW$_1$={1,p+1,2p+1,...,(k-1)p+1}, and for processor P$_3$, ROW$_3$={3,p+3,2p+3,...,(k-1)p+3}, and so on. In the ParGE algorithm, the computation of multipliers are begun while some part of the pivot row is received. For example, the computation of multipliers l_{i1} begin as soon as a_{11} is received rather than waiting for the entire pivot row to be received. This way, the communication delay is minimized, and the overall load-balance improved.

4.2.1.2 GE & LU Factorization for Banded Linear Systems

1. Serial GE (LU) for Banded Linear Systems The serial Gauss Elimination method (SerGE) given above for general dense linear systems is very inefficient for the solution of large sparse systems because of a potentially large number of zeros in such systems. The algorithm SerGE handles the matrix **A** and vector **k** in entirety, that is, all the elements of **A** and of vectors are processed according to the iterative steps of the algorithm. Since for our model problem **A** is highly sparse, SerGE would be very ill-suited indeed for the solution of such a system since a very large number of zero elements would be processed. Hence, to use SerGE to solve our model problem, an extensive modification must be in order, and , moreover, appropriate data structures capable of holding the nonzero elements of **A** and **k** are needed.

Below, we give a modified version of SerGE suitable for solution of block-tridiagonal linear systems depicted in eq.(4.2). We call the algorithm SerGET for Serial Gauss Elimination for Tridiagonal linear systems. Although SerGET is the Gaussian Elimination algorithm for block-tridiagonal systems, the same algorithm is used for LU factorization for tridiagonal systems, with slight modifications mainly aimed at retaining the resulting matrices **U** and **L**. The major distinctive feature of SerGET algorithm is a set of data structures used to hold mostly the nonzero elements of matrix **A**.

In the forward elimination step (step 1), three auxiliary vectors **u**, *l*, **y** are precomputed at every iteration step. These auxiliaries are functions of **A** only. In table 4.2, we give the operation count of the SerGET algorithm. The total number of iterative steps needed by the algorithm is $10n$. The four loops of SerGET in steps 1.2, 1.4, 1.6, and 2.2 are all sequential recurrences that must be evaluated one term at a time. This, together with the fact that the vector elements are

```
ALGORITHM SerGET()
  input: (1) a real array
         (2) k real vector
  output: (1) k real array   comment: k is overwritten with solution x
          (2) a real vector  comment: holds L & U
  Locals: (1) u real vector
          (2) l real vector
          (3) y real vector
  BEGIN
        step 1: comment Forward Elimination Step
              step 1.1: $u_1 \leftarrow k_1$
              step 1.2: for $i \leftarrow 2, N-1$ do
                        $u_i \leftarrow k_i - (a_i c_{i-1}/u_{i-1})$

              step 1.3: $l_2 \leftarrow a_2/k_1$
              step 1.4: for $i \leftarrow 2, N$ do
                        $l_i \leftarrow a_i/u_{i-1}$

              step 1.5: $y_1 \leftarrow k_1$
              step 1.6: for $i \leftarrow 2, N$ do
                        $y_i \leftarrow k_i - l_i y_{i-1}$

        step 2: comment Back Elimination Step
              step 2.1: $x_n \leftarrow y_n/u_n$
              step 2.2: for $i \leftarrow n-1, 1$ do
                        $x_i \leftarrow (y_i - x_{i+1} c_i/u_i$
  END                                                              □
```

Figure 4.3: Gauss Elimination for Tridiagonal Systems (SerGET)

referenced with unit increment and that the number of arithmetic operations is minimized, makes the algorithm ideally suited to serial computers. Equally, the

Table 4.2: Operation Count of SerGET Algorithm

Step	*Substep*	*Operation Count*	
		$\times/\div$	+/-
Forward Subst.	step 1.2	$3N-6$	
	step 1.4	$N-2$	
	step 1.6	$2N-2$	$N-1$
Back Subst.	step 2.2.4	$2N-1$	$N-1$

absence of any parallelism prevents the algorithm from taking any advantage of the parallel hardware features on a parallel computer. Golub, et al. [GOLUB 93], Ortega [ORTEGA 88], and Lambiotte [LAMBIO 75] have proposed some methods for parallelizing the algorithm for efficient computation on vector computers. But the resulting algorithms have exhibited only insignificant gains in performance over the sequential algorithm given above. Because of these difficulties stemming from the intrinsic sequential nature of the Gauss Elimination method, we conclude that SerGET is most unsuitable for solving a single tridiagonal systems of equations on parallel computers.

However, if one is faced with solving a set of m independent block-tridiagonal systems as often occurs in scientific and engineering systems, then SerGET algorithm would be best solved in parallel, and a maximum parallelism is realizable is m. For this to be possible, the vectors **u**, **l**, and **y** must be precomputed and saved, and the m systems are then executed in parallel with each system making use of the precomputed vectors which are the same for all the m systems. That is, only the left hand vector **k** would be different for each system while the precomputed vectors would be same. In this case, only step 2.2 needs be executed by each system thus resulting in a total of block operations is reduced to $3n$.

2. Parallel GE (LU) for Banded Systems. The simplicity of SerGET algorithm makes it very popular for serial computers. Only $3n$ block operations are required. It is this simplicity that makes it difficult for the more complex parallel algorithms to out-perform SerGET or SerGE algorithms. But, as stated in §1.4, we will attempt to parallelize this algorithm, ParGE, and use both the parallelized and the nonparallelized versions of the algorithm in our implementations and make conclusion on and recommendation for their application according to the result of their execution and test results. The model system in this study is a banded with a semibandwidth of $\beta = k$, where k is as defined in §2.3. Suppose Gauss Elimination or LU is to be used to factorize this system. Then multitiples of the first row are subtracted from row 2 through row $\beta + 1$. Since the first row (and the first column) has $\beta + 1$ elements, then only the first $\beta + 1$ rows and columns of **A** participate in the first stage of reduction. At the second stage of reduction, only elements 2 through $\beta + 2$ of rows 2 through $\beta + 2$ participate in the reduction, and so on. As this reduction continues, a point will be reached at which a $\beta \times \beta$ submatrix remains, and then the decomposition proceeds as with a a full matrix, that is, using SerGE. As in the parallel implementation ParGE for reduction of full linear systems, row or column wrapped interleaved storage illustrated above is recommended.

With this row (or column) interleaved assignment, at the first stage of reduction, rows $2,\ldots,\beta + 1$ will have an element in the first column to be eliminated. These β rows and the corresponding columns can then be updated in parallel. Hence, for $\beta < p$ (p is the number of available processors), only processors $2,\ldots,\beta + 1$ will be used during the first stage. Thus, we need $\beta \geq p$ in order to utilize fully all the processors. We conclude this discussion with a few words of caution: The method of parallelizing LU reduction is

a) recommended for a banded systems with large β, especially those with $\beta \geq p$

b) completely inadequate if the system has $\beta = 1$, that is, a tridiagonal system. In this case the final submatrix, being $\beta \times \beta = 1 \times 1$, would contain a single element,

c) more prone to load-imbalance even if $\beta \geq p$.

Because the parallel solution of banded system whose $\beta = 1$ is trivial, we will not try to parallelize the GE or LU methods when our system is cast in block tridiagonal form. That is, we will endeavor to parallelize only the original model system with $\beta = k$.

3. <u>Relating GE & LU IN The SerGET Algorithm</u>. Before leaving SerGET algorithm for the moment, we want to emphasize the fact that the auxiliary vectors **u** and **l** are the coefficients in the triangular decomposition of **A** into the product of a triangular matrix **L** and an upper triangular matrix **U**:

$$\mathbf{Ax} = \mathbf{LU},$$

where

$$\mathbf{L} = \begin{pmatrix} 1 & & 0 & & 0 \\ l_2 & 1 & & & \\ 0 & & & & \\ & & \cdots & & \\ & & l_{N-1} & 1 & 0 \\ 0 & & 0 & l_N & 1 \end{pmatrix} \tag{4.11}$$

and where

$$\mathbf{U} = \begin{pmatrix} u_1 & c_1 & 0 & & 0 \\ 0 & u_2 & c_2 & & \\ & & \cdots & & \\ & & & & 0 \\ & & & l_{N-1} & c_{N-1} \\ 0 & & 0 & & u_N \end{pmatrix} \tag{4.12}$$

The reader may have noted that in the above LU factorization, we have used the Croute's approach among the three most popular factorization methods characterized with the following properties:

$$\left.\begin{array}{ll} Croute: & l_{ii} = 1 \\ Doolittle: & u_{ii} = 1 \\ Choleski: & u_{ii} = l_{ii} \end{array}\right\} \forall i,\ 1 \le i \le N.$$

4.2.2 Orthogonal Factorization for Banded Systems

An alternative to the LU decomposition is the factorization

$$\mathbf{AP} = \mathbf{QR} \tag{4.13}$$

Here **A** is any mXn matrix, not just square matrix or, as in the case of Choleski factorization scheme a symmetric positive definite matrix, **P** is an nXn permutation matrix,**Q** an mXm orthogonal matrix, and **R** an mXn upper triangular matrix. There are a number of approaches to obtaining the factorization indicated in eq. (4.2.2). These usually differ in the way the permutation matrix **P** is derived and on the nature of the system. Two commonest such factorization approaches are the Householder and the Givens factorization schemes.

When matrix **A** (see eqn. (4.2.2)) are full rank, then the permutation matrix $\mathbf{P} = \mathbf{I}$, hence, eq. (4.2.2) becomes

$$\mathbf{A} = \mathbf{QR}, \tag{4.14}$$

and the orthogonal factorization is inherently numerically stable and can be used without pivoting. Most of scientific applications, such as the system used in this dissertation, are full rank, therefore, the Householder and the Given orthogonal factorizations can be used without pivoting. There are many applications which give rise to rank-deficient systems for which orthogonal factorization with some pivoting can typically be used. Pivoting is usually necessitated by the need to identify a basis for the range space of the columns of the matrix **A**. One such system arises during the so-called subset selection problem in statistics. Other applications are the solution of underdetermined or rank-deficient least squares problems ([AKE 89]) and nullspace methods of optimization ([COLE 84]).

Bischof ([BISCHOF 83]) has computed orthogonal factorization of rank-deficient systems well suited for implementation on high-performance computing systems with memory hirarchies. Such systems include Cray 2, Cray X-MP, and Cray Y-MP. Bischof used the Householder technique, the traditional technique of computing a QR factorization of a rank-deficient matrix **A** with column pivoting which can be viewed as choosing at every step the column which is farthest (i.e., with largest two-norm) from the subspace spanned by the columns previously selected. Here, **Q** is computed by a sequence of Householder transformations:

$$\mathbf{H} \equiv \mathbf{H}(\mathbf{w}) = \mathbf{I} - 2\mathbf{w}\mathbf{w}^{\mathbf{T}}, \ \|\mathbf{w}\|\ 2, = 1 \tag{4.15}$$

for which **w** is chosen to be

$$\mathbf{w} = \frac{x + (sign)(x_1)\|x\|2\, e_1}{\|x + (sign)(x_1)\|x\|2\, e_1\|2} \tag{4.16}$$

With the vector $\mathbf{w}$ so chosen, the solution vector (see eq. (4.1)) is reduced to a multiple of canonical unit vector e_1 since

$$(\mathbf{I} - 2\mathbf{w}\mathbf{w}^{\mathbf{T}})\mathbf{x} = -(sign)(x_1)\|x\|2\,e_1. \tag{4.17}$$

Of the two orthogonal transformation given above, we choose the Householder for designing the benchmark for our system. We demonstrate below the usual reduction steps involved in the orthogonal reduction by means of the Householder transformation for the case of general dense linear system to ease the demonstration. But these reductions will be dully modified for block-tridiagonal systems.

4.2.2.1 Householder Factorization For Dense Linear Systems

We do not use the above approach for orthogonal QR factorization given in eqs. (4.2.1 - 4.2.1) since the systems for which they apply are rank-deficient and the systems used in this study are full-rank. So, we use the factorization indicated in eq. (4.2.2). We give a brief description of this approach for both the Householder and the Givens factorization techniques. The conventional factorization on the full-rank matrix $\mathbf{A}$, as it has been stated before, have an inherent numerical stability and, therefore, require no explicit pivoting.

The Householder factorization is the matrix of the form

$$\mathbf{I} - \mathbf{w}\mathbf{w}^T$$

where $\mathbf{w}$ is a real column vector such that $\mathbf{w}^{\mathbf{T}}\mathbf{w} = 2$. This transformation, which is both symmetric and orthogonal, is used to compute matrix

$$\mathbf{P} = \mathbf{P}_{\mathbf{n}-1}\mathbf{P}_{\mathbf{n}-2}\dots\mathbf{P}_1 \;\; where\mathbf{P}_{\mathbf{i}} = \mathbf{I} - \mathbf{w}_{\mathbf{i}}\mathbf{w}^{\mathbf{T}}. \tag{4.18}$$

P_i in eq.(4.2.1)is determined in the following way with the w_i as the ith column vector of $\mathbf{A}$ but having the first $i-1$ elements to be zeros.

Let $\mathbf{a_i}$ be the ith column vector of $\mathbf{A}$. Also, let

$$\mathbf{u}^T = (a_{ii} - s, a_{2i}, \ldots, a_{ni}), \ \mathbf{w_i} = \mu \mathbf{u}, \tag{4.19}$$

where

$$\mathbf{u}^T = \pm(\mathbf{a_i^T a_i}^{\frac{1}{2}}, \ \gamma = (s^2 - a_{ii}s)^{-1}, \ \mu = \gamma^{\frac{1}{2}}, \tag{4.20}$$

and the sign of s is chosen to be opposite that of a_{ii} for numerical stability. Then

$$\mathbf{w_i^T w_i} = \mu^2((a_{ii} - s)^2 + \sum_{i=2}^{n} a_{ii}^2) = \mu(s^2 - a_{ii}s) = \frac{1}{\mu}$$

so that

$$\begin{aligned} a_{ii} - \mathbf{w_i w_i^T a_i} &= a_{ii} - \frac{\mu(a_{ii} - s)}{\mu} = s \\ a_j i - \mathbf{w_i w_i^T a_i} &= a_{ji} - \frac{a_{ji}\mu}{\mu} = 0 \ j = i+1, i+2, \ldots, N \end{aligned}$$

which shows that the subdiagonal elements on column i of $\mathbf{A}$ have been reduced to zeros. At the ith step of this reduction, $\mathbf{P}_i$ is determined to be $\mathbf{P}_i = \mathbf{I} - \mathbf{w_1 w_1^T}$. The Householder algorithm computes the matrices $\mathbf{P}_i$ for $i = 1, 2, \ldots n-1$ using the column vectors $\mathbf{w_i}$ as defined above. The $\mathbf{P}_i$ so determined are used to compute matrix $\mathbf{P}$ according to eq.(4.2.1). With $\mathbf{P}$, $\mathbf{R}$ and $\mathbf{Q}$ of eq.(4.2.2) are determined with

$$\mathbf{PA} = \mathbf{P_{n-1} P_{n-2} \ldots P_1 A} = \mathbf{R} \tag{4.21}$$

$$\textit{and} \ \ \mathbf{Q} = \mathbf{P}^{-1} \tag{4.22}$$

where $\mathbf{R}$ is upper triangular, and the matrices $\mathbf{P}_i$ are all orthogonal so that $\mathbf{P} = \mathbf{P}_{n-1} \ldots \mathbf{P}_1$ and $\mathbf{P}^{-1}$ are also orthogonal matrics. The bulk of the work in carrying out the Householder reduction to triangular form is updating the nonzero columns of $\mathbf{A}$ at each stage. The Householder transformation for linear dense systems is given in Figure 4.4.

```
ALGORITHM HouseHolder()
  input: a real array
  output: a real array
  Local: (1) u real vector
         (2) s real vector
         (3) $\alpha$ real vector
         (4) $\gamma$ real vector
  BEGIN
        step 1.: for k ← $1, N-1$ do
              step 1.1: $s_k \leftarrow -sgn(a_{kk}) (\sum_{l=k}^{n} a_{lk}^2)^{\frac{1}{2}}, \gamma_k = (s_k^2 - s_k a_{kk})^{-1}$
              step 1.2: $u_k^T \leftarrow (0, \dots, 0, a_{kk} - s_k, a_{k+1,k}, \dots, a_{nk})$
              step 1.3: $a_{kk} \leftarrow s_k$
              step 1.4: for j ← $k+1, N$ do
                    step 1.4.1: $\alpha_j \leftarrow (\gamma_k u_k^T) a_j$
                    step 1.4.2: $a_j \leftarrow a_j - alpha_j u_k$
  END                                                              □
```

Figure 4.4: Householder Reduction for Dense Linear Systems

4.2.2.2 Householder Factorization For Banded Linear Systems

We consider the Orthogonal reduction of a banded linear system having a semiband-width of β. The reduction, like in the case of LU reduction, also expands the band-widths above the main diagonal. When the Householder transformation $\mathbf{I} - \mathbf{w}\mathbf{w}^T$ is applied to matrix **A** in order to zero elements of the first column below the main diagonal, then **w** is of the form $\mathbf{w}^T = (*, \dots, o, \dots, 0)$ in which the first $\beta + 1$ elements are, in general, nonzero. Then the new *ith* column of **A** is $\mathbf{a_i} - \mathbf{w^T a_i w}$, where $\mathbf{a_i}$ is the *ith* column of **A**. The first Householder transformation introduces, in general, nonzero elements outside the original band. No further nonzero elements are introduced since, on the first transformation, $\mathbf{w^T a_i} = 0$ for $i > 2\beta + 1$. On the second Householder transformation, nonzeros will be introduced into $2\beta + 2$ column, starting in the second element. At each subsequent transformation, nonzeros will be introduced in one more column so that at the end of the triangular reduction, the upper triangular matrix will have a bandwidth of $2\beta + 1$.

4.3 ITERATIVE BENCHMARKS

Iterative methods are commonly used to solve large, sparse systems of linear equation for three major reasons:

1. Iterative methods require a minimum of storage, usually on the same order as the number of nonzero entries of the matrix $\mathbf{A}$ (see eq.(4.1)), whereas a direct methods (e.g., Gauss Elimination) usually cause fill-in of $\mathbf{A}$ as their computation with $\mathbf{A}$ proceeds, and, therefore, the advantages of sparcity of $\mathbf{A}$ are usually partly lost.
2. Generally, iterative methods lend themselves to easy parallelizing than do direct methods. We learned this during the parallelizing attempts of some direct methods in §4.1. Many iterative methods (e.g., Jacobi) have inherent parallel features which are usually obvious and easy to exploit, although such exploitation could be limited by less-than-satisfactory convergence rates.
3. Iterative methods reduce the amount of actual computation needed to solve linear system of equations. This is especially true for our model problem if cast in three dimensions.

Iterative methods, however, have some disadvantages over direct methods.

1. Iterative methods require initial guess of the solution vector $\mathbf{x}$ (see eq.(4.1)). Direct methods, on the other hand, at least theoretically, do not involve iteration and, therefore, do not require initial guesses of solution.

2. The rate of convergence of iterative methods are dependent on the quality of the initial guess of the solution vector, whereas, direct methods, because they do not require initial guesses of solution, theoretically should give exact solution. Of course, this is seldom the case chiefly because of limitations imposed by round-off, and other computation-related errors.

3. The convergence rates of iterative methods are generally poorer (e.g., Jacobi) than those of direct methods, and these rates are almost always improved with some mathematical means (e.g., relaxation techniques), and by reformating the methods accordingly.

In this section, we intend to develop some parallel algorithms based on some iterative methods and then use such algorithms as benchmarks for testing the new algorithm for parallel solution of the model problem. The iterative approaches that are used are the symmetric over-relaxation method or SOR, and the conjugate gradient method. We will follow the approach of algorithmic development adopted in §4.1 — a brief overview of the essential steps embodied in the formulation of the chosen method when applied for solving dense linear systems, and then parallelizing the method mainly along those steps for the solution of the model problem on a parallel processor.

4.3.1 Symmetric Over-relaxation Method (SOR)

4.3.1.1 The SOR Method For Dense Linear Systems

Assume that the linear system depicted in eq.(4.1) is dense, that an $N \times N$ matrix $\mathbf{A}$ is nonsingular, and that $\mathbf{x}$ and $\mathbf{k}$ are n-dimensional vectors. Let $\mathbf{D}$ be diagonal matrix of $\mathbf{A}$, and $\mathbf{Z} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$. Then eq.(4.1) can be rewritten as

$$\mathbf{D}^{-1}\mathbf{A}\mathbf{x} = \mathbf{D}^{-1}\mathbf{k}$$

$$\begin{aligned}\mathbf{x}+(\mathbf{D}^{-1}\mathbf{A}-\mathbf{I})\mathbf{x} &= \mathbf{D}^{-1}\mathbf{k} \\ \mathbf{x} &= (\mathbf{I}-\mathbf{D}^{-1}\mathbf{A})\mathbf{x}+\mathbf{D}^{-1}\mathbf{k}, \\ &= \mathbf{Z}\mathbf{x}+\mathbf{D}^{-1}\mathbf{k}, \qquad (4.23)\end{aligned}$$

where $\mathbf{I}$ is an $N \times N$ identity matrix. Finally, using the superscript "(i)" to denote iterative level ($i = 0, 1, \ldots,$), then eq.(4.23) may be rewritten as

$$\mathbf{x}^{(i+1)} = \mathbf{Z}\mathbf{x}^{(i)} + \mathbf{D}^{-1}\mathbf{k}. \qquad (4.24)$$

The iterative algorithm depicted in eq.(4.24) is known as the Jacobi method. It is a very important method in that, despite its slow convergence rate, it is usually regarded as a starting point for developing other iterative methods such as Gauss-Seidel, SO, SSOR, LSOR, Chebychev SOR, to name but a few. These methods have convergence rates more superior to that of the Jacobi method. Another attractive feature of the Jacobi method is that it has inherent parallelism. In Figures 4.5 and 4.6, we have given the conventional Jacobi method based upon eq.(4.24) (SerJAC) and its parallel version (ParJAC). The ease with which the parallelized algorithm is formed from the nonparallelized one testifies to the inherent parallel nature of the Jacobi method. In the SerJAC we have used the vector norm, $\| * \|$, for test of convergence at step 2. Similar test can be applied to the ParJAC.

The Jacobi method is ideally suited for implementation on parallel computers. In the case of array processor, a straight-storage scheme in which each element x_i of $\mathbf{x}$ is allocated on processor P_i is adequate. Then parallel adjustment of subsequent values of $\mathbf{x}$ is carried out on each processor using the computation of eq.(4.24). No inter-processor communication or data routing is necessary since, at each iteration level, processor P_i only uses x_i on its local storage.

The iteration provided by the Jacobi approach as given in eq.(4.24) can not be applied to our study without substantial modification mainly to speed up the rate

```
ALGORITHM  SerJAC()
  input: (1) A real  array
         (2) x,k real  vector
         (3) $\varepsilon$ real  scalar
        comment: $\varepsilon$ used to stop iter.

  output: x real vector
        comment:  x is Guessed: x=$(x_0^{(0)}, x_1^{(0)}, \ldots, x_{n-1}^{(0)})$
  BEGIN
    step 1: Precompute matrices the D & Z
    step 2: for  i $\leftarrow 1$, forever, ..., $n$  do
      $\mathbf{x}^{(i)} = \mathbf{Z}\mathbf{x}^{(i-1)} + \mathbf{D}^{-1}\mathbf{k}$
      if $\|x^i - x^{i-1}\| \leq \varepsilon$ then
        return($x^{(i)}$)
  END                                                          □
```

Figure 4.5: Serial Jacobi Method for Dense Linear Systems (SerJAC)

```
ALGORITHM  ParJAC()
  BEGIN
    step 1: Precompute matrices the D & Z
    step 2: Allocate $\mathbf{x} = (x_0, \ldots, x_{n-1})$ such that $x_i$ is on local
            memory of processor $P_i$
    step 3: forall ($P_i$) do
      $\mathbf{x}^{(i)} = \mathbf{Z}\mathbf{x}^{(i-1)} + \mathbf{D}^{-1}\mathbf{k}$
  END                                                          □
```

Figure 4.6: Parallel Jacobi Method for Dense Linear Systems (ParJAC)

of convergence. Unfortunately, as we will see shortly, speeding up convergence rate comes with a severe price — the loss of obvious parallelism in the original Jacobi method.

The fact that Jacobi method, or any method for that matter, can be efficiently implemented on parallel computers does not necessarily mean that it is a good method to use because one must also consider the number of iterations that are required to obtain a satisfactory convergence. The Jacobi method, becuse of its slow convergence rate, must be modified so that it can achieve a better rate of convergence.

The modification is usually achieved by combining the Jacobi method with the Gauss-Seidel approach. The gauss Seidel method performs iteration as given in eqs.(4.23, 4.24) above but with one important difference: in the computation of $\mathbf{x}^{(i+1)}$ from $\mathbf{x}^{(i)}$, computer values for $\mathbf{x}^{(i+1)}$ are substituted for values in $\mathbf{x}^{(i)}$ as soon as they are available during the course of computation. In other words, the computation of $\mathbf{x}^{(i+1)}$ commences as soon as $\mathbf{x}^{(i)}$ becomes available at the same iteration level, whereas in Jacobi or JOR, iteration at a level commences after cessation of iterations at previous level. The Gauss-Seidel method involves computing $\mathbf{x}^{(i+1)}$, and immediately setting $\mathbf{x}^{(i)} \leftarrow \mathbf{x}^{(i+1)}$.

The combination of the Gauss-Seidel method and overrelaxation method is called succesive overrelaxation method or SOR. Experience and theoretical results show that SOR, most of the times, converges faster than Jacobi and JOR methods. SOR amounts to using the computation:

$$\mathrm{x}^{(i+1)} = \mathcal{L}_\omega x^{(i)} + (1-\omega \mathrm{L})^{-1}\omega \mathrm{Dk} \tag{4.25}$$

where $\mathbf{D}$ is the diagonal matrix of $\mathbf{A}$, $\mathbf{Z}$ as defined before (see eq.(4.24)), $\mathbf{U}$, $\mathbf{L}$ are respectively the upper and lower triangular matrices of **Z**,

$$\mathcal{L}_\omega = (\mathbf{I} - \omega\mathbf{L})^{-1}(\omega\mathbf{U} + (1-\omega)\mathbf{I}),$$

where ω chosen such that $1 \leq \omega \leq 2$. If $\omega = 1$, it is very easy to prove that the above SOR algorithm will degenerate to the Jacobi method.

4.3.1.2 The SOR Method for Banded Linear Systems

As we already remarked, the parallel solution of the SOR algorithm as given in eq.(4.25) is not as easy or as intuitive as that of the Jacobi approach, but that the Jacobi approach has unaccetable rate of convergence. So, instead of developing a parallel SOR algorithm from eq.(4.25), we carry out such development indirectly using eq.(3.17), the elliptic equation from which we obtained the model problem. That equation (eq.(3.17)) is rewritten as eq.(4.27) below for convenience:

$$4U_{ij} - U_{i+1j} - U_{i-1j} - U_{ij+1} - Uij - 1 = k^2 f(i,j). \tag{4.26}$$

Eq.(4.26) is the result of finite difference discretization of elliptic equation over a rectangular region $(1,0) \times (1,0)$, and partitioning the region k times in both $x-$ and $y-$ coordinate directions. The result is a mesh region already given in Figure 3.1. At a mesh point (i,j), a solution u^* can be obtained from eq.(4.26) as

$$U^*_{ij} = U_{i+1j} + U_{i-1j} + U_{ij+1} + Uij - 1 + k^2 f(i,j). \tag{4.27}$$

1. <u>A Parallel SOR for Banded Systems</u>. We develop a parallel SOR algorithm that can solve eq.(4.27) simultaneously at all mesh points. Such a solution involves a simultaneous replacement of the "old" values of U on the right hand side of that equation by the "new" values of U^* on the left hand side of the equation. This approach is nothing more than the Jacobi method described before, and, as we observed then, such a simultaneous replacement is ideally suited for implementation on parallel computers. Simultaneous adjustment means that eq.(4.27) can be evaluated at all mesh points in parallel with the maximum parallelism of n^2. In Typical scientific and engineering applications, $N = 32$ to 256, so that parallelism varies from about 1000 to about 64,000. This leads to satisfactorily long vectors for efficient implementation on pipelined computers. On the other hand, for array processor such as MasPar MP-X, the mesh can probably be chosen to fit the machine size or multiples of it. Since each mesh point needs values of U from at most 4 neighbors, a nearest-neighbor communication, the obvious communication primitive to use for inter-processor communication and data routings are the Xnet or Xnetc available in MPL of the MP-X.

Because of the slow convergence rate, the above Jacobi algorithm for simultaneous adjustment of u at all mesh points will not be used. Instead we use the SOR method, which, by direct algebraic manipulation of the SOR equation given in eq.(4.24), gives in the present case the following solution which is equivalent to eq.(4.27):

$$U_{ij}^{new} = \omega U_{ij}^{*} + (1-\omega)U_{ij}^{old}, \tag{4.28}$$

where, as before, ω is a constant relaxation factor used improve the convergence rate and chosen such that $1 \leq \omega \leq 2$. The SOR equation given in eq.(4.28) degenerates to the Jacobi equation of eq.(4.27) when $\omega = 1$. For our model problem, the best convergence rate is obtained with

$$\omega = \omega_b = \frac{2}{1+(1-\rho^2)^{\frac{1}{2}}}, \tag{4.29}$$

where $\lambda = \omega_b - 1$, and ρ is the convergence factor corresponding to the Jacobi iteration. Therefore, $\rho = \cos(\pi/N)$ for the model problem.

The results of direct application of eq.(4.28) for simultaneous adjustment of u at mesh points as found in this study is less than satisfactory, in terms of convergence rate, even when $\omega = 2$. The result was improved by the so-called mesh sweep method suggested by Hockney [HOCK 81], Golub et al. [GOLUB 93], and others. One of the best pattern mesh sweep algorithms, in fact, the one adopted in this study, is the odd/even ordering with the Chebychev acceleration. Each odd/even pair is assigned to a processor. In this method, the mesh points are divided into two groups according to whether $i+j$ is odd or even. The method proceeds in half iterations, during each of which only half the points are adjusted such that the odd and even points are adjusted at each half iteration alternately. In addition, the value of ω changes at each half iteration according to:

$$\begin{aligned}
\omega^{(0)} &= 1 \\
\omega^{(\frac{1}{2})} &= \frac{1}{(1-\frac{1}{2}\rho^2)} \\
\omega^{(t+\frac{1}{2})} &= \frac{1}{(1-\frac{1}{4}\rho^2\omega^{(t)})}, \quad t = \frac{1}{2}, 1, \ldots, \infty,
\end{aligned} \tag{4.30}$$

where, as before, the superscript designates the iteration level. We will implement this modified SOR on MasPar MP-X in chapter 5.

4.3.2 Conjugate Gradient Method (CG)

The Conjugate Gradient method is the last in our set of iterative approaches for solving the type of linear system of equations proposed in this study. As always, we first show how the method is used for generating an algorithm for solving the general dense linear systems, and then modify the algorithm so formed for solving the sparse system of equations from the model problem.

```
ALGORITHM SerCG()
  input: (1) A real array
         (2) x, k real vector

  output: k real vector
            comment: k is overwritten by x
  Locals: (1) r real vector
            comment: residual vector
  Locals: (2) p real vector
            comment: p is direction vector
          (3) $\alpha, \beta$ real scalar

  BEGIN
    step 1: Choose $\mathbf{x}^{(0)}$
      Compute $\mathbf{r}^{(0)} = \mathbf{k} - \mathbf{A}\mathbf{x}^{(0)}$
      Set $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$
      Set $i = r^0$

    step 2: Compute
      $\alpha^{(i)} = (\mathbf{r}^{(i)}, \mathbf{r}^{(i)})/(\mathbf{p}^{(i)}, \mathbf{p}^{(i)})$
      $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \alpha^{(i)}\mathbf{p}^{(i)}$
      $\mathbf{r}^{(i+1)} = \mathbf{r}^{(i)} - \alpha^{(i)}\mathbf{A}\mathbf{p}^{(i)}$
      $\beta^{(i)} = (\mathbf{r}^{(i+1)}, \mathbf{r}^{(i+1)}/(\mathbf{r}^{(i)}, \mathbf{r}^{(i)})$
      $\mathbf{p}^{(i+1)} = \mathbf{r}^{(i+1)} + \beta^{(i)}\mathbf{p}^{(i)}$

    step 3: if $\|\mathbf{r}^{(i+1)}\|_2 \geq \varepsilon$ then
              goto step 2
  END                                            □
```

Figure 4.7: Conjugate Gradient Method for Dense Linear Systems (SerCG)

4.3.2.1 The CG Method For Dense Linear Systems

Given a linear system of equations as in eq.(4.1) where, as before, **A** is an $N \times N$ symmetric positive-definite, **x** and k $N-$dimensional vectors. Let $\mathbf{x}^{(0)}$ be the initial guess of the solution vector **x**. Then the conjugate gradient method minimizes an appropriate measure of the error, $\mathbf{x} - \mathbf{x}^{(k)}$ where $\mathbf{x}^{(k)}$ is the approximation of **x** at the *kth* step of iteration. With the error functional

$$E(\mathrm{x}^{(k)}) \;=\; (\mathrm{x} - \mathrm{x}^{(\mathrm{k})}, \mathrm{A}(\mathrm{x} - \mathrm{x}^{(\mathrm{k})})), \tag{4.31}$$

is the unique minimum reached when $\mathbf{x}^{(\mathbf{k})} = \mathbf{x}$, since **A** is positive-definite. The CG method minimizes the error functional over subspaces of increasing dimension. At step k, a new direction, $\mathbf{p}^{(\mathbf{k})}$, is chosen, and $\mathbf{x}^{(\mathbf{k}+1)}$ is calculated to be $\mathbf{x}^{(\mathbf{x}+1)} = \mathbf{x}^{(\mathbf{k})} + \alpha^{(\mathbf{k})}\mathbf{p}^{(\mathbf{k})}$, where $\alpha^{(\mathbf{k})}$ minimizes $\mathrm{E}(\mathbf{x}^{(\mathbf{k}+1)})$ as a function of $\alpha^{(\mathbf{k})}$. The direction vectors, $\mathbf{p}^{(\mathbf{o})}, \mathbf{p}^{(1)}, \ldots$ are **A**-orthogonal. That is, $\mathbf{p}^{(\mathbf{i})}, \mathbf{A}\mathbf{p}^{(\mathbf{j})} = \mathbf{0} \; i \neq j$. Also, minimizations in one direction actually corresponds to minimizations in the entire subspace spanned by $\{\mathbf{p}^{(0)}, \ldots, \mathbf{p}^{(\mathbf{k})}\}$. This is why the CG method terminates with the exact solution in at most n steps if exact arithmetic is used. That is, if used as a direct method, CG will converge in at most n number of steps. But, if used as an iterative method, it may converge at a significantly less number of steps if a "good" approximation to the solution is found. It is empiracally found that, the conjugate gradient method will converge fater if the eigenvalues of **A** cluster around unity. A large number of improvement techniques aimed at improving the convergence rate of the conventional CG approach concentrate on approximating **A** with matrix **M**[23] which gives a better distribution of eigenvalues than does **A**. Improvement of the convergence rate of CG is usually done through techniques collectively known as preconditioning, and they commonly involve replacement of **A** by its approximation **M**. For excellent treatment of a variety of preconditioning of

[23]Commonly chosen to be diagonal matrix of **A**.

CG, see the work of Baucom [BAUCOM 88]. Figure 4.7 is an algorithm, SerCG, for solving a dense linear system based upon the above described formulations. Note that the function $(\mathbf{x}, \mathbf{y})$ is inner product. That is, $(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$. Also, note that $\mathbf{r}$ is residual. Note also that we have used the vector norm for the convergence test at step 3 following the suggestion by Ortega [ORTEGA 88]. Vector norm has the advantage of requiring virtually no extra work since $(\mathbf{r}^{(i+1}, \mathbf{r}^{(i+1)})$ (of step 3) will be needed at step 2 of the next iteration level if convergence has not already occured.

The above conjugate gradient algorithm for dense linear systems is easily extended to solve in parallel banded systems such as used in this study. On an array processor such as MP-X, components of vectors such as **x,p,r** are assigned to individual processors and the vector operations performend on these processors. The multiplication of $\mathbf{A}\mathbf{p}^{(i)}$ follows the treatment of Section 4.2.

4.4 THE NEW TECHNIQUE

We dedicate this section for development of the new algorithm that we believe is efficient enough to satisfactorily execute the model problem. At the core of this algorithm design are two powerful direct methods — cyclic reduction and recursive doubling techniques. There is nothing new about the use of either of the two methods but their use in combination to construct an algorithm for efficient solution linear systems is the uniqueness of their application. Cyclic reduction was developed by Hockney who also was the first to use it to solve tridiagonal equations on the IBM 7090, a serial processor [HOCK 65]. This method was chosen in lieu of the Gaussian Elimination technique because it deals with periodic boundary conditions in a neater fashion than does the elimitation method. For example, cyclic reduction eliminates computing of auxiliary data structures (see

§4.1.1A) necessitated by the elimination technique. Since its first application by Hockney, this method, either in its original form or in one or more of its many modified vesions, has been extensively used by a large number of investigators to solve various sparse systems ([SWEET 73, SWEET 74, TEMP 80, SWARZ 89, GOLUB 93]).

Recursive doubling method, proposed by Stone [STONE 73] is a very efficient technique for parallelizing recursive sequential algorithms. In §4.1.1A, we demonstrated the difficulties that accompany any attempt to parallelize the GE and the LU factorization methods, and stated that these methods are very efficient to use on serial processors because of their inherent serial nature, and that any advantage that comes as a result of use of parallelized forms are miniscule, and, as a result, these methods, are more often than not, used in their serial forms. One major difficulty in parallelizing these codes is the fact of existence of recurrences in their mathematical forms given in steps 1.2, 1.6 and 2.2 which are respectively the the forward elimination and the back substitution of the SerGET algorithm in figure 4.4. For convenience, we rewrite these equations below:

$$\begin{aligned}
u_1 &= k_1 \\
u_i &= k_i - (a_i c_{i-1}/u_{i-1}), \quad 2 \leq i \leq N-1 && (4.32) \\
l_2 &= a_2/k_1 \\
l_i &= a_i/u_{i-1}, \quad 2 \leq i \leq N && (4.33) \\
y_1 &= k_1 \\
y_i &= k_i - l_i y_{i-1}, \quad 2 \leq i \leq N && (4.34) \\
x_n &= y_N/u_N \\
x_i &= (y_i - x_{i+1} c_i)/u_i && (4.35)
\end{aligned}$$

The forward elimination step is represented by eqs.(4.32, 4.33, and 4.34), while eq.(4.35) determines the back substitution step. Obviously, there are recursions in eqs.(4.32, 4.33, 4.34) of the forward elimination step, and in eq.(4.35) of the back elimination step. Eq.(4.32) is said to be nonlinear-recursive because of the occurence of the nonlinearity, u_{i-1}^{-1}, while eqs.(4.33, 4.34, and 4.35) are said to be linear-recursive.

The presence of any type of recursion in an equation renders an algorithm that simulates the execution of that equation to be strictly serial for a very obvious reason. Consider a general linear-recursive equation in which x, a, d are$n$-dimensional vectors:

$$\begin{aligned} x_1 &= d_1 \\ x_i &= a_i x_{i-1} + d_i, \quad 2 \leq i \leq N \end{aligned} \tag{4.36}$$

The value of the solution vector x at $i = 1$ is simply the value of d_1. But the vlaue of x_j $(2 \leq j \leq N)$ depends on the value of x_{j-1}, the value computed in the previous step. Thus, for x_j to be evaluated, x_{j-1} must first be computed. This algorithm is strictly serial, simple and elegant and can most be efficiently carried out on a serial machine. Parallelizing it often results in more complicated and, possibly, less efficient code. Recursive doubling method affords a very neat introduction into a recursion.

In this section, both the recursive doubling algorithm and the cyclic reduction technique are given. Then these are followed by a discussion of how these approaches are combined to solve our model problem.

4.4.1 Recursive Doubling for Linear Recursions

We now demonstrate the principal steps in the development of recursive doubling algorithm using the linear recursion provided in eq.(4.36). The idea of recursive doubling is to rewrite that equation in terms of x_{2i} instead of x_i so that successive iterations can compute $x_1, x_2, x_4, \ldots$ thus using only The computation of $x_{2i}(i = 1, 2, \ldots)$ depends on that of x_i, and it has the complexity that is double that of the computation of x_i. Hence, the name recursive doubling. A repeated substitution for y_{i-1} in this equation, the following set of equations results:

$$\begin{aligned} x_1 &= d_1 \\ x_2 &= a_2 d_1 + d_2 \\ x_3 &= a_3 a_2 d_1 + a_3 d_2 + d_3 \\ &\vdots \\ x_i &= \sum_{j=1}^{i} d_j \prod_{k=j+1}^{i} a_k \end{aligned} \tag{4.37}$$

Eq.(4.37) shows an explicit dependency of x upon the coefficients of vectors d and a. The goal is to derive a recurrence in which x_{2i} is a function of x_i. This is done in the following way: Let the sum of k_j to k_{j-i+1} be represented as $X_i(k_j, k_{j-1}, \ldots, k_{i+1}$, Let the sum of k_j to k_{j-i+1} be defined as $y_i(k_j, k_{j-1}, \cdots, k_{j-1+1})$, and the function $A_i(j)$ be defined as

$$\begin{aligned} A_i(j) &= \prod_{k=j-i+1}^{j} a_k \;\; for \;\; j \geq i \\ &= \prod_{k=1}^{j} a_k \;\; for \;\; j < i. \end{aligned} \tag{4.38}$$

and the function $D_i(j)$ as

$$\begin{aligned} D_i(j) &= \prod_{k=j-i+1}^{j} d_k \;\; for \;\; j \geq i \\ &= \prod_{k=1}^{j} d_k \;\; for \;\; j < i. \end{aligned} \tag{4.39}$$

then eq.(4.37) can be rewritten as:

$$X_{2i} = X_i(j) + X_i(j-1)A_i(j) + D_j(j). \tag{4.40}$$

```
ALGORITHM ParRD()
  input: a,d real vector
  output: x real vector
          comment: a is overwritten by x
  local: A,D real vector

  BEGIN
   step 1: Initialization Step
    forall $(1 \leq i \leq N)$ do
     $\mathbf{A}_j \leftarrow a_j \ (1 \leq j \leq N)$
     $\mathbf{D}_j \leftarrow d_j \ (1 \leq j \leq N)$

   step 2: comment: Solution of 4.40
   for $i \leftarrow 1, N/2, \text{step}i$ do

    step 2.1: forall $(1 \leq i \leq N/2)$ do
     $\mathbf{X}_j \leftarrow X_j + X_{i-1} \ (1 \leq j \leq N)$
    step 2.2: forall $(1 \leq i \leq N/2)$ do
     $\mathbf{A}_j \leftarrow A_j.A_{j-1} \ (1 \leq j \leq N)$

    step 2.3: forall $(1 \leq i \leq N/2)$ do
     $\mathbf{D}_j \leftarrow D_j.D_{j-1} \ (1 \leq j \leq N)$
  END                                    □
```

Figure 4.8: Recursive Doubling Technique for Linear-Recursive Problems

The recursive doubling computation of $A_i(j)$ is provided by the formula:

$$A_{2i}(j) = A_i(j)A_i(j-1) \quad for\ i,j \geq 1. \tag{4.41}$$

$$D_{2i}(j) = D_i(j)D_i(j-1) \quad for\ i,j \geq 1. \tag{4.42}$$

Eqs.(4.40, and 4.41,4.44) are executed together to provide the recursive doubling solution of eq.(4.36). The parallel solution of these equations are provided by the algorithm ParRD below. The total operation count of recursive doubling algorithm is $\mathrm{O}\log_2 n$. The above recursive doubling solution applies to any linear-recursive problems. Because forward elimination and back substitution steps of the LU factorization given in eq.(4.34) and eq.(4.35) respectively, are linear-recursive we can rewrite eqs.(4.40 and 4.41) for recursive doubling computations of these equations. For eq.(4.34), for example, eqs.(4.40 and 4.41) become

$$\begin{aligned} Y_{2i}(j) &= Y_i(j) + Y_i(j-1)M_i(j), \quad (4.43) \\ M_{2i}(j) &= M_i(j)M_i(j-1) \quad for i, j \geq 1, \quad (4.44) \end{aligned}$$

where

$$\begin{aligned} M_i(j) &= \prod_{k=j-i+1}^{j} (-m_k) \quad for \quad j \geq i \\ &= \prod_{k=1}^{j} (-m_k) \quad for \quad j < i. \end{aligned}$$

Eqs.(4.44, and 4.44) can be computed using ParRD algorithm given in figure 4.11.

4.4.2 Cyclic or Odd-Even Reduction

The cyclic (odd-even) reduction is used most often on block tridiagonal equations, such as eq.(4.2), resulting from finite difference approximation of elliptic equation. Example of such system is the model problem used in this study. This technique also has applicability to tridiagonal systems, although the fact that it is more complex than the GE or LU factorization makes it an unlikely candidate for a general tridiagonal system on a serial computer. However, cyclic reduction technique is a lot more convenient for solving periodic boundary condition problems than are most other solution techniques including the GE and LU reduction methods.

When first used by Hockney /citehockney65 to solve tridiagonal system arising from the finite difference discretization of Poisson equation on a rectangular domain, the number of mesh points, therefore, the number of equations was assumed to be a power of two. Further modification aimed at generalizing the technique for solving systems arising from irregular domains, for example, makes these restrictions unnecessary. But in this work, for simplicity, we will assume that the number of equations is a power of two. We summarize the assumptions in the following equations in which n has the usual meaning and q an integer:

Assumption 1: $N = \acute{n} - 1$.

Assumption 2: $\acute{n} = N^q - 1$. That is, the size of the system is odd.

The cyclic reduction technique starts by eliminating certain of the coefficients in a tridiagonal system by elementary row operations so that the modified equations contain references to only one half of the original (modified, of course) variables. Under a renumbering of the remaining unknowns, the new system is again tridiagonal, but only half as large. We demonstrate this technique using the tridiagonal system depicted in eq.(4.2). Writing three adjacent equation (from eq.(4.2)), we have for $i = 2, 4, \dots, \acute{n} - 2$,

$$\begin{aligned} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i &= k_{i-1} \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= k_{i-1} \qquad (4.45) \\ a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} &= k_{i+1}, \qquad (4.46) \end{aligned}$$

where $x_0 = x_{\acute{n}} = 0$. If the first of these equations is multiplied by $\alpha_i = -a_i/b_{i-1}$, and the last equations by $\gamma = -c_i/b_{i+1}$, and the three equations added, we obtain

$$a_i^{(1)}x_{i-2} + b_i^{(1)}x_i + c_i^{(1)}x_{i+2} = k_i^{(1)}, \qquad (4.47)$$

where

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1}, \\ c_i^{(1)} &= \gamma_i c_{i+1}, \\ b_i^{(1)} &= b_i + \alpha_i c_{i-1} + \gamma_i a_{i+1}, \qquad (4.48) \\ k_i^{(1)} &= k_i + \alpha_i k_{i-1} + \gamma_i k_{i+1}. \end{aligned}$$

Eqs.(4.47, and 4.48) relate every second variable and, when written for $i = 2, 4, \ldots, \acute{n}-2$, a tridiagonal set of equations similar to eq.(4.45) results but with different coefficients $a^{(1)}, b^{(1)}, c^{(1)}$. The system has been roughtly halved. This process can be repeated recursively until, after $\log_2 \acute{n} - 1$ levels of reduction, only one central equation remains. This remaining equation is

$$a^{(r)}_{\acute{n}/2} x_0 + b^{(r)}_{\acute{n}/2} x^{(r)}_{\acute{n}/2} + c^{(r)}_{\acute{n}/2} = k^{(r)}_{\acute{n}/2}, \tag{4.49}$$

where the superscript $r = \log_2 \acute{n} - 1$ indicates the level of reduction. Since $x_0 = x_{\acute{n}} = 0$, the solution for the central equation is obtained as

$$x_{\acute{n}/2} = k^{(r)}_{\acute{n}/2} / b^{(r)}_{\acute{n}/2}. \tag{4.50}$$

The remaining unknowns can be found by fill in process. Since $x_0, x_{\acute{n}/2}$ and $x_{\acute{n}}$, then the unknowns midway between these can be found from the equations at level $r-1$ using

$$x_i = \left(k_i^{(r-1)} - a_i^{(r-1)} x_{i-\acute{n}/4} - c_i^{(r-1)} x_{i+\acute{n}/4}\right) / b_i^{(r-1)}, \tag{4.51}$$

for $i = \acute{n}/4$ and $3\acute{n}/4$.

For the above procedure to compute on an array processor, the computational model for this study, we first the necessary modifications to the procedure (given above), then give the algorithm for the parallel execution of the modified procedure, finally, we give an example of the execution using a small tridiagonal system.

Summary of Modifications

1. The system size $= \acute{n} - 1$, where $\acute{n} = n^q, q \geq 1$.
2. At any level l, the vector $\mathrm{v}_i^{(l)}$ (i being the index of processor P_i, is computed in parallel, where $\mathrm{v}_i^{(l)} = (a_i, b_i, c_i, k_i)$.

3. If size does not satisfy (1), add dummy equations as needed for the condition to be satisfied. The added equation should be according to formulation:

$$\left.\begin{array}{lcl} a_i^{(l)} & = & c_i^{(l)} = k_i^{(l)} = 0 \\ b_i^{(l)} & = & 1 \end{array}\right\} \; for\ i \leq 0,\ i \geq N+1, \qquad (4.52)$$

or $\mathbf{p}_i^{(l)} = (0, 1, 0, 0)$.

4. The maximum level of computation is $\log \acute{N} - 1$.

5. The filling in process given above (see eq.(4.51)) is not necessary. The solution vector $\mathbf{x}$ is to be computed in parallel using computation

$$x_i = k_i^{(q)} / b_i^{(q)} \qquad (4.53)$$

The parallel algorithm of figure 13 above is based upon the procedure described above. We call the algorithm ParCR In the ParCR algorithm given above, execution in steps 1 and 2 always goes all the way to the maximum number of levels, $\acute{n}-1$ after which the solution vector (actually, each component of the vector is evaluated by a respectiveprocessor) in parallel in step 3. But many studies have shown that execution of cyclic reduction algorithm to this number of levels may not give the most efficient computation of the given linear system. That is, the most efficient computation, for a given system, may at some level l_{max} $(1 \leq l_{max} < \log \acute{n})$. In this study, we will use ParCR at various levels of execution and, with a given system, determine l_{max}. The procedural steps involved in the use of the new algorithm is summarized in the algorithm of Figure 4.10.

4.5 CHAPTER SUMMARY

In this chapter, an algorithm based upon the cyclic reduction and recursive doubling direct methods is developed to solve the model problem developed in chapter

```
ALGORITHM ParCR()
  input: (1) $l$ int level
  input: (2) a,b,c,k real vector
             (3) n,l int comment: size & level
  output: x real vector
          comment: a is overwritten by x
  local: $\alpha, \gamma$ real

  BEGIN
   for ($l \leftarrow 1, l$) do
    Step 1: Update $\alpha \& \gamma$
     forall ($1 \le i \le \log_2 \acute{n} - 1$) do
      $\alpha_i = -a_i / b_{i-2^{l-1}}$
      $\gamma_i = -c_i / b_{i+1}$
    Step 2: Comment: Compute P

      $a_i = \alpha_i a_{i-2^{(l-1)}}$
      $c_i = \gamma_i c_{i+2^{(l-1)}}$
      $b_i = b_i + \alpha_i c_{i-2^{(l-1)}} + \gamma_i a_{i+2^{(l-1)}}$
      $k = k_i + \alpha_i k_{l-2^{(l-1)}} + \gamma_i k_{i+^{(l-1)}}$
    Step 3: Comment: Compute x using eq.(4.53)
     forall ($1 \le i \le \acute{n} - 1$) do
      $x_i = k_i^{(q)} / b_i^{(q)}$
  END                                       □
```

Figure 4.9: Cyclic (Odd/Even) Reduction for the New Technique

```
ALGORITHM  New()
  BEGIN
    Step 1: Set $l = 1$.  comment: level $\leftarrow 1$
    Step 2: Invoke ParCR.  comment:

      Step 2.1: Solve resulting system using ParRD
      Step 2.2: Substitute the results into original system
      Step 2.3: Repeat step 2 & 3 until final solution x is obtained

    Step 3 If $l \neq log_2\acute{n} - 1$ then
      (a) Set $l \leftarrow l + 1$
      (b) goto step 2
  END                                                          □
```

Figure 4.10: The New Algorithm Computing a Tridiagonal System at Various Reduction Levels

2. Five benchmarks algorithms, Gauss Elimination, LU and Orthogonal reduction, Conjugate Gradient, and Symmetric Over-relaxation methods, are used to execute the same model problem and their execution times compared to that of the new algorithm. All but one of these benchmarks are parallelized. Gauss Elimination algorithm is used in its serial form of its simplicity and elegance.

Chapter 5
Implementations and Experimental Results

The last thing one knows in constructing a work is.
Blaise Pascal.

Concepts without percepts are empty. Percepts without concepts are blind.
Immanuel Kant.

The purpose of this chapter is two-fold: to give the implementation of the new algorithm and its benchmarks on the MasPar MP-X (model 1208B), the computational model whose architecture and hardware/software features immediately relevant for our study were presented in chapter 2, and also to present the results of this implementation and conclusion thereof. The implementation was carried out in a series of experiments each of which involving the execution of the developed algorithms with different set of parameters. Data structures, such as deemed appropriate to facilitate the implementation and to enhance an efficient inter-processor communication and data routings needed, were developed and used in all the experiments. The definition and implementation of the data structures are treated in §5.1. The construction of the grid of the rectangular domain wherein the model problem is approximated is treated in §5.2. The description of the experiments and the determination of the parametric data used for conducting them are given in §5.3. The efficiency of the new algorithm execution was compared to those of the benchmarks using three evaluation measures. One measure was deviation of results of the algorithms — the new algorithm and the benchmarks — from the exact solution. The exact solution was determined at the specific mesh points of the domain where the the problem was approximated. The deviation was measured in terms of average absolute error. The second evaluation measure was the total cpu time required for each computation. The last measure was the number of non-trivial floating point operations involved in each execution determined in terms of the mega flop (Mflops) rates of the computations concerned. The implication of these results is duly discussed and the conclusion reached as to whether the design and implementation of the new methodology was worth the effort and, if so, to identify the computational situations that merit its application. The results and the conclusion based upon the results are given in §5.4.

5.1 DATA STRUCTURES

There are various methods for storing sparse matrices in order to take advantage of their structures. Several of these methods are only applicable for matrices of a particular form. For the matrices arising from the model problem used in this study, two of such data structures were mentioned in §4.1. These are the straight and row(column) interleaved storages which were used for demonstrating how nonzero entries of matrix A could be handled in order to facilitate inter-processor communication and data routings called for in some computation situations, while, at the same time, preventing the occurence of such common problems as fill-in and load-imbalance. No particular parallel computer was envisaged in such discussions. In this section, we give a more efficient data structuring that was used for the MasPar MP-X programming.

Another commonly used data structure for sparse systems is the so-called diagonal storage. This storage scheme, when properly designed to take advantage of the underlying hardware features of the particular parallel system, can be used for efficient programming of any parallel processors — array, vector, or multiprocessors. One such modification might aim at making the storage scheme to be able to handle a more general form of sparse systems. One common way to do that is through the use of the standard "a, ja, ia format". In this format **a** is a real vector containing all the nonzero entries of the matrix **A**, ja is an integer vector containing the column positions of each row of **a**, and ia is an integer vector of length n pointing to the beginning of each row in **a**. Below is a demonstration of this data structuring using a 5×5 sparse matrix. Note that **A** does not have to be tridiagonal, pentadiagonal, nor any matrix of a particular structure. It is a general sparse matrix of any type and structure:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 8 & 3 \\ 0 & 2 & 0 & 0 & 9 \\ 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 \\ 0 & 5 & 0 & 1 & 0 \end{pmatrix}.$$

$$\begin{aligned} \mathbf{a} &= (\,8,\ 3,\ 2,\ 9,\ 7,\ 4,\ 3,\ 5,\ 1), \\ ja &= (\,4,\ 5,\ 2,\ 5,\ 1,\ 3,\ 4,\ 2,\ 4), \\ ia &= (\,1,\ 3,\ 5,\ 6,\ 8). \end{aligned}$$

This modified diagonal storage scheme would demand $2N^2 + N = O(N^2)$ of memory to store the sparse system. Of course, such a quadratic requirement is too costly, and, therefore, unacceptable for this study.

An alternative storage scheme is to use diagonal storage without any modification, but, rather exploit the linear system's structure more judiciously in defining the structures. The model system, as noted in chapter 3 is a symmetric-banded system with a semi-bandwidth of $\beta = n$ where n, as it might be recalled, is the number of equal partitions of the mesh region in both x- and y-coordinate directions, the width of each mesh being k. The matrix **A**'s structure, for any value of n ($n \geq 1$), is depicted in figure 5.1. Note that this system is is block-tridiagonal, and also that, irrespective of the value of k, it always has 5 nonzero diagonal vectors

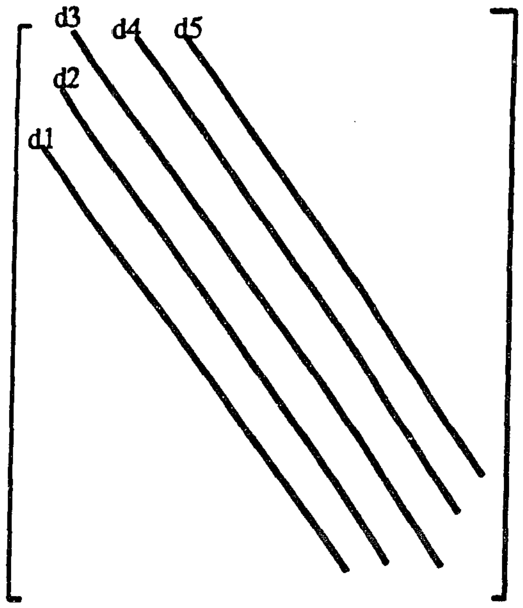

Figure 5.1: Structure of Matrix A Resulting from Model Problem, $\beta = n$

denoted in this report as $d1$, $d2$, $d3$, $d4$, and $d5$, where the main diagonal vector is $d3$ and the nth subdiagonal vector $d1$. The rest of the vectors are indexed in order as shown in figure 5.2.

In this study, these 5 diagonal vectors were defined and used as vexes (see chapter 2 for the meaning of a vex). As an example, for $n = 3$, the matrix **A** and the 5 vexes are:

$$\mathbf{A} = \begin{pmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{pmatrix}.$$

$$\begin{aligned} \mathbf{d1} &= (\,0,\ 0,\ 1,\ 1) \\ \mathbf{d2} &= (\,0,\ 1,\ 0,\ 1) \\ \mathbf{d3} &= (\,-4,\ -4,\ -4,\ -4) \\ \mathbf{d4} &= (\,10,\ 1,\ 0) \\ \mathbf{d5} &= (\,1,\ 1,\ 0,\ 0). \end{aligned}$$

For general n, the pattern of storage for the 5 vexes is given below:

$$\begin{array}{lcl}
\mathbf{d1} & = & (\, 0,\ \underbrace{0,\ \ldots,\ 0}_{\leftarrow\ n\ \rightarrow},\ 1,\ \ldots,\ 1) \\
\mathbf{d2} & = & (\, 0,\ \underbrace{1,\ \ldots,\ 1}_{\leftarrow\ n\ \rightarrow},\ 0,\ 1,\ \ldots,\ 1, \ldots) \\
\mathbf{d3} & = & (\, \underbrace{-4,\ -4,\ \ldots,\ -4}_{\leftarrow\ N\ \rightarrow}) \\
\mathbf{d4} & = & (\, \underbrace{1,\ 1,\ \ldots,\ 0}_{\leftarrow\ n\ \rightarrow},\ 1,\ 1,\ \ldots,\ 1,\ 0,\ \ldots) \\
\mathbf{d5} & = & (\, \underbrace{1,\ 1,\ \ldots,\ 1}_{\leftarrow\ n\ \rightarrow},\ 0,\ \ldots,\ 0)
\end{array} \tag{5.1}$$

The MPL code fragment that was used for defining the 5 vexes for a general value of n is given in figure 5.2. Note that the matrix $\mathbf{A}$ is never stored explicitly. The vexes are used for all computations involving $\mathbf{A}$.

5.2 THE GRID CONSTRUCTION

The rectangular domain $(0,1) \times (0,1)$ on which the model problem was approximated according to the approach discussed in chapter 3, was partitioned n times in both the $x-$ and $y-$coordinate directions and with the mesh width of $k = \frac{1}{(n+1)}$, that is, n mesh lines in each direction. The resulting system was of dimension $N \times N$, where $N = n^2$. Once n was determined, the grid coordinates were computed using the MPL fragment in Figure 5.3 on page 107.

/* The following are the input parameters to this MPL code fragment: */
(a) N The system's dimension
(b) k The mesh width */
(c) n The no. of partitions of mesh */

```
#include   mpl.h   /* Must include this header file */
  {
    plural double *d1,*d2,*d3,*d4,*d5;   /* declares vexes*/
    int nb, i;   /* No. of memory layers , loop-counter */
    double val    /* temp. value */

      nb = (n-1≪lnproc)+1   /* nb determined based on value of n */
      d1 = getstord(nb);    /* dynamic allocation of vex d1 */
      d2 = getstord(nb);    /* dynamic allocation of vex d2 */
      d3 = getstord(nb);    /* dynamic allocation of vex d3 */
      d4 = getstord(nb);    /* dynamic allocation of vex d4 */
      d5 = getstord(nb);    /* dynamic allocation of vex d5 */

      val=(double)-4;   /* val = -4 */
      p_vexcon(n, &val, d3, 0);   /* d3 = (-4, -4, ..., -4) */
      p_vexcopy(n-n, d3, n, d5,0);   /* d5 ¡= d3 */
      p_vexcopy(n-1, d3, 0, d2,1);   /* d2 ¡= d3 */
      p_vexcopy(n-1, d3, 1, d4,1);   /* d4 ¡= d3 */

      val = (double) 0    /* val ¡== 0 */
      for(i=0; i¡n; i+=n){
        p_vexput(&val, d2, i);   /* introduces zero into d2 */
        p_vexput(&val, d4, i+n-1);   /* introduces zero into d4 */
      }
  }
```

Figure 5.2: The Basic Data Structures and their Allocation

/* The following are the input parameters to this MPL code fragment: */
(a) n The system's dimension
(a) h The mesh width */

```
#include   mpl.h  /* Must include this header file */
#include   "pmml.h"


  {
     plural double *d1,*d2,*d3,*d4,*d5;  /* declares vexes*/
     plural double *x,*y;  /* x & y coords. */
     double val    /* temp. value */

        nb = (n-1≪lnproc)+1  /* nb determined based on value of n */
        x = getstord(nb);   /* dynamic allocation of vex x */
        y = getstord(nb);   /* dynamic allocation of vex y */

        val=(double)1;   /* val = 1 */
        p_vexcon(N, &val, d3, 0);   /* d3 ¡= (1,...,1) */
        p_vexinx(N,  x, 0);   /* u = (0,1,...,n − 1) */
        p_vexcopy(N, x, 0, y,0);   /* v ¡= u */
        p_vexcopy(N, d3, 0, d2,1);   /* d2 ¡= d3 */
        p_vexcopy(N, d3, 1, d4,1);   /* d4 ¡= d3 */
        p_vexfix(N, y, 0);   /* fixes v in all layers */

        val = (double) n    /* val ¡== 0 */
        p_vexaxpy(N, &k , y, 0, x, 0);   /* x ¡= y*k */
        p_vexadd(N, d3, 0, x,0);   /* x ¡= x+d3 */
        p_vexcal(N, &h, x, 0);   /* x ¡= x*h */
  }
```

Figure 5.3: The Mesh Region and the Mesh Point Coordinates

5.3 THE EXPERIMENTS

5.3.1 The System Size Determination and Usage

In chapter 2, we identified the computational model, namely, the MasPar MP-X, for the solution of the model problem. We also discussed those hardware and software features of this parallel processor, and, in particular, stated the fact that the MasPar 1208B is actually half the size of the MPP, and that

- MasPar 1208B has $2^{13} = 8192$ processing elements or PEs in a 128×64 machine array,
- each PE has a local (nonsharable) memory of size (PMEMSZ is MPL's jargon) of $2^{16} = 65536$ bytes.

In order to realize the maximum potential (in terms of parallelism) of this machine, any program that executes on it should involve all the 8192 PEs. That is, for any program to execute efficiently and with maximum parallelism, the program's size should be made to fit the machine size as closely as possible. In this study, we determined the maximum n value that caused the model problem to fit exactly the nyproc$\times$nxproc processor array of the 1208B. That value was determined to be $305 \times 305 = 93025$ using the following simple arithmetic calculation intermixed with the MPL notation:

$$\begin{aligned} n &= \text{(int)sqrt}((\textbf{availmem} \ll \text{lnproc})/(11.0\text{*sizeof(double)})); \\ &\qquad\qquad (\textbf{availmem} = \text{1000Bytes.}) \\ n &= [(1000 \times 2^13) \div 88.0]^{\frac{1}{2}}, \text{ (since "double" type has size of 8 bytes.)} \\ &= 305, \quad \Rightarrow N = n^2 = 305^2 = 93025. \end{aligned}$$

Thus, 305 is the maximum n value that caused the model problem to fit exactly the machine size of 128×64 processors. That meant that the maximum number of memory layers or blocks to accomodate the vexes was

$$n^2 - 1 \gg lnxproc = \lfloor 93,000 \div 128 \rfloor = 726,$$

and the maximum number of layers for the veys was

$$n^2 - 1 \gg lnyproc = \lfloor 93,000 \div 64 \rfloor = 1453.$$

5.3.2 Exact Solution

The mathematical model used in this study was the elliptic equation — Poisson's – that is given in eq.(3.15) whose independent variable u was approximated as U with a 5-star stencil of the central finite difference approximation technique to give eq.(3.20). The approximation of the Poisson's equation was carried out on a square domain $(0,1) \times (0,1)$ by first partitioning the domain in both coordinate directions into n meshes of the same mesh width of k, and then applying the 5-star stencil on each grid point (i,j) to produce eq.(3.20). Eqs.(3.15) and (3.20) are rewritten out below as eqs.(5.2) and (5.3) respectively, for convenience:

$$\begin{aligned} -\frac{\partial^2 u(x,y)}{\partial x^2)} - \frac{\partial^2 u(x,y)}{\partial y^2} &= f(x,y). && (5.2) \\ 4U_{ij} - U_{i+1\ j} - U_{i-1\ j} - U_{i\ j+1} - Ui\ j - 1 &= k^2 f(i,j). && (5.3) \end{aligned}$$

The forcing function $f(x,y)$ of the Poisson's equation, eq.(5.2), can be any function, but, in this study, it was chosen to satisfy the Dirichlet boundary conditions as specified in chapter 3. We solved eq.(5.3) in 2 ways. The first way was generating the linear system of equations by expanding eq.(5.3) for grid point (i,j), $\forall (i,j) \in \Gamma \bigcup \Omega$, where Ω is the square domain and Γ its boundary. In other words, eq.(5.3) was written for the entire domain and its boundaries provided both the approximated function $u(i,j)$ and the forcing function $f(i,j)$ were continuous in $\Gamma \bigcup \Omega$. The choice of the function $f(i,j)$ was such that both the

Dirichlet boundary conditions and the condition of continuity were satisfied. Once eq.(5.3) was written for all the grid points, the linear system given in eq.(3.14) was generated. The linear system was then solved by the methods discussed in chapter 4, in keeping with the objective of this study as given in chapter 1.

Another method of solving eq.(5.3) was by direct evaluation of $U(i,j)$ for each grid point (i,j) by making use of the point's position in the mesh (i.e., utilizing its indices "i" and "j") and the fact of the the occurence of the constant coefficient "-4" for each point. Because the values of U at the 4 neighboring grid points must be considered in computing the value of U for a given point (i,j), this computation was carried out in serial using the MPL fragment given in Figure 5.4. The results of this computation were the exact solution of U at all the grid points. As we will explain shortly, the exact solution was used in determining the average absolute errors associated with the determining the solution of the linear system by the algorithms developed in chapter 4.

5.3.3 Experimental Approach

An investigation of the efficiency and the applicability of the new method in solving the model problem was conducted by executing both the new algorithm and its benchmarks, and then comparing all the results of the executions using the three evaluation metrics mentioned in the introduction of this chapter. A total of 4 experiments were conducted each involving one of the system incremental size (N) of $1600, 10000, 40000$, and 93025, that is, at the n values of $40, 100, 200$, and 305 respectively. In the case of the new algorithm, only the maximum level of reduction, that is, $\lceil log(N) \rceil - 1$, of the cyclic reduction (ParCR() of Figure 4.13) was used.

The results of the various executions were compared with those of the exact solution which were computed per grid point using the MPL fragment provided in Figure 5.4. The benchmark results were compared to those of the new approach using the three evaluation metrics — average absolute error, cpu times, and mega flop rates.

The preprocessing step which used the LU factorization to reduce matrix **A** into **L** and **U** before using the parallel recursive doubling (ParRD() of Figure 4.12) was not included in the determination of the evaluation metrics. That is, only the linear recursion of eq.(4.36) was determined using the recursive doubling algorithm, ParRD() (Figure 4.12), and the operations involved in that determination were considered in the evaluation measure, while the nonlinear recursion of eq.(4.33) was not solved by the recursive doubling technique but was computed serially as a preprocessing step, and, as such, its computations were not considered in the evaluation measure determination.

All the data structures used were represented as the MasPar MPL data structures which are vex, vey and mat. Also, to ensure efficiency of computation, the MasPar MPLABB routines were used as much as possible.

5.4 EXPERIMENTAL RESULTS

Given in Figures 5.5 – 5.8 are the results obtained by executing the model problem at the system sizes of 1600, 10000, 40000 and 93025 respectively. Tables 5.1 – 5.4 give the comparison of the results in terms of average absolute errors with respect to the exact solution. The exact solution was determined by computing the values of the dependent variable U at all the grid points. Figures 5.9 and 5.10 give the cpu times and the mega flop rates of the respective execution. In all the figures and tables of the results, the keys "Exact Soln", "New", "SOR", "GE", "CG", and

"Ortho" stand for the computational result of exact, Symmetric-Overrelaxation, Gauss Elimination, Conjugate Gradient, and Orthogonal methods respectively. The orthogonal reduction used was the Householder factorization as explained in chapter 4. The LU reduction was not used as a benchmark since, as explained in §4.1, it is equivalent to the GE method, but, as stated above, the LU reduction was involved in the preprocessing step of the recursive doubling algorithm. Of the benchmarks, only the GE method was used as a serial code for the reason already given in §4.1 — serial usage is simple and elegant and the benefits that accrue from the parallel implementation of GE are miniscule, hence, in many applications of GE given in the literature, serial implementation of GE is are the rule. The rest of the benchmarks, however, were implemented in parallel in accordance with the steps of their respective pseudocoded algorithms given in chapter 4.

5.4.1 Results and Observations

Below are some of the most obvious observations that can be made from the results:

Observation 1: The results shown in Figures 5.5 – 5.9, even by visual inspection, seem reasonable approximations of the values of U at the indicated grid points as they appear to satisfy the Dirichlet boundary conditions which were specified to be zero at the four sides of the rectangular domain. That is, $u(x,y) = 0,\ \forall(x,y) \in \Gamma$, where, as may be recalled from chapter 3, Γ was the set of the boundary points of the domain. Since the values of U, according to the boundary conditions, were zero at the boundaries, one would expect the results, at least, to tend to this value at the boundaries. An examination of Figures 5.5 – 5.8 indicates that such is the case. Note that n was the width of the domain in the two coordinate directions. Hence, at an integral multiples of n, the values of U clustered about zero.

/* The following are the input parameters to this MPL code fragment: */
(a) $d1, d3$ The vexes determined in code of fig.5.2 */
(b) N The size of the system */

(c) n The no. of partitions of mesh */

(d) **x, y** /* x & y vexes. */

```
#include   mpl.h   /* Must include this header file */
   {
      plural double *d1, *d3, *x, *y;   /* declares d3 vex*/
      plural double *u;   /* declares exact u vex */
      int nb, i;   /* No. of memory layers , loop-counter */
      double val   /* temp. value */

         nb = (N-1≪lnproc)+1   /* nb determined based on value of n */
         d3 = getstord(nb);   /* dynamic allocation of vex d1 */
         u_exact = getstord(nb);   /* dynamic allocation of vex exact u */

         val=(double)1;   /* val = 1 */
         p_vexcon(n, &val, u_exact, 0);   /* u_exact = (1,1,...,1) */
         p_vexsub(N, y, 0, u_exact,0);   /* u_exact ¡= 1-y */
         p_vexmul(N, y, 0, u_exact,0);   /* u_exact ¡= y(1-y) */
         p_vexcon(N, &val, d1, 0);   /* d1 = (1,1,...,1) */
         p_vexsub(N, x, 0, d1,0);   /* d1 ¡= 1-x */
         p_vexmul(N, x, 0, u_exact,0);   /* u_exact ¡= x */
         p_vexmul(N, d1, 0, u_exact,0);   /* u_exact */
   }
```

Figure 5.4: The Exact Solution of the Model Problem

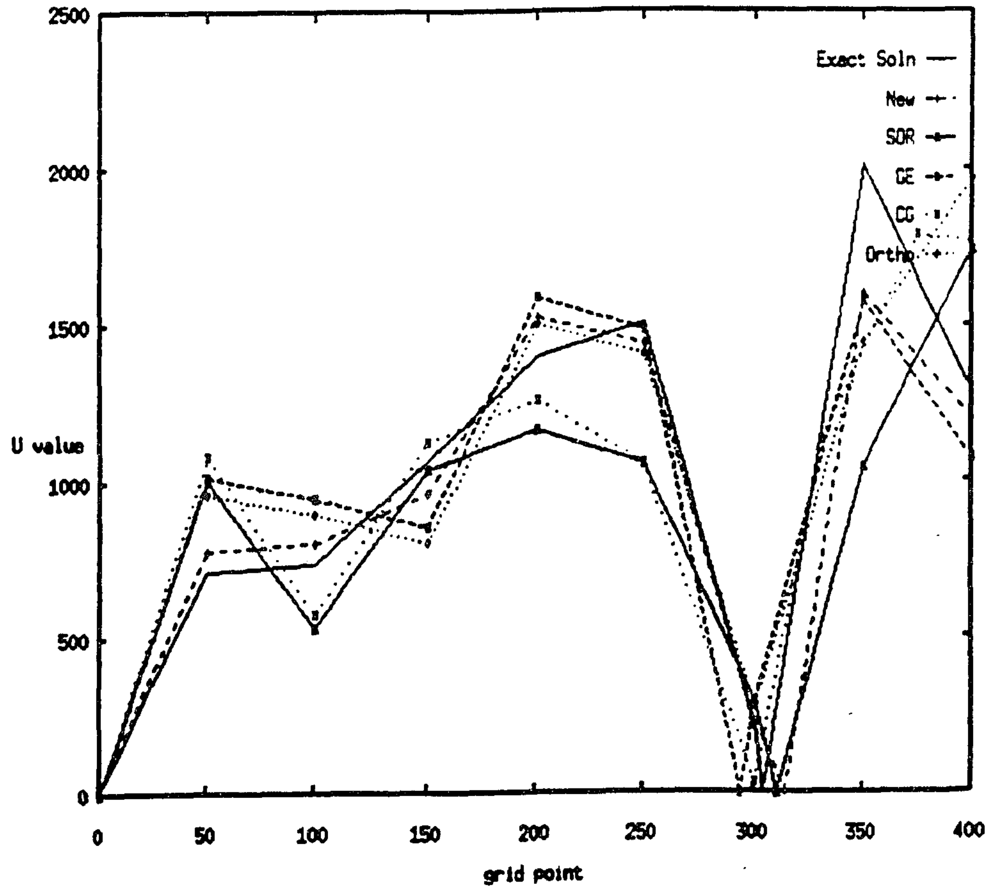

Figure 5.5: Values of U at the First 400 Grid Points when $n = 305$ $N = 93,025$

Table 5.1: Errors of Results when $n = 305$, $N = 93,025$

METHOD	New	SOR	GE	CG	Ortho
ERROR	0.01033	0.021615	0.017166	0.026013	0.025367

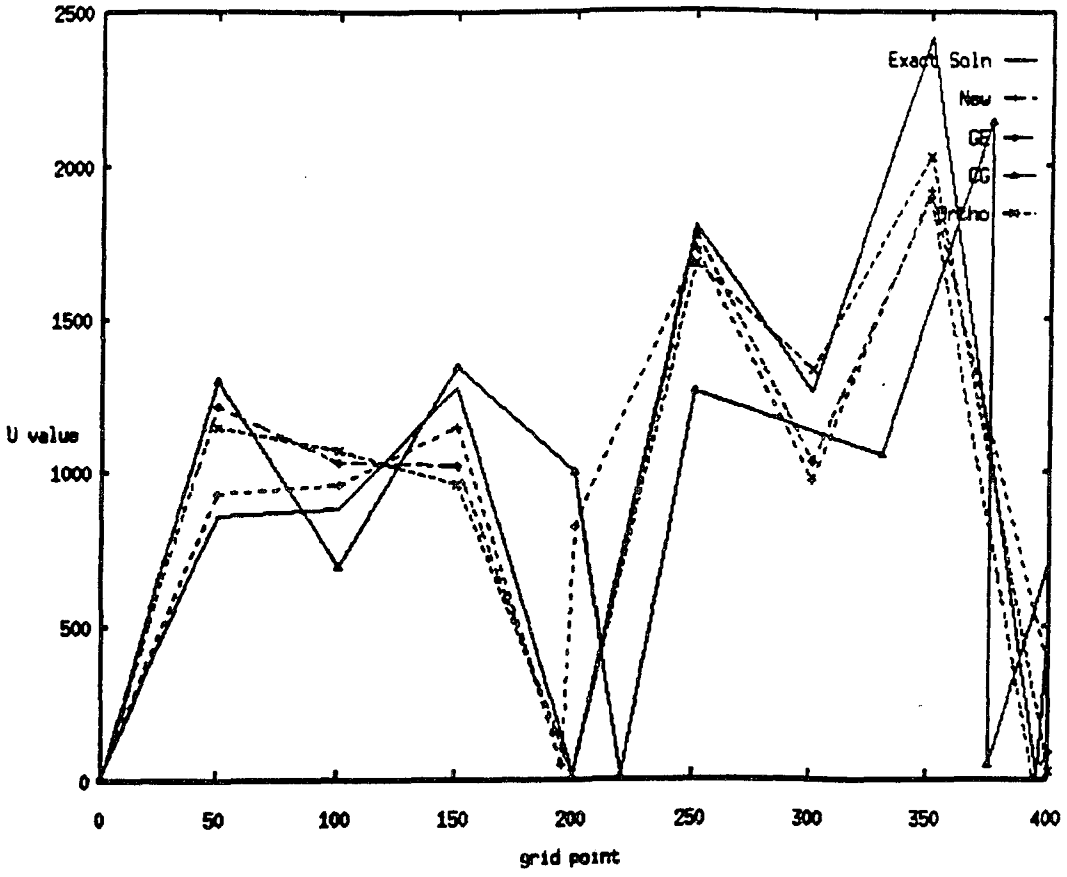

Figure 5.6: Values of U at the First 400 Grid Points when $n = 200$, $N = 40,000$

Table 5.2: Errors of Results when $n = 200$, $N = 40,000$

METHOD	New	SOR	GE	CG	Ortho
ERROR	0.016102	0.021992	0.030503	0.034183	0.029903

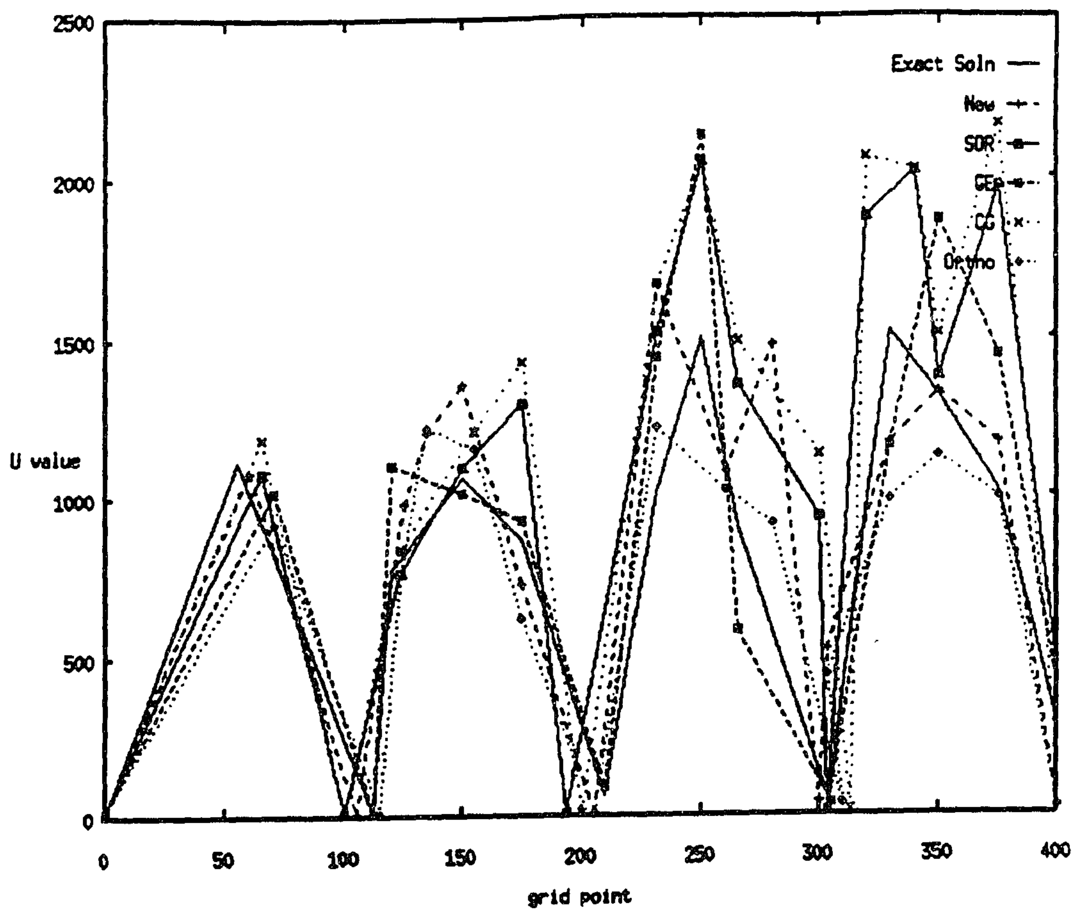

Figure 5.7: Values of U at the First 400 Grid Points when $n = 100$, $N = 10,000$

Table 5.3: Errors of Results when $n = 100$, $N = 10,000$

METHOD	New	SOR	GE	CG	Ortho
ERROR	0.024933	0.025965	0.031008	0.035128	0.026581

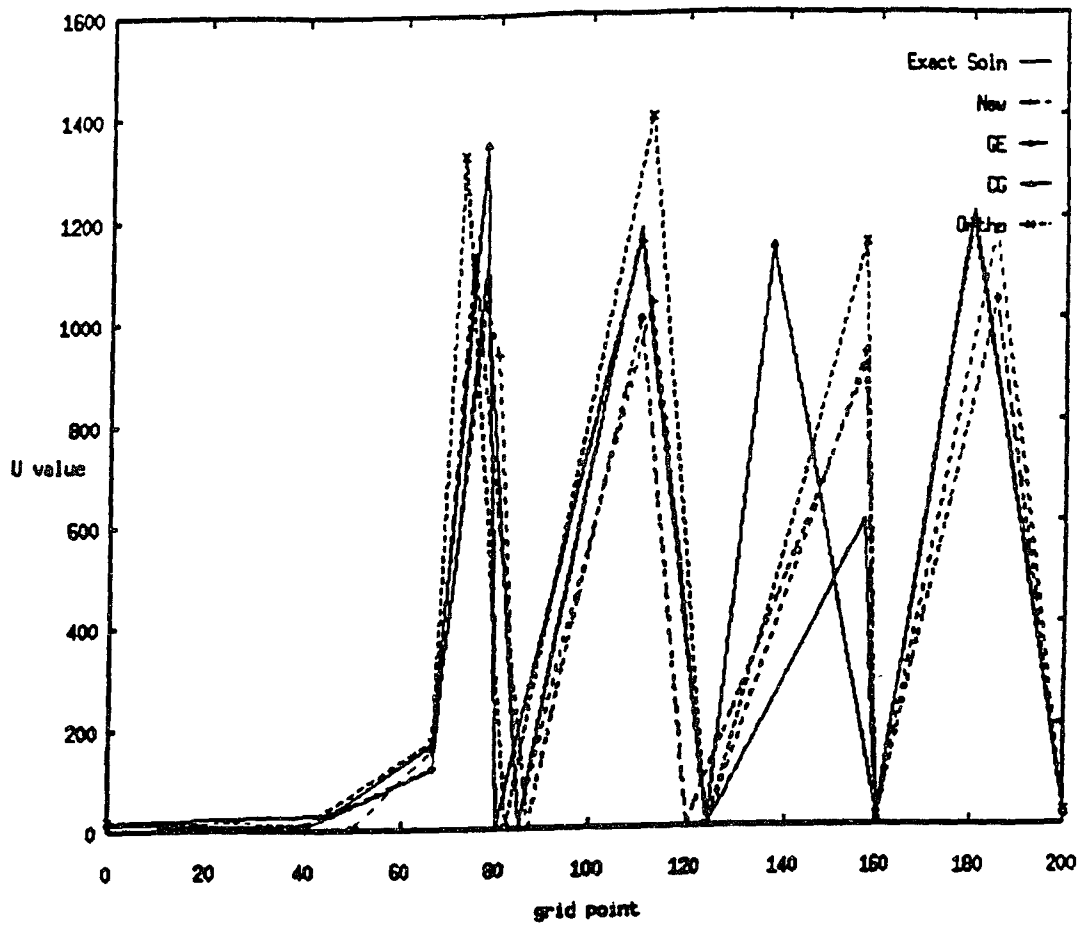

Figure 5.8: Values of U at the First 200 Grid Points when $n = 40$, $N = 1,600$

Table 5.4: Errors of Results when $n = 40$, $N = 1,600$

METHOD	New	SOR	GE	CG	Ortho
ERROR	0.025323	0.027955	0.038929	0.039192	0.027404

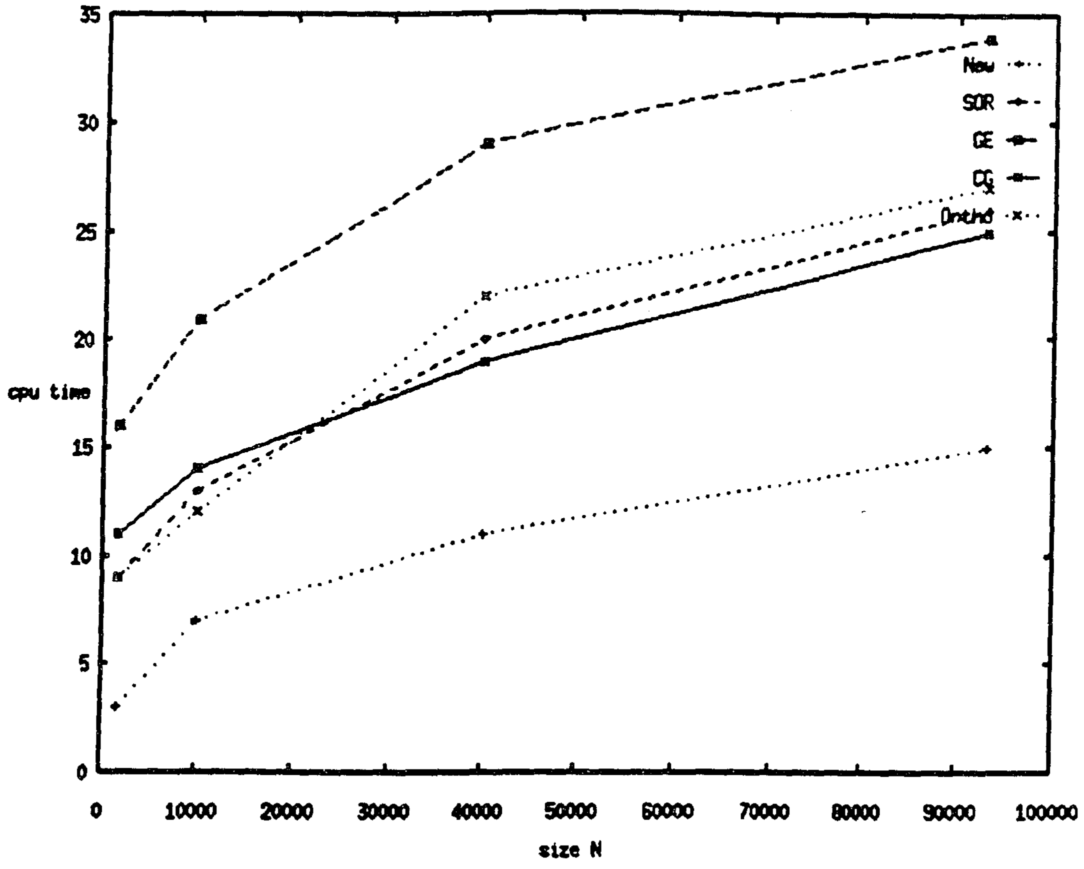

Figure 5.9: CPU Times of the Algorithms

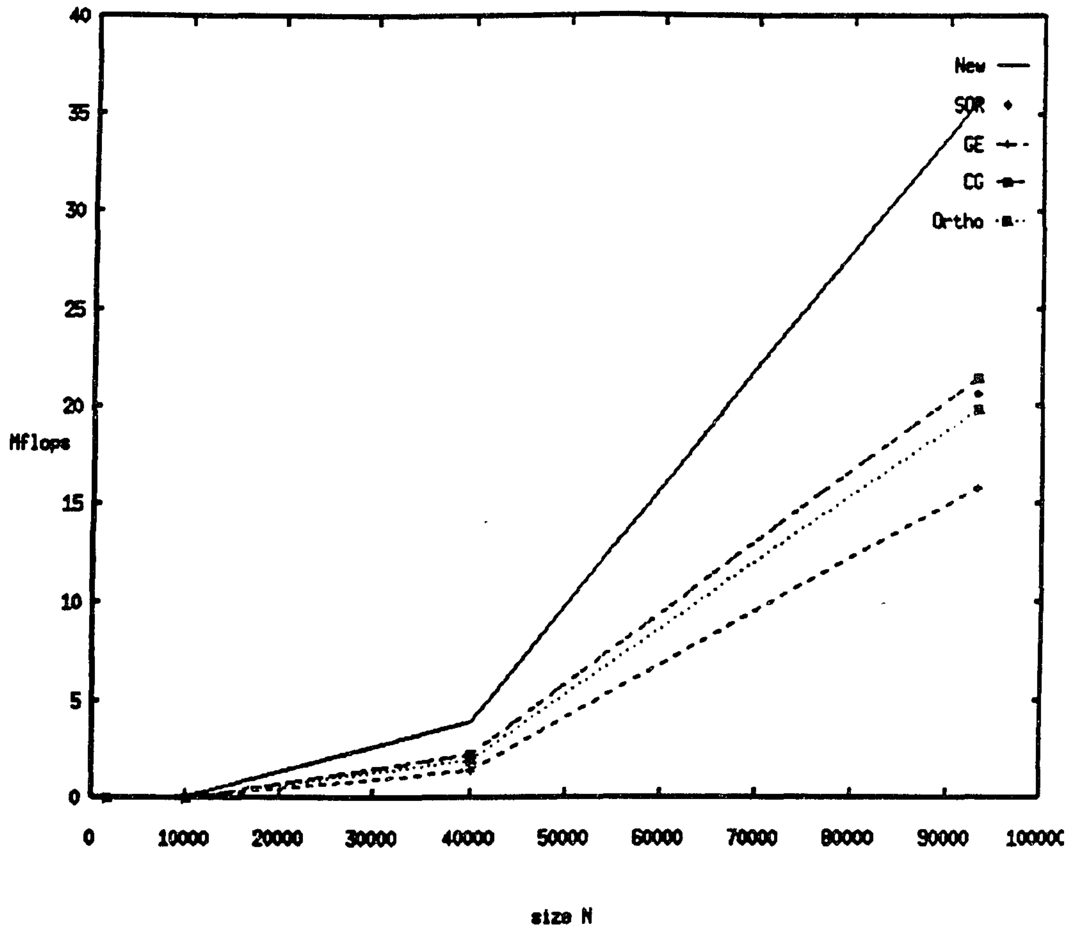

Figure 5.10: Megaflop Rates of the Algorithms

For example, when $n = 305$ (Figure 5.5), $U(305m, y)(m \geq 0) \Rightarrow 0$. Similar observations can be made in the cases in which $n = 200, 100, 40$ (Figures 5.6, 5.7, 5.8 respectively). The clustering was repeated cyclically with multiple n values which marked the boundaries.

Observation 2: It is further observed that, when n was very large such that $n^2 = N \gg p$ (p being the number of processors or PEs), that is, when the domain was partitioned into a fine mesh, the results generated were very close approximations to the exact solutions of U at the grid points. The largest n value used in this study was $n = 305$, the value which ensured that the problem size N ($N = n^2$) fit the machine array exactly. At that n value, one would expect the approximated solutions to be closest to the exact values and less so where n assumed smaller values. A glance at Tables 4.1 - 5.4 shows that this is exactly the case. The closeness of the approximations to the actual values with $n = 305$ is evidenced by the small deviations (or the absolute errors) of the approximated results indicated in Table 5.1. But as n became smaller, the absolute errors were larger and, at the smallest n value of 40, the absolute errors were largest (see Figures 5.6, 5.7, and 5.8 and the corresponding Tables 5.2, 5.3 and 5.4). Thus at $n = 305$, the new technique gave the smallest deviation of 1.03%, and GE with the next best result of 1.72%, and CG the worst result of 2.60%. Householder and SOR gave the deviations of 2.50 and 2.16% respectively. The new method, therefore, gave the best result at $n = 305$. The deviations grew larger with decreasing magnitudes of n for all the algorithms, and all registered the worst results at $n = 40$, the lowest semi-bandwidth of the model problem, with the new technique giving the worst deviation of 4.5% and the Householder and SOR with the best value of 2.70 and 2.83% respectively, while GE and CG gave almost the same deviations of 3.93 and 3.91% respectively. The new technique was about 2.18 more accurate than the benchmarks at $n = 305$, while being about 3.38 worse than the benchmarks at $n = 40$.

These results were as expected: the larger the n value, the finer the mesh, and, therefore, the larger the system size N and, as a result, the closer the approximated results to the exact solutions at the specific mesh points. On the other hand, the smaller the n value, the coarser the mesh and the less accurate the approximations at the mesh points.

Observation 3: Another indicator of the superior performance of the new method are the cpu times given in figure 5.9. In all the system sizes, the new approach outperformed all the benchmarks. At $n = 305$ or $N = 93,025$, the new method with a cpu time of 15.09 seconds, was faster than all the benchmarks with the cpu times of 25.03 seconds (CG), 26.13 seconds (Householder), 25.11 seconds (SOR). The GE, with 34.74 seconds fared the worst in this evaluation category. It was far slower than the parallel codes. Thus, at the maximum system size, the new approach was about 1.7 times as fast as the Householder reduction, CG and SOR, and about 2.3 times as fast as GE. At the rest of the system size (100000, 10000, 16000) the new approach was still the fastest being about 1.6, 1.8, 2.0, and 2.6 times as fast as CG, SOR, Householder reduction, and GE respectively at $N = 40000$, and about about 2.6 times as fast as SOR and Householder, 3.6 times as fast as CG and 5.3 times as fast as GE at $N = 1600$. The GE is the slowest at all size categories because it was implemented in serial and, therefore, did not have the parallelism advantages of the other codes that were implemented in parallel.

Observation 4: In terms of the megaflop rates (see Figure 5.10), the new technique again outperformed the benchmarks at a comfortable margin. This was no surprise since the megaflop rate is directly dependent on the cpu time. The megaflop rate is a measure of how many millions of floating point arithmetic operations the system is capable of performing every second when executing a given program. At $N = 93025$, the new approach had the megaflop rate of 35.22 while the benchmark rates ran in a narrow range of 14.86 (GE) to 23.31 (CG). Thus, the new method's rate was 1.71 times greater than those of the benchmarks on the average. Interestingly, the megaflop rates of all algorithms fell rapidly with decrease in system size. Thus at $N = 40000$, the new approach's rate came down to 3.85, and those of the benchmarks ranged narrowly from 1.86 (GE) to 1.93 (CG) making the new method's rate about 2.05 higher than those of the benchmarks. At $N = 1600$, the new approach's rate fell precipitously to .56 and those of the benchmarks to .46 on the average, thus making the new method's rate about 1.08 higher than the benchmarks'. The new method's megaflop rate was practically the same of the benchmarks. The reason could be that, at the lowest system size of $N = 1600$, all the algorithms — the new approach and the benchmarks — were executing at the minimal parallelism level affordable by the machine. That is, at $N = 1600$, the parallelism features of the machine were not exploited to the fullest, hence, the poor showing in terms of the megaflop rates of execution.

Observation 5: The executions of the developed algorithms within the pametric constraints described in this report, were computationally very stable. That is, they neither failed nor oscillated wildly from the expected path of the computation. This computational stability was due to the fact that the model system used in this study is diagonally-dominant because (see eq.(4.4)) the diagonal elements were of constant values of -4, and, also, the system had no more than 2 other entries each of value 1 per row. That made the system very diagonally-dominant. Because of this property, the system was stable and, as a consequence, none of the developed algorithms required row or column pivoting.

From the above observations, it is obvious that the new technique fared better than all the benchmarks when $N \gg 8,192$, as was the case when $N = 93025, 10000$, and 10000, but did not do as well in all the three evaluation measures when $N \leq 1600$. The superior computational capability of the new technique which manifested in the the above results was due to a number of factors some of which were hinted in the discussion above. We summarize the most salient of these factors below:

1. The new technique was designed by combining two powerful algorithmic tools — cyclic reduction and recursive doubling — to execute the model problem. The Cyclic reduction method, although it was not designed originally for parallel computing, has considerable inherent parallelism. When used with such auxiliary algorithms as the Gaussian elimination, the Choleski decomposition or other factorization techniques, the cyclic reduction can be a very effective tool for solving linear systems of equations, not just the type of system modeled for this study. The maximum benefits of the cyclic reduction are realizable when the algorithm is well designed even if the auxiliaries are implemented serially as is usually the case. Some of these auxiliary algorithms involve recursions which, without the benefit of such other techniques as recursive doubling to solve them in parallel, must be solved sequentially. When recursions are computed in parallel by the auxiliary, the use of the auxiliary with the cyclic reduction technique can potentially be very effective parallel algorithm. The combination of the cyclic reduction and recursive doubling technique to design a new algorithm is what this study set out to achieve. In this study, the auxiliary algorithm was the LU factorization method the linear recursions of which was handled in parallel by the recursive doubling method.

From the indicated results and observations, we conclude that the new algorithm, is a very computationally effective tool for solving the model problem when the size of the system solved is very large or when the system size fits the machine array.

2. MasPar 1208B, just like any parallel processor, performs at the peak of its computational capability when all the processing elements are involved in computing, which, in the context of our experiments, means having the system size to be a lot larger than the 8,192 processing elements. Under this cicumstance, one would expect the highest system's performance in terms of the cpu times, the mega flop rates. According to the above observations, such was the case with the sizes 63025, 40000, and 10000 since these were significantly greater than 8192, the total number of PEs of MasPar model 1208B. There was a clear degradation in performance of the new algorithm when the size of 1600 was used. The explanation is that, with high N value, there was a better utilization of the machine as many PEs became involved in the computation. That is, the load-imbalance problem, though it existed in all levels of computation of the algorithms proposed in this study, did not have a very damaging effect on the performance at the three high N values as it it did when the size was 1,600 or lower (results for $N, 1,600$ are not shown).

3. Both the recursive doubling and the cyclic reduction techniques which form the core of the new algorithmic technique utilized $\mathbf{O} \log_2(N)$ operations while each of the benchmarks required significant more operational steps, for example, the exhobitant $\mathbf{O}(N^3)$ operations by the GE (see §4.1.1A).

5.4.2 Conclusion

From the above results and observations, we come to the following conclusions:

1. The new method developed in this study is a very effective tool for efficient parallel execution of symmetric-banded linear systems, and, although it is specifically designed for large block-tridiagonal linear systems ($\beta = 1$), it can be extended to solve a general banded system with some modification mainly involving the cyclic reduction component of the algorithm. With the present model problem or its modification, the new algorithm performs optimally when

   - the system size N is much larger than 8,192, the number of processing elements. That is, $N \gg 8,192$,
   - the system size is made fits the machine size. Under this condition, the machine will perform at the peak rate because of the better utilization of the machine's parallel resources. Such a utilization will generally lead to mininal load-imbalance.

2. The machine's performance deteriorates rapidly when the system size N is such that $N \leq 1,600$. In fact, in such a situation, the machine's performance may even be worse than a serial's machine's performance.

Bibliography

[AKE 89] Ake, B̄., **Difference Methods – Solutions of Equations in** R^n, Handbook of Numerical Analysis, Elsevier Pub., 1989.

[AKI 89] Aki, S.G., **The Design and Analysis of Parallel Algorithms**, Prentice-Hall, Inc. , New Jersey.

[ALMASI 89] Almasi, G.S., et al., **Highly Parallel Computing**, Benjamin-Cumming, Redwood City, CA.

[AMES 77] Ames, W.F., **Numerical Methods for Partial Differential Equations**, Academic Press, Inc., New York, (1977).

[ANDER 89] Anderson, E., and Y. Saad, **Solving Large Triangular Systems on Parallel Computers**,Int'l. Jour. High Speed Comp, , pp. 73-96.

[BATCH 74] Batcher, K., **STARAN Parallel Processor System Handbook** , AFIPS Conf. , AFIPS Proc. 43, NCC, (1974), pp. 405-410..

[BATCH 82] Batcher, K., **MPP: A Supersystem for satellite Image Processing** , 1982 Nat. Comp. Conf. (AFIPS), **51**.

[BAUCOM 88] Baucomb, C. L., **Reduced Systems and the Preconditioned Conjugate Gradient Method on a Multiprocessor**, M.S. Thesis, Dept. of Comp. Sci. , Univ. of Illinois at Urbana-Champaign, (1988).

[BELL 85] Bell, C. G., **Multis: A New Class of Multiprocessor Computers** ,Science, **228**, (1985), pp. 462-467.

[BELYT 84] Belytschko, M. , et al., **Hourglass Control in Linear and Nonlinear Programs**, Comp. Meth. in Appl. Mech. and Eng. , **43**, (1984), pp. 251-276.

[BELYT 87] Belytschko, M., and N.Gilbertsen, **Concurrent and Vectorized Mixed Time, Explicit Nonlinear Structural Dynamics Algorithm**, In Parallel Computations and Their Impact on Mechanics ed., ASME, New York, pp. 280-291.

[BELYT 89] Belytschko, M., et al. , **Parallel Processors and Nonlinear Structural Dynamics Algorithms Software**, Semiannual Progress Report, (DEc. 1988-May 1989).

[BIRK 83] Birkhoff, G., and R.E. Lynch, **Numerical Solution of Elliptic Problems**, SIAM Pub. , (1983).

[BISCHOF 83] Bischof, C.H., **A Block QR Factorization Algorithm Using Restricted Pivoting**, Journ. of Supercomp, (1989), pp.248-257.

[BOIS 72] Boisvert, R.F., and R.A. Sweet, **Sources and Development of Mathematical Software**, In W. Cowell ed. , Prentice-Hall, Inc. Englewood Cliffs, New Jersey, (1982).

[BOUK 72] Bouknight, H. , et al., **The Illiac IV System** , Proc. IEEE, **60**, (1972), pp. 369-379., Reading, Ma.

[BURN 88] Bouknight, D.S., **Finite Element Analysis: From Concepts to Applications**, Addison-Wesley, New Jersey, (1988).

[BUZBEE 70] Buzbee, B.L. , et al., **On Direct Methods for Solving Poisson's Equation**, SIAM Jour. Num. Anal., **7**, pp. 627-656.

[CHAZAN 69] Chazan, D., and W. Miranker **Chaotic Relaxation**, Linear Algebra & its Applications, **2**, (1969), pp. 199-222.

[COLE 84] Coleman, T.F., **Large Sparse Numerical Optimization**, Lecture Notes in Computer Science, **165**, Springer-Verlag, New York, (1984).

[DADDY 92] Daddy, W.L., **Mechanisms for Concurrent Computing**, Int'l, Conf. on the 5th Gen. Syst., pp. 154-156.

[DASGUP 89a] Dasgupta, S., **Computer Architecture: A Modern Synthesis**,Vol. 1, John Wiley & Sons, Inc.

[DASGUP 89b] Dasgupta, S., **Computer Architecture: A Modern Synthesis**,Vol. 2, John Wiley & Sons Inc.

[DIGIT 81a] **The VAX Architecture Handbook**, Digital Equipment Corporation, Maynard, MASS.

[DIGIT 81b] **The VAX Architecture Handbook**, Digital Equipment Corporation, Maynard, MASS.

[DIGIT 85]	**The VAX 8600 Processor**, Digital Technical Jour. , Digital Equipment Corporation, Hudson, MASS.
[DONGA 86]	Dongara, J.J., **A Survey of High-Performance Computers**, Digest of papers: IEEE Com. Soc. COMCON, (Spring 1986), pp. 8-11.
[DUNCAN 90]	Duncan, R., **A Survey of Parallel Computer Architectures**, **23**, No.2, (Feb. 1990), pp. 5-16.
[EAGER 89]	Eager, D.L., et al. , **Speedup Versus Efficiency in Parallel Systems**, IEEE Trans. on Comp., **C-38**, No. 3, (1989), pp. 408-423.
[FLYNN 66]	Flynn, M., **Very High Speed Computing Systems**, Proc. IEEE, **54**, pp. 1901-1909.
[FLYNN 72]	Flynn, M. , **Some Computer Organizations and Their Differences**, IEEE Trans. Comp. **C-21**, (1972), pp. 948-960.
[FOX 86]	Fox, G., **Questions and Unexpected Answers in Concurrent Computation**, Tech. Rep. No. CP-288, CAL Tech. Concurr. Comp. Prog., CALTECH, Jet Propulsion Labs, Pasadena, Cal.
[GARVEY 93]	Garvey, S.D., **An Efficient Method for Solving the Eigenvalue Problem for Matrices Having A Skew-Symmetric (or Skew-Hermitian) Component of Low Rank**, Int'l Jour. for Num. Meth. in Eng., **36**, (1993), pp. 4151-4163.
[GEUDER 93]	Geuder, U. , et al., **GRIDS User's Guide**, Computer Science Report, (April 1993), Comp. Sci. Dept., Univ. of Stuttgart, Germany.
[GILM 71]	Gilmore, P., **Numerical Solution of Partial Differential Equations by Association Processing**, Proc. FJCL, AFIPS Press, Montvale, New Jersey, (1971), pp. 411-418.
[GILOI 81]	Giloi, W.K., **A Complete Taxonomy of Computer Architecture Based on the Absolute Data Type View**, In Blaauw & Hăndler ed., (1981), pp. 19-38.
[GILM 83]	Gilmore, P., **The Massively Parallel Processor (MPP): A Large Scale SIMD Processor**, Proc. of SPIE, **431**, (Aug. 1983).

[GILOI 83] Giloi, W.K., **Towards a Taxonomy of Computer Architecture Based on the Machine Data Type View**, Proc. of the 10th Annual Int'l. Symp. on Comp. Arch., IEEE. Comp. Soc. Press, New York, (1983), pp. 6-15.

[GLAD 79] Gladwell, J., and R. Wait, **A Survey of Numerical Methods for Partial Differential Equations**, Oxford Univ. Press, (1979).

[GOLUB 93] Golub, G. and J.M. Ortega, **Scientific Computing: An Introduction With Parallel Computing**, Academic Press, Inc. , Boston and New York.

[GOODMN 81] Goodman, J.R., and C.H Sequin, **Hypertree: A Multiprocessor Interconnection Topology**, IEEE Trans. Comp., **C-303**, No. 12, (Dec. 1981), pp. 923-933.

[GOLD 72] Goldstine, H.H., **The Computer from Pascal to von Neumann**, Princeton University Press, Princeton, New Jersey, (1972).

[FOX 86] Goodman, J.R., and C.H. Sequin, **Hypertree: A Multiprocessor Interconnection.**

[GOODYR 74] **Application of STARAN to Fast Fourier Transforms** , Report GER 16109, (May 1972).

[GRIT 88] Grit, D.H., and J.R. McGraw, **Programming Divide- and Conquer for a MIMD Mechine**, Softw. Prac. and Exper., **15**, No. 1, (1988), pp. 41-53.

[GUŠEV 91] Gu{eev, M., and D.J. Evans, **Some Nonlinear Transformation of Matrix Vector Multiplication Algorithm**, Tech. Rep., **635**, Loughborough Univ. of Tech., PARC, Dept. of Comp. Stud., (Oct. 1991).

[GUŠEV 92a] Gu{eev, M., and D.J. Evans, **A Fast Symbolic Design and Double Pipelines**, Tech. Rep., **665**, Loughborough Univ. of Tech., PARC, Dept. of Comp. Stud., (Jan. 1992).

[GUŠEV 92b] Gu{eev, M., and D.J. Evans, **Speeding up the Systolic Design in the Bidirectional Linear Arrays**, Tech. Rep., **702**, Loughborough Univ. of Tech., PARC, Dept. of Comp. Stud., (April 1992).

[HÄND] Händler, W., **The Impact of Classification Schemes on Computer Architecture"**, Proc. of the 1977 Int'l Conf. on Par. Proc., (1977), pp. 7-15.

[HAYNES 82] Haynes, L. et al., **A Survey of Highly Parallel** , Computer, **15**, No. 1, (Jan. 1982), pp. 9-24.

[HELLER 78] Heller, D., **A Survey of Parallel Algorithms in Numerical Linear Algebra**, SIAM Rev. **20**, (Oct. 1978), pp. 740-777.

[HEUB 82] Heuber, K.H., and E.A. Thornton, **The Finite Element Method for Engineers**, John Wiley & Sons, Inc. (1982).

[HENSY 91] Hennessy, J.L., and N.P. Jouppi, **Computer Technology and Architecture: An Evolving Interaction, Computer and Artificial Intelligence**, McGraw-Hill Bk. Co., New York.

[HOCK 65] Hockney, R.W., **A Fast Direct Solution of Poisson's Equation Using Fourier Analysis**, J. ACM,**12**, (march 1965), pp. 95-113.

[HOCK 70] Hockney, R.W., **The Potential Calculation and Some Applications**, Meth. Comp. Phys., **9**, pp.135-211.

[HOCK 81] Hockney, R.W., and J.W. Eastwood, **Computer Simulation Using Pariticles**, McGraw-Hill Pub. Co., Inc., New York, (1981).

[HOCK 87] Hockney, R.W., **Classification and Evaluation of Parallel Computer Systems**, In Springer-Verlag Notes in Comp. Sci., No. 295, (1987), pp. 13-25.

[HOCK 88] Hockney, R. and C.Jesshope, **Parallel Computers**, Adam-Hilger, Bristol and Philadelphia.

[HOLL 93] Hollingsworth, R., **Advanced Semiconductor Technology**, Comm. ACM, **36**, No. 2, (Feb. 1993), pp. 83-105.

[HWANG 84] Hwang, K., and F.A. Briggs, **Computer Architecture and Parallel Processing**, McGraw-Hill Pub. Co., New York.

[HWANG 87] Hwang, K., and J. Ghosh, **Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers** , IEEE Trans. Comp., (Dec. 1987).

[HWANG 89] Hwang, K., and F. Degroot, **Parallel Processing for Supercomputers and Artificial Intelligence** , McGraw-Hill Pub. Co., New York.

[KUCK 77] Kuck, D.J., **Parallel Processing of Ordinary Programs,** Advances in Computers, **15**, Academic Press, New York, (1977), pp. 119-179.

[KUCK 82] Kuck, D.J., and R. Stokes, **The Burroughs Scietific Processor (BSP)**, IEEE Trans. Comp. , **C-31**, pp. 363-376.

[LAMBIO 75] Lambiotte, J., **The Solution of Linear Systems of Equations on a Vector Computer**, Ph.D Diss., Univ. Virginia, Charlottesville.

[LEIGH 92] Leighton, F. T., **Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypertrees** Morgan Kauffmann, (1992).

[LEVES 90] Levesque, J.M., **A Parallel Programming Environment,** IEEE Par. Prog., (1990), pp. 291-294.

[LÖHNER 93] Löhner, R.I., **Some Useful Renumbering Strategies for Unstructured GRIDS**, Int'l. Jour. for Num. Meth. in Eng., **36**, (June 1993), pp. 3259-3270.

[MACK 87] Mackintosh, A. R., and W. Miranker **The First Electronic Computer**, Physics Today, (March 1987), pp. 25-32.

[MAYR 69] Mayr, E., **Principles of Systematic Zoology**, McGraw-Hill Pub. Co., New York.

[MPPE] **MasPar Programming Environment (MPPE)**, Softwr. version 2.2, Document Part No. 9305-0000, Rev.A6, (Dec., 1991).

[MPPSYS] **MasPar System Overview**, Document Part No. 9300-0100, Rev.A4, (Dec., 1991).

[MPPPLS] **MasPar Programming Language (ANSI Compatible MPL)**, Softwr. ver.2.2, Document Part No. 9302-0001, Rev.A1, (Dec., 1991).

[MPML] **MasPar MasPar Mathematics Library (MPML)**, Softwr. ver.2.2, Document Part No. 9302-0400, Rev.A3, (Dec., 1991).

[MPCMDS]	MasPar **MasPar Commands Reference Manual**, Softwr. ver.2.2, Document Part No. 9302-0300, Rev.A7, (Dec., 1991).
[METRO 80]	Metropolis, N., and J. Worlton, **A Trilogy on Errors in the History of Computing**, Annals Hist. oc Comp., **2**, No. 1, pp. 49-59.
[MIKLOŠ 84]	Mikloško, J., and V.E. Kotov, **Algorithms, Software & Hardware of Parallel Computers**, Springer-Verlag, (1984), New York.
[MIRAN 71]	Miranker, W., **A Survey of Parallelism in Numerical Analysis**, SIAM Rev., **13**, (1971), pp. 524-547.
[MODI 88]	Modi, J. J., **Parallel Algorithms and Matrix Computations**, Clarendon Press, Oxford, UK, (1988).
[MORSE 78]	Morse, S. P. , et al., **Intel Microprocessors: 8008 to 8086**, Intel Corporation, Aloha, OR, Reprinted in Siewiorek, Bell, and Newell (1982), pp. 615-642.
[MURA 71]	Muraoka, Y., **Parallelism Exposure and Exploitation in Programs**, Tech. Report No. 71-424, Comp. Sci. Dept., Univ. of Illinois, (1971).
[NIEVER 64]	Nievergelt, J., **Parallel Methods for integrating Ordinary Differential Equations**, Comm. ACM, **7**, (1964), pp. 737-743.
[ORTEGA 85]	Ortega, J.M., and G. Voigt, **Solution of Partial Differential Equations on Vector and Parallel Computers, A Survey**, SIAM Rev., **27**, No. 2, (June 1985), pp. 149-290.
[ORTEGA 88]	Ortega, J.M., **Introduction to Parallel and Vector Solution of Linear Systems**, Plennum Press, New York.
[ORTIZ 89]	Ortiz, M., **Concurrent Algorithms for Transient Nonlinear Finite Element Analysis**, Tech. Report, Div. of Eng. Brown Univ. Rhode Island.
[ORTIZ 86]	Ortiz, M., and B. Nour-Omid, **Unconditional Stable Concurrent Procedures for Transient Finite Element Analysis**, Comp. Meth. Appl. Mech., **58**, pp. 151-174.
[PEASE 74]	Pease, M.C., , **The C(2,m) Algorithm for Matrix Inversion**, Tech. Rep., Stanford Res. Inst., Menlo Pork, CA, (1974).

[RAND 75]	Randell, B., **Origins of Digital Computers**, Springer-Verlag, New York, (1975).
[RICE 83]	Rice, J.R., **Numerical Methods, Software, and Analysis**, In IML Reference ed., McGraw-Hill Pub. Co., (1983).
[ROSS 74]	Ross, H.H., **Biological Systematics**, Addison-Wesley, Reading, MA.
[RUDOL 72]	Rudolph, J., **A production Implementation of an Associative Array Processor – STARAN** , Proc. Fall Joint Comp. Conf., AFIPS Press, Montvale, NJ, (1972), pp. 229-241..
[RUSS 73]	Ruse, M., **The Philosophy of Biology**, Hutchinson Univ. Lib., London.
[SCHAE 77]	Schaefer, D.H. and J.P. Strong, **Tse Computers**, Proc. of the IEEE, **65**, No. 1, (Jan. 1977).
[SCHW 83]	Schwartz, J., **A Taxonomy Table of Parallel Computers, Based on 55 Designs**, Courant Inst., NYU, New York, (Nov. 1983).
[SCHUM 76]	Schumann, U., and R.A. Sweet, **A Direct Method for the Solution of Poisson's Equation with Neumann Boundary Conditions on a Staggered Grid of Arbitrary Size**, Jour. Comp. Phys., **20**, pp. 171-182.
[SIEWIO 82]	Siewiorek, D.P., et al. , **Computer Structures: Principles and Examples**, MacGraw-Hill Pub. Co. , New York, (1982).
[SKILL 88]	Skillcorn, D.B., **Taxonomy for Computer Architectures**, Computer, **21**, No. 21, (Nov. 1987), pp. 46-57..
[SOKAL 63]	Flynn, M.R., and P.H.A. Sneath, **Principles of Numerical Taxonomy**, Freeman, San Francisco.
[STONE 73]	Stone, H.S., **Problems of Parallel Computations,** (Complexity and Parallel Numerical Algorithms ed.), edited by J.F. Traub, Aca. Press, New York & London, pp. 1-16.
[STONE 80]	Stone, H.S., **Parallel Computers** (Stone ed.), In Stone ed. , SRA, (1980).
[STONE 90]	Stone, H.S., **High-Performance Computer Architecture** , Addison-Wesley, Reading, Ma.

[STONE 91] Stone, H.S., and J. Cocke, **Computer Architecture in the 1990s**, Computer, **24** , No.9, (Sept. 1991), pp. 30-38.

[SUN 93a] Sun, Xian-He, and J. D. Joslin, **A Simple Parallel Prefix Algorithm for Compact Finite-Difference Schemes**, ICASE Rep. No. 93-16, NASA Langley Research Center, Hampton, VA.

[SUN 93b] Sun, Xian-He, **On Parallel Diagonal Dominant Algorithm**, 1993 Int'l Conf. on Par. Alg., pp. III-10 - III-17.

[SWARZ 74] Swarztrauber, P. N., **A Direct Method for the Discrete Solution of Seperable Elliptic Equations**, SIAM Jour. Num. Anal. , **11**, pp. 1136-1150.

[SWARZ 89] Swarztrauber, P. N., and R.A. Sweet, **Vector and Parallel Methods for the Direct Solution of Poisson's Equation**, Journ. Comp. Appl. Math. , **27**, pp. 241-263.

[SWEET 73] Sweet, R. A., **Direct Methods for the Solution of Poisson's Equation on a Staggered Grid**, Jour. Comp. Phys., **12**, (1973), pp. 422-428.

[SWEET 74] Sweet, R. A., **Cyclic Reduction Algorithm**, SIAM Jour. Num. Anal., **11**, (1974), pp. 506-520.

[SWEET 93] Sweet, J.R., **The Design and analysis of Parallel Algorithms**, Oxford Univ. Press, Oxford & New York (1993).

[TEMP 80] Sweet, J.R., **On the FACR(l) Algorithm for the Discrete Poisson Equation**, Jour. Comp. Phys., **34**, pp. 314-329.

[TENT 94] Tentner, A.M., et al., **On Advances in Parallel Computing for Reactor Analysis and Safety**, Comm. ACM, **37**, No.4, (April 1994), pp.55-63.

[TRAUB 74] Traub, J., **Complexity of Sequential and Parallel Numerical Algorithms**, Aca. Press, New York.

[WILSON 71] Wilson, E.L., **SAP – A General Structural Analysis Program**, SESM Report 70-20, Det. of Civil Eng., Univ. of CAL, Berkeley.

[WILSON 72] Wilson, E.L., and J. Penzieu, **Evaluation of Orthogonal Damping Matrices**, Int'l Jour. for Num. Math. in Eng., **4**, No. 1, (1972).

[VICH 82]	Vichnevetsky, R., **Computer Methods for Partial Differential Equations, 1: Elliptic Equations and the Finite Element Method**, (1981), and **2: Initial Value Problems**, (1982), Prentice-Hall, Inc., Englewood Cliffs, New Jersey.
[VON 45]	von Neumann, J., **The Draft of a Report on EDVAC**, Memorandum, reprinted in Randell (1975), pp. 355-364.
[LAMB 75]	Lambiotte, J., and R. Voigt **The Solution of Tridiagonal Linear systems on the CDC STAR-100 Computer**, ACM Trans. Math. Softw., **1**, pp. 308-329.
[VOIGT 77]	Voigt, R., **The Influence of Vector Computer Architecture on Numerical Algorithms**(Kuch ed.), pp. 229-244.
[WILKES 68]	Wilkes, M.V., **Computers Then and Now**, 1967 Turing Lecture, Jour. ACM, **15**, No. 1, pp. 1-7.
[PRESS 86]	Press, W.H., et al., **Numerical Recipe: The Art of Scientific Computings**, Cambridge Univ. Press, (1986).
[WULF 72]	Wulf, W. , and C. Bell, **C. mmp – A Multiprocessor** , Proc. AFIPS Press, Reston, VA, (1972), pp. 765-777..

Vita

Okon Hanson Akpan was born in Uyo, Nigeria on May 31, 1952. He attended Lutheran High School, Obot Idim where he graduated with West African School Certificate in December, 1968. After this, he attended Hope Waddel Training Institude, Calabar where he studied Pre-medicine. He Obtained his HSC diploma in pre-medical studies from the University of Cambridge, England in December, 1970. From January, 1971 to December, 1973, Okon taught mainly the physical subjects in Itam Secondary School, Nigeria.

Okon came to the U.S.A in January, 1974 to attend Maryville College, Tennesse from where He graduated with a B.A degree in Mathematics in December, 1976 following a successful completion of the studies in that discipline. In January of the following year, he began his Chemical Engineering training in the University of Tennessee, Knoxville and received the M.S. degree in Chemical Engineering in June, 1980.

Having taught in the MechanicaL Engineering Department, Southern University, Baton Rouge from January, 1981 to May 1985, Okon attended the University of Southwestern Louisiana, Lafeyette from August, 1985 to May, 1988 to study for an advanced degree in Computer Science. He graduated with an M.S. degree in Computer Science in May, 1988. He then taught for the Computer Science Department at Xavier University, New Orleans from August, 1988 to May, 1991. Following this, Okon pursued his doctoral studies in Computer Science in the Louisiana State University, Baton Rouge, and graduated with a Ph.D degree in December, 1994.

Okon, now a naturalized U.S.A. citizen, is married to former Miss Victoria Ekpe Okon of Uyo, Nigeria. He has three daughters: Ituen, Itoro, and Mayen who are fourteen, ten and four years of age respectively.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Okon Hanson Akpan

Major Field: Computer Science

Title of Dissertation: A New Method for Efficient Parallel Solution of Large Linear Systems on a SIMD Processor

Approved:

J B Jones
Major Professor and Chairman

[illegible]
Dean of the Graduate School

EXAMINING COMMITTEE:

A-Elamawy

[illegible]

[illegible]

Charles N. Delzell

[illegible]

Date of Examination:

September 12, 1994