

This item was submitted to [Loughborough's Research Repository](#) by the author.  
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

## **Design and analysis of numerical algorithms for the solution of linear systems on parallel and distributed architectures**

PLEASE CITE THE PUBLISHED VERSION

PUBLISHER

© Rosni Abdullah

PUBLISHER STATEMENT

This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: <https://creativecommons.org/licenses/by-nc-nd/4.0/>

LICENCE

CC BY-NC-ND 4.0

REPOSITORY RECORD

Abdullah, Rosni. 2019. "Design and Analysis of Numerical Algorithms for the Solution of Linear Systems on Parallel and Distributed Architectures". figshare. <https://hdl.handle.net/2134/33138>.



**Pilkington Library**

Author/Filing Title ..... ABDULLAH, R .....

Accession/Copy No. .... 040147105 .....

|               |                  |
|---------------|------------------|
| Vol. No. .... | Class Mark ..... |
|---------------|------------------|

|                    |                  |
|--------------------|------------------|
| <u>26 JUN 1998</u> | <u>LOAN COPY</u> |
|--------------------|------------------|

0401471055



BADMINTON PRESS  
1111 BROOK ST.



**DESIGN AND ANALYSIS OF NUMERICAL ALGORITHMS  
FOR THE SOLUTION OF LINEAR SYSTEMS  
ON PARALLEL AND DISTRIBUTED ARCHITECTURES**


By

Rosni Abdullah

A Doctoral Thesis submitted in partial fulfilment of the award  
of Doctor of Philosophy of the Loughborough University

February 1997

© by Rosni Abdullah 1997

|  |           |
|--|-----------|
|  Loughborough<br>University<br>public library |           |
| Date   | June 97   |
| Class  |           |
| Doc<br>ID  | 040147105 |

979102877

## *Acknowledgements*

*In the Name of Allah  
The Most Beneficent, The Most Merciful*

*"All the praises and thanks be to Allah, the Lord of the 'Alamin, the most Beneficent, the most Merciful"*

*(Al Fatiha:1-3)*

This thesis would not have been possible without the help and support of many people. First and foremost, I would like to express my sincere and deepest gratitude to Professor D.J.Evans for his dedicated and excellent supervision which includes the invaluable advice, discussion, co-operation, guidance and most of all for being very understanding of my situation as a student and a mother at the same time. Thank you too to Dr. M. Harrison from the Department of Mathematics for his invaluable comments. Thanks is also due to Professor Alty, my Director of Research who has been very supportive and helpful. I would also like to thank Professor Schroder, Head of Department and Director of PARC for his comments.

I am also grateful to the technical and support staff of the Computer Studies Department, particularly Dr W.Yousif who has given me a lot of technical advice and Mrs. J. Poulton who has provided the secretarial support. Also, thank you to fellow colleagues who have helped me in one way or another.

I also wish to extend my gratitude to Universiti Sains Malaysia for granting me the scholarship and study leave to pursue my Ph.D.

I would also like to thank my parents, friends and family who have supported and helped me in one way or another throughout my stay here.

Last but not least, a very special thank you to my husband, Azman, and my three lovely daughters Nur Sakinah, Aishah and Yusraa who have given me constant support, encouragement and most all the continuing love and never ending patience that has kept me going all these while.

## **Design and Analysis of Numerical Algorithms for the Solution of Linear Systems on Parallel and Distributed Architectures**

### **ABSTRACT**

The increasing availability of parallel computers is having a very significant impact on all aspects of scientific computation, including algorithm research and software development in numerical linear algebra. In particular, the solution of linear systems, which lies at the heart of most calculations in scientific computing is an important computation found in many engineering and scientific applications.

In this thesis, well-known parallel algorithms for the solution of linear systems are compared with implicit parallel algorithms or the Quadrant Interlocking (QI) class of algorithms to solve linear systems. These implicit algorithms are  $(2 \times 2)$  block algorithms expressed in explicit point form notation.

Both the direct and iterative methods for solving linear systems are investigated. For the direct methods, the Gaussian Elimination (GE) and LU factorisation are compared with Parallel Implicit Elimination (PIE) and Quadrant Interlocking Factorisation (QIF) respectively. The comparison is made for both shared memory parallel computers and distributed memory parallel computers. The investigation of direct methods on shared memory parallel computers also included partial pivoting for PIE and QIF. The Givens QR method is compared with the QZ method on a shared memory parallel computer. The classical iterative methods of Jacobi, Gauss Seidel, Jacobi OverRelaxation (JOR) and Successive OverRelaxation (SOR) methods were compared to the Quadrant Interlocking (QI) iterative methods. Both the synchronous and asynchronous iterative algorithms were investigated on a shared memory parallel computer.

The shared memory parallel computer used to implement the algorithms was the Sequent Balance. When investigating algorithms on the distributed memory parallel computer, a cluster of Dec-Alpha workstations was used and the Parallel Virtual Machine (PVM) software was employed to make the workstations appear as a single parallel computing resource.

From a detailed analysis of the computational complexity of the algorithms it is clearly seen that the implicit methods have less memory accesses than their explicit forms. Consequently, results of the implicit direct methods on a shared memory parallel

computer revealed a gain of 20% in execution time over the classical methods while the results on the distributed memory architecture showed a reduction of almost 50% in communication time. Even with partial pivoting, both PIE and QIF are more superior in terms of execution time as compared to GE and LU respectively. Results of QZ on shared memory parallel computer has shown a gain of about 10% over the Givens QR method. The QI iterative methods for both the synchronous and asynchronous iterations have shown a faster rate of convergence on the shared memory parallel computer.

As the aim of the implicit methods is to express the algorithms in explicit point form, the saving on BLAS overheads results in more efficient algorithms.



## CONTENTS

|  |       |    |
|--|-------|----|
| <b>Acknowledgements</b>                                      | ..... | i  |
| <b>Abstract</b>  | ..... | ii |
| <b>List of Algorithms</b>                                    | ..... | x  |
| <br><b>Chapter 1</b>   |       |    |
| <b>Introduction</b>  | ..... | 1  |
| 1.1 The Heat Transfer Field Problem                          | ..... | 3  |
| 1.2 Least Squares Problem                                    | ..... | 6  |
| 1.3 Guide to Thesis  | ..... | 8  |
| <br><b>Chapter 2</b>   |       |    |
| <b>Basic Mathematical Concepts and Methods in Linear</b>     |       |    |
| <b>Algebra</b>   | ..... | 10 |
| 2.1 Basic Matrix Algebra                                     | ..... | 10 |
| 2.2 Diagonal Dominance and Irreducibility                    | ..... | 15 |
| 2.3 Eigenvalues and eigenvectors                             | ..... | 18 |
| 2.4 Vector and matrix norms                                  | ..... | 21 |
| 2.5 Positive definite and special matrices                   | ..... | 22 |
| 2.6 Property $\mathcal{A}$ and consistently ordered matrices | ..... | 23 |
| 2.7 Rate of convergence                                      | ..... | 25 |
| 2.8 Direct methods for the solution of linear systems.       | ..... | 29 |
| 2.8.1 Gaussian Elimination                                   | ..... | 30 |
| 2.8.2 LU factorisation                                       | ..... | 32 |
| 2.8.3 QR factorisation                                       | ..... | 34 |
| 2.8.4 Partial Pivoting                                       | ..... | 35 |
| 2.8.4.1 Partial Pivoting for Gaussian                        |       |    |
| Elimination  | ..... | 36 |
| 2.8.4.2 Partial Pivoting for LU factorisation                | ..... | 37 |
| 2.9 Iterative Methods  | ..... | 38 |
| 2.9.1 The model problem                                      | ..... | 39 |
| 2.9.2 The Jacobi iterative method                            | ..... | 40 |

|        |   |    |
|--------|---|----|
| 2.9.3  | Successive Displacement or Gauss-Seidel method              | 41 |
| 2.9.4  | Tests for convergence                                       | 42 |
| 2.9.5  | Accelerated convergence for Jacobi and Gauss-Seidel methods | 43 |
| 2.10   | Computational Complexity                                    | 45 |
| 2.10.1 | Gaussian Elimination  | 45 |
| 2.10.2 | LU factorisation  | 45 |
| 2.10.3 | QR factorisation  | 46 |
| 2.10.4 | Iterative methods   | 46 |

### Chapter 3

#### A survey of current methods in solving linear equations on parallel computers

|         |  |    |
|---------|--|----|
| 3.1     | Parallel Architecture  | 48 |
| 3.2     | Parallel Algorithms  | 54 |
| 3.2.1   | Design of parallel algorithms                                    | 55 |
| 3.2.2   | Analysis of parallel algorithms                                  | 57 |
| 3.2.3   | Programming Methods  | 60 |
| 3.2.3.1 | Programming the Sequent Balance                                  | 60 |
| 3.2.3.2 | Programming in PVM   | 62 |
| 3.3     | Survey Of Parallel Algorithms For The Solution Of Linear Systems | 63 |
| 3.3.1   | Parallel Direct Linear System Solvers                            | 64 |
| 3.3.1.1 | Parallelisation of LU and GE                                     | 65 |
| 3.3.1.2 | Parallelisation of the QR method                                 | 71 |
| 3.3.1.3 | Other related work   | 73 |
| 3.3.2   | Parallel iterative linear system solvers                         | 78 |

## Chapter 4

### Parallel Implicit Elimination (PIE) and Quadrant

### Interlocking Factorisation (QIF) on Shared Memory

|   |     |
|---|-----|
| Architecture  | 81  |
| 4.1 Parallel Implicit Elimination method (PIE)                        | 82  |
| 4.2 Quadrant Interlocking Factorisation method (QIF)                  | 88  |
| 4.3 Partial Pivoting for PIE and QIF                                  | 94  |
| 4.3.1 Partial Pivoting for PIE  | 94  |
| 4.3.2 Partial Pivoting for QIF  | 95  |
| 4.4 Implementation of PIE and QIF on shared memory architecture       | 96  |
| 4.4.1 Parallel Elimination algorithm                                  | 97  |
| 4.4.2 QIF algorithm   | 99  |
| 4.4.3 Bi-directional substitution algorithm                           | 100 |
| 4.4.4 Bi-directional solution algorithm                               | 100 |
| 4.5 Computational Complexity and Shared Memory Access                 | 102 |
| 4.5.1 Computational Complexity and Shared Memory Access Count for PIE | 102 |
| 4.5.2 Computational Complexity and Shared Memory Access Count for QIF | 106 |
| 4.5.3 Computational Complexity and Shared Memory Access Count for GE  | 109 |
| 4.5.4 Computational Complexity and Shared Memory Access Count for LU  | 112 |
| 4.5.5 A Summary   | 114 |
| 4.6 Numerical Results   | 115 |
| 4.6.1 Execution time of PIE, QIF, GE and LU                           | 116 |
| 4.6.2 Speedup of PIE, QIF, GE and LU                                  | 117 |
| 4.6.3 Efficiency of PIE, QIF, GE and LU                               | 118 |
| 4.6.4 Temporal Performance of PIE, QIF, GE and LU                     | 119 |
| 4.7 Summary   | 121 |

## **Chapter 5**

### **Parallel Implicit Elimination (PIE) and Quadrant**

#### **Interlocking Factorisation (QIF) on Distributed Memory**

|  |       |     |
|--|-------|-----|
| <b>Architecture</b>  | ..... | 123 |
| 5.1 Cluster Computing and PVM  | ..... | 124 |
| 5.2 Implementation of PIE and QIF in PVM                             | ..... | 127 |
| 5.3 Theoretical analysis of communication for PIE, GE, QIF<br>and LU | ..... | 132 |
| 5.3.1 Theoretical analysis of communication for GE<br>algorithm      | ..... | 132 |
| 5.3.2 Theoretical analysis of communication for PIE<br>algorithm     | ..... | 133 |
| 5.3.3 Theoretical analysis of communication for LU<br>algorithm      | ..... | 134 |
| 5.3.4 Theoretical analysis of communication for QIF<br>algorithm     | ..... | 135 |
| 5.4 Prediction of communication times                                | ..... | 137 |
| 5.5 Numerical results  | ..... | 141 |
| 5.6 Summary  | ..... | 145 |

## **Chapter 6**

|  |       |     |
|--|-------|-----|
| <b>QZ decomposition on a shared memory multiprocessor</b>              | ..... | 147 |
| 6.1 QZ decomposition method  | ..... | 148 |
| 6.2 Sequential and Parallel Algorithms for QR and QZ                   | ..... | 152 |
| 6.3 Computational Work and Memory Accesses of the QR<br>and QZ methods | ..... | 156 |
| 6.3.1 QR method  | ..... | 156 |
| 6.3.2 QZ method  | ..... | 157 |
| 6.3.3 A Summary  | ..... | 160 |
| 6.4 Numerical Results  | ..... | 161 |
| 6.5 The Greedy Approach  | ..... | 164 |
| 6.6 Summary  | ..... | 165 |

## **Chapter 7**

### **Quadrant Interlocking (QI) Iterative methods on Shared Memory Parallel Computer**

|       |  |           |
|-------|--|-----------|
|       | .....  | 166       |
| 7.1   | Quadrant Interlocking Matrix Splitting Strategy  | ..... 167 |
| 7.1.1 | Simultaneous Quadrant Interlocking iterative<br>method (J.Q.I.)  | ..... 170 |
| 7.1.2 | Simultaneous Overrelaxation iterative method<br>(J.O.Q.I.)   | ..... 170 |
| 7.1.3 | Successive Quadrant Interlocking iterative<br>method (S.Q.I.)  | ..... 171 |
| 7.1.4 | Successive Overrelaxation iterative method<br>(S.O.Q.I.)   | ..... 171 |
| 7.2   | The Model Problem  | ..... 172 |
| 7.3   | Sequential and Parallel Algorithms for the Classical and<br>QI iterative methods                         | ..... 173 |
| 7.4   | Computational Complexity and Shared Memory Access<br>Analysis for the Classical and QI iterative methods | ..... 180 |
| 7.4.1 | Jacobi and Gauss-Seidel iterative methods  | ..... 180 |
| 7.4.2 | JQI and SQI methods  | ..... 180 |
| 7.4.3 | Summary  | ..... 181 |
| 7.5   | Numerical Results  | ..... 182 |
| 7.6   | A Simplified Model Study   | ..... 192 |
| 7.7   | Summary  | ..... 194 |

## **Chapter 8**

|                                |       |     |
|--------------------------------|-------|-----|
| <b>Summary and Future Work</b> | ..... | 195 |
|--------------------------------|-------|-----|

|                   |       |     |
|-------------------|-------|-----|
| <b>References</b> | ..... | 198 |
|-------------------|-------|-----|

|                 |       |     |
|-----------------|-------|-----|
| <b>Appendix</b> | ..... | 208 |
|-----------------|-------|-----|

|   |       |     |
|---|-------|-----|
| Parallel Implicit Elimination program listing (Balance) | ..... | 208 |
|---|-------|-----|

|   |       |     |
|---|-------|-----|
| Quadrant Interlocking Factorisation program listing (Balance) | ..... | 212 |
|---|-------|-----|

|   |       |     |
|---|-------|-----|
| QZ Decomposition program listing (Balance)                          | ..... | 217 |
| Parallel Implicit Elimination master program listing (PVM)          | ..... | 224 |
| Parallel Implicit Elimination slave program listing (PVM)           | ..... | 227 |
| Quadrant Interlocking Factorisation master program listing<br>(PVM) | ..... | 233 |
| Quadrant Interlocking Factorisation slave program listing<br>(PVM)  | ..... | 235 |

## LIST OF ALGORITHMS

|                     |   |       |     |
|---------------------|---|-------|-----|
| Algorithm 2.9.2.1   | Jacobi Algorithm  | ..... | 41  |
| Algorithm 2.9.3.1   | Gauss-Seidel Algorithm  | ..... | 41  |
| Algorithm 3.3.1.1.1 | Sequential forward elimination algorithm                      | ..... | 66  |
| Algorithm 3.3.1.1.2 | Sequential row oriented algorithm for back<br>substitution    | ..... | 68  |
| Algorithm 3.3.1.1.3 | Sequential column oriented algorithm for<br>back substitution | ..... | 68  |
| Algorithm 3.3.1.1.4 | Sequential algorithm for factorisation stage in<br>LU         | ..... | 71  |
| Algorithm 3.3.1.1.5 | Sequential algorithm for forward substitution                 | ..... | 71  |
| Algorithm 3.3.1.2   | Sequential QR Decomposition                                   | ..... | 72  |
| Algorithm 4.4.1.1   | Sequential algorithm for implicit elimination                 | ..... | 98  |
| Algorithm 4.4.1.2   | Parallel algorithm for implicit elimination                   | ..... | 98  |
| Algorithm 4.4.2.1   | Sequential algorithm for WZ factorisation                     | ..... | 99  |
| Algorithm 4.4.2.2   | Parallel algorithm for WZ factorisation                       | ..... | 99  |
| Algorithm 4.4.3.1   | Sequential algorithm for bi-directional<br>substitution       | ..... | 100 |
| Algorithm 4.4.3.2   | Parallel algorithm for bi-directional<br>substitution         | ..... | 100 |
| Algorithm 4.4.4.1   | Sequential algorithm for bi-directional<br>solution           | ..... | 101 |
| Algorithm 4.4.4.2   | Parallel algorithm for bi-directional solution                | ..... | 101 |
| Algorithm 4.5.1.1   | Sequential algorithm for parallel elimination                 | ..... | 103 |
| Algorithm 4.5.1.2   | Sequential algorithm for bi-directional<br>solution           | ..... | 103 |
| Algorithm 4.5.2.1   | Sequential algorithm for WZ factorisation                     | ..... | 106 |
| Algorithm 4.5.2.2   | Sequential algorithm for bi-directional<br>substitution       | ..... | 107 |
| Algorithm 4.5.3.1   | Forward elimination algorithm of GE                           | ..... | 109 |
| Algorithm 4.5.3.2   | Backsubstitution algorithm of GE                              | ..... | 109 |

|                   |  |       |     |
|-------------------|--|-------|-----|
| Algorithm 4.5.4.1 | Factorisation stage in LU                                    | ..... | 112 |
| Algorithm 4.5.4.2 | Forward substitution   | ..... | 112 |
| Algorithm 5.2.1   | Partitioning of data to slave programs                       | ..... | 128 |
| Algorithm 5.2.2   | WZ factorisation algorithm                                   | ..... | 129 |
| Algorithm 5.2.3   | Bi-directional substitution algorithm                        | ..... | 129 |
| Algorithm 5.2.4   | Bi-directional Solution algorithm                            | ..... | 130 |
| Algorithm 5.2.5   | Parallel Implicit Elimination algorithm                      | ..... | 130 |
| Algorithm 5.2.6   | LU factorisation algorithm                                   | ..... | 131 |
| Algorithm 5.2.7   | Forward Substitution Algorithm                               | ..... | 131 |
| Algorithm 5.2.8   | Back Substitution Algorithm                                  | ..... | 131 |
| Algorithm 5.2.9   | Forward Elimination Algorithm                                | ..... | 131 |
| Algorithm 6.2.1   | Sequential QR Decomposition                                  | ..... | 152 |
| Algorithm 6.2.2   | Sequential QZ Decomposition                                  | ..... | 153 |
| Algorithm 6.2.3   | Parallel QR Decomposition                                    | ..... | 154 |
| Algorithm 6.2.4   | Parallel QZ Decomposition                                    | ..... | 155 |
| Algorithm 7.3.1   | Sequential Jacobi Algorithm                                  | ..... | 174 |
| Algorithm 7.3.2   | Sequential Gauss-Seidel Algorithm                            | ..... | 175 |
| Algorithm 7.3.3   | Sequential JOR Algorithm                                     | ..... | 175 |
| Algorithm 7.3.4   | Sequential SOR Algorithm                                     | ..... | 175 |
| Algorithm 7.3.5   | Sequential JQI Algorithm                                     | ..... | 175 |
| Algorithm 7.3.6   | Sequential SQI Algorithm                                     | ..... | 176 |
| Algorithm 7.3.7   | Sequential JOQI Algorithm                                    | ..... | 176 |
| Algorithm 7.3.8   | Sequential SOQI Algorithm                                    | ..... | 177 |
| Algorithm 7.3.9   | Parallel Jacobi and JOR Algorithms                           | ..... | 177 |
| Algorithm 7.3.10  | Parallel JQI and JOQI Algorithms                             | ..... | 177 |
| Algorithm 7.3.11  | Parallel Gauss-Seidel/SOR Algorithms<br>(Red-Black Ordering) | ..... | 178 |
| Algorithm 7.3.12  | Parallel SQI/SOQI Algorithms (Red Black<br>Ordering)         | ..... | 178 |
| Algorithm 7.3.13  | Asynchronous Iterative Algorithm                             | ..... | 179 |



## Chapter 1

### Introduction

An important field in computer science has emerged from the need for high performance computing machines consisting of a large number of processors, interconnected by high speed networks, having fast access to a large memory, and able to work together on large-scale computationally intensive applications. There seems to be a general agreement among researchers in many fields of science and engineering that parallel computing is the most practical means to satisfy the ever-increasing need for higher computing power.

Rapid developments in very large scale integration (VLSI) and communication technologies have created a situation whereby the hardware designers are able to build machines with thousands of powerful computing nodes at a relatively low cost. These machines are naturally more complex than the machines with a single processor. Whatever achievements may be made in new technologies for building faster processors and memory modules, it is necessary to have a good understanding of how to group these processors so that they are able to work together on large problems. The inevitability of this direction is shown by the existence of many important applications exhibiting great potential for being split into parts and being solved in parallel.

The increasing availability of advanced-architecture computers is having a very significant impact on all aspects of scientific computation, including algorithm research and software development in numerical linear algebra.

Linear algebra, in particular, the solution of linear systems of equations, lies at the nucleus of most calculations in scientific computing. The solution of a system of linear equations is an important computation found in many engineering and scientific applications.

These applications include electromagnetic scattering, computational fluid dynamics, airline wing design, radar cross-section studies, supercomputer benchmarking, flow around ships and other offshore constructions, diffusion of solid bodies in a liquid, noise reduction and diffusion of light through small particles.

Excellent numerical methods for solving these problems on uniprocessor machines have long been developed and many reliable and high quality codes are available for different cases of linear systems. However, for the past 15 years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems on parallel computers. Earlier work in developing parallel algorithms for linear algebra problems involved hypothetical parallel computers with  $n^2$  or more processors, where  $n$  is the dimension of the matrix. As parallel computers became more widely available, these algorithms were then tested on real parallel computers with a limited number of processors usually less than  $n$ . Later development involved redesigning and reorganising the parallel algorithms to obtain an improvement in performance [Dongarra 84]. These parallel algorithms have also been developed to form parallel software libraries. More recent development in the design of linear algebra algorithms for advanced architecture computers is that the frequency with which data is moved between different levels of the memory hierarchy must be minimised in order to attain high performance [Dongarra 95b]. The main algorithmic approach for exploiting both vectorisation and parallelism is the use of block-partitioned algorithms, particularly in conjunction with highly tuned kernels for performing matrix-vector and matrix-matrix operations (the Level 2 and 3 BLAS) [Dongarra 95b].

Earlier algorithms have been developed bearing in mind a sequential execution. When these algorithms are parallelised, they may not offer the best performance. This thesis focuses on numerical algorithms which have been designed to suit the parallel nature of computation. The performance of these parallel algorithms will then be compared against that of the classical methods which have been parallelised and implemented on the same platform. Analysis of sequential algorithms are based on computational count and memory requirements of the algorithm. However, new dimensions are involved in analysing parallel algorithms. These include speedup, efficiency, temporal performance and data movement between processors.

In particular, attention will be given to numerical algorithms for the solution of linear systems, covering both the direct and iterative methods of solving linear systems.

Section 1.1 covers the heat transfer problem as an application that will yield a sparse linear system. The least squares problem which results in a dense linear system will be discussed in section 1.2. Section 1.3 provides an outline of the thesis.

## 1.1 The Heat Transfer Field Problem

A well known problem that generates a sparse linear system of equations to be solved is the heat transfer field problem. Field problems do not give rise to finite sets of equations by a direct interpretation of the physical properties. Instead, the system is approximated to discrete form by choosing points or nodes at which to assign the basic variables. The errors involved in making the approximation will be small if more nodes are chosen. Therefore, the user will have to decide on the number of nodes which is likely to give sufficient accuracy.

Consider the heat transfer problem of Figure 1.1.1 in which the material surface AB is maintained at high temperature  $T_H$  while the surface CDEF is maintained at a low temperature  $T_L$ , and the surfaces BC and FA are insulated. Estimates may be required for the steady state temperature distributions within the material and also the rate of heat transfer, assuming that the geometry and boundary conditions are constant in the third direction.

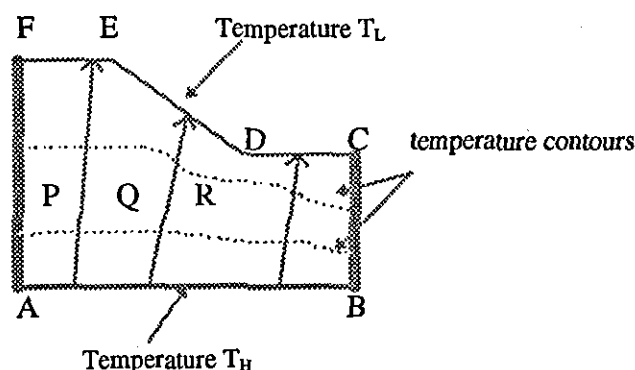


Figure 1.1.1 A heat transfer problem showing possible heat flow lines

There will be heat flow lines such as the lines P, Q, and R in Figure 1.1.1 and also temperature contours which must be everywhere mutually orthogonal. If  $T$  is the temperature and  $q$  the heat flow per unit area, then the heat flow will be proportional to the conductivity of the material,  $k$ , and to the local temperature gradient. If  $s$  is measured along a line of heat flow then

$$q = -k \frac{\delta T}{\delta s} \quad (1.1.1)$$

Alternatively, the heat flow per unit area can be separated into components  $q_x$  and  $q_y$  in the directions  $x$  and  $y$  respectively. For these components, it can be shown that

$$q_x = -k \frac{\delta T}{\delta x} \quad \text{and} \quad q_y = -k \frac{\delta T}{\delta y} \quad (1.1.2)$$

Consider the element with volume  $dx \times dy \times 1$  shown in Figure 1.1.2. In the equilibrium state the net outflow of heat must be zero, hence

$$\left( \frac{\delta q_x}{\delta x} + \frac{\delta q_y}{\delta y} \right) dx dy = 0 \quad (1.1.3)$$

Substituting for  $q_x$  and  $q_y$  from equations (1.1.2) gives

$$\frac{\delta^2 T}{\delta x^2} + \frac{\delta^2 T}{\delta y^2} = 0 \quad (1.1.4)$$

provided that the conductivity of the material is constant. This equation must be satisfied throughout the region ABCDEF, and, together with the boundary conditions, gives a complete mathematical statement of the problem. The boundary conditions for the problem of discussion are:

$$\text{on AB:} \quad T = T_H \quad (1.1.5)$$

$$\text{on CDEF:} \quad T = T_L \quad (1.1.6)$$

$$\text{on BC and FA:} \quad q_x = \frac{\delta T}{\delta x} = 0 \quad (1.1.7)$$

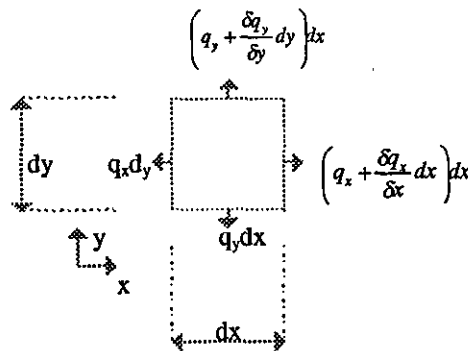


Figure 1.1.2 Heat flow across the boundaries of an element  $dx \times dy \times 1$

Equation (1.1.4) is known as Laplace's equation. Other problems which give rise to equations of Laplace form are analysis of sheer stress in shaft objects due to pure torsion and potential flow analysis in fluid mechanics.

Few analytical solutions exist for Laplace's equation, therefore, it is necessary to apply numerical techniques such as the finite difference or the finite element method to obtain an approximate solution.

In the finite difference method a regular mesh is designated to cover the whole area of the field, and the field equation is approximated by a series of difference equations involving the magnitudes of the required variable at the mesh points.

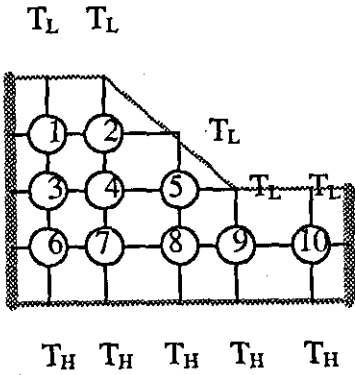


Figure 1.1.3 A finite difference mesh

Figure 1.1.3 shows a square mesh suitable for an analysis of the heat transfer problem in which the temperatures of the points 1 to 10, designated by  $T_1, T_2, \dots, T_{10}$ , are unknowns whose values are to be computed. For a square mesh the finite difference equation corresponding to Laplace's equation is

$$T_G + T_H + T_K + T_L = 4T_J \tag{1.1.8}$$

where the points G, H, K and L adjoin point J as shown in Figure 1.1.4.

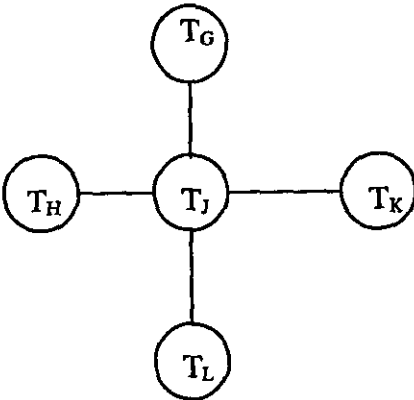


Figure 1.1.4 Finite difference linkage for Laplace's equation

Equation (1.1.8) which states that the temperatures at any point is equal to the average of the temperatures at the four neighbouring points, is likely to be less accurate when a

large mesh rather than a small one is chosen. Taking J to be each of the points 1 to 10 in turn yields 10 equations, for example, when

$$J = 4: T_2 + T_3 - 4T_4 + T_5 + T_7 = 0$$

and

$$J = 5: T_4 - 4T_5 + T_8 = -2T_L$$

The insulated boundaries BC and FA may be considered to act as mirrors, so that, for instance, with J=1, the point immediately to the left of point 1 can be assumed to have a temperature of  $T_1$ , giving

$$-3T_1 + T_2 + T_3 = -T_L$$

The full set of equations may be written in matrix form as

$$\begin{bmatrix} 3 & -1 & -1 & & & & & & & \\ -1 & 4 & & -1 & & & & & & \\ -1 & & 3 & -1 & & -1 & & & & \\ & -1 & -1 & 4 & -1 & & -1 & & & \\ & & & -1 & 4 & & & -1 & & \\ & & -1 & & & 3 & -1 & & & \\ & & & -1 & & -1 & 4 & -1 & & \\ & & & & -1 & & -1 & 4 & -1 & \\ & & & & & & & -1 & 4 & -1 \\ & & & & & & & & -1 & 3 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \\ T_9 \\ T_{10} \end{bmatrix} = \begin{bmatrix} T_L \\ 2T_L \\ 0 \\ 0 \\ 2T_L \\ T_H \\ T_H \\ T_H \\ T_L + T_H \\ T_L + T_H \end{bmatrix} \quad (1.1.9)$$

with the coefficient matrix being a sparse one. Sparse linear systems are normally solved by iterative methods.

## 1.2 Least Squares Problem

The least squares problem for overdetermined equations will result in a dense linear system. In most problems, the object is to obtain the best fit to a set of equations using insufficient variables to obtain an exact fit. A set of  $n$  linear equations involving  $m$  variables  $x_i$ , where  $n > m$ , may be described as overdetermined. Taking the  $k$ -th equation as typical, then

$$a_{k1}x_1 + a_{k2}x_2 + \dots + a_{km}x_m = b_k \quad (1.2.1)$$

It is not normally possible to satisfy these equations simultaneously, and therefore for any particular proposed solution one or more of the equations are likely to be in error.

Let  $e_k$  be the error in the  $k$ -th equation such that

$$a_{k1}x_1 + a_{k2}x_2 + \dots + a_{km}x_m = b_k + e_k \quad (1.2.2)$$

The most acceptable solution or best fit will not necessarily satisfy any of the equations exactly but will minimise an appropriate function of all the errors  $e_k$ . If the reliability of each equation is proportional to a weighting factor  $w_k$ , then a least squares fit finds the solution which minimises

$$\sum_{k=1}^n w_k e_k^2 \quad (1.2.3)$$

Since this quantity can be altered by adjusting any of the variables, then the equations of the form

$$\frac{\partial}{\partial x_i} \sum w_k e_k^2 = 0 \quad (1.2.4)$$

must be satisfied for all the variables  $x_i$ . Substituting for  $e_k$  from equation (1.2.2) and differentiating with respect to  $x_i$  gives

$$\left( \sum_{k=1}^n w_k a_{ki} a_{k1} \right) x_1 + \left( \sum_{k=1}^n w_k a_{ki} a_{k2} \right) x_2 + \dots + \left( \sum_{k=1}^n w_k a_{ki} a_{km} \right) x_m = \sum_{k=1}^n w_k a_{ki} b_i \quad (1.2.5)$$

Since there are  $n$  equations of this form, a solution can be obtained.

Alternatively, these equations may be derived in matrix form by rewriting equations (1.2.2) as

$$Ax = b + e \quad (1.2.6)$$

where  $A$  is an  $n \times m$  matrix and  $b$  and  $e$  are column vectors of order  $n$ .

Proceeding in the same way and if incremental changes in the variables are represented by the column vector  $[dx]$  then the corresponding incremental changes in the errors  $[de]$  satisfy

$$A [dx] = [de] \quad (1.2.7)$$

However, from the required minimum condition for the sum of the weighted squares, it follows from (1.2.4) that

$$\sum_{k=1}^n w_k e_k de_k = 0 \quad (1.2.8)$$

which can be expressed in matrix form as

$$[de]^T W e = 0 \quad (1.2.9)$$

where  $W$  is a diagonal matrix of the weighting factors. Substituting for  $e$  and  $[de]$  using equations (1.2.6) and (1.2.7) gives

$$[dx]^T A^T W(Ax-b) = 0 \quad (1.2.10)$$

Since this equation is valid for any vector  $[dx]$ , it follows that

$$A^T W A x = A^T W b \quad (1.2.11)$$

It can be verified that this equation is equivalent to (1.2.5). Furthermore, the coefficient matrix  $A^T W A$  is symmetric and positive definite when  $W > 0$ . Thus, for positive weights there is only one minimum and one unique solution.

### 1.3 Guide to Thesis

The aim of this thesis is to investigate the Quadrant Interlocking (QI) class of algorithms (both direct and iterative methods) for the solution of linear systems. The investigation includes analysing the computational complexity of the algorithms, analysing the data movement involved and the implementation of the parallel algorithms. A similar investigation is done on the classical algorithms used in the solution of linear systems and a comparison is then made between the QI class of algorithms and the classical algorithms.

Basic mathematical concepts needed to understand the methods for solving linear systems are discussed in chapter 2. In chapter 3, a survey of current methods in solving linear equations on parallel computers is presented. Some basic insight to parallel architectures, parallel algorithms and parallel programming methods are also given here.

Chapter 4 covers the QI direct methods, namely Parallel Implicit Elimination (PIE) and Quadrant Interlocking Factorisation (QIF) methods and their implementation on a shared memory machine, the Sequent Balance. The performance of these methods are then compared to that of the classical and most commonly used direct methods, Gaussian Elimination (GE) and the LU factorisation respectively. A discussion on the computational complexity and shared data accesses for the algorithms is also given.

In chapter 5, the PIE and QIF methods are investigated on the distributed memory architecture using the Parallel Virtual Machine (PVM) software on a cluster of workstations. In particular, the communication complexity of the algorithms are analysed and a comparison is made with the classical methods GE and LU.

An orthogonal version of the QI method, known as the QZ method is examined in chapter 6. The performance of this algorithm will be compared to that of the Givens



QR method. The computational complexity and shared data accesses for both algorithms are also discussed.

In chapter 7 of the thesis, the iterative version of the QI method is discussed. This includes the Simultaneous Quadrant Iterative (JQI) method which is compared with the Jacobi (J) method, the Jacobi Over-relaxation Quadrant Iterative (JOQI) method which is compared to the Jacobi Over-relaxation (JOR) method, the Successive Quadrant Iterative method which is compared to the Gauss-Seidel method and the Successive Over-relaxation Quadrant Iterative (SOQI) method which is compared to the Successive Over-relaxation (SOR) method. Both the synchronous and asynchronous versions of the iterative methods are discussed and the results presented. Lastly, a discussion and suggestions for future work pertaining to this research will be given in chapter 8.

## Chapter 2

### Basic Mathematical Concepts and Methods in Linear Algebra

In this chapter the important topics in matrix computation are presented. Here the concepts are narrowed down to that of relevance to the study of the solution of linear systems. Sections 2.1 through 2.7 cover the basic concepts of matrix computation. The solution  $\mathbf{x}$  to the system of simultaneous linear equations with the  $(n \times n)$  matrix  $A$  and  $(n \times 1)$  vector  $\mathbf{b}$ , can be denoted in the matrix form  $A\mathbf{x}=\mathbf{b}$ . The methods for solving such systems, either direct or iterative, depend on some matrix properties of  $A$  such as irreducibility, diagonal dominance and positive definiteness of the coefficient matrix of the system. These and some other properties along with some basic fundamentals of matrix theory will be discussed in these sections.

As mentioned above, practical methods for the solution of linear systems fall mainly into two classes; direct and iterative. The direct methods find the solution in a finite number of steps and are guaranteed to succeed. The iterative methods start with an arbitrary first approximation to  $\mathbf{x}$  and then improve this estimate in a convergent sequence of steps. Section 2.8 will cover the direct methods and section 2.9 covers the iterative methods. The computational complexity of the methods is covered in section 2.10.

#### 2.1 Basic Matrix Algebra

Matrices are important to numerical analysis because they provide a concise method for specifying and manipulating large numbers of linear equations. The collection of equations and their unknowns is called a linear system if each one can be expressed in the form,

$$a_1x_1 + a_2x_2 + \dots + a_mx_m = b,$$

with  $x_i, a_i, b \in \mathfrak{R}, i=1(1)m$ . The  $x_i, i=1(1)m$  are the unknowns,  $a_i$  are the coefficients and  $b$  the constant or right hand side (RHS) term.

Hence, an  $n$  equation system is as illustrated in (2.1.1).

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1m}x_m &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2m}x_m &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nm}x_m &= b_n, \end{aligned} \tag{2.1.1}$$

A matrix is defined as a two dimensional array with each element denoted as  $a_{ij}$  where  $i$  specifies the row and  $j$  specifies the column of the array in which the element appears. For example, a matrix  $A$  with  $n$  rows and  $m$  columns is said to be of size  $(n \times m)$ . The matrix  $A$  can be denoted as

$$A = [a_{i,j}] = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix} \quad (2.1.2)$$

When  $n=1$ , the matrix is a row matrix or a row vector and when  $m=1$ , it is a column matrix or a column vector. Vectors are normally denoted by a small underlined letter  $\underline{a}$  or a bold letter  $\mathbf{a}$ . In this thesis, all vectors are denoted in bold. An element  $a_i$  represents the  $i$ th element of vector  $\mathbf{a}$ . The term vector without qualification will refer to a column vector. Therefore, a vector  $\mathbf{b}$ , for example, whose elements are  $b_1, b_2, \dots, b_n$  and is of order  $n$  is denoted by

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad (2.1.3)$$

A matrix  $A$  is said to be a square matrix of order  $n$  if  $n = m$ . In this thesis, the term matrices imply square matrices, unless stated otherwise. The italic capital letters will be used to denote matrices.

A linear system of equations can be represented by

$$A\mathbf{x} = \mathbf{b} \quad (2.1.4)$$

with  $A$  an  $n \times m$  coefficient matrix, and  $\mathbf{x}, \mathbf{b}$  vectors. The system is homogeneous if the components of  $\mathbf{b}$ ,  $b_i = 0$ ,  $i=1(1)n$  and always has a trivial solution with  $x_i = 0$ ,  $i=1(1)m$ . Any solution with some  $x_i \neq 0$  is termed a nontrivial solution. A non-homogeneous system has a particular solution if  $\mathbf{u} \in \mathbb{R}^n$  satisfies (2.1.4) when substituted for  $\mathbf{x}$ , and the set of all vectors satisfying (2.1.4) gives the general solution. This thesis is restricted to linear systems with mainly square coefficient matrices, and which often arise from physical problems. Fortunately, in general, such systems if solvable produce only a single or unique solution obviating the need to deal with general solution sets.

The set of elements  $a_{i,i}$ ,  $i = 1, 2, \dots, n$  of a matrix  $A$  is a principal (main) diagonal of  $A$ . The transpose of a matrix  $A=[a_{i,j}]$  is denoted as  $A^T$  and is obtained by interchanging the rows and columns of  $A$ , i.e., the element  $a_{i,j}$  of  $A$  becomes  $a_{j,i}$  of  $A^T$ . Thus the transpose of  $A$  in (2.1.2) is:

$$A = [a_{i,j}] = \begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{n,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{n,2} \\ \cdots & \cdots & \cdots & \cdots \\ a_{1,m} & a_{2,m} & \cdots & a_{n,m} \end{bmatrix} \quad (2.1.5)$$

The determinant of a square matrix can be denoted either as  $\det(A)$  or  $|A|$ . An inverse of a given square matrix, denoted by  $A^{-1}$ , if it exists, is also a square matrix such that

$$AA^{-1} = A^{-1}A = I$$

where  $I$  is the identity (unit) matrix having the same order as  $A$  and is defined as follows:

$$a_{i,i} = 1, \text{ for all } i = 1, 2, \dots, n$$

$$a_{i,j} = 0, \text{ for all } i, j = 1, 2, \dots, n \text{ and } i \neq j.$$

If the inverse of  $A$  exists, then  $A$  is non-singular, otherwise it is singular. On the other hand,  $A$  is non-singular if  $\det(A) \neq 0$  and singular if  $\det(A) = 0$ .

If  $x$  and  $y$  are real numbers, then the conjugate of the complex number  $a = x + iy$  is  $\bar{a} = x - iy$ . If the elements of a matrix  $A$  are complex numbers, the conjugate of  $A$  is the matrix  $\bar{A}$  whose elements are conjugates of the corresponding elements of  $A$ , i.e., if  $A=[a_{i,j}]$  then  $\bar{A} = [\bar{a}_{i,j}]$ .

The Hermitian transpose or conjugate transpose of  $A$  denoted by  $A^H$ , is the transpose of  $\bar{A}$  and also the conjugate of  $A^T$ , i.e.,

$$A^H = (\bar{A})^T = \overline{A^T} = [\bar{a}_{j,i}]$$

The sum of the diagonal elements of a matrix  $A$  is called the trace of  $A$ , denoted by  $\text{trace}(A)$ , i.e.,

$$\text{trace}(A) = \sum_{i=1}^n a_{i,i}$$

A permutation matrix  $P = [p_{i,j}]$  is a matrix which has the entries of ones and zeroes with exactly only one non-zero entry in each row and each column. For example,

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad (2.1.6)$$

is a permutation matrix of order 4. For any permutation matrix  $P$ , we have

$$PP^T = P^TP = I$$

Hence  $P^T = P^{-1}$ .

For any two vectors  $\mathbf{a}$  and  $\mathbf{b}$ , both of the same order say  $n$ , the inner product of  $\mathbf{a}$  and  $\mathbf{b}$  is defined as

$$(\mathbf{a}, \mathbf{b}) = \mathbf{a}^H \mathbf{b} = \sum_{i=1}^n \overline{a_i} b_i$$

Further, for any matrix  $A$ ,

$$(\mathbf{a}, A\mathbf{b}) = (A^H \mathbf{a}, \mathbf{b}).$$

Given a matrix  $A = [a_{ij}]$ , the integers  $i$  and  $j$  are associated with rows and columns with respect to  $A$  if  $a_{ij} \neq 0$  or  $a_{ji} \neq 0$ .

The matrix  $A = [a_{ij}]$  of order  $n$  is

- Symmetric, if  $A = A^T$ .
- Orthogonal, if  $A^T = A^{-1}$ .
- Hermitian, if  $A^H = A$ .
- Null, usually denoted by  $0$  if  $a_{ij} = 0$ , for  $i, j = 1, 2, \dots, n$ .
- Diagonal, if  $a_{ij} = 0$  for  $i \neq j$ , i.e.,  $|i - j| > 0$ , where  $|i - j|$  represents the modulus of any number  $(i - j)$  and  $a_{ij} \neq 0$  for  $i=j$ .

$$A = \begin{bmatrix} x & & & 0 \\ & x & & \\ & & \ddots & \\ & & & x \\ 0 & & & & x \end{bmatrix} \quad x \text{ denotes a possible non-zero element.}$$

Figure 2.1.1 Diagonal matrix

- Banded, if  $a_{ij} = 0$ , for  $|i - j| > r$ , where  $2r+1$  is the bandwidth of  $A$ .
- Tridiagonal, if  $r = 1$  ( $r$  as in  $f$  above). See figure 2.1.2 for example.

$$A = \begin{bmatrix} x & x & & & \\ x & x & x & & 0 \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \\ 0 & & & x & x & x \\ & & & & x & x \end{bmatrix} \quad \text{where } x \text{ denotes a possible non-zero element.}$$

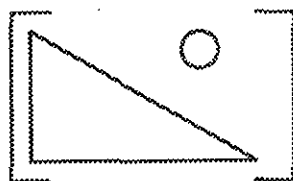
Figure 2.1.2 Tridiagonal matrix

- h) Quindigonal, if  $r = 2$ , see figure 2.1.3.

$$A = \begin{bmatrix} x & x & x & & & \\ x & x & x & x & & 0 \\ x & x & x & x & x & \\ & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & x & x & x & x & x \\ 0 & & & x & x & x & x \\ & & & & x & x & x \end{bmatrix} \quad \text{where } x \text{ denotes a possible non-zero element.}$$

Figure 2.1.3 Quindigonal matrix

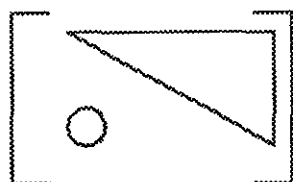
- i) Lower triangular, (strictly lower triangular), if  $a_{ij} = 0$ , for  $i < j$ , ( $i \leq j$ ).



where  $a_{ij} = 0$  for  $i < j$ .

Figure 2.1.4 A lower triangular matrix

- j) Upper triangular, (strictly upper triangular), if  $a_{ij} = 0$ , for  $i > j$ , ( $i \geq j$ ).



where  $a_{ij} = 0$  for  $i > j$ .

Figure 2.1.5 An upper triangular matrix

- k) Sparse, if a relatively large number of the elements  $a_{ij}$  are zero.  
l) Dense, if a relatively large number of the elements  $a_{ij}$  are non-zero.  
m) Block Diagonal, if each  $D_i$ ,  $i = 1, 2, \dots, n$ , (see figure 2.1.6) is a square matrix, but not necessarily of the same order.

$$A = \begin{bmatrix} D_1 & & & \\ & D_2 & & 0 \\ & & \ddots & \\ & 0 & & D_{n-1} \\ & & & & D_n \end{bmatrix}$$

Figure 2.1.6 Block Diagonal matrix

- n) Block Tridiagonal, if each  $D_i$ ,  $i = 1, 2, \dots, n$ , is a square matrix, but not necessarily of the same order, while the  $E$ 's and  $F$ 's are rectangular matrices, as shown in figure 2.1.7.

$$A = \begin{bmatrix} D_1 & F_1 & & & \\ E_2 & D_2 & F_2 & & 0 \\ & E_3 & D_3 & F_3 & \\ & & & & \\ & 0 & & E_{n-1} & D_{n-1} & F_{n-1} \\ & & & & E_n & D_n \end{bmatrix}$$

Figure 2.1.7 Block Tridiagonal matrix

## 2.2 Diagonal Dominance and Irreducibility

### Definition 2.2.1

A matrix of order  $n$  is diagonally dominant if

$$|a_{i,i}| \geq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{i,j}|, \text{ for all } 1 \leq i \leq n \quad (2.2.1)$$

and for at least one  $i$

$$|a_{i,i}| > \sum_{\substack{j=1 \\ j \neq i}}^n |a_{i,j}|. \quad (2.2.2)$$

If (2.2.2) holds for all  $i$ , then  $A$  has strong diagonal dominance.

The irreducibility of a matrix  $A$  as defined by [Young 71] is as follows:

### Definition 2.2.2

A matrix of order  $n$  is said to be irreducible if  $n=1$  or if  $n > 1$  and given any two non-empty disjoint subsets  $S$  and  $T$  of  $W$ , the set of the first  $n$  positive integers, such that  $S \cup T = W$ , there exists  $i \in S$  and  $j \in T$  such that  $a_{i,j} \neq 0$ . [Varga 62] stated that a matrix of order 1 is irreducible if its single element is non-zero, otherwise reducible.

**Theorem 2.2.1**

$A$  is irreducible if and only if there does not exist a permutation matrix  $P$  such that  $P^1AP$  has the form

$$P^1AP = \begin{bmatrix} F & O \\ G & H \end{bmatrix} \quad (2.2.3)$$

where  $F$  and  $H$  are square matrices and  $O$  is the null matrix.

**Theorem 2.2.2**

A matrix of order  $n$  is irreducible if and only if  $n = 1$  or, given any two distinct integers  $i$  and  $j$  with  $i, j = 1, 2, \dots, n$  then  $a_{i,j} \neq 0$  or there exists  $i_1, i_2, \dots, i_n$  such that

$$a_{i,i_1} a_{i_1,i_2} \dots a_{i_n,j} \neq 0. \quad (2.2.4)$$

The concept of irreducibility can be shown graphically. Given a matrix  $A$ , a directed graph is constructed as follows: Label  $n$  distinct points (or nodes) in the planes as  $1, 2, \dots, n$ . For any non-zero element  $a_{i,j}$  of the matrix, connect the point  $i$  to the point  $j$  by means of a path directed from  $i$  to  $j$ . For non-zero diagonal elements  $a_{i,i}$ , the path goes from  $i$  to itself forming a loop.

**Definition 2.2.3**

A directed graph is strongly connected if for any ordered pair of nodes  $P_i$  and  $P_j$ , there exists a directed path

$$\vec{P_i P_{k_1}}, \vec{P_{k_1} P_{k_2}}, \vec{P_{k_2} P_{k_3}}, \dots, \vec{P_{k_{r-1}} P_{k_r = j}}$$

connecting  $P_i$  to  $P_j$ . Such a path has length  $r$ .

**Theorem 2.2.3**

A square matrix is irreducible if and only if its directed graph  $G(A)$  is strongly connected.

**Definition 2.2.4**

An irreducible matrix which is also diagonally dominant with strict inequalities holding for at least one  $i$  in definition 2.2.2 is said to be irreducibly diagonally dominant. If definition 2.2.2 holds for all  $i$ , then  $A$  has strong diagonal dominance.

As an example, consider the matrix  $P$  given below.

$$P = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad (2.2.5)$$



The directed graph of  $P$  in (2.2.5) is given as in figure 2.2.1.

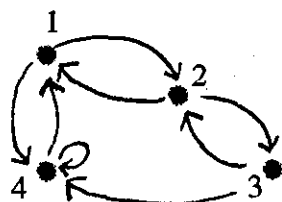


Figure 2.2.1

The matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

in which its directed graph is given in figure 2.2.2 is not irreducible.



Figure 2.2.2

The directed graph in figure 2.2.2 is not connected since there is no path from point 3 to point 1. Also, point 1 or point 2 cannot be reached starting from point 2 or point 3 respectively.

Some fundamental theorems follow.

#### Theorem 2.2.4

If  $A$  is an irreducible matrix with diagonal dominance, then  $\det(A) \neq 0$  and none of the diagonal elements of  $A$  are zero.

#### Corollary 2.2.1

From theorem 2.2.4, if  $A$  is strongly diagonally dominant, then  $\det(A) \neq 0$ .

There is now a sufficient condition for an Hermitian matrix to be positive definite.

#### Theorem 2.2.5

If  $A$  is an Hermitian matrix with non-negative diagonal elements and with diagonal dominance, then  $A$  is non-negative definite. If  $A$  is also irreducible or non-singular, then  $A$  is positive definite. Positive definite matrices will be discussed in more detail in section 2.5.

### 2.3 Eigenvalues and eigenvectors

Suppose that  $A$  is a matrix of order  $n$  and  $x \neq 0$  is a vector of the same order. An eigenvalue of  $A$  is a real or a complex number  $\lambda$  such that

$$Ax = \lambda x \quad (2.3.1)$$

It is also called a characteristic or latent root of  $A$ . An eigenvector of  $A$  is a vector  $x$  such that  $x \neq 0$  and (2.3.1) holds for some  $\lambda$ . This vector is sometimes called the characteristic or latent vector of  $A$ . The equation (2.3.1) can be written as

$$(A - \lambda I)x = 0 \quad (2.3.2)$$

The non-trivial solution  $x \neq 0$  to equation (2.3.2) exists if and only if the matrix of the system is singular, thus leading to the theorem 2.3.1.

#### Theorem 2.3.1

The number  $\lambda$  is an eigenvalue of  $A$  if and only if

$$\det(A - \lambda I) = 0 \quad (2.3.3)$$

(2.3.3) is a polynomial equation, referred to as the characteristic equation of  $A$  of the form

$$(-1)^n \det(A - \lambda I) = \lambda^n - \text{trace}(A) \lambda^{n-1} + \dots + (-1) \det(A) = 0 \quad (2.3.4)$$

Since the sum and product of the roots of (2.3.4) are  $\text{trace}(A)$  and  $\det(A)$  respectively, theorem 2.3.2 can be derived.

#### Theorem 2.3.2

If  $A$  is a matrix of order  $n$  with eigenvalues  $\lambda_1, \lambda_2, \dots, \lambda_v$ , where  $v \leq n$ , then

$$\det(A) = \prod_{i=1}^v \lambda_i, \quad \text{trace}(A) = \sum_{i=1}^v \lambda_i. \quad (2.3.5)$$

The left hand side of (2.3.3) is called the characteristic polynomial of  $A$ , which can also be written as,

$$\alpha_0 + \alpha_1 \lambda + \dots + \alpha_{n-1} \lambda^{n-1} + (-1)^n \lambda^n = 0 \quad (2.3.6)$$

where  $\alpha_i, i=0, 1, \dots, n-1$  are constants.

It is clear that since the coefficient of  $\lambda^n$  is not zero, the equation (2.3.6) always has  $n$  roots either real or complex which are the eigenvalues of the matrix  $A$ , namely as  $\lambda_1, \lambda_2, \dots, \lambda_n$  (not necessarily having the same values), each of them possessing a unique corresponding eigenvector.

In physical problems, all the eigenvalues of (2.3.2) are rarely found. In particular, it is often necessary to determine the largest eigenvalue in modulus, where it is often termed as the dominant eigenvalue or spectral radius. One of the methods for obtaining the dominant eigenvalue with its corresponding eigenvector is called the Power Method.

**Definition 2.3.2**

Given a matrix  $A$  of order  $n$  with eigenvalues  $\lambda_i$ ,  $1 < i < n$ , then

$$\rho(A) = \max |\lambda_i| \quad (2.3.7)$$

is the spectral radius of  $A$ .

**Definition 2.3.3**

The P-condition number of the matrix  $A$  is defined as  $P=b/a$ , where  $a, b \in \mathbb{R}$  satisfy  $a \leq \lambda_i \leq b$ ,  $i=1(1)n$  and are the largest and smallest eigenvalues respectively.

The spectral radius is extremely useful (particularly in iterative solution of linear systems) because it allows the structure and properties of a coefficient matrix via the eigenvalues to influence the performance of the solution technique.

**Definition 2.3.4**

Two matrices  $A$  and  $B$  of order  $n$  are similar if there exists a non-singular matrix  $P$  such that

$$B = P^{-1}AP \quad (2.3.8)$$

Matrix  $B$  is said to be obtained from matrix  $A$  by a similarity transformation and if  $B$  is symmetric then  $P$  will be orthogonal, i.e.,  $P^{-1} = P^T$ , and hence

$$B = P^TAP \quad (2.3.9)$$

The advantage of such a transformation is that the eigenvalues of  $A$  and  $B$  are the same. This can be shown as follows:

Let  $\lambda$  and  $x$  be the eigenvalue and the eigenvector of the matrix of  $A$  respectively. Hence,

$$Ax = \lambda x \quad (2.3.10)$$

then premultiply by  $P^{-1}$  to have

$$P^{-1}Ax = \lambda P^{-1}x. \quad (2.3.11)$$

Thus if  $\mathbf{u} = P^{-1}\mathbf{x}$ , then

$$\mathbf{x} = P\mathbf{u}. \quad (2.3.12)$$

Substituting (2.3.12) into (2.3.11),

$$P^{-1}AP\mathbf{u} = \lambda\mathbf{u} \quad (2.3.13)$$

$$\text{or} \quad B\mathbf{u} = \lambda\mathbf{u}. \quad (2.3.14)$$

Thus,  $\lambda$  is the eigenvalue of  $B$  and  $\mathbf{u}$  is the corresponding eigenvector.

### Theorem 2.3.1 (Gerschgorin's theorem)

If  $A = [a_{ij}]$  is a matrix of order  $n$ , then all the eigenvalues of  $A$  lie in the union of the discs,

$$|\lambda - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|, \quad 1 \leq i \leq n. \quad (2.3.15)$$

Proof:

Let  $\lambda$  and  $\mathbf{x}$  be the eigenvalue and the eigenvector of matrix  $A$  respectively.  $\mathbf{x}$  can be normalised so that  $\max_i |x_i| = 1$ . Hence, the following is obtained from (2.3.1).

$$\lambda\mathbf{x} = A\mathbf{x} \quad (2.3.16)$$

$$\lambda x_i = \sum_{j=1}^n a_{ij} x_j, \quad 1 \leq i \leq n \quad (2.3.17)$$

$$\text{i.e., } (\lambda - a_{ii})x_i = \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j, \quad 1 \leq i \leq n. \quad (2.3.18)$$

Now,  $|x_k| = 1$ , then

$$|\lambda - a_{kk}| \leq \sum_{\substack{j=1 \\ j \neq k}}^n |a_{kj}| |x_j| \leq \sum_{\substack{j=1 \\ j \neq k}}^n |a_{kj}| = D_k. \quad (2.3.19)$$

Thus the eigenvalue  $\lambda$  lies within the disc  $D_k$ , say, and since  $\lambda$  is arbitrary, then it follows that all the eigenvalues of  $A$  must lie in the union of discs, i.e.,

$$|\lambda - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|, \quad 1 \leq i \leq n. \quad (2.3.20)$$

### Corollary 2.3.1

If  $A = [a_{ij}]$  is a matrix of order  $n$  and we have

$$v_1 = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|, \quad (2.2.21)$$

$$v_2 = \max_{1 \leq j \leq n} \sum_{i=1}^n |a_{ij}|, \quad (2.2.22)$$

$$\text{then } \rho(A) \leq \min(v_1, v_2). \quad (2.2.23)$$

The condition (2.2.23) is a direct consequence of the fact that  $A$  and  $A^T$  have the same eigenvalues.

## 2.4 Vector and matrix norms

This section discusses the size or magnitude of a vector or matrix. This measure is called a norm and is enclosed within  $\|$ .

### Definition 2.4.1

The norm of a vector  $\mathbf{u}$ , denoted by  $\|\mathbf{u}\|$ , is a non-negative number satisfying the following three axioms:

$$1) \|\mathbf{u}\| = 0, \text{ for } \mathbf{u} = 0 \text{ and } \|\mathbf{u}\| > 0 \text{ if } \mathbf{u} \neq 0. \quad (2.4.1)$$

$$2) \|\alpha \mathbf{u}\| = |\alpha| \cdot \|\mathbf{u}\| \text{ for any complex scalar } \alpha, \quad (2.4.2)$$

$$3) \|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\| \text{ for any vectors } \mathbf{u} \text{ and } \mathbf{v}. \quad (2.4.3)$$

The axiom (2.4.3) is called the triangle inequality. From (2.4.3), the axiom (2.4.4) can be derived.

$$\|\mathbf{u} - \mathbf{v}\| \geq \|\mathbf{u}\| - \|\mathbf{v}\| \quad (2.4.4)$$

The most commonly used norms are the  $L_1$ ,  $L_2$  and  $L_\infty$  norms of  $\mathbf{u}$  and they are defined as follows:

### Definition 2.4.2

If  $\mathbf{u} = [u_1, u_2, \dots, u_n]^T$  is a vector of order  $n$  then

$$L_1 = \|\mathbf{u}\|_1 = \sum_{i=1}^n |u_i| \quad (2.4.5)$$

$$L_2 = \|\mathbf{u}\|_2 = \left[ \sum_{i=1}^n |u_i|^2 \right]^{1/2}, \quad (\text{Euclidean norm}) \quad (2.4.6)$$

$$L_\infty = \|\mathbf{u}\|_\infty = \max_i |u_i|, \quad (\text{maximum or uniform norm}) \quad (2.4.7)$$

For these three special cases, the general case for  $L_p$  for  $p > 1$  can be defined as:

$$L_p = \|\mathbf{u}\|_p = \left[ \sum_{i=1}^n |u_i|^p \right]^{1/p} \quad (2.4.8)$$

The matrix norm can be defined in a similar manner.

### Definition 2.4.3

A norm of a matrix  $A$  of order  $n$ , denoted as  $\|A\|$  is a scalar satisfying the following axioms:

$$1) \|A\| > 0 \text{ and } \|A\| = 0 \text{ if and only if } A = [0] \quad (2.4.9)$$

$$2) \|\alpha A\| = |\alpha| \cdot \|A\| \text{ for any scalar } \alpha. \quad (2.4.10)$$

$$3) \|A + B\| \leq \|A\| + \|B\| \text{ for any matrices } A \text{ and } B. \quad (2.4.11)$$

$$4) \|AB\| \leq \|A\| \cdot \|B\| \text{ for any matrices } A \text{ and } B. \quad (2.4.12)$$

In the same way,  $L_1$ ,  $L_2$  and  $L_\infty$  are given by

$$L_1 = \|A\|_1 = \max_j \sum_{i=1}^n |a_{i,j}| \quad (\text{maximum absolute column sum}) \quad (2.4.13)$$

$$L_2 = \|A\|_2 = \left( \text{maximum of } A^H A \right)^{\frac{1}{2}} \quad (\text{spectral norm}) \quad (2.4.14)$$

$$L_\infty = \max_i \sum_{j=1}^n |a_{i,j}|, \quad (\text{maximum absolute row sum}) \quad (2.4.15)$$

A norm compatible with the  $L_2$  vector norm is the Euclidean norm or Schur norm and is defined as follows:

$$L_2 = \|A\|_E = \left[ \sum_{i,j} |a_{i,j}|^2 \right]^{\frac{1}{2}} \quad (2.4.16)$$

### Definition 2.4.4

A matrix norm  $\|A\|$  is said to be compatible with a vector norm  $\|u\|$  if

$$\|Au\| \leq \|A\| \cdot \|u\|, \text{ for all } u \neq 0. \quad (2.4.17)$$

### Definition 2.4.5

A matrix norm is said to be subordinate to the corresponding vector norm if it can be constructed in the following form:

$$\|A\| = \max_{u \neq 0} \frac{\|Au\|}{\|u\|} \quad (2.4.18)$$

or equivalent to

$$\|A\| = \max \|Au\|, \quad \|u\| = 1. \quad (2.4.19)$$

## 2.5 Positive definite and special matrices

There are many definitions for the property of positive definiteness of a matrix  $A$ .

### Definition 2.5.1

If a matrix  $A$  of order  $n$  is Hermitian, and the quadratic form

$$(\mathbf{x}, A\mathbf{x}) > 0 \quad (2.5.1)$$

for  $\mathbf{x} \neq 0$ , then  $A$  is positive definite.

The matrix  $A$  is non-negative definite if  $(\mathbf{x}, A\mathbf{x}) \geq 0$ . The other definition of positive definiteness of a matrix  $A$  is stated by the following theorem.

### Theorem 2.5.1

The necessary and sufficient condition for a Hermitian or a real symmetric matrix  $A$  to be positive definite is that, the eigenvalues of  $A$  are all positive.

### Theorem 2.5.2

An irreducible, diagonally dominant matrix which is also symmetric and positive real diagonal elements is positive definite. The proof of these theorems may be found in [Young 71].

Following are definitions of some special matrices.

### Theorem 2.5.3

If  $A = [a_{ij}]$  is a real matrix of order  $n$ , then it is said to be

- 1) an L-matrix if

$$a_{ii} > 0, i = 1, 2, \dots, n \quad \text{and} \quad (2.5.2)$$

$$a_{ij} \leq 0, i, j = 1, 2, \dots, n. \quad (2.5.3)$$

- 2) a Stieltjes matrix if  $A$  is positive definite and if (2.5.3) holds.

- 3) an M-matrix if  $A$  is non-singular, if (2.5.3) holds and if  $A^{-1} > 0$ .

It should be noticed that by stating  $A > 0$ , it means that all elements of the matrix  $A$  are real and non-negative. A simple matrix  $A$  of order 3 which satisfies these three definitions of special matrices is

$$A = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix}$$

## 2.6 Property $\mathcal{A}$ and consistently ordered matrices

The property  $\mathcal{A}$  of a matrix  $A$  of order  $n$  can be defined as follows:

### Definition 2.6.1

A matrix  $A = [a_{ij}]$  of order  $n$  is said to have property  $\mathcal{A}$  if there exists two disjoint subsets  $S$  and  $T$  of  $W = \{1, 2, \dots, n\}$  such that if  $i \neq j$  and if either  $a_{ij} \neq 0$  and  $a_{ji} \neq 0$ , then  $i \in S$  and  $j \in T$  or else  $i \in T$  and  $j \in S$ .

The property  $\mathcal{A}$  can also be defined as follows:

### Definition 2.6.2

A matrix  $A$  of order  $n$  has property  $\mathcal{A}$  if there exists a permutation matrix  $P$  such that  $PAP^T$  has the form

$$PAP^T = \begin{bmatrix} D_1 & F \\ E & D_2 \end{bmatrix} \quad (2.6.1)$$

where  $D_1$  and  $D_2$  are square diagonal matrices.

A consistently ordered matrix is defined as follows:

### Definition 2.6.3

A matrix  $A$  of order  $n$  is consistently ordered if for some  $t$  there exists disjoint subsets  $S_1, S_2, \dots, S_t$  of  $W = \{1, 2, \dots, n\}$  such that

$$\sum_{k=1}^t S_k = W$$

and are such that if  $i$  and  $j$  are associated, then  $j \in S_{k+1}$  if  $j > i$  and  $j \in S_{k-1}$  if  $j < i$  where  $S_k$  is the subset containing  $i$ .

The following theorem can be used as an alternative definition of a consistently ordered matrix.

### Theorem 2.6.1

If  $A$  is a T-matrix, then  $A$  is consistently ordered.

The proof of this theorem can be referred to in [Young 71].

The ordering vector for the matrix  $A$  is defined in the following definition.

### Definition 2.6.4

A column vector  $v$  of order  $n$  with integer elements is an ordering vector for the matrix  $A$  of order  $n$  if for any pair of associated integers  $i$  and  $j$  with  $i \neq j$ ,  $|v_i - v_j| = 1$ .

A compatible ordering vector for the matrix  $A$  is defined as follows:

### Definition 2.6.5



An ordering vector  $v = [v_1, v_2, \dots, v_n]^T$  for the matrix  $A$  of order  $n$  is a compatible ordering vector of  $A$  if

- 1)  $v_i - v_j = 1$  if  $i$  and  $j$  are associated and  $i > j$ .
- 2)  $v_i - v_j = -1$  if  $i$  and  $j$  are associated and  $i < j$ .

### Theorem 2.6.2

There exists an ordering vector for a matrix  $A$  if and only if  $A$  has property  $\mathcal{A}$ .

Once again, the proof for the theorem can be referred to in [Young 71].

## 2.7 Rate of convergence

A linear stationary iterative method is defined as

$$u^{(k+1)} = Bu^{(k)} + c \quad (2.7.1)$$

where  $B$  is known as the iteration matrix.

In practice, even if the method (2.7.1) converges, it may converge very slowly. Therefore, it is important to evaluate the effectiveness of an iterative method. In order to do this, both the computation required for each iteration and the number of iterations required for convergence at a given accuracy must be considered.

A sequence of matrices  $A^{(1)}, A^{(2)}, A^{(3)}, \dots$  all of the same order is said to converge to a limit  $A$  if

$$\lim_{k \rightarrow \infty} A^{(k)} = A. \quad (2.7.2)$$

### Theorem 2.7.1

The sequence of matrices  $A^{(1)}, A^{(2)}, A^{(3)}, \dots$  converges to  $A$  if and only if for every matrix norm  $\| \cdot \|_p$ ,

$$\lim_{k \rightarrow \infty} \|A^{(k)} - A\|_p = 0. \quad (2.7.3)$$

### Theorem 2.7.2 [Varga 62]

If  $A$  is a matrix of order  $n$ , then  $A$  is convergent if and only if

$$\rho(A) < 1. \quad (2.7.4)$$

### Theorem 2.7.3

The matrix  $I - B$  is non singular and the series  $I + B + B^2 + \dots$  converges if and only if  $\rho(B) < 1$ . Moreover, if  $\rho(B) < 1$ , then

$$(I - B)^{-1} = I + B + B^2 + \dots = \sum_{i=0}^{\infty} B^i. \quad (2.7.5)$$

The proof of these theorems can be found in [Young 71].

The basic convergence criterion of the method (2.7.1) is as follows:

The method (2.7.1) converges if

$$\lim_{k \rightarrow \infty} u_i^{(k)} = u_i, \text{ for all } i \quad (2.7.6)$$

and for all starting vectors  $\mathbf{u}^{(0)}$ .

#### Theorem 2.7.4

The iterative method (2.7.1) is convergent if and only if

$$\rho(B) < 1. \quad (2.7.7)$$

Therefore, from equation (2.4.14), we have the following corollary.

#### Corollary 2.7.1

A sufficient condition for the convergence of (2.7.1) is merely that

$$\|B\| < 1. \quad (2.7.8)$$

It follows from this that a sufficient condition for convergence is that  $\|B\| < 1$ . It is not a necessary condition because  $\|G\|$  can exceed one even when  $\rho(G) < 1$ .

The necessary condition for an iterative process to converge is for the spectral radius of  $B$  to be less than 1, i.e.  $\rho(B) < 1$ . The spectral radius is the largest value of  $|\lambda_i|$  for a given matrix  $B$ . The system with a smaller spectral radius converges faster. The test for convergence can be done without actually calculating the eigenvalues. By Gerschgorin's theorem,  $\rho(B) \leq \max(\|B\|_1, \|B\|_\infty)$ . Thus if the maximum sum of the moduli of the elements of the rows or columns of  $B$  is less than 1, then the iterative process will converge. However, the estimate of the upper bound of  $\rho$  is used only as a guide to establish convergence and is not accurate enough to establish and compare rates of convergence.

As stated earlier, the rate of convergence of the method (2.7.1) may be determined by calculating the number of iterations at a predetermined accuracy. In practice, the usual approach is to iterate until the norm of the error vector  $\mathbf{e}^{(k)}$  is reduced to less than some given tolerance, say  $\epsilon$ , of the norm of the initial vector  $\mathbf{e}^{(0)}$ . If the error vector after  $k$  iterations is defined as

$$\mathbf{e}^{(k)} = \mathbf{u}^{(k)} - \mathbf{u} \quad (2.7.9)$$

where  $\mathbf{u}$  is the exact vector solution, then applying the method (2.7.1) would result in

$$\mathbf{e}^{(k+1)} = B\mathbf{e}^{(k)} \quad (2.7.10)$$

and therefore

$$\begin{aligned} \mathbf{e}^{(k)} &= B\mathbf{e}^{(k-1)} \\ &= B^2\mathbf{e}^{(k-2)} = B^k\mathbf{e}^{(0)}. \end{aligned} \quad (2.7.11)$$

From (2.7.11),

$$\begin{aligned} \|\mathbf{e}^{(k)}\| &= \|B^{(k)}\mathbf{e}^{(0)}\| \\ &\leq \|B^{(k)}\| \|\mathbf{e}^{(0)}\| \end{aligned} \quad (2.7.12)$$

If  $\mathbf{e}^{(0)} \neq 0$  then,

$$\frac{\|\mathbf{e}^{(k)}\|}{\|\mathbf{e}^{(0)}\|} \leq \|B^k\|. \quad (2.7.13)$$

It is required that

$$\|\mathbf{e}^{(k)}\| \leq \varepsilon \|\mathbf{e}^{(0)}\| \quad (2.7.14)$$

where  $\|\cdot\|$  denotes  $\|\cdot\|_2$  as defined in section 2.4. By theorem (2.7.4) it is known that  $\|B^k\|$  converges to zero as  $k$  approaches infinity if and only if  $\rho(B) < 1$ . Hence equation (2.7.14) can be satisfied by choosing  $k$  sufficiently large such that

$$\|B^k\| \leq \varepsilon. \quad (2.7.15)$$

If  $k$  is large enough so that  $\|B^k\| < 1$ , then it follows that (2.7.15) may be written as

$$k \geq -\frac{\log \varepsilon}{(-\frac{1}{k} \log \|B^k\|)} \quad (2.7.16)$$

From this inequality, the lower bound for the number of iterations for the iterative method (2.7.1) can be determined.

### Definition 2.7.1

For any convergent iterative method of the form (2.7.1), the quantity

$$R_k(B) = -\frac{1}{k} \log \|B^k\| \quad (2.7.17)$$

is called the average rate of convergence after  $k$  iterations.

If  $R_k(B_1) < R_k(B_2)$  for matrices  $B_1$  and  $B_2$  then for  $k$  iterations,  $B_2$  is iteratively faster than  $B_1$ .

### Definition 2.7.2

The asymptotic average rate of convergence is defined by

$$R(B) = \lim_{k \rightarrow \infty} R_k(B) = -\log \rho(B) \quad (2.7.18)$$

This is true since,

$$\rho(B) = \lim_{k \rightarrow \infty} (\|B^k\|)^{1/k} \quad (2.7.19)$$

as proved by [Young 71].

It is usual for iterative methods to converge slowly for substantially large problems corresponding to the values of  $\rho(B)$  being only slightly less than one, and a rate of convergence nearly zero.

Now, upon replacing  $\|B^k\|$  by  $[\rho(B)]^k$  in equation (2.7.16), it can be seen that  $\varepsilon \approx [\rho(B)]^k$ , and the number of iterations  $k$  can be estimated as

$$k \approx \frac{-\log \varepsilon}{-\log \rho(B)} = \frac{-\log \varepsilon}{R(B)} \quad (2.7.20)$$

However, the value of  $k$  from (2.7.20) could be very much lower when compared with the number required, in which  $\|B^k\|$  will behave like  $k[\rho(B)]^{k-1}$ , rather than  $[\rho(B)]^k$ , as mentioned by [Young 71]. In this case the smallest value of  $k$  such that

$$k[\rho(B)]^{k-1} \leq \varepsilon \quad (2.7.21)$$

estimates the number of iterations required more accurately.

There are many ways to perform the convergence tests in order to determine the number of iterations for a given tolerance value. It is obvious that a different stopping criteria will yield a different number of iterations. However, a better test will yield a better accuracy. In this thesis, the average test will be used, i.e.,

$$\frac{\|u_i^{(k+1)} - u_i^{(k)}\|}{\|1 + u_i^{(k)}\|} < \varepsilon \quad \text{for all } i, \quad (2.7.22)$$

where  $\varepsilon = 10^{-5}$ .

For small values of  $u_i^{(k)}$ , this test approximates the absolute test  $\|u_i^{(k+1)} - u_i^{(k)}\|$ , and for large values of  $u_i^{(k)}$ , it approximates the relative test, i.e.,  $(\|u_i^{(k+1)} - u_i^{(k)}\| / \|u_i^{(k)}\|)$ , for all  $i$ .

## 2.8 Direct methods for the solution of linear systems.

The previous sections have introduced a mathematical basis for linear systems of the form

$$Ax = b \quad (2.8.1)$$

Sections 2.8 and 2.9 will in turn discuss the methods of constructing the solution to the system and assume for convenience that  $A$  is non-singular ( $A^{-1}$  exists) so that the solution is unique. The methods discussed are intended for use on computers and so the solution of the system is generally only approximate due to rounding errors introduced by the finite word (length) calculations. However, the growth of errors is bounded in practice and results are acceptable especially if double precision arithmetic is used.

The choice of solution method depends on a number of factors including the structure and size of the matrix  $A$ , the number of arithmetic operations required to construct the solution, the amount of storage required for  $A$  and  $b$  in (2.8.1), and the control of rounding error growth (stability).

In this section direct methods to solve linear systems of equations are considered. These methods are applicable to dense matrices and have the advantage of producing a solution after a fixed number of operations proportional to the matrix order. Furthermore, in most cases the accuracy of the solution is usually stable and adequate for our purposes. The methods discussed are the Gaussian Elimination (GE) in section 2.8.1, LU factorisation in section 2.8.2 and QR factorisation in section 2.8.3. The coefficient matrices in these methods are assumed to be dense and of type real.

Equation (2.8.1) is an example of an implicit system where the solution vector cannot be derived without modification to the system. A brute force approach is to solve (2.8.1) by converting it to an equivalent explicit form, using the fact that  $A$  is non-singular. This yields

$$x = A^{-1}b \quad (2.8.2)$$

and  $x$  is constructed explicitly by performing matrix-vector multiplication. This implicit-explicit conversion is fine if  $A^{-1}$  is already known, or easily constructed. However, this is not generally the case.

Direct methods are aimed at a compromise which manipulates  $A$  and  $b$  to produce a semi-explicit form,

$$A'x = b', \quad (2.8.3)$$

where  $x$  can be derived from an ordered substitution process, and  $A'$  is a matrix with an easily solvable structure, and  $b'$  is a modified right hand side.

### 2.8.1 Gaussian Elimination

This method is used to solve a system of linear equations by transforming it into an upper triangular system (i.e. one in which all of the coefficients below the leading diagonal are zero) using elementary row operations. The solution of the upper triangular matrix is then obtained using a back substitution process.

Consider a general system  $Ax=b$  of  $n$  equations and  $n$  unknowns. In matrix form, it can be written as (2.8.1.1).

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} \quad (2.8.1.1)$$

The system in (2.8.1.1) is transformed using forward elimination to

$$Ux = b', \quad (2.8.1.2)$$

where  $U$  is an upper triangular matrix and  $b'$  is the modified right hand side vector. This matrix is then easily solved by back substitution.

#### The forward elimination process

To obtain the upper triangular matrix from the system in (2.8.1.1), all the elements below the diagonal will be zeroed out or eliminated. The first row is used to eliminate elements in the first column below the diagonal. The first row is then known as the pivotal row and the element  $a_{11}$  is called the pivotal element. The values  $\frac{a_{k1}}{a_{11}}$  are multiples of row 1 (also known as the multipliers) that are to be subtracted from row  $k$ , for  $k=2, 3, \dots, n$  times the first equation from the second. The result after the first stage of elimination is (2.8.1.3).

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & \dots & a_{3n}^{(1)} \\ 0 & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2^{(1)} \\ \dots \\ b_n^{(1)} \end{bmatrix} \quad (2.8.1.3)$$

Equations 2 to  $n$  of (2.8.1.3) constitute a linear system of  $n-1$  equations with  $n-1$  unknowns  $x_2, x_3, \dots, x_n$  to solve. Thus the elimination process described earlier can be applied recursively to this subsystem. To obtain the upper triangular matrix in (2.8.1.2), the elimination process is repeated  $(n-1)$  times on matrix  $A$ .

### The back substitution process

After transforming the system in (2.8.1.1) to the form  $Ux=b'$ , where

$$U = \begin{bmatrix} u_{1,1} & \dots & u_{1,j} & \dots & u_{1,n} \\ 0 & \ddots & \vdots & & \vdots \\ 0 & 0 & u_{j,j} & \dots & u_{j,n} \\ 0 & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & 0 & u_{n,n} \end{bmatrix} \quad (2.8.1.4)$$

and the  $i$ th element of  $b'$  is  $b'_i$ . To obtain the solution vector  $x$ , we begin with the final equation in the system (2.8.1.4) which is

$$u_{nn}x_n = b'_n,$$

which gives

$$x_n = b'_n / u_{nn}$$

We then work backwards to substitute  $x_n$  into the  $(n-1)$ th equation and solve for  $x_{n-1}$ . In general, the  $i$ -th equation in (2.8.1.4) is

$$u_{ii}x_i + u_{i,i+1}x_{i+1} + \dots + u_{in}x_n = b'_i$$

After determining  $x_n, x_{n-1}, \dots, x_{i+1}$ , the unknown  $x_i$  is given by

$$x_i = \frac{1}{u_{ii}} \left( b'_i - \sum_{j=i+1}^n u_{ij}x_j \right) \quad (2.8.1.5)$$

The algorithm for GE is given in chapter 3.

GE is best suited for a system of linear equations with one right-hand side. When there are more than one right-hand side, it is better to separate the modification of the

coefficient matrix from the right-hand side. The advantage is that if we have a new right-hand side whose value is not known at the start of the elimination process, only the right-hand side needs to be modified. This is achieved by matrix factorisation methods.

## 2.8.2 LU factorisation

Factorisation is an alternative to triangularisation which avoids modification of the right hand side of (2.8.1) and so is more convenient for multiple right hand sides. A well-known matrix factorisation is the LU method where the coefficient matrix  $A$  is decomposed into a pair of factors  $L$  and  $U$ , such that:

$$A=LU \quad (2.8.2.1)$$

where  $L$  is a unit lower triangular matrix and  $U$  is an upper triangular matrix as shown in (2.8.2.2).

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,j} & \cdots & a_{1,n} \\ \vdots & & \vdots & & \vdots \\ a_{i,1} & \cdots & a_{i,j} & \cdots & a_{i,n} \\ \vdots & & \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,j} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \vdots & \ddots & 0 & 0 & 0 \\ l_{i,1} & \cdots & 1 & 0 & 0 \\ \vdots & & \vdots & \ddots & 0 \\ l_{n,1} & \cdots & l_{n,j} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{1,1} & \cdots & u_{1,j} & \cdots & u_{1,n} \\ 0 & \ddots & \vdots & & \vdots \\ 0 & 0 & u_{j,j} & \cdots & u_{j,n} \\ 0 & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & 0 & u_{n,n} \end{bmatrix} \quad (2.8.2.2)$$

From (2.8.2.1), the linear system  $Ax=b$  can be written as,

$$LUx=b \quad (2.8.2.3)$$

To obtain the solution of equation  $Ax=b$  by using this method, an auxiliary column vector  $y$  must be introduced such that,

$$Ux=y, \quad (2.8.2.4)$$

and then the triangular system,

$$Ly=b, \quad (2.8.2.5)$$

must be solved by a forward substitution process before the triangular system in (2.8.2.4) is solved for  $x$  by back substitution as previously described for GE in section 2.8.1.

The forward elimination process of GE in section 2.8.1 can be used to construct the triangular factorisation of the matrix  $A$ . The matrix  $L$  will be constructed from an identity matrix placed at the left as in (2.8.2.6).

$$A=IA \quad (2.8.2.6)$$



For each row operation used to construct the upper triangular matrix, the multipliers (discussed in section 2.8.1) of each row are placed in the corresponding location in the identity matrix.

After obtaining the factorisation, the solution can be obtained in two steps:

- Solve  $Ly=b$  for  $y$  using forward substitution.
- Solve  $Ux=y$  for  $x$  using backward substitution.

In the forward substitution process, the matrix form is as shown in (2.8.2.7).

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \\ l_{31} & & \ddots & \\ \vdots & & & 1 & 0 \\ l_{n1} & & & l_{n-1,n-1} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad (2.8.2.7)$$

The value of  $y_1$  is easily obtained and is equal to  $b_1$ . The known value of  $y_1$  is then substituted into the second equation

$$l_{21}y_1 + y_2 = b_2$$

and now the value of  $y_2$  is known. This substitution continues and in the  $i$ th stage, the equation to be solved is

$$l_{i1}y_1 + l_{i2}y_2 + \dots + l_{ii}y_i = b_i$$

After obtaining the values of  $y_1, y_2, \dots, y_{i-1}$ ,

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij}y_j$$

The back substitution process has been described in section 2.8.1.

The algorithm for the LU factorisation is given in chapter 3.

A number of methods for producing  $L$  and  $U$  factors which satisfy (2.8.2.1) are known. They can be classified according to whether the diagonal element  $l_{ii}$  or  $u_{ii}$  are set to 1 or equal ( $l_{ii} = u_{ii}$ ). The methods are:

- Doolittle's method which is  $l_{ii} = 1, i=1(1)n$  formulated as

$$\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}, \quad j \geq i \\ l_{ji} &= \frac{1}{u_{ii}} \left[ a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki} \right] \quad j > i, i=1(1)n \end{aligned} \quad (2.8.2.8)$$

(b) Crout's method is  $u_{ii} = 1, i=1(1)n$  given as

$$u_{ij} = \frac{1}{l_{ii}} \left[ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right] \quad j \geq i$$

$$l_{ji} = a_{ji} - \sum_{k=1}^{i-1} l_{jk} u_{ki} \quad j > i, i = 1(1)n$$
(2.8.2.9)

(c) Choleski's method ( $l_{ii} = u_{ii}$  effectively)

$$l_{ii} = \left( a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2 \right)^{\frac{1}{2}}$$

$$l_{ij} = \frac{\left( a_{ji} - \sum_{k=1}^{i-1} l_{ik} l_{jk} \right)}{l_{ii}} \quad , i > j, \quad j = 1(1)n$$
(2.8.2.10)

Note that if any  $l_{ii}$  or  $u_{ii}$  is zero, the methods break down. The formulae (2.8.2.8) and (2.8.2.9) are general methods for non-singular matrices, while (2.8.2.10) is applicable only for symmetric positive definite matrices and  $A=LL^T$ . A root-free form of Choleski with  $A=LDL^T$  ( $D$  is a diagonal matrix) is also possible. Due to the fact that only the entries of  $L$  are computed in (2.8.2.10) savings in memory and computation time can be made over GE and LU. However, the method requires a relatively complex square root calculation.

### 2.8.3 QR factorisation

The aim of the QR factorisation is also to transform the system  $Ax=b$  into one that is easy to solve. In this method, the coefficient matrix  $A$  is decomposed into an orthogonal matrix  $Q$  and an upper triangular matrix  $R$ . Since  $Q$  is orthogonal  $Q^T Q = I_n$ . Hence if we premultiply the equation

$$Ax=b \tag{2.8.3.1}$$

by  $Q^T$  and replace  $A$  by  $QR$ , we will obtain:

$$Q^T(QR)x = Q^T b$$

$$Rx = Q^T b.$$

There are at least three ways in which the orthogonal matrix  $Q$  can be computed:

- (a) Product of Householder reflection matrices.
- (b) Product of Givens rotation matrices.
- (c) Generation of a set of orthogonal vectors from the columns of  $A$  using modified Gram-Schmidt orthogonalisation.

In this thesis, the second method is considered, i.e. obtaining the orthogonal matrix  $Q$  by taking the product of Givens rotation matrices. Here, each rotation eliminates a single element. The complete decomposition process can be described as:

$$Q^T A = Q_r^T Q_{r-1}^T \cdots Q_2^T Q_1^T A = R \quad (2.8.3.2)$$

where  $Q_1^T, Q_2^T, \dots, Q_r^T$  are  $(n \times n)$  Givens transformations and  $R$  is the resulting triangular matrix. Substituting for  $A$  from equation (2.8.3.2) in equation (2.8.3.1) and considering the orthogonality property of  $Q^T Q = I$ , results in:

$$Rx = Q^T b \quad (2.8.3.3)$$

which can be solved by the normal back substitution process.

The rotation matrix is:

$$P = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \quad (2.8.3.4)$$

where the angles of rotation are determined by the formulae:

$$\begin{aligned} \sin \theta &= \frac{a_{i,j}}{\sqrt{(a_{i,j}^2 + a_{i-1,j}^2)}} \\ \cos \theta &= \frac{a_{i-1,j}}{\sqrt{(a_{i,j}^2 + a_{i-1,j}^2)}} \end{aligned} \quad (2.8.3.5)$$

The algorithm for QR factorisation using the Givens method is given in chapter 3.

#### 2.8.4 Partial Pivoting

Pivoting is a widely used technique for improving the numerical stability of matrix elimination and matrix factorisation methods. The GE and LU methods described in sections 2.8.1 and 2.8.2 are appropriate when the diagonal elements are dominant. This is often the case but in general, attention must be given to numerical instability resulting when the diagonal elements are not dominant.

There are two basic well known pivoting schemes; partial pivoting and complete pivoting. In partial pivoting an element of largest modulus in the column of each

reduced matrix is chosen as the pivot. Elements of rows which have previously been pivoted are not considered. At the  $k^{\text{th}}$  stage of partial pivoting, the rows of the matrix are interchanged such that the largest element in the  $k^{\text{th}}$  column is used as the pivot. That is, after the pivoting,

$$|a_{kk}| = \max |a_{ik}|, \text{ for } i=k, k+1, \dots, n.$$

In the complete pivot scheme, the pivot at each stage of the reduction is chosen as the element of largest magnitude in the submatrix of rows which have not been pivoted up to now, regardless of the position of the element in the matrix. This may require both row and column interchanges. At the  $k^{\text{th}}$  step of complete pivoting, both the rows and columns of the matrix are interchanged so that the largest number in the remaining matrix is used as pivot. That is, after pivoting,

$$|a_{kk}| = \max |a_{ij}|, \text{ for } i=k, k+1, \dots, n \text{ and } j=k, k+1, \dots, n.$$

The advantage of partial pivoting is that the pivotal row is always multiplied by a number whose magnitude is less than unity before its subtraction from other rows in the elimination process. Any rounding errors present in the pivotal row are thereby decreased in absolute magnitude, the propagated effect of these errors being decreased correspondingly.

The propagation of rounding errors for full pivoting is less, theoretically, but the main disadvantage is more time being consumed to search for the largest pivot.

In this section, we describe partial pivoting strategies for GE in section 2.8.4.1 and LU in section 2.8.4.2. The QR method is already stable because each row is being multiplied by the rotation matrix consisting of cosine and sine functions whose values are always less than one and hence does not need pivoting.

#### 2.8.4.1 Partial Pivoting for Gaussian Elimination

It is not always advisable to carry out the forward elimination in section 2.8.1 even when the matrix of coefficients is non-singular. The reason is that division by  $a_{ii}$  is not always possible if it is very small as it leads to unacceptable rounding errors. Partial pivoting is performed to overcome this problem and to maintain numerical stability against growth of rounding error.

Consider the  $k^{\text{th}}$  elimination stage in section 2.8.1, we search below the  $k^{\text{th}}$  column of  $A^{(k-1)}$  (the pivotal column) for the element of largest modulus. Suppose that it is  $a_{rk}^{(k-1)}$

for some  $r \geq k$ . Then we interchange equations  $r$  and  $k$  and the elimination process proceeds using the new  $k^{\text{th}}$  row (the pivotal row) to perform the eliminations.

#### 2.8.4.2 Partial Pivoting for LU factorisation

Partial pivoting for LU consists of  $n$  major steps in the  $r^{\text{th}}$  step of which we determine the  $r^{\text{th}}$  column of  $L$ , the  $r^{\text{th}}$  row of  $U$  and  $r'$ . The configuration at the beginning of the  $r^{\text{th}}$  step for the case  $n=5, r=3$  is given by:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} & c_1 & s_1 \\ l_{21} & u_{22} & u_{23} & u_{24} & u_{25} & c_2 & s_2 \\ l_{31} & l_{32} & a_{33} & a_{34} & a_{35} & b_3 & s_3 \\ l_{41} & l_{42} & a_{43} & a_{44} & a_{45} & b_4 & s_4 \\ l_{51} & l_{52} & a_{53} & a_{54} & a_{55} & b_5 & s_5 \end{bmatrix}$$

The  $r^{\text{th}}$  major step is as follows:

(a) For  $t=r, r+1, \dots, n$ :

Calculate  $a_{tr} - l_{t1}u_{1r} - l_{t2}u_{2r} - \dots - l_{t,r-1}u_{r-1,r}$ , storing the result as  $s_t$  at the end of row  $t$ .

(b) Suppose  $|s_{r'}|$  is the maximum of  $|s_t|$  ( $t=r, \dots, n$ ). Then store  $r'$  and interchange the whole rows of  $r$  and  $r'$  including the  $l_{ri}, a_{ri}, b_r, s_r$ . The new  $s_r$  is now  $u_{rr}$  and overwrite this on  $a_{rr}$ .

(c) For  $t=r+1, \dots, n$ :

Compute  $s_t / u_{rr}$  to give  $l_{tr}$  and overwrite on  $a_{tr}$ .

(d) For  $t=r+1, \dots, n$ :

Compute  $a_{ti} - l_{r1}u_{1i} - l_{r2}u_{2i} - \dots - l_{r,r-1}u_{r-1,i}$  to give  $u_{ti}$ . Overwrite  $u_{ti}$  on  $a_{ti}$ .

(e) Compute  $b_r - l_{r1}c_1 - l_{r2}c_2 - \dots - l_{r,r-1}c_{r-1}$  to give  $c_r$ . Overwrite  $c_r$  on  $b_r$ .

## 2.9 Iterative Methods

It is well known that iterative methods, utilising the great speeds of modern-day computers, are extensively used in large scale computations for solving linear equations that arise from finite difference approximations to ordinary differential equations and partial differential equations. Iterative methods are infinite methods and produce only approximation to the solution. They are easy to define and hence are very widely used. These methods consist of repeated application of a simple algorithm. One begins with an initial approximation and then successively modifies the approximation according to some rules. To be useful, the iteration must converge and it is not considered to be effective unless the convergence is rapid.

Iterative methods have two advantages over direct methods. First, although these methods do not yield a solution in a finite number of steps, one can terminate after a finite number of iterations when it has produced a sufficiently good approximate to the solution. Secondly, most iterative schemes require simple arithmetic operations only on the non-zero entries of the coefficient matrix, hence being suitable for sparse matrices for which elimination methods would be relatively very laborious and need a lot of storage. Iterative methods are used mainly in those problems for which convergence is known to be rapid so that the solution is obtained with much less work than that of a direct method.

Iterative methods are built around a partition (or splitting) of matrix  $A$ . This section presents the conventional diagonal splitting of  $A$  into  $A=D-L-U$  where  $D$  is the main diagonal of  $A$ ,  $-L$  and  $-U$  are strictly lower and upper triangular elements of  $A$  respectively. The simplest method, Jacobi, is covered in section 2.9.2 and the Gauss-Seidel method is covered in section 2.9.3. Accelerated versions of Jacobi, the Jacobi Overrelaxation method (JOR), and the accelerated version of Gauss-Seidel, the Successive Overrelaxation method (SOR), are covered in section 2.9.5. The model problem on which these methods have been applied is discussed in section 2.9.1.

Given the matrix system

$$Ax=b \quad (2.9.1)$$

consider then the splitting

$$A=D-L-U \quad (2.9.2)$$

where  $D$  is the diagonal matrix of  $A$ ,  $-L$  and  $-U$  are strictly lower and upper triangular elements of  $A$  respectively.

From (2.9.1) the iterative formula of the form

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{c} \quad (2.9.3)$$

can be deduced. By choosing an initial value  $\mathbf{x}_0$ , we can use (2.9.3) to generate  $\mathbf{x}_1, \mathbf{x}_2, \dots$  etc. Provided the process is convergent, successive values of  $\mathbf{x}_k$  will give a closer approximation to the actual solution  $\mathbf{x}$ .

### 2.9.1 The model problem

The model problem used to test the iterative methods in this research study is taken from a set of coupled ordinary differential equations (O.D.E.). The problem is described briefly in this section.

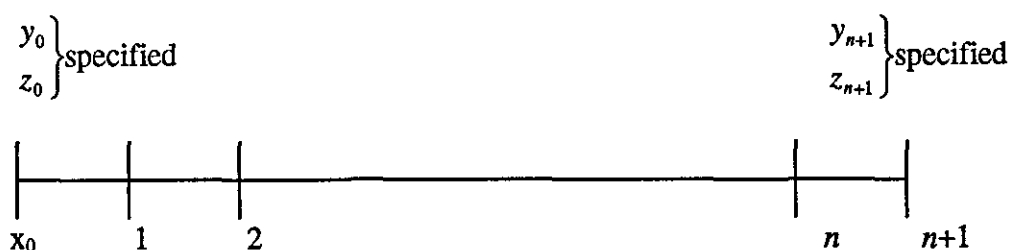
Given the problem,

$$-\frac{d^2 y}{dx^2} + py + qz = f_1(x), \quad (2.9.1.1)$$

$$-\frac{d^2 z}{dx^2} + qz + py = f_2(x), \quad (2.9.1.2)$$

with specified boundary values at the end points of the interval (0,1).

Discretisation of the equations (2.9.1.1) and (2.9.1.2) by central difference operators on an equally spaced grid points of  $x_i, i=1,2,\dots,n$ , where  $x_i = x_0 + ih$ .



will result in the following set of finite difference equations,

$$\frac{(-y_{i-1} + 2y_i - y_{i+1}))}{h^2} + py_i + qz_i = f_1(x_i), i = 1, 2, \dots, n, \quad (2.9.1.3)$$

$$\frac{(-z_{i-1} + 2z_i - z_{i+1}))}{h^2} + qz_i + py_i = f_2(x_i), \text{ and } h = \frac{1}{n+1} \quad (2.9.1.4)$$

with the values  $y_0, y_{n+1}, z_0$ , and  $z_{n+1}$  specified as above and  $p = q = 1/h^2$ .

By setting up equations (2.9.1.3) and (2.9.1.4) in matrix form, and numbering the  $y_i$ , ( $i=1,2 \dots, n$ ) values in increasing order and  $z_i$  values in reverse order, the following matrix equation is obtained.

$$\left[ \begin{array}{cccc|cccc} 3 & -1 & & 0 & & & & -1 \\ -1 & 3 & & & & & & \\ & & \ddots & & & & & \\ 0 & & -1 & 3 & & \ddots & & \\ \hline & & & & \ddots & & & \\ & & & & & 3 & -1 & 0 \\ & & & & & -1 & \ddots & \\ & & & & & & 3 & -1 \\ -1 & & & & & & 0 & -1 & 3 \end{array} \right] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ z_n \\ z_{n-1} \\ \vdots \\ z_1 \end{bmatrix} = \begin{bmatrix} h^2 f_1(x_1) + y_0 \\ h^2 f_1(x_2) \\ \vdots \\ h^2 f_1(x_n) + y_{n+1} \\ h^2 f_2(x_n) + z_{n+1} \\ h^2 f_2(x_{n-1}) \\ \vdots \\ h^2 f_2(x_1) + z_0 \end{bmatrix} \quad (2.9.1.5)$$

The matrix  $A$  of (2.9.1.5) can be expressed as

$$A = 3 [I - B] \quad (2.9.1.6)$$

where  $B$  is a  $nxn$  real symmetric matrix given explicitly by

$$\frac{1}{3} \begin{bmatrix} 0 & 1 & & & 1 \\ 1 & 0 & 1 & & \\ & 1 & \ddots & \ddots & \\ & \ddots & \ddots & 0 & 1 \\ 1 & & & 1 & 0 \end{bmatrix} \quad (2.9.1.7)$$

## 2.9.2 The Jacobi iterative method

The simplest of the iterative methods is that attributed to Jacobi, also known as the Simultaneous Displacement method. It is not widely used in practice but it's theoretical interest may provide a convenient starting point to establish convergence analysis.

In matrix form and using the splitting in (2.9.2) the following iterative equation is obtained.

$$\mathbf{x}^{(k+1)} = D^{-1}(L+U)\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \quad (2.9.2.1)$$

In the form of the equation (2.9.3)

$$B = D^{-1}(L+U), \quad \mathbf{c} = D^{-1}\mathbf{b}. \quad (2.9.2.2)$$

For the model problem in section 2.9.1, the algorithm for the Jacobi iterative method is shown in Algorithm 2.9.2.1.



### Algorithm 2.9.2.1 Jacobi Algorithm

*Step 1: Set  $x_i^{(k)} = 0$ , for  $i=1$  to  $n$ .*

*Step 2: for  $i=1$  to  $n$*

$$x_i^{(k+1)} = (b_i + x_{i-1}^{(k)} + x_{i+1}^{(k)} + x_{n-i+1}^{(k)}) / 3$$

*Step 3: Repeat step 2 until convergence is achieved.*

### 2.9.3 Successive Displacement or Gauss-Seidel method

A simple modification of the Jacobi method leads to the Gauss-Seidel method also known as the Successive Displacement method. Instead of waiting to use the updated values at the end of an iteration, the updated values are used as soon as they are available.

In matrix form and using the splitting in (2.9.2) the following iterative equation is obtained.

$$\mathbf{x}^{(k+1)} = (D-L)^{-1}U\mathbf{x}^{(k)} + (D-L)^{-1}\mathbf{b} \quad (2.9.2.3)$$

In terms of equation (2.9.3),

$$\mathbf{B} = (D-L)^{-1}U, \quad \mathbf{c} = (D-L)^{-1}\mathbf{b} \quad (2.9.2.4)$$

For the model problem in section 2.9.1 the algorithm for the Gauss-Seidel iterative method is as shown in Algorithm 2.9.3.1.

### Algorithm 2.9.3.1 Gauss-Seidel Algorithm

*Step 1: Set  $x_i^{(k)} = 0$ , for  $i=1$  to  $n$ .*

*Step 2: for  $i=1$  to  $n$*

*if  $i < n/2$  then*

$$x_i^{(k+1)} = (b_i + x_{i-1}^{(k+1)} + x_{i+1}^{(k)} + x_{n-i+1}^{(k)}) / 3$$

*else*

$$x_i^{(k+1)} = (b_i + x_{i-1}^{(k+1)} + x_{i+1}^{(k)} + x_{n-i+1}^{(k+1)}) / 3$$

*Step 3: Repeat step 2 until convergence is achieved*

#### 2.9.4 Tests for convergence

In order for an iterative process to converge, from section 2.7, the spectral radius of the iterative matrix must be less than one. Evaluating the spectral radius can be a long process. However, in certain cases, a decision on convergence can be made without actually evaluating the eigenvalues but by performing the following tests.

- a) If any of the diagonal elements of  $A$  are zero, the Jacobi and Gauss-Seidel methods cannot be used as  $D$  and  $(D+L)$  will be singular.
- b) By Gerschgorin's theorem of (2.2.23), we have

$$\rho(B) \leq \min(\|B\|_1, \|B\|_\infty)$$

Thus, if the maximum sum of the moduli of elements of the rows or columns of  $B$  is less than one, then the iterative process will converge.

In the model problem of section 2.9.1

$$\rho_{\text{Jacobi}}(B) \leq 1$$

- c) In the case of the Jacobi method, rule (b) requires that  $A$  be diagonally dominant. In the model problem of section 2.9.1,  $A$  is diagonally dominant by Definition 2.2.1.
- d) If each equation (2.9.2.3) is divided by its diagonal element, the Gauss-Seidel iterative matrix becomes  $(I-L)^{-1}U$  and in this form the Stein-Rosenberg theorem may be applied.

Theorem 2.9.4.1 (Stein-Rosenberg)

If  $B = -(L+U)$  is a non-negative matrix, then either

- (a)  $\rho_{\text{Jacobi}} = \rho_{\text{Gauss-Seidel}} = 0$ ,
- (b)  $\rho_{\text{Jacobi}} = \rho_{\text{Gauss-Seidel}} = 1$ ,
- (c)  $0 < \rho_{\text{Gauss-Seidel}} < \rho_{\text{Jacobi}} < 1$  or
- (d)  $1 < \rho_{\text{Jacobi}} < \rho_{\text{Gauss-Seidel}}$ .

It follows immediately from this theorem that if the Jacobi iterative matrix is non-negative then the Jacobi and Gauss-Seidel methods both converge or diverge together and in the case of convergence the Gauss-Seidel method will always be at least as fast as the Jacobi and in most cases even faster.

- (e) If  $A$  is real, symmetric and positive definite, then the Gauss-Seidel iterative matrix (2.9.2.3) has spectral radius less than one. The proof can be found in [Cohen 73].

### 2.9.5 Accelerated convergence for Jacobi and Gauss-Seidel methods

The rate of convergence of the Jacobi and the Gauss-Seidel methods may be accelerated by the intelligent use of a weighting or acceleration function  $\omega$ . A measure of the rate of convergence of a particular method can be obtained by taking the norm of difference of two successive iterative values, that is

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \epsilon \quad (2.9.5.1)$$

where  $\epsilon$  is a small specified quantity.

Consider the case of the Jacobi method. By (2.9.2.1)

$$\begin{aligned} D\mathbf{x}^{(k+1)} - D\mathbf{x}^{(k)} &= (L + U)\mathbf{x}^{(k)} + \mathbf{b} - D\mathbf{x}^{(k)} \\ &= \mathbf{b} - (D - L - U)\mathbf{x}^{(k)} \\ &= \mathbf{b} - A\mathbf{x}^{(k)} \end{aligned} \quad (2.9.5.2)$$

The change in  $\mathbf{x}$  between two successive steps of the calculation is dependent upon the residual vector of the previous step. Therefore, by taking a different proportion of this quantity we might accelerate convergence. This leads to

$$D(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \omega(\mathbf{b} - A\mathbf{x}^{(k)}) \quad (2.9.5.3)$$

where  $\omega$  is some selected weighted factor.

From (2.9.5.3) the iterative relationship

$$\mathbf{x}^{(k+1)} = (I - \omega D^{-1}A)\mathbf{x}^{(k)} + \omega D^{-1}\mathbf{b} \quad (2.9.5.4)$$

defines the Jacobi Overrelaxation method (JOR).

The rate of convergence is governed by the spectral radius of the iterative matrix which is

$$(I - \omega D^{-1}A) \quad (2.9.5.5)$$

In the Jacobi method, the convergence was governed by  $(I - D^{-1}A)$ . In order to get an accelerated convergence, it is reasonable to choose  $\omega$  such that the largest eigenvalue  $\lambda_2$  of  $(I - \omega D^{-1}A)$  is smaller than  $\lambda$  of  $(I - D^{-1}A)$  [Cohen 73].

A similar treatment can be given to the Gauss-Seidel case where

$$\begin{aligned}
 D(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) &= L\mathbf{x}^{(k+1)} + U\mathbf{x}^{(k)} + \mathbf{b} - D\mathbf{x}^{(k)} \\
 &= \mathbf{b} - (D - L - U) \mathbf{x}^{(k)} + L\mathbf{x}^{(k+1)} - L\mathbf{x}^{(k)} \\
 &= \mathbf{b} - A\mathbf{x}^{(k)} + L(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})
 \end{aligned} \tag{2.9.5.6}$$

To obtain the accelerated version of Gauss-Seidel also known as the Successive Overrelaxation method (SOR), multiply the right hand side of (2.9.5.6) by the weighting function  $\omega$  to give

$$\begin{aligned}
 D(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) &= \omega(\mathbf{b} - A\mathbf{x}^{(k)}) + \omega L(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) \\
 D\mathbf{x}^{(k+1)} - \omega L\mathbf{x}^{(k+1)} &= \omega(\mathbf{b} - A\mathbf{x}^{(k)}) - \omega L\mathbf{x}^{(k)} + D\mathbf{x}^{(k)} \\
 &= \omega\mathbf{b} - \omega A\mathbf{x}^{(k)} - \omega L\mathbf{x}^{(k)} + D\mathbf{x}^{(k)} \\
 \mathbf{x}^{(k+1)} &= (D - \omega L)^{-1} \omega\mathbf{b} - (D - \omega L)^{-1} \omega A\mathbf{x}^{(k)} + (D - \omega L)^{-1} (D - \omega L)\mathbf{x}^{(k)} \\
 &= (D - \omega L)^{-1} \omega\mathbf{b} - (D - \omega L)^{-1} \omega A\mathbf{x}^{(k)} + I\mathbf{x}^{(k)} \\
 &= (D - \omega L)^{-1} \omega\mathbf{b} + [I - (D - \omega L)^{-1} \omega A]\mathbf{x}^{(k)}
 \end{aligned} \tag{2.9.5.7}$$

For the model problem in section 2.9.1 the algorithms for JOR and SOR are similar to that of Jacobi and Gauss-Seidel respectively, except that  $\omega$  is inserted in the iteration formulas as appropriate.

## 2.10 Computational Complexity

The operation count procedure for a method is a form of computational complexity analysis. One estimates the work of a certain computation, in this case the solution of  $Ax=b$ , in terms of basic machine arithmetic operations like multiply and addition. In this section, the computational complexity for the different methods to solve linear systems will be examined. The computational complexity for GE is covered in section 2.10.1, LU factorisation in section 2.10.2, QR factorisation in 2.10.3, and the iterative methods in section 2.10.4. A more detailed discussion will be presented in later chapters.

### 2.10.1 Gaussian Elimination

A detailed discussion on the measure of work (in terms of the number of multiply and addition operations involved) for GE can be found in [Fox 64], and is summarised as follows:

|                   |  |
|-------------------|--|
| Elimination:      | $\frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}$ |
| Backsubstitution: | $\frac{1}{2}n(n-1)$                            |
| Total:            | $\frac{n^3}{3} + n^2 - \frac{n}{3}$            |

### 2.10.2 LU factorisation

A detailed discussion on the measure of work (in terms of multiply and addition operations) for LU can be found in [Fox 64], and is summarised as follows:

|                   |   |
|-------------------|---|
| Elimination:      | $\frac{n^3}{3} - \frac{n}{3}$               |
| Backsubstitution: | $\frac{1}{2}[n(n-1)] + \frac{1}{2}[n(n-1)]$ |
| Total:            | $\frac{n^3}{3} + n^2 - \frac{n}{3}$         |

### 2.10.3 QR factorisation

QR factorisation is more expensive than Gaussian elimination or LU factorisation. However, this extra cost may be compensated by the fact that QR does not involve pivoting. This may be an advantage for structured matrices and parallel computation.

In order to evaluate the trigonometric values in (2.8.3.6), a total of 2 multiply + 1 add + 2 div + 1 sqrt is required. This is a total of

$$(2m+1a+2div+1sqrt) \text{ for } n(n-1)/2 \text{ points.} \quad (2.10.3.1)$$

The calculation of new entries for the reduced matrix and RHS is

$$\sum_{j=1}^n (2m+1a)(2j+1)(j-1) \quad (2.10.3.2)$$
$$\approx 2n^3 \text{ operations} = O(n^3)$$

The back substitution process is of complexity  $O(n^2)$ .

The dominating operation is the updating procedure in (2.10.3.2), hence the QR factorisation is said to be of order  $O(n^3)$ .

### 2.10.4 Iterative methods

In terms of computational work, the iterative methods seem to be an attractive scheme. Both the Jacobi and the Gauss-Seidel methods only require  $2n$  mults +  $3n-2$  adds per iteration. Slightly more work is required for the overrelaxation methods JOR and SOR due to the computation involving the weighting factors,  $\omega$ . However, this does not change the order of complexity of the methods which is at most  $O(n^2)$ .

### Chapter 3

#### A survey of current methods in solving linear equations on parallel computers

Parallel processing, the simultaneous execution of multiple processors to solve a single computational problem co-operatively, is a fast growing technology that permeates many areas of computer science and engineering. A parallel computer is a set of processors that are able to work co-operatively to solve a computational problem. There are many application areas where the available power of a sequential computer is insufficient to obtain the desired results. These problems can be classified into two categories. First, problems characterised by inordinate size and complexity, such as detailed weather or cosmological modelling that often require hours or days of conventional processing. Second is real-time problems that require computations to be performed within a strictly defined time period and are typically driven by external events.

Uniprocessor performance growth has been, and continues to be impressive. However, there are signs of diminishing returns from smaller and faster devices, as they begin to struggle with injecting power into, and heat out of, smaller and smaller volumes. Costs of the development process are also growing faster than performance so that each new generation of processors costs more to build than the previous ones, and offer smaller performance improvements. Eventually, the speed of light is an upper limit on single device performance, although we are some way off from that particular limitation. On the other hand, there is no inherent limit to the expansion, and therefore, computational power, of parallel architectures. Therefore, parallelism is the only long-term growth path for powerful computation.

Parallel computers offer the potential to concentrate computational resources, whether it be processors, memory or input/output bandwidth, on important computational problems. Parallelism brings unprecedented speed and cost-effectiveness but also raises a new set of complex and challenging problems to solve, most of which are software related. Parallel processing comprises algorithms, computer architecture, programming and performance analysis.

There has been a lot of work done on the development of parallel algorithms for Linear Algebra problems, including systems of linear equations, linear least squares problems and algebraic eigenvalue problems. This literature survey was conducted with the view

that the thesis is focused on parallel algorithms for the solution of linear systems on both shared-memory and distributed-memory architectures. In this chapter, a survey of current methods in solving systems of linear equations on parallel computers are covered. Section 3.1 presents an overview of parallel computers focusing on the architectures used for the experiments in this thesis, which were the shared-memory model and message-passing environments for workstation clusters. Section 3.2 covers the design and analysis of parallel algorithms as well as parallel programming methods employed on the parallel architectures used in this thesis. Section 3.3 is the survey on parallel methods for the solution of linear systems. The methods in focus are both the direct and iterative methods with the direct solvers covering Gaussian Elimination (GE), LU factorisation and QR factorisation while the iterative solvers include Jacobi, Gauss-Seidel, Jacobi OverRelaxation (JOR) and Successive OverRelaxation (SOR).

### 3.1 Parallel Architecture

One contributing factor to the widespread use and success of the traditional, sequential computer is that most of the time the user need not be concerned with hardware details. However, at this point in the development of parallel processing, some knowledge of the major hardware categories and their strengths and weaknesses is required.

A parallel architecture provides an explicit, high level framework for expressing and executing parallel programming solutions by providing multiple processors, whether simple or complex, that cooperate to solve problems through concurrent execution [Duncan 92].

There are two main classes of parallel machines: the Single Instruction Multiple Data (SIMD) machines in which the same task, usually of small granularity, is executed simultaneously on different data and the Multiple Instruction Multiple Data (MIMD) class in which different tasks can be executed on different processors. The distinctive aspect of SIMD execution consists of the control unit broadcasting a single instruction to all processors, which execute the instruction in lockstep fashion on local data. The MIMD architectures consist of multiple processors that can execute independent instruction streams. Thus, MIMD computers support parallel solutions that require processors to operate in a largely autonomous manner.



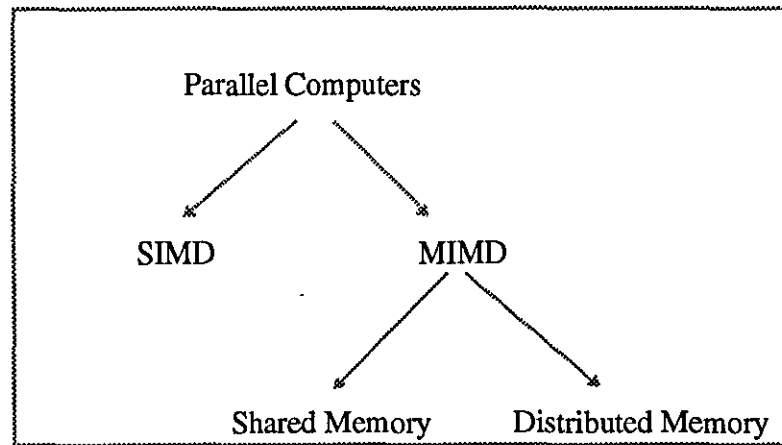


Figure 3.1.1 The two main classes of parallel computers

MIMD processors come into two distinct classes: shared-memory machines and distributed-memory/message-passing machines (Figure 3.1.1). In shared-memory machines, processors have access to a large global random access memory of which they have the same view. The software processes, executing on different processors, co-ordinate their activities by reading and modifying data values in the shared-memory. The co-ordination is achieved via different mechanisms that synchronise attempts to access the shared data. Processors are provided with a small fast local memory, in the form of data registers or cache. Access to the global memory is either via a high speed bus or a switching network. Figure 3.1.2 shows a simplified diagram of a shared memory parallel computer.

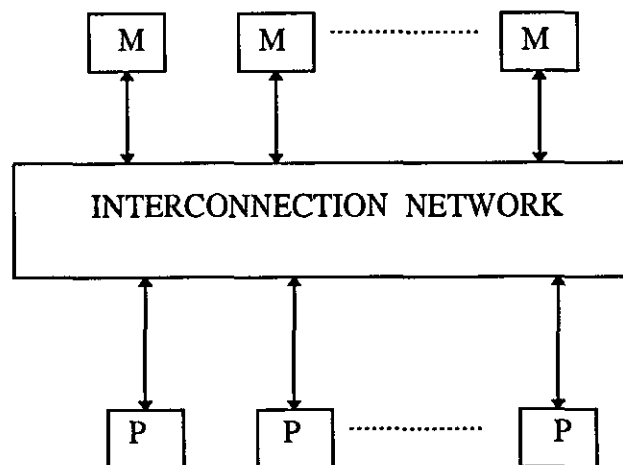


Figure 3.1.2 A typical shared memory architecture

Message-passing models comprise of a number of identical processors, each provided with a small private random access memory and interconnected in a regular topology. Processors in distributed-memory systems have no direct access to the memory of other processors. Figure 3.1.3 shows the diagram of a typical message passing architecture. In order to utilise multiple processors on one task, it is necessary to exchange information between processors by sending packets of data (messages) between them using an available communication network. Software libraries to facilitate such exchange of data are called message-passing environments. The emergence of message-passing environments has made distributed computing available to application programmers. Shared memory computers are relatively easy to program but difficult to scale up to large numbers of processors. On the other hand, distributed memory computers hold an advantage over shared memory machines when it comes to massive parallelism. However, there are at least two problems associated with using distributed memory machines which are low machine efficiency due to inter processor communication and they are difficult to program.

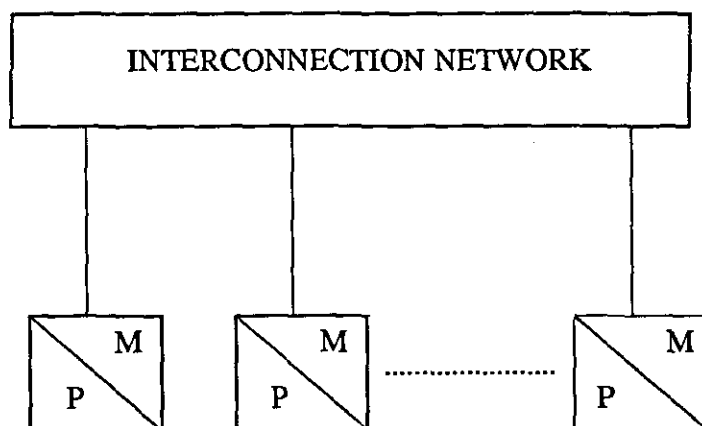


Figure 3.1.3 A typical message passing architecture

Another very important trend in high performance parallel architectures is the appearance of the distributed computing systems that are based on clusters of workstations interconnected via fast local area networks (LAN) which are presently implemented with the high bandwidth fibre optic technology. Communication in such systems is based on message-passing. These systems may utilise already available computing resources, frequently workstations of different brands and/or models

running distributed network computing software tools such as Parallel Virtual Machine (PVM), Message Passing Interface (MPI), Linda etc. For some applications, the cost effectiveness and flexibility of network distributed computing make it a very attractive alternative to the traditional parallel computing platform. Distributed computing is a process whereby a set of computers connected by a network are used collectively to solve a single large problem. As more and more organisations have high speed local area networks interconnecting many general purpose workstations, the combined computational resources may exceed the power of a single high performance computer. The most important factor in distributed computing is cost. Large Massively Parallel Processors (MPP) typically cost more than ten million dollars. In contrast, users see very little cost in running their problems on a local set of existing computers. A number of environments exist that make distributed computing available to the application programmer. Environments like PVM [Sunderam 94b], Message Passing Interface (MPI) [Walker 94], Linda [Carriero 94] and others [McBryan 94] have recently appeared allowing users to consider their computer networks as a virtual parallel machine. Such software contributes to the popularity of parallelism. Indeed, they allow us to do parallel computation without any access to 'real' parallel machines. The algorithms in this thesis were implemented on two classes of machines. First is the Sequent Balance 8000, a multiprocessor comprising of twelve identical processors which share a single common memory. All the processors, memory modules, and input/output controllers plug into a single high speed bus. The processors employ dynamic load balancing whereby processors automatically schedule themselves to ensure that they are kept busy as long as there are executable processes available. Each processor has 8 Kbytes of cache RAM. Sequent computers run the Dynix operating system, a version of Unix 4.2bsd that also supports most utilities, libraries, and system calls provided by Unix System V. The Sequent supports C and Fortran and both languages include extensions that allow programs to specify explicit parallelism. The schematic diagram of the Sequent Balance 8000 is shown in Figure 3.1.4.

The second class of machine was a cluster of Dec-Alpha workstations. This pool of processors was used as a virtual parallel machine cooperating on a task. This virtual parallel machine is made possible by the availability of message-passing environments such as PVM, MPI, Linda etc. In this research PVM is used to support the

parallelisation of programs using a loosely coupled network of workstations. Hence a description of PVM is given in more detail. Figure 3.1.5 shows a typical workstation cluster.

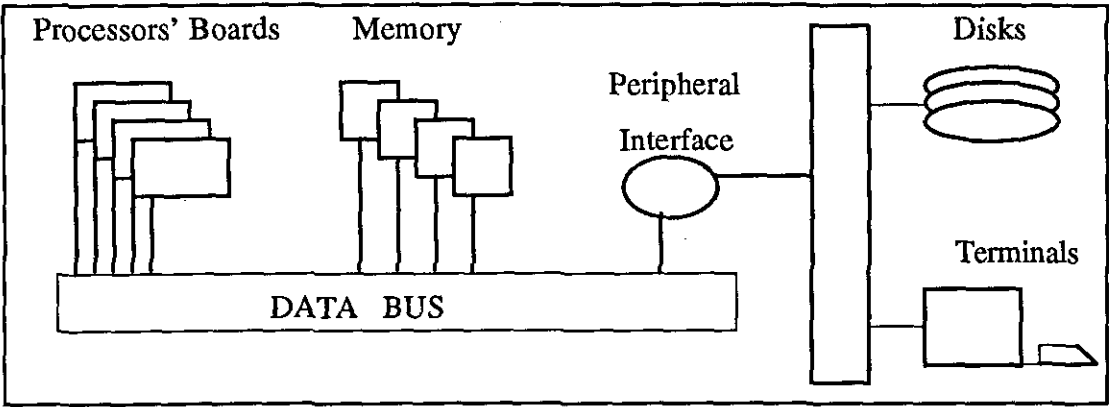


Figure 3.1.4 Sequent Balance 8000 Architecture

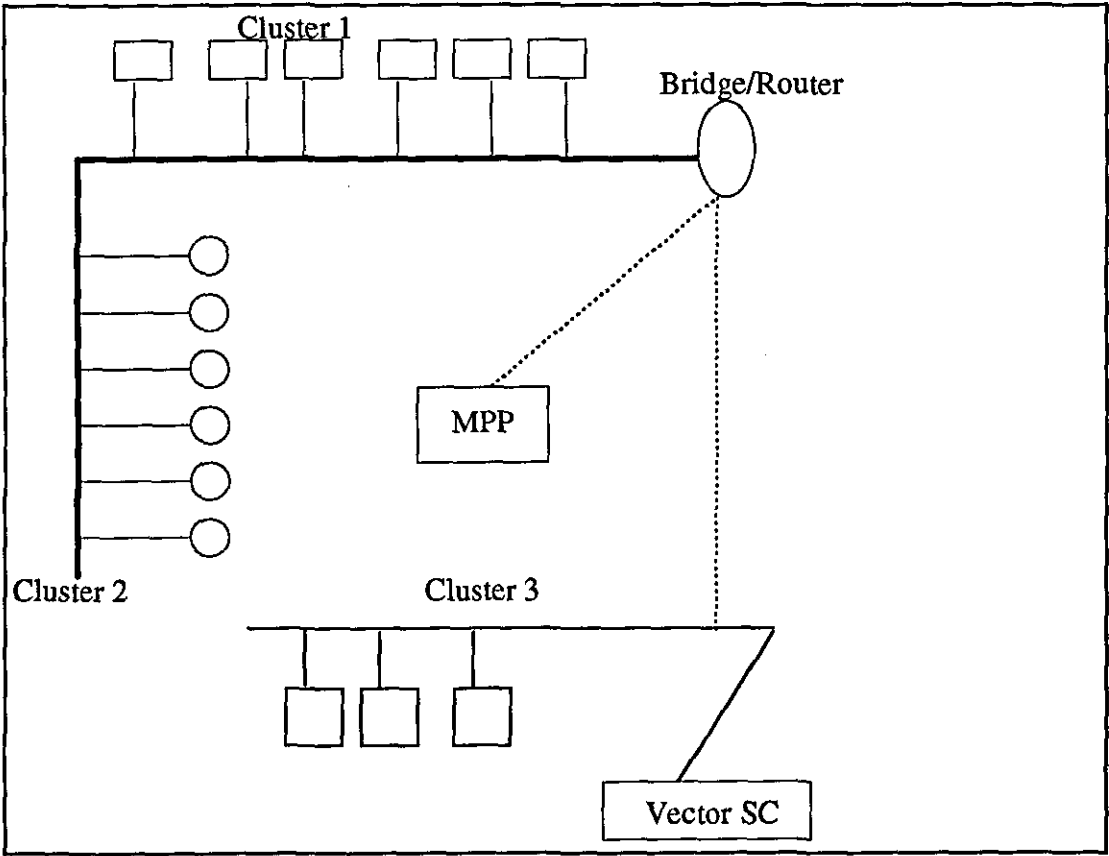


Figure 3.1.5 PVM Architectural Overview

PVM (Parallel Virtual Machine) is a portable message-passing programming system, designed to link separate host machines to form a 'virtual machine' which is a single,

manageable computing resource. The virtual machine can be comprised of hosts of varying types, (making it a heterogeneous environment) in physically remote locations or it can be composed of hosts of similar type making it a homogeneous environment. PVM applications can be composed of any number of separate processes written in a mixture of C, C++ and Fortran. The system is portable to a variety of architectures, including workstations, multiprocessors, supercomputers and personal computers. PVM is an on going research project started in 1989 at Oak Ridge National Laboratory (ORNL) and now involving people from ORNL, the University of Tennessee and Carnegie Mellon University. In the interests of advancing science, the software is available freely over the Internet. The latest version of the PVM software and documentation is always available through netlib which is a software distribution service set up on the Internet containing a wide range of computer software. Software can be obtained from netlib via anonymous file transfer protocol (ftp), world wide web (WWW), xnetlib or email. The ftp address is netlib2.cs.utk.edu and the files are in directory pvm3. Using a world wide web tool like Xmosaic or Netscape, PVM files can be accessed by using the address <http://www.netlib.org/pvm3/index.html>. Xnetlib is an X-Window interface that allows a user to browse or query netlib for available software and to automatically transfer the selected software to the user's computer. To get Xnetlib, send email to netlib@ornl.gov with the message send xnetlib.shar from xnetlib or anonymous ftp from cs.utk.edu pub/xnetlib. The PVM software can also be requested by email by sending email to netlib@ornl.gov with the message: send index from pvm3. [Geist 94a]

The PVM package is small (about 1 Mbytes of C source code) and easy to install. It needs to be installed only once on each machine to be accessible to all users. The installation does not require special privileges on any of the machines and thus can be done by any user.

The PVM user interface requires that all message data be explicitly typed. PVM performs machine-independent data conversions when required, thus allowing machines with different integer and floating point representation to pass data.

Applications can be parallelised by using message-passing constructs common to most distributed-memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel. PVM supplies the functions

to automatically start up tasks on the virtual machine and allows the task to communicate and synchronise with each other. In particular, PVM handles all message conversion that may be required if two computers use different data representations.

The PVM system is composed of two parts. The first part is a daemon which resides on all the computers that make up the virtual computer. Second is a library of PVM interface routines which contains user-callable routines for passing messages, spawning processes, co-ordinating tasks, and modifying the virtual machine. Application programs must be linked with this library to use PVM.

The programming methods for PVM will be discussed in later sections.

### **3.2 Parallel Algorithms**

The growth of parallel computers has led to most common algorithms being re-evaluated according to new viability and performance criteria. There are two approaches to deal with this; either one can look at how existing methods are best parallelised, or design new methods that are specially adapted to parallel machines. In this thesis, we take the latter approach of investigating the performance of methods which have been devised for parallel computation.

In order to utilise the enormous computing potential offered by the growth of parallel processing technology, it is necessary to devise parallel algorithms that can keep a large number of processors working in parallel towards the completion of one overall computation. The usefulness of a parallel computer largely depends on the invention of suitable parallel algorithms that operate efficiently on such a computer, and on the design of parallel languages in which these new algorithms can be expressed.

A parallel algorithm can be viewed as a collection of independent task modules that can be executed in parallel and that communicate with each other during the execution of the algorithm.[Kung 80] There are three orthogonal dimensions of the space of parallel algorithms, namely concurrency control, module granularity and communication geometry. The concurrency control enforces desired interactions among task modules so that the overall execution of the parallel algorithm will be correct. Module granularity of a parallel algorithm refers to the maximal amount of computation a typical task module can do before having to communicate with other modules. This reflects whether or not the algorithm tends to be communication

intensive. A parallel algorithm with a small module granularity will require frequent intermodule communication. Communication geometry is the geometric layout of the task modules of the parallel algorithm connected together to represent intermodule communication.

There are two classes of parallel algorithms which correspond to the two main classes of parallel computer architecture; the SIMD and MIMD classes discussed in Section 3.1. The two classes of parallel algorithms are the synchronous algorithms and the asynchronous algorithms.

Synchronous algorithms are algorithms with processes that contain interaction points by which the process can communicate with other processes. These interaction points divide a process into stages. At the end of each stage, a process may communicate with other processes before starting the next stage. Therefore, there is a need for synchronisation at these points and this degrades the performance of the algorithm.

An asynchronous algorithm is a parallel algorithm where communication between processes are achieved through global variables or shared data. To ensure correctness, the operations on shared data are programmed as critical sections. A critical section is a piece of code that accesses shared data. The main characteristic of an asynchronous algorithm is that the processes never wait for inputs at any time but continue or terminate according to whatever information is currently contained in the shared data. However, processes may be blocked from entering critical sections since access to critical sections follows the First In First Out (FIFO) rules.

### 3.2.1 Design of parallel algorithms

Parallel algorithm design is not an easy task. There is a need to manage explicitly the execution of multiple processors and co-ordinate interprocessor interaction. The parallel algorithm design process can be structured as four distinct stages: partitioning, communication, agglomeration and mapping [Foster 95]. In this methodology, machine-independent issues such as concurrency are considered early and machine-specific aspects of design are delayed until late in the design process.

In the *partitioning phase*, the computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as number of processors in the target computer are ignored and attention is focused in

recognising opportunities for parallel computation. The *communication stage* determines the communication required to co-ordinate task execution as well as define the appropriate communication structures and algorithms. In the *agglomeration stage*, the task and communication structures defined in the first and second stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs. Lastly, in the *mapping stage*, each task is assigned to a processor in a manner that attempts to satisfy the computing goals of maximising processor utilisation and minimising communication costs. Decomposition of data among nodes or processors requires synchronisation. If such a synchronisation is to be done frequently, the overhead can be quite significant. The cost of synchronisation is quite expensive, hence, a goal of the parallel algorithm designer should be to make the grain size as big as possible and avoid excessive use of the system bus (in a shared memory multiprocessor) or large number of exchanged messages (in message passing systems) for communication. Mapping can be specified statically or determined at run time by load balancing algorithms. This design process is a highly parallel process with many concerns being considered simultaneously.

Some parallelising techniques that can be used in the design of a parallel algorithm include *data parallelism*, *data partitioning*, *relaxed algorithm*, *synchronous iteration*, *replicated workers* and *pipelined computation*.

In *data parallelism*, a large number of different data items are subjected to similar processing all in parallel. This technique is useful in numerical algorithms that deal with large arrays and vectors.

*Data partitioning* technique is a technique where the data space is naturally partitioned into adjacent regions, each of which is operated on in parallel by a different processor. There may be occasional exchange of data values across the region boundaries. This technique is suitable for iterative numerical algorithms. Data partitioning is suitable for implementation on message-passing computers because computing is only on local data. Therefore, interprocessor communication is infrequent.

In a *relaxed algorithm*, each parallel process computes in a self sufficient manner with no synchronisation or communication between processes. This suits both shared-memory and message-passing architectures. On the other hand, in *synchronous*



*iteration*, each processor performs the same iterative computation on a different portion of data. However here the processors must be synchronised at the end of each iteration to ensure that no processor starts the next iteration before all the processors have finished the previous iteration. This technique suits very well the standard numerical algorithms used in science and engineering. *Synchronous iteration* is suitable for implementation on shared-memory architectures.

*Replicated workers* is a technique which maintains a central pool of similar computational tasks. There is a large number of worker processes that retrieve tasks from the pool, carry out the required computation, and possibly add new tasks to the pool. The overall computation terminates when the pool is empty. This technique is useful for combinatorial problems, for example tree or graph search where it is not known in advance how large the central pool is.

Lastly, in *pipelined computation*, processes are arranged in some regular structure such as ring or two-dimensional mesh. The data then flows through the entire process structure, with each process performing a certain phase of the overall computation. This is suitable for message-passing computers because of the orderly pattern of data flows and the lack of need for globally accessible data.

### 3.2.2 Analysis of parallel algorithms

The study of algorithms will not be complete without looking at the analysis of algorithms. The same applies for the study of parallel algorithms. The performance of a sequential algorithm is measured by time and memory space. However, performance of parallel computation requires not only an operation complexity analysis of algorithms, but also a careful look at the architecture of the parallel computer and specific parallel programming issues such as synchronisation and load balancing. Ortega and Voigt in [Ortega 85] showed that computational complexity, the basis for algorithm selection for decades is still relevant for vector computers because each computation costs some units of time but it is much less relevant for parallel computers for two reasons. First, parallel computers can support extra computation at no extra cost if the computation can be organised properly. Secondly, parallel computers are subject to new overhead costs required by communication and synchronisation that are not reflected by computational complexity.

The time taken by a parallel algorithm can be defined as the elapsed time of the process in the program which finishes last, where the elapsed time of the process is the sum of a) basic processing time which is the sum of the time taken by each stage b) blocked time, which is the total time that the process is blocked at the end of a stage because it is waiting for inputs in a synchronised algorithm or for entering a critical section in an asynchronous algorithm and c) execution time of synchronisation overhead; i.e. synchronisation housekeeping operations and implementing critical sections [Kung 80]. In some cases, the time complexity of the algorithm is governed by the time required by the overhead operations rather than the actual computations themselves. Therefore, on parallel computers, the goal of minimum execution time is not necessarily synonymous with performing the minimum number of arithmetic operations as in serial computers.

As mentioned above, the performance of a parallel algorithm cannot be evaluated in isolation from the parallel architecture. The combination of a parallel algorithm and the parallel architecture upon which it is implemented is known as a parallel system. There are various metrics that can be used to evaluate the performance of parallel systems. Some of the metrics are *run time*, *speedup*, *efficiency* and *cost*.

The *run time* of a serial program (denoted  $T_s$ ) is the time elapsed between the beginning and the end of its execution on a single processor. The run time of a parallel program (denoted  $T_p$ ) is the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution. Most scientists in the field agree that the most relevant measure of run time is actual wall clock elapsed time, that is the time that would be measured on an external clock that records the time of the day.

*Speedup* ( $S$ ) is the measure of relative benefit of parallelising a given application over a sequential implementation. It is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with  $p$  identical processors. Mathematically,

$$S = \frac{T_1}{T_p} \quad (3.2.2.1)$$

where  $T_1$  is the time for the fastest serial algorithm on a single processor.

Theoretically, speedup can never exceed the number of processors  $p$ . However, in practice, a speedup greater than  $p$  (superlinear speedup) is sometimes obtained due to either a non-optimal sequential algorithm or to hardware characteristics that put the sequential algorithm at a disadvantage. An example of this is the data for a problem might be too large to fit the main memory of a single processor and hence the secondary storage is used. This will degrade the performance of the sequential algorithm. Speedup can be used to study, in isolation, the scaling of one algorithm or benchmark on one computer. However, it must not be used to compare different algorithms on the same computer or the same algorithms on different computers because  $T_1$  may change.

To compare the performance of different algorithms for the solution of the same problem, the *temporal performance* metric can be used. Temporal performance is defined as the inverse of the execution time where the unit is solutions per second or timesteps per second [Hockney 96]. Mathematically,

$$P = \frac{1}{T} \quad (3.2.2.2)$$

The algorithm with the highest performance executes in the least time and therefore is the better algorithm. Notice that the number of floating point operations does not appear in this definition because the aim of algorithm design is not to perform the most arithmetic per second but rather it is to solve the problem in the least time, regardless of the amount of arithmetic involved.

*Efficiency* ( $E$ ) is a measure of the fraction of time for which a processor is usefully employed. It is defined as the ratio of speedup to the number of processors. Mathematically,

$$E = \frac{S}{p} \quad (3.2.2.3)$$

Ideally, speedup is equal to  $p$  and hence efficiency is 1. However, in practice, speedup is less than  $p$  and efficiency is between zero and one, depending on the degree of effectiveness with which the processors are utilised.

The *cost* of solving a problem on a parallel system is the product of run time and the number of processors used. The cost of solving a problem on a single processor is the execution time of the fastest known sequential algorithm. A parallel system is said to

be cost-optimal if the cost of solving a problem on a parallel computer is proportional to the execution time of the fastest known sequential algorithm on a single processor.

Another important aspect of parallel computation is data access, for even if there are enough processors to exploit the available parallelism, they can only do so if they can access the necessary operands, which are typically results of operations by other processors [Gentleman 78]. Either the processing shares a common memory, in which case there is a need to account for memory cycles or they have their own private memories in which case there is a need to account for data movement between the private memories.

### **3.2.3 Programming Methods**

Although the programming methods discussed in this section pertain to the specific architecture used in this research, nevertheless, the discussion is applicable to other shared memory and message-passing architectures too.

#### **3.2.3.1 Programming the Sequent Balance**

The hardware configuration and the operating system (Dyrix) of the Sequent Balance were described in Section 3.1. As for programming methods, the Sequent supports two basic kinds of parallel programming, that is multiprogramming and multitasking. Multiprogramming is an operating system feature that allows a computer to execute multiple unrelated programs concurrently. Multitasking is a programming technique that allows a single application to consist of multiple processes executing concurrently. The two basic multitasking methods are *data partitioning* and *function partitioning*. Most applications naturally lend themselves to either data partitioning or function partitioning methods.

*Data partitioning* involves creating multiple, identical processes and assigning a portion of the data to each process. This method is also called homogeneous multitasking as it involves identical tasks executed in parallel. Data partitioning is appropriate for applications that perform the same operations repeatedly on large collections of data. In programming terms, it is suitable for applications that require loops to perform calculations on arrays or matrices. Data partitioning is implemented

by executing the loop iterations in parallel. Applications such as matrix multiplication, ray tracing or signal processing adapt well to data partitioning.

*Function partitioning* involves creating multiple, identical processes and having them simultaneously perform different operations on a shared data set. It is also called heterogeneous multitasking as it involves different tasks executed in parallel. Function partitioning is suitable for applications which must perform many different operations on the same data. In programming terms, it is suitable for applications that include many unique functions. Applications such as flight simulation, program compilation and traditional process control adapt well to function partitioning.

There are applications which involve both *data* and *function partitioning*. However, most applications adapt most easily to *data partitioning*. The advantages of *data partitioning* over *function partitioning* are ease of load balancing, minimal programming effort and programs which adapt automatically to the number of processors in a system.

In Dynix, a new process is created by using a system call called fork. The new (or child) process is a duplicate copy of the old (or parent) process with the same data, register contents and program counter.

Multitasking programs include both shared and private data. Shared data is accessible by both parent and child processes. Private data is accessible only by one process.

Tasks in multitasking programs can be scheduled among processes using three types of algorithms: pre-scheduling, static scheduling or dynamic scheduling. In pre-scheduling, the task division is determined by the programmer before the program is completed. In static scheduling, tasks are scheduled by the processes at runtime but they are divided in some predetermined way while in dynamic scheduling, each process schedules its own tasks at runtime by checking a task queue.

Dynamic scheduling provides dynamic load balancing. All processes are kept working as long as there is still work to be done. Since the workload is evenly divided among the processes, the work can be completed sooner. On the other hand, static scheduling provides static load balancing. Since the division of tasks is statically determined, several processors may be idle while one processor completes its job. However, dynamic scheduling involves more overhead than static scheduling because each time a

process schedules another task for itself, it must check the task queue to make sure there is work to do and it must remove that task from the queue.

Process synchronisation on the Sequent is achieved via a lock that ensures only one process at a time can access a shared data structure.

### **3.2.3.2 Programming in PVM**

In order to develop applications for the PVM system, the traditional paradigm for programming distributed memory multiprocessors such as the nCUBE or the Intel family of multiprocessors can be followed. Significant differences exist in terms of task management, initialisation phases prior to actual computation, granularity choices and heterogeneity.

PVM supports three parallel programming models. The first and most common model for PVM applications is termed as the "crowd" computation where a collection of closely related processes, typically executing the same code, perform computations on different portions of the workload. These processes periodically exchange intermediate results. This paradigm can be further subdivided into two categories: the master-slave (or host-node) and the node-only model. In the master-slave model, there is a separate "control" program termed the master which is responsible for process spawning, initialisation, collection and display of results, and perhaps timing of functions. The slave programs perform the actual computation and they are either allocated their workloads by the master (statically or dynamically) or perform the allocation themselves. The node only model is where multiple instances of a single program execute, with one process taking over the non-computational responsibilities in addition to contributing to the computation itself.

The second paradigm is the "tree" computation. Here, processes are spawned (usually dynamically as the computation progresses) in a tree-like manner establishing a tree-like, parent-child relationship. Although this paradigm is less commonly used, it is suitable for applications where the total workload is not known "a priori", for example in branch-and-bound algorithms, alpha-beta search, and recursive divide and conquer algorithms.

The third paradigm is termed "hybrid" as it is the combination of the tree model and the crowd model. At any point during application execution, the process relationship structure may resemble an arbitrary and changing graph.

In terms of workload allocation, PVM supports both data partitioning and function partitioning. Data partitioning may be done statically, where each process knows in advance its share of workload or dynamically, where a control process allocates portions of the workload to processes as soon as they are free. The main difference between these two approaches is scheduling, the former being static scheduling where workloads for each process are fixed and the latter is dynamic scheduling where the workloads of processes varies as the computation progresses. In the general PVM environment, static scheduling is not necessarily an advantage. This is because PVM environments based on networked clusters are prone to external influences such as varying CPU speed, different memory and other system attributes.

In order to fully utilise the PVM system, two important parallelisation decisions must be made. First, is with respect to structure which concerns choosing the appropriate paradigm while the second is with regard to efficiency. Decisions with regard to efficiency when parallelising for distributed memory environments must attempt to minimise the amount of data communication and maximise computation.

### **3.3 Survey Of Parallel Algorithms For The Solution Of Linear Systems**

The fundamental importance of Linear Algebra problems in science and engineering has placed algorithms for matrix computations in the forefront of research on parallel algorithms. Linear algebra problems, including systems of linear equations, linear least squares problems, and algebraic eigenvalue problems, are fundamental to the computational solution of differential equations, optimisation problems, and the analysis of various discrete structures. Matrix algorithms have been in the vanguard of algorithm development on multiprocessors not only because they are building blocks on which many other scientific computations are based but also because they serve as realistic prototypes that present many of the fundamental challenges of parallel computation in a pure form. Thus, the development of parallel algorithms for matrix computation has received strong emphasis from researchers in parallel computing.

There is a surprisingly long tradition of research on parallel algorithms for solving computational problems in linear algebra, especially the solution of various types of linear equations. Much of the early research concentrated simply on attaining the maximum possible concurrency in solving a given type of problem and often employed highly simplified and unrealistic models of parallel computation, for example unlimited numbers of processors, no communication costs, no memory contention, no synchronisation overhead etc. This was understandable in view of the lack of widely available multiprocessors then to be able to conduct numerical experiments. The advent of VLSI in recent years has made possible the commercial development of multiprocessors having a substantial number of processors, and this in turn has given new impetus to the development and testing of parallel algorithms in realistic multiprocessor environments.

Practical methods for the solution of linear systems of the form

$$Ax=b \quad (3.3.1)$$

where  $A$  is the coefficient matrix of order  $(n \times n)$ ,  $b$  is the known column vector of right hand side values and  $x$  is the unknown column vector, fall mainly into two classes: direct and iterative. This section presents a survey of the parallelisation of both classes of methods.

### 3.3.1 Parallel Direct Linear System Solvers

This section gives a survey of research on parallel implementation of various direct methods to solve dense linear systems. In particular are covered Gaussian Elimination (GE), LU factorisation and QR decomposition.

The application of linear algebra to the solution of many numerical problems has led to an attempt to categorise the most common components in linear algebra computations which is the BLAS (Basic Linear Algebra Subprograms). Linear algebra computations can then be performed by making calls to these BLAS building blocks. BLAS is defined as relatively low level linear algebra operations which are intended as basic building blocks from which higher level linear algebra routines can then be constructed [Freeman 92].

There are three level of BLAS which are categorised according to their floating point operation counts. Level 1 BLAS are vector-vector operations and were first conceived



in the 1970s for traditional serial machines. They require only  $O(n)$  floating point operations which is too small a granularity for vector and parallel computers. Therefore, a larger grain ( $O(n^2)$  floating point operations) Level 2 BLAS were proposed.

Level 2 BLAS involve matrix-vector operations and are better suited for implementation on computers with a vector pipeline unit because of their greater granularity. Since Level 2 BLAS involve  $O(n^2)$  floating point operations on  $O(n^2)$  data items, they then have a compute/communication ratio of  $O(1)$  which makes efficient implementation on parallel computers, especially message-passing computers, difficult. This led to the introduction of Level 3 BLAS (involving matrix-matrix operation) which have a greater granularity and more importantly a compute/communication ratio of  $O(n)$ .

The ScaLAPACK software library, which was released in December of 1994, extend the LAPACK library to run scalably on distributed memory concurrent computers. LAPACK provides routines for solving systems of simultaneous linear equations, least squares solutions of linear systems of equations, eigenvalue problems and singular value problems. Like LAPACK, the ScaLAPACK routines are based on block partition algorithms in order to minimise the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are distributed memory versions of the level 2 and level 3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations.

### **3.3.1.1 Parallelisation of LU and GE**

Many algorithms for the solution of linear systems such as the GE and LU decomposition have been designed for parallel systems, both shared memory and distributed memory. The differences between the different parallel implementations are in the way the coefficient matrix  $A$  is distributed amongst the processors and the way pivoting is carried out to ensure numerical stability.

The GE and LU methods have been described in Chapter 2. In this section, their parallel implementations are described. The main concern in the parallel

implementation is to distribute the computational workload equally across the processors.

The parallel algorithm can be constructed in terms of Levels 1 and 2 BLAS or by writing the parallel code from scratch. A good discussion of the construction of the algorithms in terms of the BLAS routines can be found in [Freeman 92].

The first stage of GE is the forward elimination whose sequential algorithm is given in Algorithm 3.3.1.1.1. A study of the algorithm's data dependencies revealed that the  $k$  loop must be done in sequential order. However, the middle and innermost loop can be done in parallel. This means that once a pivot row  $k$  is determined, the modification of all unmarked rows may occur simultaneously. Within each row, once the multiplier  $a[i][k]/a[k][k]$  has been computed, modifications to elements  $k+1$  to  $n$  of each row can proceed simultaneously.

**Algorithm 3.3.1.1.1 Sequential forward elimination algorithm**

```

for k=1 to n-1 do
    for i=k+1 to n do
         $a[i][k]=a[i][k]/a[k][k]$ 
        for j=k+1 to n
             $a[i][j]=a[i][j]-a[i][k]*a[k][j]$ 
        end for j
    end for i
end for k

```

On a shared memory machine, independent loop iterations indexed by  $i$  can be assigned to the multiple processors either by partitioning it into a contiguous group of rows (known as block partitioning) or by interleaving the rows (called wrapped, interleaved storage). If  $n=kp$ , and in the block row partitioning, the first  $k$  rows of  $A$  are assigned to processor 1, the second  $k$  rows to processor 2 and so on. For the wrapped interleaved partitioning, rows 1,  $p+1$ ,  $2p+1$ , ... are stored in processor 1; rows 2,  $p+2$ ,  $2p+2$ , ... in processor 2 and so on. Figure 3.3.1.1.1 illustrates an example of block data partitioning for a  $20 \times 20$  matrix while Figure 3.3.1.1.2 shows an example of wrapped interleaved partitioning for a  $20 \times 20$  matrix.

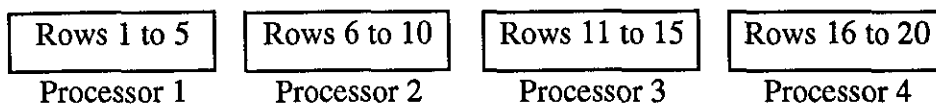


Figure 3.3.1.1.1 A 20X20 matrix distributed amongst 4 processors as a contiguous block

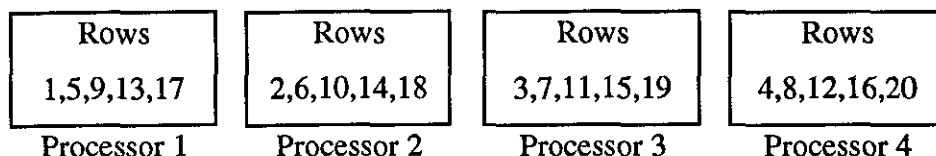


Figure 3.3.1.1.2 A 20X20 matrix distributed amongst 4 processors in an interleaved way

On a message-passing architecture where each processor has its own private memory, the main issue to consider is the distribution of rows of coefficient matrix  $A$  to the memories. Again, the rows can be distributed as a contiguous block or in an interleaved fashion. In order to determine the pivot row, the processors must interact with each other. Once the pivot row has been determined, the processor that owns the pivot row must broadcast its elements to the other processors so that they can update the unmarked rows that they have in their memories.

If an algorithm partitions the matrix  $A$  by row, it is known as a row oriented algorithm. An alternative distribution of data is assigning to each processor an interleaved group of columns (known as column oriented algorithm) and the vector  $\mathbf{b}$ . This way, the processors need not interact to determine the pivot row. Instead, each processor must broadcast elements of the column and the identity of the pivot row to the other processors.

Another important strategy in designing algorithms for message-passing architecture is to maximise grain size and minimise communication among the processors. If the messages are small, it makes sense to combine messages to be sent to a certain processor in order to reduce message passing overhead. For example, in the row oriented version of parallel GE, the processor controlling the pivot row must send the elements of the pivot row to other processors. It makes more sense to broadcast the entire row rather than sending one element at a time.

Likewise, in the column oriented parallel GE, the processor controlling a column must broadcast the entire column and the identity of the pivot row together to the other processors.

It is obvious that the difference between the row oriented parallel GE and the column oriented parallel GE lies in the communication/computation ratio. In the row-oriented version, processors work together to determine the pivot row. For a given system of size  $n$  and  $p$  processors, each processor examines at least  $n/p$  values but once a processor has determined the local maximum within its subset of data, it must communicate with the other processors to determine the global maximum. In iteration  $i$  of the column oriented algorithm, each processor must perform  $(n-i)$  comparisons but no communication is required. Both algorithms require a broadcast step after the pivot row has been found.

The second stage of GE is solving a triangular system known as back substitution. One possible algorithm is the row oriented algorithm shown in Algorithm 3.3.1.1.2.

**Algorithm 3.3.1.1.2 Sequential row oriented algorithm for back substitution**

```

x[n]=b[n]/a[n][n]
for i=n-1 downto 1
  x[i]=b[i]
  for j=i+1 to n do
    x[i]=x[i]-a[i][j]*x[j]
  end for j
  x[i]=x[i]/a[i][i]
end for i

```

The outer loop  $i$  controls computation of the solution vector. For any  $r$ ,  $r < p$ ,  $x_p$  must be available before  $x_r$  can be computed. Therefore, this loop cannot be executed in parallel. The inner loop  $j$  can be parallelised but it has synchronisation restrictions.

Another alternative is the column oriented back substitution shown in Algorithm 3.3.1.1.3.

**Algorithm 3.3.1.1.3 Sequential column oriented algorithm for back substitution**

```

for k=n downto 1
  x[k]=b[k]/a[k][k]
  for i=1 to k-1
    x[i]=x[i]-x[k]*a[i][k]
  end for i
end for k

```



Implementation of the back substitution algorithm on a shared memory system involves assigning independent loop iterations to the different processors. On a distributed memory implementation, the decisive factor is again, the distribution of the matrix  $A$  to the processors. If the matrix  $A$  had been distributed by columns, then the column oriented back substitution would be suitable. Otherwise, if it had been distributed by rows, then the row oriented algorithm would be suitable. For both methods, once a solution is obtained, the processor which owned the row for which the solution was obtained will have to broadcast the result to the other processors. This will enable the other processors to use this new value in the substitution process of the rows that it has in its own memory.

In GE, the forward elimination process is applied to the coefficient matrix and the right hand side vector simultaneously. If we have several right hand sides, then each stage of the forward elimination must be applied to each right hand side. A better way is to separate the modification of the coefficient matrix from that of the right hand side. This is achieved by first applying the forward elimination to the coefficient matrix. Then, after this stage is completed, the forward elimination can then be applied to the right hand side. This method is known as the LU factorisation.

As mentioned in Chapter 2, LU factorisation is a method which decomposes the coefficient matrix into two factors  $L$  and  $U$  where  $L$  is the lower triangular matrix and  $U$  is the upper triangular matrix. The solution phase consists of the forward substitution on the  $L$  matrix followed by a back substitution on the  $U$  matrix.

The sequential algorithm for the factorisation stage of LU is shown in Algorithm 3.3.1.1.4 while the sequential algorithm for the forward substitution stage of the solution process is shown in Algorithm 3.3.1.1.5. The algorithms assume that the right hand side vector  $\mathbf{b}$  is augmented to the coefficient matrix  $A$  and the loop indices begin from 0 to match the code which was written in C.

The factorisation stage can be parallelised by partitioning the inner loop  $i$  to the available processes. The outer loop  $k$ , however, has to be processed in sequential order. The forward substitution can be parallelised in a similar way as the back substitution.

Algorithm 3.3.1.1.4 Sequential algorithm for factorisation stage in LU

```
for k = 0 to n-2
  for i=k+1 to n-1
    m=a[i,k]/a[k,k]
    a[i,k] = m
    for jr=k+1 to n-1
      a[i,jr] = a[i,jr] - m * a[k,jr]
    end for jr
  end for i
end for k
```

Algorithm 3.3.1.1.5 Sequential algorithm for forward substitution

```
for nv = 1 to n-1
  for j = 0 to nv -1
    a[nv,n] = a[nv,n] - a[nv,j] * a[j,n]
  end for j
end for nv
```

The incorporation of partial pivoting to maintain numerical stability introduces additional considerations into the data partitioning issues. If  $A$  is partitioned by column wrapping, then the search for the pivot element takes place in a single processor. Once the pivot row is determined, it must be transmitted to the other processors. In row wrapped partitioning, the search for the maximal element in the current column must take place across all the processors. Once again, when the pivot row is determined, it must be broadcast to the other processors.

### 3.3.1.2 Parallelisation of the QR method

The difficulties with implementing interchange strategies on parallel architectures suggest that orthogonal reductions to triangular form may have advantages.

It was observed by [Gentleman 75] that the orthogonal reduction to triangular form by Givens rotation or Householder transformation has a certain natural parallelism. In general, the Givens process works by taking linear combinations of rows of the matrix, chosen to make the new elements below the diagonal zero. There are several possible orderings of the transformations and choices of row pairs which will also produce an upper triangular matrix. Any two rows may be chosen to produce a required zero as long as the previous elements of the two rows have already been made zero. The QR method has been described in detail in Chapter 2. The sequential algorithm for Givens QR rotation is shown in Algorithm 3.3.1.2. Again, the loop index begins from 0 to

match the code which has been written in C and the right hand side vector **b** is augmented to the coefficient matrix **A**.

The basic principle underlying the parallelisation of Givens QR decomposition is that independent rotations, that is rotations applied to different rows, are allocated to different processors. It is assumed that each processor can read two rows from memory and apply a plane rotation to these so as to introduce a zero element and return the modified rows to memory. Several parallel algorithms have been proposed along this line where the main aim is to perform the decomposition in the least number of steps.

### Algorithm 3.3.1.2 Sequential QR Decomposition

```

for k = 0 to n-2
  for i = k+1 to n-1
    { annihilate A(i,k) }
    d = sqrt(A(i,i)*A(i,i) + A(i-1,i)*A(i-1,i))
    cos_t = A(i,i)/d
    sin_t = A(i-1,i)/d
    A(i,i) = d
    for j = 1 to n
      s1 = cos_t * A(i,j) + sin_t * A(i-1,j)
      s2 = cos_t * A(i-1,j) - sin_t * A(i,j)
      A(i,j) = s1
      A(i-1,j) = s2
    end for j
  end for i
end for k

```

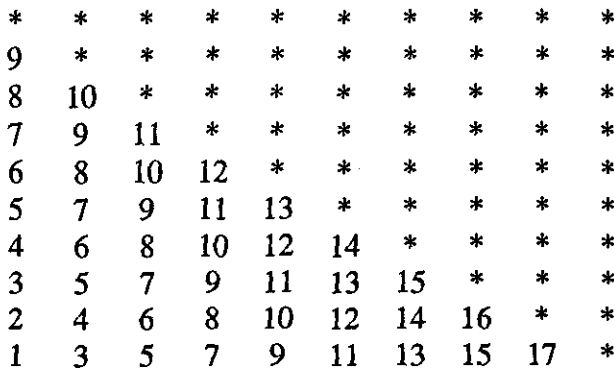


Figure 3.3.1.2.1 Annihilation pattern for parallel Givens rotation

The parallelisation of Givens rotation employed in this thesis is based on [Freeman 92] where the main aim is to perform several Givens rotations concurrently. For example, the first step uses a rotation in the  $(n-1,n)$  plane to annihilate the  $(n,1)$  element. The



second step uses a rotation in the  $(n-2, n-1)$  plane to annihilate the  $(n-1, 1)$  element. On the third step, two rotations can be performed simultaneously, i.e. rotations in the  $(n-3, n-2)$  and  $(n-1, n)$  planes to zero out the  $(n-2, 1)$  and  $(n, 2)$  elements. The order in which the elements of a matrix are zeroed out is called the annihilation pattern. Figure 3.3.1.2.1 shows the annihilation pattern for the parallel Givens rotation employed in this thesis. For simplicity, the annihilation pattern is given for a  $10 \times 10$  matrix. The integers indicate the steps at which the given elements are annihilated.

After performing the QR factorisation of a matrix, the solution of the system of equations can be obtained through the back substitution process.

### 3.3.1.3 Other related work

Quite a number of earlier researchers have investigated the parallelisation of GE, LU and QR and their implementation on parallel computers. Following is a brief account of some of the earlier work.

Sameh and Kuck in (Sameh 77) conducted a survey of direct parallel algorithms for solving systems of linear equations. The survey included triangular, dense and tridiagonal systems. Of relevance to this thesis is the solution of dense systems, where two parallel methods were surveyed. Emphasis of the survey was on the speedup of the parallel algorithm over the corresponding sequential algorithm. The first algorithm surveyed was the LU factorisation via Gaussian elimination without pivoting. It was assumed that  $(n-1)^2$  processors were available, hence the factorisation required  $3(n-1)$  steps. When pivoting was incorporated, Gaussian elimination then requires  $O(n \log n)$  steps. The second algorithm was parallel Givens reduction which required  $O(n)$  steps with  $O(n^2)$  processors.

Lord et al. [Lord 83] re-examined the GE algorithm, assuming  $n/2$  processors, on the HEP computer with 8 processors. They showed that the algorithm then required  $n^2-1$  steps.

Darmohray and Brooks III [Darmohray 87] investigated the performance of parallel Gauss and parallel Gauss-Jordan elimination algorithms on the Cerberus multiprocessor simulator, a simulator for a scalable shared-memory multiprocessor with fully pipelined functional units. The parallel implementation of their algorithms made extensive use of barrier synchronisation. The two barrier implementations used

were the butterfly barrier which is a software technique, and a hardware barrier. Their work was an attempt to compare two parallel algorithms that solved the same problem. A better speedup in one algorithm does not imply better performance. Absolute running times of the algorithms must also be compared. They have shown that an algorithm which requires more operations but synchronises less frequently and is more load-balanced can run faster than an algorithm with fewer operations for certain choices of problem size and numbers of processors.

The era of the hypercube saw the trend of parallel linear system solvers moving towards a message-passing implementation. Although the work in this thesis involves message-passing on workstation clusters, work pertaining to the hypercube is relevant as it has message-passing in common between them. The only feature of the hypercube algorithm that are specifically dependent on the structure of the hypercube are the details of the broadcast algorithm and the use of a globally connected host. Thus, the same general algorithm, should work on any message-passing, distributed memory multiprocessor that support broadcasting and has a globally connected host.

[Geist 85] developed a message-passing algorithm to form the LU factors of general non-singular matrices on a hypercube multiprocessor. Partial pivoting was performed to ensure numerical stability. He proposed a new algorithm that best masked the work of pivoting by letting the host processor which remained idle during factorisation to determine the next pivot row while the nodes continue with the factorisation. Thus, the work of pivoting is masked. The only degradation caused by pivoting is due to load imbalance rather than any additional work or communication. The load imbalance produced by random pivoting caused 5% -14% increase in execution time. The mapping of rows of matrix onto the processors seemed to give the highest efficiencies and the best overall load balance.

Subsequently, Geist and Heath [Geist 87] discovered that the performance of the same algorithm on an Intel Hypercube can be quite different from the simulator results. In particular, because the host had limited buffer space, and the host-to-node communication is sequential and substantially slower than the node-to-node communication, it is no longer practical to involve the host in the process of pivot selection. In the solution proposed by Geist and Heath [Geist 87], a node processor is

designated as the manager with extra responsibility of determining the pivot and informing all other nodes of the pivot row number.

A potential weakness of the implementations of [Geist 85] and [Geist 87], is that they did not deal with the possible load unbalancing of the computation that could be caused by an unfortunate sequence of pivot choices. Although Geist and Heath reported in [Geist 87] that their experience suggested that this imbalance is quite low, about 5-15% in execution time of the factorisation phase, there are examples where it is much higher [Chu 87]

Chu and George in [Chu 87] modified Geist and Heath's [Geist 87] general approach to incorporate dynamic load balancing. They proposed that the pivot row is exchanged with a row in a designated processor so that all the processors have the same number of equations still to be factored. The scheme had shown to be effective in maintaining a balanced load distribution throughout the factoring process with very modest communication cost and the overall execution time was not very much affected for the chosen test problems..

Another work on parallel algorithms for the solution of linear systems using a variant of Gaussian Elimination with partial pivoting on the hypercube was done by [Chamberlain 87]. He proposed an algorithm where the matrix was stored by rows and the maximal element in a row was taken as a pivot. This is known as column pivoting. Columns were then interchanged to put the maximal element on the diagonal. No communication was required to determine the pivot element, hence the number of messages was reduced. The determination of the pivot was done sequentially by one processor. The only communication was the distribution of the pivot rows. Elimination and distribution of the pivot rows were overlapped, hence reducing the time a processor spends waiting for pivot rows. Numerical results obtained by Chamberlain showed that the method provided substantial gains over the normal row pivoting.

Dekker et al [Dekker 94] provided a survey of research on the parallelisation of GE with pivoting on supercomputers and distributed memory systems. They have also indicated the main BLAS operations used by GE algorithms. The implementation on the distributed memory system (which was a Meiko computing surface with 64 processors) used a threshold pivoting strategy which yielded a good compromise between numerical stability and the reduction of data traffic between local memories,

needed for row interchanges. Threshold pivoting means that instead of an element of largest size, a suitable element which is at least a given fraction of the largest element is selected as pivot.

Computing the orthogonal decomposition of an  $m \times n$  matrix is a classical problem in scientific computing. Two well known methods are available for solving such a problem, that is the Householder reduction and the Givens rotation. The parallelisation of Givens method gives rise to very interesting algorithmic problems.

An implementation of the parallel Givens method proposed in [Wright 91] divided the computation into a number of stages. In the first stage, roughly half of the first column is made zero by rotations which are all independent by taking the row  $j$  with row  $n-j+1$  for  $j=1, \dots, \lfloor n/2 \rfloor$ . In the second stage, a similar scheme is applied to reduce the remaining non-zeros in the first column and start on the second column. The independent modifications in each stage can be assigned to parallel tasks in either a predetermined way or dynamically. The end of each stage requires synchronisation as the next stage cannot start until the previous one is complete.

Lord et al. [Lord 83] discussed Givens transformations for full systems, motivated by multiprocessor systems and the Delnecor HEP in particular. As opposed to the annihilation pattern of Sameh and Kuck (Figure 3.3.1.3.1), which assumed the use of  $O(n^2)$  processors, they assumed that  $p \leq O(n/2)$  and gave 2 possible annihilation patterns (Figure 3.3.1.3.2).

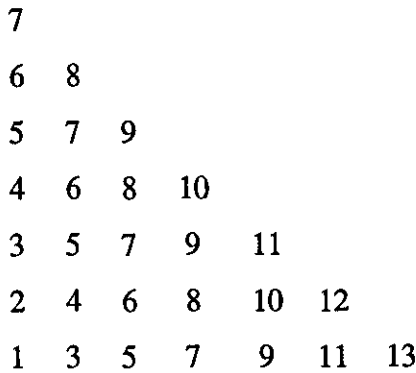


Figure 3.3.1.3.1 Annihilation pattern of Sameh and Kuck

For both figures 3.3.1.3.1 and 3.3.1.3.2, an integer  $r$  in position  $(i,k)$  means that the element  $a(i,k)$  has been annihilated in the  $r^{\text{th}}$  step.

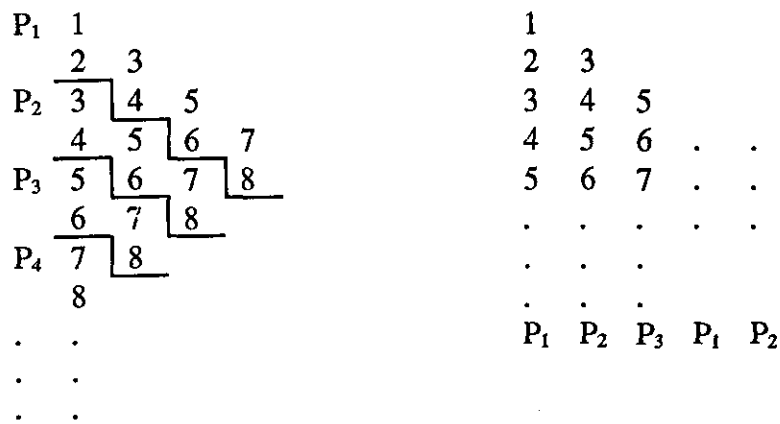


Figure 3.3.1.3.2 Annihilation pattern of Lord et al

Modi and Clarke [Modi 84] have introduced the greedy algorithm that performs simultaneously, at each step, all disjoint rotations. Elements in one column are being annihilated from bottom to top and those in each row from left to right. Figure 3.3.1.3.3 illustrates the annihilation pattern for an 8x8 matrix for the greedy algorithm.

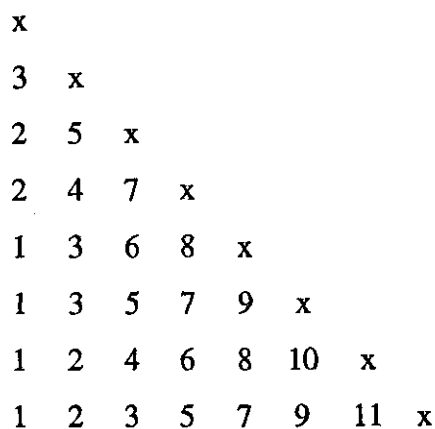


Figure 3.3.1.3.3 Annihilation pattern for the greedy algorithm for an 8x8 matrix

Cosnard et. al in [Cosnard 86] have proved that for any value  $m$  and  $n$  for an  $m \times n$  matrix with  $m \geq n$  and for  $p = \lfloor m/2 \rfloor$ , the greedy algorithm is optimal.

In [Modi 88], another class of algorithm called the Fibonacci scheme and its performance is discussed. However, it was observed that the Fibonacci scheme seemed to be less efficient than the greedy algorithm.

### 3.3.2 Parallel iterative linear system solvers

The methods discussed in the previous section were direct methods where the exact solution is computed in a determinable number of arithmetic operations. These methods are suitable for dense systems. However, they may be impractical for large sparse systems for the following two reasons:

- a. The methods require  $O(n^3)$  operations and for large values of  $n$ , the computational cost may be high.
- b. It is a waste of storage to store zero coefficients and also a waste of computing time to perform arithmetic operations on zero elements.

In this section, the parallelisation of iterative methods is discussed.

A lot of research has been done in the parallelisation of sequential iterative methods for solving the sets of linear equations generated from the discretisation of elliptic partial differential equations. The work on parallel iterative methods in this thesis is geared towards multiprocessor machines. This section gives a survey of research on parallel iterative solvers. In particular are covered the Jacobi, Gauss-Seidel, Jacobi Overrelaxation (JOR) and Successive Overrelaxation (SOR) iterative methods.

Indeed the Jacobi method has been recognised as an ideal algorithm for parallelisation as the calculation of each component  $x_i^{(k+1)}$ ,  $i=1,2,\dots,n$  can be done independently. On a shared-memory computer, the loop which calculates the updated components is distributed amongst the processors, either in a contiguous block or in a wrapped, interleaved manner. On a distributed memory machine, the rows of the coefficient matrix  $A$  are distributed to the processors and computation in the separate processors can proceed concurrently. At the end of each iteration, each processor broadcasts the components of  $x^{(k+1)}$  that it has computed to the other processors.

The parallel Jacobi scheme, however, suffers from the same drawback as the sequential Jacobi, which is very slow convergence rates. Whereas the Jacobi iteration is often cited as a "perfect" parallel algorithm, the Gauss-Seidel and SOR iterations are considered to be the opposite. The usual serial code for Gauss-Seidel would have new values at each point to replace the old as soon as they are updated. It is this recursive process that is not amenable to parallelism. Several different parallel modifications have been performed on the classical Gauss-Seidel and SOR. One set of methods involve the multicolouring of grid elements and the update of those elements of like

colour. The simplest of these methods is the Red-Black method, in which two colours are assigned to the grid elements in a checkerboard manner. Then, all grid elements of one colour can be updated in a Jacobi-like sweep in odd-numbered passes, while those of the second colour are updated in even numbered passes. Work has been done on this by [Evans 84]. Implementation of the Jacobi or SOR iterations on multiple processors require a suitable distribution of the work amongst the processors so as to minimise the processor idleness. In order to carry out these iterations in their mathematical form, we also need to ensure that the processors are synchronised before the beginning of each iteration, or in the case of the multicolour SOR method, before the beginning of each Jacobi sweep. This synchronisation can be carried out in a number of ways but, in essence, it requires that each processor wait after completion of its part of the computation until all processors have completed their work and the next iteration can begin. This adds two forms of overhead to the computation: one is the work required to verify that every processor is ready for the next iteration, and the other is the idle time that some processors may experience while waiting for all processors to complete their tasks.

Another parallel variation of SOR has been developed by Patel and Jordan [Patel 84] where each processor is assigned the task of updating one row of grid points. Since SOR re-uses updated values as soon as they are available, each processor must wait for the previous processor's iterative updates before it can begin updating in its row. Synchronisation between the processors was controlled by full/empty flags assigned to each memory location. These flags were built into the Heterogeneous Element Processor (HEP) on which their algorithm was developed.

Bonomo and Dyksen [Bonomo 89] proposed pipeline iterative techniques as an effective means to parallelise basic serial iterative methods for the solution of linear systems.

An alternative that has special appeal in the case of the Jacobi or SOR iterations is to let the processors run asynchronously. This idea goes at least to the chaotic relaxation methods of Chazan and Miranker [Chazan 69] and has been studied in some detail by [Baudet 78], following work of [Kung 76]. No attempt is made to synchronise each iterative sweep. This method avoids two problems which are inherent in any algorithm that attempts to synchronise sweeps: first is the extra computational work that must be

performed by each processor after each sweep to verify when it can begin the next sweep, and second is the extra time incurred by a processor in waiting for all other processors to finish a sweep. However, the asynchrony makes the analysis of the algorithms and proofs of its convergence rate difficult.

Parallel versions of the Jacobi and Successive Overrelaxation methods to solve systems of linear equations on a network of transputers were implemented by Cunha and Hopkins [Cunha 91]. The JOR method was parallelised by distributing the data in a row wise fashion among the processors. The parallelisation of SOR was less obvious due to data-dependency issues. The SOR method was rewritten to obtain a suitable iteration to parallelise. An adaptive version of the parallel SOR was also developed. All the methods were tested on two model problems, one dense and the other sparse. The JOR method showed a high degree of efficiency for dense systems. The SOR methods were far more effective on sparse systems.



## Chapter 4

### Parallel Implicit Elimination (PIE) and Quadrant Interlocking Factorisation (QIF) on Shared Memory Architecture

The solution of systems of linear equations of the form

$$Ax = b \quad (4.1)$$

where  $A$  is an  $n \times n$  non-singular matrix,  $x$  is the unknown vector and  $b$  is the right hand side vector, is probably one of the activities in scientific computing that uses the most computer time. This is due to the fact that the solution of systems of linear equations occur in most real life applications.

The well-known Gaussian elimination (GE) method and the LU factorisation are, to date, perhaps the most commonly used algorithms to solve linear systems on sequential computers. Since the advent of parallel computers, the GE and LU methods has been parallelised to run on parallel computers. This technique of exploiting parallelism is known as "vectorising existing software". Both GE and LU are essentially algorithms in which elimination and factorisation are done serially. Evans introduced PIE [Evans 93b] and QIF [Evans 79] which are more suitable for parallel computation and are aimed at a parallel machine from the outset. Most of the earlier studies on QIF have been done on hypothetical parallel computers [Evans 79] and [Shanechi 80]. Later Evans and Bekakos [Evans 88] introduced a parallel version of this method on a wavefront computer. The first attempt of comparing these algorithms with their classical methods equivalence, namely GE and LU respectively was made by Yalamov [Yalamov 95]. In his work, he compared QIF to LU and has obtained a 20% gain in execution time. However, there was no evidence as to what contributed to the gain in execution time.

The work in this chapter involves analysing the performance of PIE and QIF on a shared memory parallel computer and comparing their performance with GE and LU. An attempt is also made to justify the results obtained in the investigation.

Sections 4.1 and 4.2 presents the PIE and QIF methods respectively. Partial pivoting strategies for both PIE and QIF are covered in section 4.3. In section 4.4, both the sequential and parallel algorithms for PIE and QIF are outlined. The computational complexity and shared memory access analysis of PIE and QIF are given in section 4.5

while the results of PIE and QIF implemented on the Sequent Balance, a shared memory parallel computer, are given in section 4.6. A summary of the chapter is given in section 4.7

#### 4.1 Parallel Implicit Elimination method (PIE)

PIE is a scheme that simultaneously eliminates two elements at a time, instead of just one as in GE. The basis of PIE is to transform the coefficient matrix  $A$  to that of butterfly form as illustrated by (4.1.1) when  $n$  is odd, and (4.1.2) when  $n$  is even. This transformation is known as parallel elimination.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n-1} & a_{1n} \\ 0 & a_{22}^{(1)} & \cdots & a_{2,n-1}^{(1)} & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & a_{n-1,2}^{(1)} & \ddots & a_{n-1,n-1}^{(1)} & 0 \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix} \quad (4.1.1)$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & \cdots & a_{1,n-1} & a_{1n} \\ 0 & a_{22}^{(1)} & \cdots & \cdots & \cdots & a_{2,n-1}^{(1)} & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & a_{n-1,2}^{(1)} & \cdots & \cdots & \cdots & a_{n-1,n-1}^{(1)} & 0 \\ a_{n1} & a_{n2} & \cdots & \cdots & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix} \quad (4.1.2)$$

The solution stage, known as bi-directional solution, begins at the central value  $x_{\frac{n+1}{2}}$  if  $n$  is odd and the two values  $x_{\frac{n}{2}}$  and  $x_{\frac{n+2}{2}}$  if  $n$  is even. It involves backward and forward substitution processes concurrently solving  $(2 \times 2)$  systems to evaluate the values  $x_i$  where  $i=1, 2, \dots, n$ .

### Parallel Elimination Procedure

Consider the following shorthand notation for the coefficient matrix  $A$ .

$$A = \begin{bmatrix} a_{11} & \underline{a_{1j}^T} & a_{1n} \\ \underline{a_{i1}} & \underline{A_{ij}} & \underline{a_{in}} \\ a_{n1} & \underline{a_{nj}^T} & a_{nn} \end{bmatrix}, \quad i, j = 2(1)n-1 \quad (4.1.3)$$

Consider the transformation matrix  $W_j$  consisting of the vector elements  $\omega_{i1}$  and  $\omega_{in}$ ,  $i=2, 3, \dots, n-1$  i.e.

$$W_j = \begin{bmatrix} 1 & 0 & 0 \\ -\omega_{i1} & I_{n-2} & -\omega_{in} \\ 0 & 0 & 1 \end{bmatrix} \quad (4.1.4)$$

Elimination is achieved by taking the product

$$W_j A = \begin{bmatrix} a_{11} & \underline{a_{1j}^T} & a_{1n} \\ -a_{11}\omega_{i1} + I_{n-2}\underline{a_{i1}} - a_{n1}\omega_{in} & -\omega_{i1}\underline{a_{ij}^T} + \underline{A_{ij}} - \omega_{in}\underline{a_{nj}^T} & -a_{1n}\omega_{i1} + I_{n-2}\underline{a_{in}} - a_{nn}\omega_{in} \\ a_{n1} & \underline{a_{nj}^T} & a_{nn} \end{bmatrix} \quad (4.1.5)$$

and, choosing the values of vectors  $\omega_{i1}$  and  $\omega_{in}$  such that,

$$\begin{aligned} a_{11}\omega_{i1} + a_{n1}\omega_{in} &= \underline{a_{i1}}, \\ a_{1n}\omega_{i1} + a_{nn}\omega_{in} &= \underline{a_{in}}, \end{aligned} \quad (4.1.6)$$

for  $i=2, 3, \dots, n-1$ , to give,

$$\begin{aligned} \omega_{i1} &= \frac{-a_{n1}\underline{a_{in}} + a_{nn}\underline{a_{i1}}}{a_{11}a_{nn} - a_{1n}a_{n1}} \\ \omega_{in} &= \frac{a_{11}\underline{a_{in}} - a_{1n}\underline{a_{i1}}}{a_{11}a_{nn} - a_{1n}a_{n1}} \end{aligned} \quad (4.1.7)$$

Hence, the determination of  $\omega_{i1}$  and  $\omega_{in}$  requires the solution of (2x2) sets of equations which can be solved by any one of the following three methods:

#### a) Cramer's Rule

This well-known formula is not usually recommended for the computer solution of large linear systems because of the excessive computational requirements it requires. However, for the solution of the (2x2) systems which occur abundantly in the PIE scheme, it is quite adequate provided the system is not too ill-conditioned.

To solve the system in (4.1.6) the following quantities are evaluated:

$$\underline{\omega}_{i1} = \frac{(-a_{n1} \underline{a}_{in} + a_{nn} \underline{a}_{i1})}{(a_{11} a_{nn} - a_{1n} a_{n1})} \quad (4.1.8)$$

$$\underline{\omega}_{in} = \frac{(a_{11} \underline{a}_{in} - a_{1n} \underline{a}_{i1})}{(a_{11} a_{nn} - a_{1n} a_{n1})}$$

b) Elimination

Apply a column pivoting strategy to the (2x2) system to improve stability. Then, proceed as follows:

1. Compute the multiplier

$$\alpha = a_{1n}/a_{11},$$

2. Compute the quantities

$$(a_{nn} - \alpha a_{n1}) \text{ and } (\underline{a}_{in} - \alpha \underline{a}_{i1}), \quad (4.1.9)$$

3. Determine

$$\underline{\omega}_{in} = (\underline{a}_{in} - \alpha \underline{a}_{i1}) / (a_{nn} - \alpha a_{n1}),$$

4. Evaluate

$$\underline{\omega}_{i1} = (\underline{a}_{i1} - a_{n1} \underline{\omega}_{in})/a_{11}.$$

c) Symmetric Elimination

By applying the column pivoting strategy to the (2x2) system the algorithm can be described as follows:

1. Compute the multipliers,

$$\alpha = -a_{1n}/a_{11}, \quad \gamma = -a_{n1}/a_{nn},$$

2. Compute

$$(a_{nn} - \alpha a_{n1}), (a_{11} - \gamma a_{1n}), (\underline{a}_{in} - \alpha \underline{a}_{i1}) \text{ and } (\underline{a}_{i1} - \gamma \underline{a}_{in}), \quad (4.1.10)$$

3. Evaluate

$$\underline{\omega}_{in} = (\underline{a}_{in} - \alpha \underline{a}_{i1}) / (a_{nn} - \alpha a_{n1}), \quad \underline{\omega}_{i1} = (\underline{a}_{i1} - \gamma \underline{a}_{in}) / (a_{11} - \gamma a_{1n}).$$

In the first step of the elimination, the elements  $a_{21}$  and  $a_{2n}$  will be eliminated by solving for  $\omega_{i1}$  and  $\omega_{in}$  where  $i=2$ . Once  $\omega_{21}$  and  $\omega_{2n}$  are obtained, the values  $a_{22}, a_{23}, \dots, a_{2,n-1}$  will be updated. After this initial step, the matrix (4.1.11) is obtained.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1, n-1} & a_{1n} \\ 0 & \overset{(1)}{a_{22}} & \dots & \overset{(1)}{a_{2, n-1}} & 0 \\ a_{31} & a_{32} & \dots & a_{3, n-1} & a_{3n} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \dots & a_{n-1, n-1} & a_{n-1, n} \\ a_{n1} & a_{n2} & \dots & a_{n, n-1} & a_{nn} \end{bmatrix} \quad (4.1.11)$$

Repeating the elimination steps for  $i=3,4,\dots,n-1$  will result in the reduced matrix (4.1.12).

$$\left[ \begin{array}{c|c|c|c|c} a_{11} & a_{12} & \dots & a_{1, n-1} & a_{1n} \\ 0 & \overset{(1)}{a_{22}} & \dots & \overset{(1)}{a_{2, n-1}} & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & \overset{(1)}{a_{n-1,2}} & \dots & \overset{(1)}{a_{n-1, n-1}} & 0 \\ \hline a_{n1} & a_{n2} & \dots & a_{n, n-1} & a_{nn} \end{array} \right] \quad (4.1.12)$$

Equations 2, . . . ,  $n-1$  are then treated as a new subsystem and the parallel elimination process is repeated on the submatrix outlined in (4.1.12). The final transformation yields  $WA = \prod W_i A = Z$ .

To obtain the eliminated matrix as in (4.1.1) for  $n$  odd and (4.1.2) for  $n$  even, the elimination process is repeated  $(n-1)/2$  times.

#### Bi-directional Solution Procedure

For the solution process of PIE, cases of odd and even-sized matrices are considered separately. When  $n$  is odd, the bi-directional solution process begins with the matrix as in (4.1.1) and the solution begins from the centre with

$$a_{\frac{n+1}{2}, \frac{n+1}{2}} x_{\frac{n+1}{2}} = b_{\frac{n+1}{2}} \quad (4.1.13)$$

Solving for  $x_{\frac{n+1}{2}}$  in (4.1.13) will result in

$$x_{\frac{n+1}{2}} = \frac{b_{\frac{n+1}{2}}}{a_{\frac{n+1}{2}, \frac{n+1}{2}}} \quad (4.1.14)$$

This value of  $x_{\frac{n+1}{2}}$  is substituted up into equation  $(n-1)/2$  and substituted down into equation  $(n+3)/2$ . Next the following two equations will be solved:

$$\begin{aligned}
 a_{\frac{n-1}{2}, \frac{n-1}{2}} x_{\frac{n-1}{2}} + a_{\frac{n-1}{2}, \frac{n+3}{2}} x_{\frac{n+3}{2}} &= b_{\frac{n-1}{2}} - a_{\frac{n-1}{2}, \frac{n+1}{2}} x_{\frac{n+1}{2}} \\
 a_{\frac{n+3}{2}, \frac{n-1}{2}} x_{\frac{n-1}{2}} + a_{\frac{n+3}{2}, \frac{n+3}{2}} x_{\frac{n+3}{2}} &= b_{\frac{n+3}{2}} - a_{\frac{n+3}{2}, \frac{n+1}{2}} x_{\frac{n+1}{2}}
 \end{aligned}
 \quad (4.1.15)$$

to yield the values for  $x_{\frac{n-1}{2}}$  and  $x_{\frac{n+3}{2}}$ .

This procedure continues outwards from the centre forwards and backwards i.e. bi-directionally until the final equations to be solved are

$$\begin{aligned}
 a_{11}x_1 + a_{1n}x_n &= b_1 - \sum_{i=2}^{n-1} a_{1i}x_i \\
 a_{n1}x_1 + a_{nn}x_n &= b_n - \sum_{i=2}^{n-1} a_{ni}x_i
 \end{aligned}
 \quad (4.1.16)$$

When  $n$  is even, the bi-directional solution process begins with (4.1.2) and the initial stage (4.1.10) is omitted. The remaining steps are similar to the case when  $n$  is even.

Finally, a comparison of the PIE and GE schemes for  $n=6$  can be made diagrammatically as shown in figures 4.1.1, 4.1.2, 4.1.3, 4.1.4 and 4.1.5. The shaded part in the diagram indicates the sub-matrix that needs to be updated for both the GE and PIE methods.

Step 1:

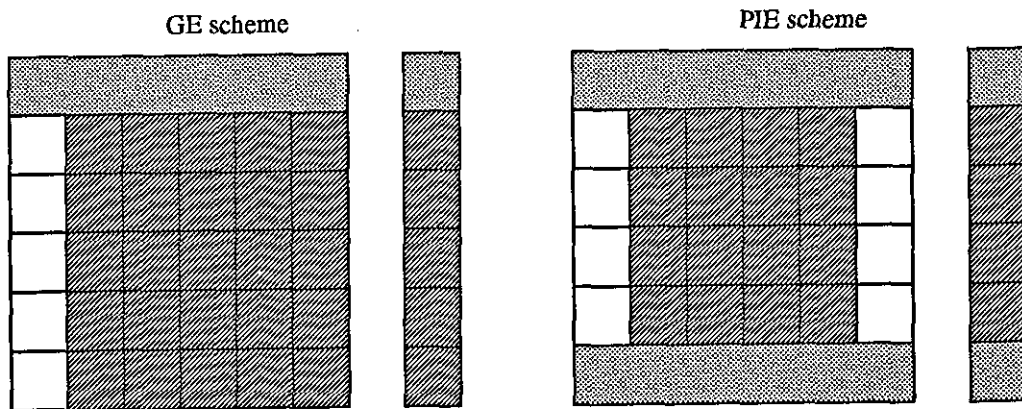


Figure 4.1.1 - Column 1 of GE being eliminated and columns 1 and 6 of PIE eliminated.

Step 2:

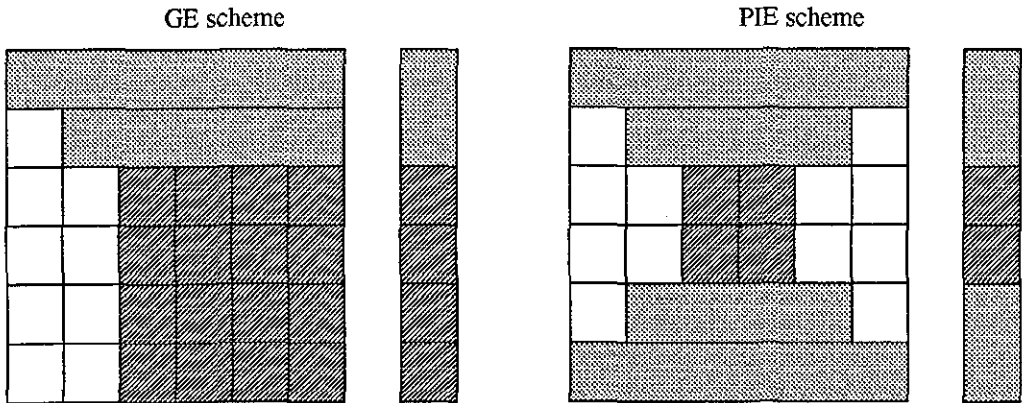


Figure 4.1.2 - Column 2 of GE being eliminated and columns 2 and 5 of PIE eliminated. PIE method is completed now.

By the second step, the PIE method is already completed while the GE method has three more steps to go. The resulting reduced matrix for PIE depicted in Figure 4.1.3 can now be solved for by a bi-directional solution process.

Step 3:

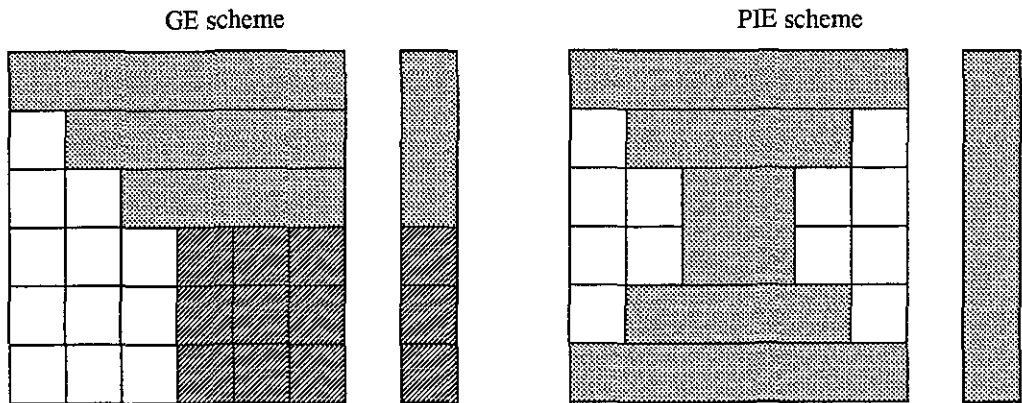


Figure 4.1.3 - Column 3 of GE being eliminated. PIE method is complete.

Step 4:

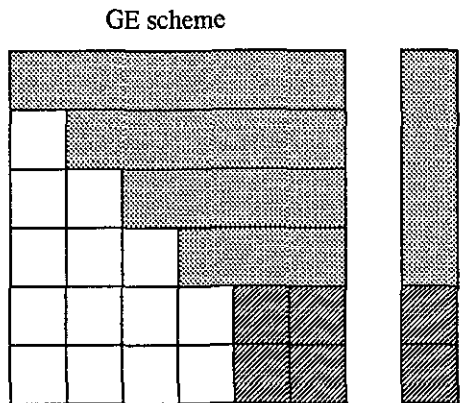


Figure 4.1.4 - Column 4 of GE being eliminated.

Step 5:

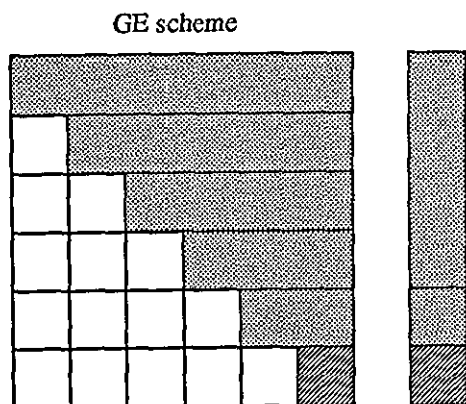


Figure 4.1.5- Column 5 of GE being eliminated.

After 5 steps in the elimination process of GE for a 6x6 matrix, the shaded area in Figure 4.1.6 represents the elements of the triangular matrix and the solution can then be obtained by a back substitution process.

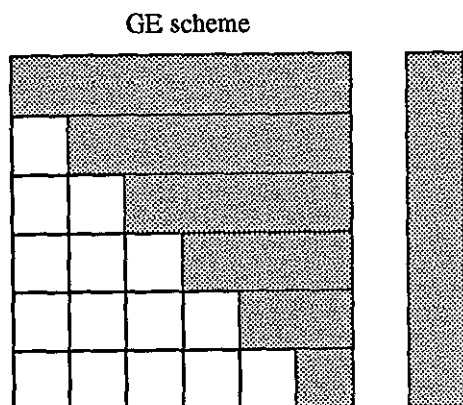


Figure 4.1.6- Forward elimination of GE complete.

## 4.2 Quadrant Interlocking Factorisation method (QIF)

QIF is a method that decomposes the coefficient matrix  $A$  into two interlocking quadrant factors of butterfly form denoted by  $W$  and  $Z$  or as,

$$A=WZ \quad (4.2.1)$$

where  $W = \begin{bmatrix} 1 & 0 \\ w_{i1} & w_{n-2} \\ 0 & 1 \end{bmatrix}, i=2, 3, \dots, n-1,$



$$Z = \begin{bmatrix} z_{11} & \underline{z_{1i}^T} & z_{1n} \\ & \underline{Z_{n-2}} & \\ z_{n1} & \underline{z_{ni}^T} & z_{nn} \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} a_{11} & \underline{a_{1i}^T} & a_{1n} \\ \underline{a_{i1}} & \underline{A_{n-2}} & \underline{a_{in}} \\ a_{n1} & \underline{a_{ni}^T} & a_{nn} \end{bmatrix}$$

which represents the partitioned forms

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ w_{2,1} & 1 & 0 & 0 & 0 & 0 & w_{2,n} \\ & w_{3,2} & 1 & 0 & 0 & w_{3,n-1} & \\ \vdots & \vdots & & \ddots & & \vdots & \vdots \\ & w_{n-2,2} & 0 & 1 & 0 & w_{n-2,n-1} & \\ w_{n-1,1} & 0 & 0 & 0 & 0 & 1 & w_{n-1,n} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad (4.2.2)$$

and

$$Z = \begin{bmatrix} z_{1,1} & z_{1,2} & \cdots & \cdots & \cdots & z_{1,n-1} & z_{1,n} \\ 0 & z_{2,2} & \cdots & \cdots & \cdots & z_{2,n-1} & 0 \\ 0 & 0 & \ddots & & & 0 & 0 \\ 0 & 0 & & \ddots & & 0 & 0 \\ 0 & 0 & & & \ddots & 0 & 0 \\ 0 & z_{n-1,2} & \cdots & \cdots & \cdots & z_{n-1,n-1} & 0 \\ z_{n,1} & z_{n,2} & \cdots & \cdots & \cdots & z_{n,n-1} & z_{n,n} \end{bmatrix} \quad (4.2.3)$$

with a similar partitioning for  $A$ .

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \quad (4.2.4)$$

Equating terms in the factorisation in (4.2.1) results in

$$\begin{bmatrix} a_{11} & \underline{a_{1i}^T} & a_{1n} \\ \underline{a_{i1}} & \underline{A_{n-2}} & \underline{a_{in}} \\ a_{n1} & \underline{a_{ni}^T} & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \underline{w_{i1}} & \underline{W_{n-2}} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} z_{11} & \underline{z_{1i}^T} & z_{1n} \\ & \underline{Z_{n-2}} & \\ z_{n1} & \underline{z_{ni}^T} & z_{nn} \end{bmatrix} \quad (4.2.5)$$

Then the following relations are obvious.

$$\begin{aligned} a_{11} &= z_{11} & a_{n1} &= z_{n1} \\ \underline{a_{1i}^T} &= \underline{z_{1i}^T} & \text{and} & \underline{a_{ni}^T} &= \underline{z_{ni}^T} \\ a_{1n} &= z_{1n} & z_{nn} &= a_{nn} \end{aligned} \quad (4.2.6)$$

Then, there are also the following relations

$$\begin{aligned} Z_{11}W_{j1} + Z_{n1}W_{in} &= a_{j1} \\ Z_{1n}W_{j1} + Z_{nn}W_{in} &= a_{in} \end{aligned} \quad (4.2.7)$$

which represents a series of (2x2) linear systems.

As discussed in PIE, the solution to the (2x2) system can be obtained by using one of three methods, namely Cramers rule, simple elimination or symmetric elimination.

Finally, in the determination of the reduced matrix  $A'_{n-2}$ ,

$$W_{j1}Z_{1i}^T + W_{n-2}Z_{n-2} + W_{in}Z_{ni}^T = A_{n-2}$$

Thus,

$$\begin{aligned} W_{n-2}Z_{n-2} &= A_{n-2} - W_{j1}Z_{1i}^T - W_{in}Z_{ni}^T \\ &= A'_{n-2} \end{aligned}$$

The system is now recursively repeated for  $n-2, n-4, \dots, 2$  for  $n$  even and  $n-2, n-4, \dots, 1$ , for  $n$  odd.

#### Solution process:

By using (4.2.1) the linear system in (4.1) can be written in the form,

$$WZx=b, \quad (4.2.8)$$

and the solution vector  $x$  can be obtained by solving the two alternative systems,

$$Wy=b, \quad (4.2.9)$$

known as bi-directional substitution,

and

$$Zx=y. \quad (4.2.10)$$

known as bi-directional solution.

#### Bi-directional substitution:

To solve (4.2.9) we have,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ w_{2,1} & 1 & 0 & 0 & 0 & 0 & w_{2,n} \\ \vdots & w_{3,2} & 1 & 0 & w_{3,n-1} & \vdots & \vdots \\ & \vdots & & \ddots & \vdots & & \\ \vdots & w_{n-2,2} & 0 & & 1 & w_{n-2,n-1} & \vdots \\ w_{n-1,1} & 0 & 0 & 0 & 0 & 1 & w_{n-1,n} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \quad (4.2.11)$$

from which the values of  $y$  can be obtained by the process of bi-directional substitution, i.e.,

$$\begin{aligned}
 y_1 &= b_1; \\
 y_n &= b_n \\
 y_2 &= b_2 - w_{21}y_1 - w_{2n}y_n; \\
 y_{n-1} &= b_{n-1} - w_{n-1,1}y_1 - w_{n-1,n}y_n \\
 &\vdots \\
 y_i &= b_i - w_{i1}y_1 - w_{in}y_n - \dots - w_{i,i-1}y_{i-1} - w_{i,n+1-i}y_{n+1-i} \\
 y_{n+1-i} &= b_{n+1-i} - w_{n+1-i,1}y_1 - w_{n+1-i,n}y_n - \dots - w_{n+1-i,n-i}y_{n-i} - w_{n+1-i,2}y_2
 \end{aligned}
 \tag{4.2.12}$$

If  $n$  is odd, the final equation to solve is

$$y_{\frac{n+1}{2}} = b_{\frac{n+1}{2}} - w_{\frac{n+1}{2},1}y_1 - w_{\frac{n+1}{2},n}y_n - \dots - w_{\frac{n+1}{2},\frac{n-1}{2}}y_{\frac{n-1}{2}} - w_{\frac{n+1}{2},\frac{n+3}{2}}y_{\frac{n+3}{2}} \tag{4.2.13}$$

and if  $n$  is even, the last two equations to be solved are

$$\begin{aligned}
 y_{\frac{n}{2}} &= b_{\frac{n}{2}} - w_{\frac{n}{2},1}y_1 - w_{\frac{n}{2},n}y_n - \dots - w_{\frac{n}{2},\frac{n}{2}-1}y_{\frac{n}{2}-1} - w_{\frac{n}{2},\frac{n}{2}+1}y_{\frac{n}{2}+1} \\
 y_{\frac{n}{2}+1} &= b_{\frac{n}{2}+1} - w_{\frac{n}{2}+1,1}y_1 - w_{\frac{n}{2}+1,n}y_n - \dots - w_{\frac{n}{2}+1,\frac{n}{2}}y_{\frac{n}{2}} - w_{\frac{n}{2}+1,\frac{n}{2}+2}y_{\frac{n}{2}+2}
 \end{aligned}
 \tag{4.2.14}$$

The substitution process begins from the top and bottom moving inwards bi-directionally and each time substituting for two values simultaneously.

Finally the solution  $x$  can be obtained from the process of bi-directional solution, similar to that of PIE, as described in section 4.1.

A diagrammatic comparison of LU and QIF can also be made. Figures 4.2.1 through 4.2.5 illustrate the steps taken to perform LU factorisation on a simple (6x6) coefficient matrix. Figures 4.2.6 and 4.2.7 illustrate the steps taken to perform QIF on a simple (6x6) coefficient matrix.

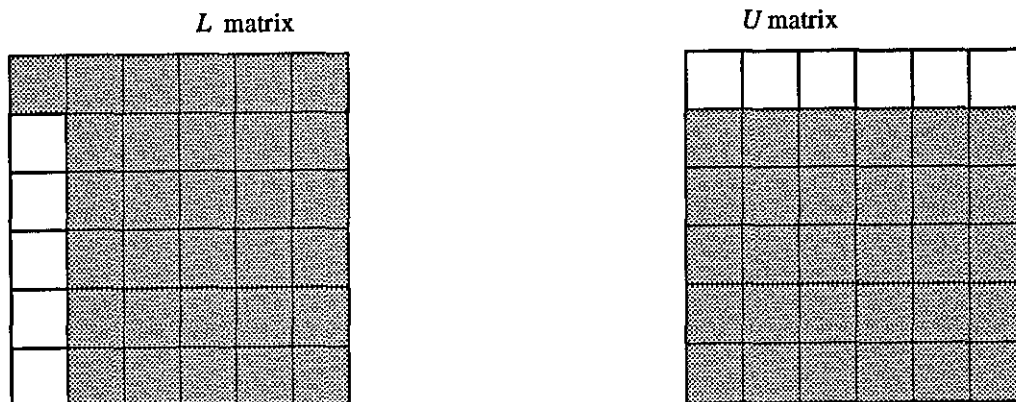


Figure 4.2.1 - First step of LU factorisation

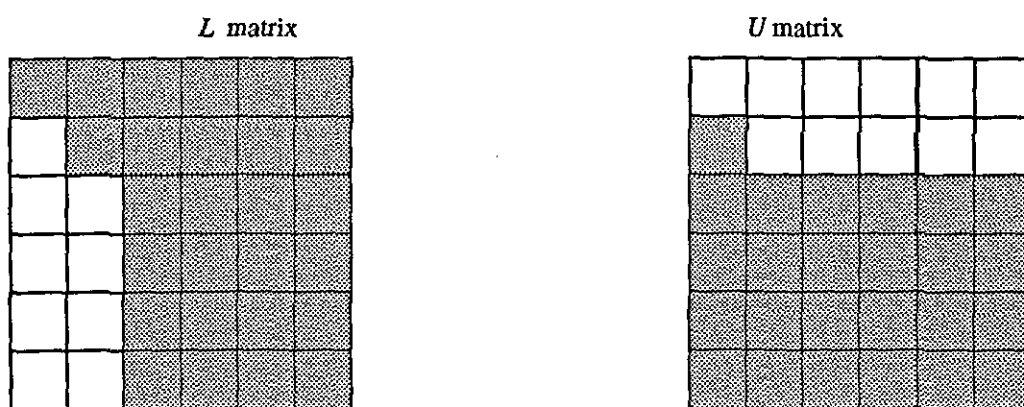


Figure 4.2.2 - Second step of LU factorisation

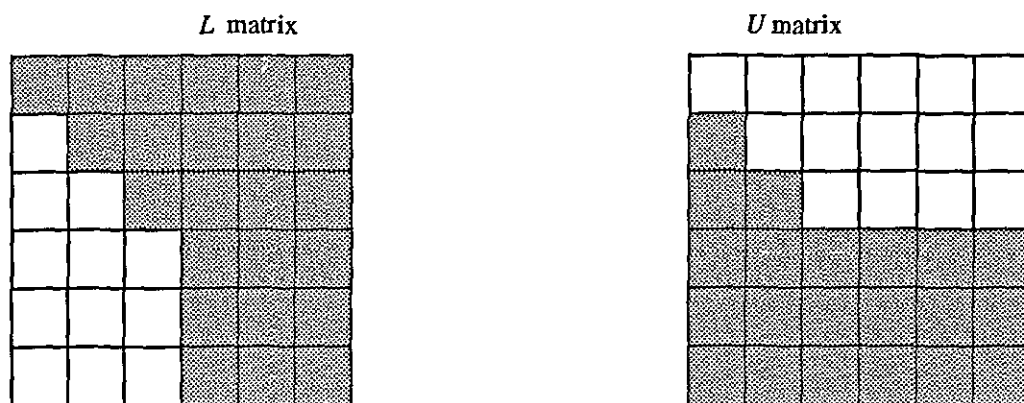


Figure 4.2.3 - Third step of LU factorisation

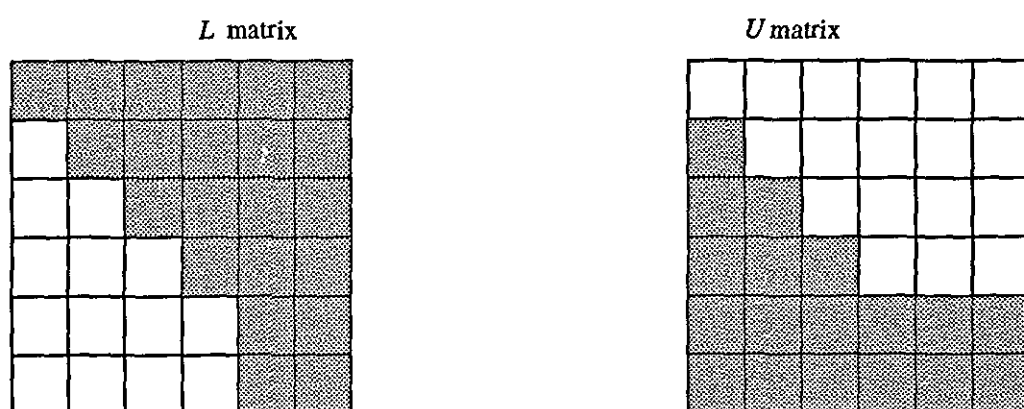


Figure 4.2.4 - Fourth step of LU factorisation

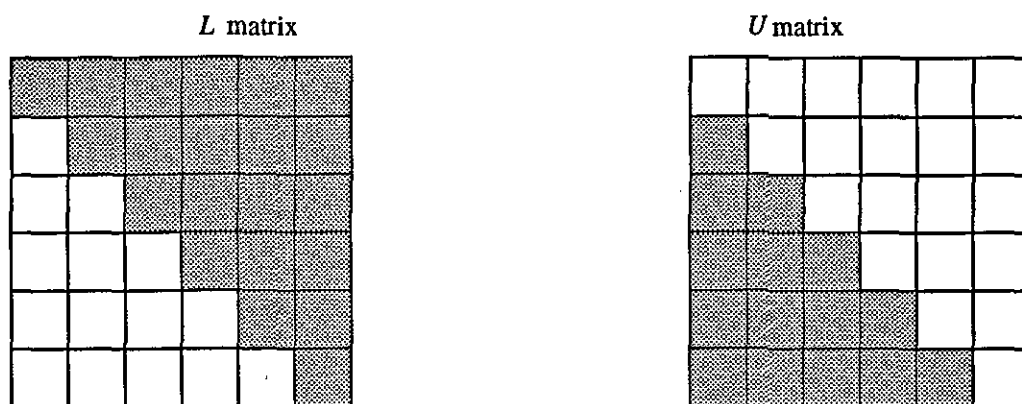


Figure 4.2.5 - Fifth step of LU factorisation

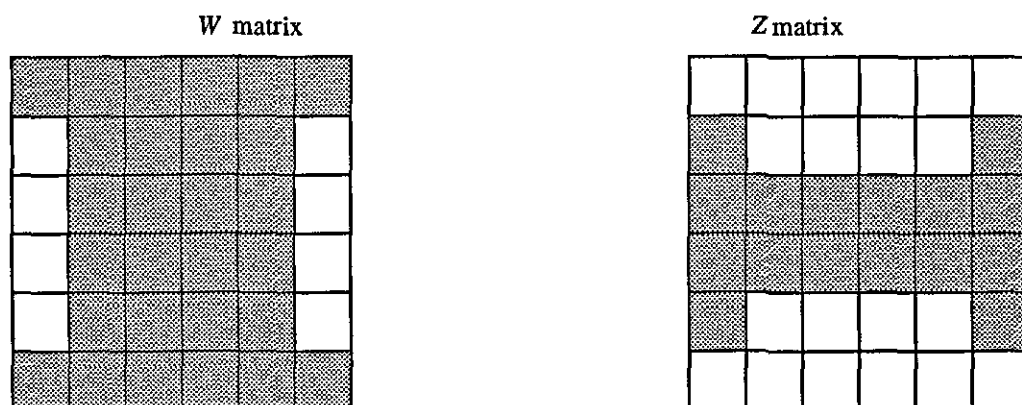


Figure 4.2.6 - First step of WZ factorisation

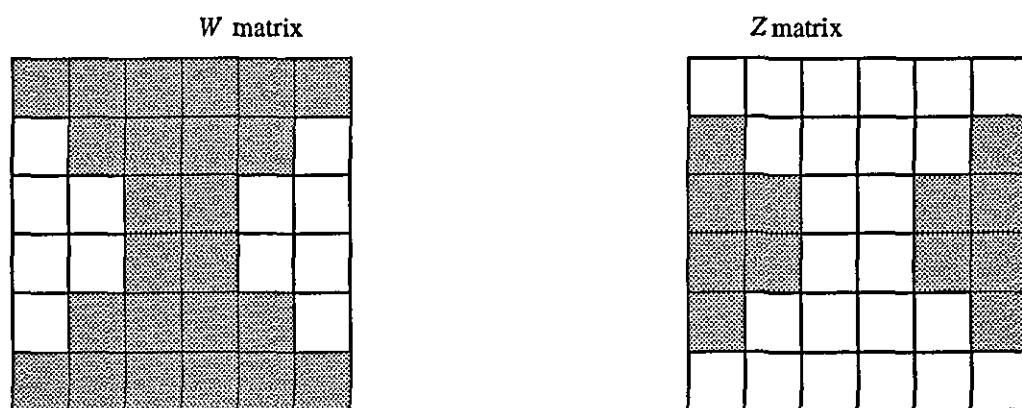


Figure 4.2.7 - Second step of WZ factorisation

In comparison to LU factorisation which takes 5 steps to factorise, QIF only takes 2 steps.

### 4.3 Partial Pivoting for PIE and QIF

The PIE and QIF methods described in section 4.1 and 4.2 respectively are appropriate when the diagonal elements are dominant. However, the methods will break down when the diagonal elements are not dominant. Therefore, attention must be given to the numerical instability resulting when the matrix is not diagonally dominant. In order to ensure that these schemes are numerically stable, there is a need to limit the growth of elements in the reduced coefficient matrix. In this section, partial pivoting for PIE and QIF will be discussed.

#### 4.3.1 Partial Pivoting for PIE

The pivots for PIE are no longer single elements, instead, at the  $k^{\text{th}}$  stage of elimination, the following multipliers will be obtained.

$$w_{ik} = \frac{a_{ik}^{(k-1)} a_{n-k+1, n-k+1}^{(k-1)} - a_{i, n-k+1}^{(k-1)} a_{n-k+1, k}^{(k-1)}}{\Delta} \quad i = k+1, \dots, n-k$$

$$w_{i, n-k+1} = \frac{a_{k, n-k+1}^{(k-1)} a_{i, k}^{(k-1)} - a_{k, i}^{(k-1)} a_{i, n-k+1}^{(k-1)}}{\Delta}$$

where

$$\Delta = a_{kk}^{(k-1)} a_{n-k+1, n-k+1}^{(k-1)} - a_{k, n-k+1}^{(k-1)} a_{n-k+1, k}^{(k-1)}$$

The safest strategy is to choose the largest  $\Delta$  but this involves an exhaustive search over

$$\frac{r!}{2!(r-2)!} = \frac{r^2 - r}{2} \quad \text{where } n > r > 1$$

possible pivots at each step. This will subsequently cause an increase in the total computational complexity. Following is a description of the partial pivoting strategy for PIE.

At the  $k^{\text{th}}$  step of the elimination process,

(a) Find a row  $i_0$ , such that

$$\left| a_{i_0, k}^{(k-1)} \right| = \max_{k \leq i \leq n-k+1} \left| a_{ik}^{(k-1)} \right|$$

and exchange the rows  $i_0$  and  $k$  or denote it by a flag.

(b) Find a row number  $i_1$ , such that

$$\left| a_{i_1, n-k+1}^{(k-1)} - \frac{a_{k, n-k+1}^{(k-1)}}{a_{kk}^{(k-1)}} a_{i_1, k}^{(k-1)} \right| = \max_{k+1 \leq i \leq n-k+1} \left| a_{i, n-k+1}^{(k-1)} - \frac{a_{k, n-k+1}^{(k-1)}}{a_{kk}^{(k-1)}} a_{ik}^{(k-1)} \right|$$

and exchange the rows  $i_1$  and  $n-k+1$ , again denoting this by a flag.

The subsequent steps are as described in (4.1.6) where the relevant entries in columns  $k$  and  $n-k+1$  are eliminated concurrently.

The above partial pivoting strategy ensures that

$$|w_{ik}| \leq 2 \quad k = 1, \dots, n \quad i = k+1, \dots, n-k$$

and

$$|w_{i,n-k+1}| \leq 1 \quad k = 1, \dots, n \quad i = k+1, \dots, n-k$$

This has been proven by Barulli and Evans in [Barulli 96].

### 4.3.2 Partial Pivoting for QIF

When the computation of the  $W$  and  $Z$  matrix factors is performed as described in section 4.2, the elements of  $W$  and  $Z$  may become too large causing rounding errors and the process can break down. The factorisation process can be reorganised to include partial pivoting. The steps are described below.

(a) Find the largest element  $a_{i1}$  in column 1 and interchange row  $i$  with row 1.

Evaluate multipliers  $m_{i1} = \frac{a_{i1}}{a_{11}} \quad i > 1$ .

(b) Calculate the updated elements  $a'_{in} = a_{in} - m_{i1}a_{1n}$  for all  $i > 1$  in the  $n^{\text{th}}$  column only.

(c) Find the row with the largest updated element  $a'_{in}$  calculated in step (b) and interchange with row  $n$ .

(d) Repeat steps (e) through (j) for values of  $k$  from 1 to  $n/2$ .

(e) Evaluate the  $W$  matrix elements for columns  $k$  and  $n-k+1$  by solving the  $(2 \times 2)$  linear system.

$$\begin{aligned} z_{kk}w_{jk} + z_{n-k+1,k}w_{j,n-k+1} &= a_{j,k}^{(k)} \\ z_{k,n-k+1}w_{jk} + z_{n-k+1,n-k+1}w_{j,n-k+1} &= a_{j,n-k+1}^{(k)} \end{aligned} \quad \text{for } j=k+1(1)n-k,$$

(f) Now calculate the prospective  $Z$  matrix elements for row  $k$ . Let  $S_i$  be the partial row/column products for the  $k^{\text{th}}$  column. Then evaluate for  $i=k$  to  $n-k+1$ :

$$S_i = a_{ik} - w_{i1}z_{1k} - \dots - w_{i,k-1}z_{k-1,k} - w_{in}z_{nk} - \dots - w_{i,n-(k-1)+1}z_{n-(k-1)+1,k}$$

These are the possible choices of the  $k^{\text{th}}$  pivotal row. Suppose  $S_{k'}$  is the maximum of  $S_i$ , for  $i=k, k+1, \dots, n-k+1$ . Interchange row  $k'$  with row  $k$ .

(g) Now calculate the prospective Z matrix elements for row  $n-k+1$ . Let  $T_i$  be the partial row/column products for the  $n-k+1$ th column.

Then evaluate for  $i=k+1$  to  $n-k+1$ :

$$T_i = a_{i,n-k+1} - w_{i1}z_{n-k+1} - \dots - w_{i,k-1}z_{k-1,n-k+1} - w_{in}z_{n,n-k+1} - \dots - w_{i,n-(k-1)+1}z_{n-(k-1)+1,n-k+1}$$

These are the possible choices of the  $n-k+1$  pivotal row. Suppose  $T_k$  is the maximum of  $T_i$  for  $i=k+1, \dots, n-k+1$ . Interchange row  $k''$  with row  $n-k+1$ .

(h) Let  $S_k$  be  $z_{kk}$  and  $T_k$  be  $z_{n-k+1,n-k+1}$ .

(i) Now update row  $k$  from column  $k+1$  to  $n-k+1$ .

Evaluate for  $j=k+1$  to  $n-k$ :

$$z_{n-k+1,j} = a_{n-k+1,j} - w_{n-k+1,1}z_{1k} - \dots - w_{n-k+1,k-1}z_{k-1,k} - w_{n-k+1,n}z_{nk} - \dots - w_{n-k+1,n-(k-1)+1}z_{n-(k-1)+1,k}$$

(j) Finally update the right-hand side vector  $b$ . Evaluate

$$\begin{aligned} b'_k &= b_k - w_{k1}b_1 - \dots - w_{k,k-1}b_{k-1} - \dots - w_{kn}b_n - \dots - w_{k,n-k+1}b_{n-k+1} \\ b'_{n-k+1} &= b_{n-k+1} - w_{n-k+1,1}b_1 - \dots - w_{n-k+1,k-1}b_{k-1} - \\ &\quad w_{n-k+1,n}b_n - \dots - w_{n-k+1,n-(k-1)+1}b_{n-(k-1)+1} \end{aligned}$$

#### 4.4 Implementation of PIE and QIF on shared memory architecture

This section describes the sequential and parallel algorithms for PIE and QIF. In understanding the algorithms, it is vital that the underlying assumptions made about the data structure employed as well as the terminologies used to describe the algorithms be explicitly stated. The parallelisation of the algorithms considered in this section are in the context of a shared-memory parallel computer. Parallelisation of all these algorithms were achieved by suitably partitioning the rows amongst the processors in a cyclic fashion.

Assumptions:

- Coefficient array  $A$  is two-dimensional and begins from index 0 to  $n-1$  for both rows and column.
- Right hand side vector  $b$  is augmented to the coefficient array  $A$  and becomes column  $n$  of matrix  $A$ .
- $W$  and  $Z$  matrices are overwritten in the original matrix  $A$ . The solution  $x$  is overwritten in the augmented right-hand side vector  $b$ . This way memory used is saved and the calculation process can be accelerated by means of an in place implementation of the algorithm. Equations (4.2.6) will not have to be evaluated.



- Algorithms assume that  $n$  is even.
- Loops begin from 0 to match the code written in C.

The algorithm is not described in any particular programming language, and hence, not directly transferable to a running program without some slight modifications. Statements within { } are comments to describe certain parts of the algorithm.

When describing the parallel algorithm, the loop enclosed between *par* and *end par* indicates that the loop iterations can be done in parallel. The actual syntax of assigning the loop iterations to the available processors is compiler dependent.

As described in section 4.1, PIE consists of two phases, i.e. :

- (i) parallel elimination phase
- (ii) bi-directional solution phase

As described in section 4.2, QIF consists of three phases, i.e.:

- (i) factorisation of  $W$  and  $Z$
- (ii) bi-directional substitution
- (iii) bi-directional solution

The parallel elimination algorithm is presented in section 4.4.1, the factorisation of  $W$  and  $Z$  is described in section 4.4.2, and the bi-directional substitution is described in section 4.4.3. Since the bi-directional solution phase is common to both PIE and QIF, it is described only once in section 4.4.4.

#### 4.4.1 Parallel Elimination algorithm

The sequential algorithm for performing the implicit elimination process of PIE is shown in Algorithm 4.4.1.1. The outer loop indexed by  $k$  in Algorithm 4.4.1.1 represents the  $k^{\text{th}}$  level of elimination and this loop must be obeyed sequentially as the outer matrix has to be reduced before the inner matrix commences its reduction. The inner loop  $i$  performs the elimination of  $w_k$  and  $w_{n-1-k}$  of row  $i$ . This can be done in parallel; i.e., each processor can eliminate the  $w_k$  and  $w_{n-1-k}$  elements of different rows as well as update the remaining elements of the rows independently. Hence, the iterations of this loop are independent of each other and can be performed independently by more than one processor. These loop iterations can be assigned to the processors in two ways. They can be assigned as a block of contiguous loop iterations or the loop iterations can be alternately assigned to the different processors in a cyclic

manner. In the work of this thesis, the loop iterations were assigned to the processors in a cyclic manner so that all the processors have an equal load. The parallel algorithm for the implicit elimination is shown in Algorithm 4.4.1.2.

Algorithm 4.4.1.1: Sequential algorithm for implicit elimination

```

cp1=0, cp2=n-1
for k=0 to (n/2)-1 do {number of levels to perform elimination}
    for i=k+1 to n-k-2 do {number of eliminations in each level}
        m= -A[cp1,cp2]/A[cp1,cp1]
        { solve the 2x2 system}
        x2=(A[i,cp2]-m*A[i,cp1])/(A[cp2,cp2]-m*A[cp1,cp2])
        x1=(A[i,cp2]-A[cp2,cp2]*x2)/A[cp2,cp1]
        for j1=cp1+1 to cp2-1 do
            {update the rest of the row }
            A[i,j1]=A[i,j1]-(x1*A[cp1,j1]+x2*A[cp2,j1])
        end for j1
        {update rhs}
        A[i,n]=A[i,n]-(x1*A[cp1,n]+x2*A[cp2,n])
    end for j
    cp1=cp1+1
    cp2=cp2-1
end for k

```

Algorithm 4.4.1.2: Parallel algorithm for implicit elimination

```

cp1=0, cp2=n-1
for k=0 to (n/2)-1 do {number of levels to perform elimination}
    par
        for i=k+1 to n-k-2 do {number of eliminations in each level}
            m= -A[cp1,cp2]/A[cp1,cp1]
            { solve the 2x2 system}
            x2=(A[i,cp2]-m*A[i,cp1])/(A[cp2,cp2]-m*A[cp1,cp2])
            x1=(A[i,cp2]-A[cp2,cp2]*x2)/A[cp2,cp1]
            for j1=cp1+1 to cp2-1 do
                {update the rest of the row }
                A[i,j1]=A[i,j1]-(x1*A[cp1,j1]+x2*A[cp2,j1])
            end for j1
            {update rhs}
            A[i,n]=A[i,n]-(x1*A[cp1,n]+x2*A[cp2,n])
        end for j
    end par
    cp1=cp1+1
    cp2=cp2-1
end for k

```

#### 4.4.2 QIF algorithm

The sequential algorithm for the factorisation stage of QIF is given in Algorithm 4.4.2.1 while the parallel algorithm is given in Algorithm 4.4.2.2. The outer loop indexed by  $k$  in Algorithm 4.4.2.1 represents the  $k^{\text{th}}$  level of factorisation and this loop must be obeyed sequentially as the outer matrix has to be factorised before the inner matrix commences its factorisation. The inner loop indexed by  $i$  performs the formation of  $w_k$  and  $w_{n-1-k}$  of row  $i$ . The calculation of  $w_k$  and  $w_{n-1-k}$  elements of the different rows and the formation of the  $z$  values in these rows can be done independently. Hence, the iterations of this loop are split amongst the processors available in a cyclic manner.

Algorithm 4.4.2.1: Sequential algorithm for WZ factorisation

```

cp1=0, cp2=n-1
for k=0 to (n/2)-1 do
    for i=k+1 to n-k-2 do
        m= -A[cp1,cp2]/A[cp1,cp1]
        {solve the following 2x2 system, to get w's}
        x2=(A[i,cp2]-m*A[i,cp1])/(A[cp2,cp2]-m*A[cp1,cp2])
        x1=(A[i,cp2]-A[cp2,cp2]*x2)/A[cp2,cp1]
        A[i,k]=x1
        A[i,n-1-k]=x2
        for j1=k+1 to n-2-k do
            {update to form z's }
            A[i,j1]=A[i,j1]-(x1*A[cp1,j1]+x2*A[cp2,j1])
        end for i
        cp1=cp1+1
        cp2=cp2-1
    end for k

```

Algorithm 4.4.2.2: Parallel algorithm for WZ factorisation

```

cp1=0, cp2=n-1
for k=0 to (n/2)-1 do
    par
        for i=k+1 to n-k-2 do
            m= -A[cp2,cp1]/A[cp1,cp1]
            {solve the following 2x2 system, to get w's}
            x2=(A[i,cp2]-m*A[i,cp1])/(A[cp2,cp2]-m*A[cp1,cp2])
            x1=(A[i,cp2]-A[cp2,cp2]*x2)/A[cp2,cp1]
            A[i,k]=x1
            A[i,n-1-k]=x2
            for j1=k+1 to n-2-k do
                {update to form z's }
                A[i,j1]=A[i,j1]-(x1*A[cp1,j1]+x2*A[cp2,j1])
            end for i
        end par
        cp1=cp1+1
        cp2=cp2-1
    end for k

```

#### 4.4.3 Bi-directional substitution algorithm

The sequential algorithm for performing the bi-directional substitution for QIF is given in Algorithm 4.4.3.1 while the parallel algorithm is shown in Algorithm 4.4.3.2.

The outer loop indexed by  $k$  obtains the values  $y_k$  and  $y_{n-1-k}$  of equation (4.2.5). This must be done sequentially. The inner loop indexed by  $i$  substitutes the values of  $y_k$  and  $y_{n-1-k}$  into its occurrences in the  $i^{\text{th}}$  row. This substitution can be done independently and can be parallelised. Once again, the loop iterations are split amongst the processors available in a cyclic manner.

Algorithm 4.4.3.1: Sequential algorithm for bi-directional substitution

```
for k=0 to (n/2)-1
  for i=k+1 to n-k-2
    A[i,k]=A[k,n]*A[i,k]
    A[i,n-1-k]=A[n-1-k,n]*A[i,n-1-k]
    A[i,n]=A[i,n]-(A[i,k]+A[i,n-1-k])
  end for i
end for k
```

Algorithm 4.4.3.2: Parallel algorithm for bi-directional substitution

```
for k=0 to (n/2)-1
  par
    for i=k+1 to n-k-2
      A[i,k]=A[k,n]*A[i,k]
      A[i,n-1-k]=A[n-1-k,n]*A[i,n-1-k]
      A[i,n]=A[i,n]-(A[i,k]+A[i,n-1-k])
    end for i
  end par
end for k
```

#### 4.4.4 Bi-directional solution algorithm

The sequential algorithm for the bi-directional solution procedure is shown in Algorithm 4.4.4.1 while the parallel version is given in Algorithm 4.4.4.2. The outer loop indexed by  $k$  in Algorithm 4.4.4.1 solves for  $x_k$  and  $x_{n-1-k}$  of equation (4.2.10). This must be done sequentially. The inner loop  $j$  substitutes for known values of  $x$  in rows  $k$  and  $n-1-k$  of the submatrix  $(k \times k)$ . This substitution can be partitioned and performed independently by different processors.

Algorithm 4.4.4.1: Sequential algorithm for bi-directional solution

```

Let m1=n/2-1
Let m2=n/2
for k=0 to n/2-1
    p1=A[m2,m1]/A[m1,m1]
    p2=A[m2,m2]-p1*A[m1][m2]
    x2=(A[m2,n]-p1*A[m1,n])/p2
    x1=(A[m1,n]-A[m1,m2]*x2)/A[m1,m1]
    for j=0 to m1-1
        A[j,n]=A[j,n]-(x1*A[j,m1]+x2*A[j,m2])
        A[n-1-j,n]=A[n-1-j,n]-(x1*A[n-1-j,m1]+x2*A[n-1-j,m2])
    end for j
    m1=m1-1
    m2=m2+1
end for k

```

Algorithm 4.4.4.2: Parallel algorithm for bi-directional solution

```

Let m1=n/2-1
Let m2=n/2
for k=0 to n/2-1
    p1=A[m2,m1]/A[m1,m1]
    p2=A[m2,m2]-p1*A[m1][m2]
    x2=(A[m2,n]-p1*A[m1,n])/p2
    x1=(A[m1,n]-A[m1,m2]*x2)/A[m1,m1]
    par
        for j=0 to m1-1
            A[j,n]=A[j,n]-(x1*A[j,m1]+x2*A[j,m2])
            A[n-1-j,n]=A[n-1-j,n]-(x1*A[n-1-j,m1]+x2*A[n-1-j,m2])
        end for j
    end par
    m1=m1-1
    m2=m2+1
end for k

```

In this section the implementation of PIE and QIF on a shared-memory architecture has been discussed. The algorithms presented did not include partial pivoting. The incorporation of partial pivoting is a straight forward one as the procedures for partial pivoting has been given in algorithmic form for PIE in section 4.3.1 and QIF in section 4.3.2.

These algorithms have also been implemented on a distributed architecture and this is covered in chapter 5 as the issues of implementation differ on the shared memory and distributed architectures.

#### 4.5 Computational Complexity and Shared Memory Access

In this section a discussion on the computational complexity or operational count of PIE and QIF is given. Another factor that seemed to have an influence on the performance of parallel algorithms is the number of times the shared memory is being accessed either by a read from shared memory or writing to shared memory. This will be referred to as the shared memory access count in this thesis. The computational complexity and shared memory access count of PIE and QIF will be covered in sections 4.5.1 and 4.5.2 respectively. Since the aim of this thesis is to compare the performance of PIE and QIF with GE and LU respectively, then it is necessary that a discussion of the computational complexity and shared memory access count of GE and LU be done too. These are covered in sections 4.5.3 and 4.5.4 for GE and LU respectively. A summary of the computational complexity and shared memory access count for all the methods is given in section 4.5.5.

The operational count includes the computational steps of add and multiply operations. Throughout this section, the add operator will be denoted by  $a$  and the multiply operator will be denoted by  $m$ .

##### 4.5.1 Computational Complexity and Shared Memory Access Count for PIE

The computational complexity and shared memory access count developed in this section are based on the sequential algorithm for PIE developed in section 4.3.1. The algorithms are reproduced here where Algorithm 4.5.1.1 is the parallel elimination procedure and Algorithm 4.5.1.2 is the bi-directional solution in PIE.

The amount of work involved in the calculation of the multipliers for the solution of  $(2 \times 2)$  linear systems is quite obvious, i.e.,

$$W_1 = (n/2)m \quad (4.5.1.1)$$

In the solution of the  $(2 \times 2)$  systems, 3 mults, 2 divs and 3 adds are required. (Assuming a division operation and a multiply operation is the same and is termed as a multiply) This yields a total of

$$\begin{aligned}
W_2 &= \sum_{k=2(2)}^{n-2} k(5m+3a) \\
&= \left( \frac{n^2}{4} - \frac{n}{2} \right) (5m+3a) \\
&= \left( \frac{5}{4}n^2 - \frac{5}{2}n \right) m + \left( \frac{3}{4}n^2 - \frac{3}{2}n \right) a
\end{aligned} \tag{4.5.1.2}$$

Algorithm 4.5.1.1: Sequential algorithm for parallel elimination

```

cp1=0, cp2=n-1
for k=0 to (n/2)-1 do {number of levels to perform elimination}
    for i=k+1 to n-k-2 do {number of eliminations in each level}
        m = -A[cp1,cp2]/A[cp1,cp1]
        { solve the 2x2 system}
        x2=(A[i,cp2]-m*A[i,cp1])/(A[cp2,cp2]-m*A[cp1,cp2])
        x1=(A[i,cp2]-A[cp2,cp2]*x2)/A[cp2,cp1]
        for j1=cp1+1 to cp2-1 do
            {update the rest of the row }
            A[i,j1]=A[i,j1]-(x1*A[cp1,j1]+x2*A[cp2,j1])
        end for j1
        {update rhs}
        A[i,n]=A[i,n]-(x1*A[cp1,n]+x2*A[cp2,n])
    end for j
    cp1=cp1+1
    cp2=cp2-1
end for k

```

Algorithm 4.5.1.2: Sequential algorithm for bi-directional solution

```

Let m1=n/2-1
Let m2=n/2
for k=0 to n/2-1
    p1=A[m2,m1]/A[m1,m1]
    p2=A[m2,m2]-p1*A[m1,m2]
    x2=(A[m2,n]-p1*A[m1,n])/p2
    x1=(A[m1,n]-A[m1,m2]*x2)/A[m1,m1]
    for j=0 to m1-1
        A[j,n]=A[j,n]-(x1*A[j,m1]+x2*A[j,m2])
        A[n-1-j,n]=A[n-1-j,n]-(x1*A[n-1-j,m1]+x2*A[n-1-j,m2])
    end for j
    m1=m1-1
    m2=m2+1
end for k

```

In the updating stage, 2 mults and 2 adds are performed on every element of the reduced array of order (kxk), giving a total of

$$\begin{aligned}
W_3 &= \sum_{k=2(2)}^{n-2} k^2 (2m+2a) \\
&= \left( \frac{n^3}{6} - \frac{3}{2}n^2 + \frac{n}{3} \right) (2m+2a) \\
&= \left( \frac{n^3}{3} - 3n^2 + \frac{2}{3}n \right) m + \left( \frac{n^3}{3} - 3n^2 + \frac{2}{3}n \right) a
\end{aligned} \tag{4.5.1.3}$$

and similarly, the updating of the right-hand side vector also requires 2 mults and 2 adds resulting in a total of

$$\begin{aligned}
W_4 &= \sum_{k=2(2)}^{n-2} k(2m+2a) \\
&= \left( \frac{n^2}{4} - \frac{n}{2} \right) (2m+2a) \\
&= \left( \frac{n^2}{2} - n \right) m + \left( \frac{n^2}{2} - n \right) a
\end{aligned} \tag{4.5.1.4}$$

Thus the total computational complexity in the elimination stage is

$$\begin{aligned}
W_{\text{elim}} &= W_1 + W_2 + W_3 + W_4 \\
&= \left( \frac{n^3}{3} - \frac{5}{4}n^2 - \frac{7}{3}n \right) m + \left( \frac{n^3}{3} - \frac{7}{4}n^2 - \frac{11}{6}n \right) a.
\end{aligned} \tag{4.5.1.5}$$

In the bi-directional solution process, the 2x2 linear system requires 6 mults and 3 adds, giving a total of

$$\begin{aligned}
W_5 &= \frac{n}{2}(6m+3a) \\
&= (3n)m + \left( \frac{3}{2}n \right) a
\end{aligned} \tag{4.5.1.6}$$

while the substitution stage requires 4 mults and 4 adds resulting in a total of

$$\begin{aligned}
W_6 &= \sum_{k=2(2)}^{n-2} k(4m+4a) \\
&= \left( \frac{n^2}{4} - \frac{n}{2} \right) (4m+4a) \\
&= (n^2 - 2n)m + (n^2 - 2n)a.
\end{aligned} \tag{4.5.1.7}$$



Hence, the total computational count for the bi-directional solution stage is

$$\begin{aligned} W_{\text{bi-soln}} &= W_5 + W_6 \\ &= (n^2 + n)m + \left(n^2 - \frac{n}{2}\right)a.. \end{aligned} \quad (4.5.1.8)$$

Thus, the total computational count for the PIE method is

$$\begin{aligned} W_{\text{PIE}} &= W_{\text{elim}} + W_{\text{bi-soln}} \\ &= \left(\frac{n^3}{3} + \frac{n^2}{4} - \frac{4}{3}n\right)m + \left(\frac{n^3}{3} - \frac{3}{4}n^2 - \frac{7}{3}n\right)a. \end{aligned} \quad (4.5.1.9)$$

The detailed outline of the shared memory access count for PIE is as follows:

The evaluation of multipliers for the solution of a (2x2) system requires 2 accesses to shared memory.

$$M_1 = \left(\frac{n}{2}\right)2 = n \quad (4.5.1.10)$$

The solution of the (2x2) linear systems require 7 shared memory accesses.

$$\begin{aligned} M_2 &= \sum_{k=2(2)}^{n-2} (k)7 \\ &= \left(\frac{7}{4}n^2 - \frac{7}{2}n\right) \end{aligned} \quad (4.5.1.11)$$

Four accesses to shared memory are needed to update the (kxk) submatrix,

$$\begin{aligned} M_3 &= \sum_{k=2(2)}^{n-2} (k^2)4 \\ &= \left(\frac{2}{3}n^3 - 6n^2 + \frac{4}{3}n\right) \end{aligned} \quad (4.5.1.12)$$

and another 4 shared memory accesses are required to update the right hand side vector.

$$\begin{aligned} M_4 &= \sum_{k=2(2)}^{n-2} (k)4 \\ &= (n^2 - 2n) \end{aligned} \quad (4.5.1.13)$$

Hence, total shared memory accesses in the elimination stage is

$$\begin{aligned} M_{\text{elim}} &= M_1 + M_2 + M_3 + M_4 \\ &= \frac{2}{3}n^3 - \frac{13}{4}n^2 - \frac{19}{6}n. \end{aligned} \quad (4.5.1.14)$$

In the bi-directional solution process, the solution of the (2x2) linear system requires 9 accesses to shared memory, resulting in a total of

$$M_5 = (9/2)n. \quad (4.5.1.15)$$

The substitution stage of the bi-directional solution process requires 8 accesses to shared memory giving

$$\begin{aligned} M_6 &= \sum_{k=2(2)}^{n-2} (k)8 \\ &= (2n^2 - 4n) \end{aligned} \quad (4.5.1.16)$$

Therefore, the total shared memory accesses in the bi-directional solution process is

$$\begin{aligned} M_{\text{bi-soln}} &= M_5 + M_6 \\ &= \left( 2n^2 + \frac{1}{2}n \right) \end{aligned} \quad (4.5.1.17)$$

and the total shared memory accesses for the PIE method is then

$$\begin{aligned} M_{\text{PIE}} &= M_{\text{elim}} + M_{\text{bi-soln}} \\ &= \frac{2}{3}n^3 - \frac{5}{4}n^2 - \frac{8}{3}n. \end{aligned} \quad (4.5.1.18)$$

#### 4.5.2 Computational Complexity and Shared Memory Access Count for QIF

The work in this section refers to Algorithm 4.5.2.1 for the factorisation of  $A$  into  $W$  and  $Z$ , Algorithm 4.5.2.2 for the bi-directional substitution process and Algorithm 4.5.1.2 for the bi-directional solution procedure.

##### Algorithm 4.5.2.1: Sequential algorithm for WZ factorisation

```

cp1=0, cp2=n-1
for k=0 to (n/2)-1 do
    for i=k+1 to n-k-2 do
        m= -A[cp1,cp2]/A[cp1,cp1]
        { solve the following 2x2 system, to get w's }
        x2=(A[i,cp2]-m*A[i,cp1])/(A[cp2,cp2]-m*A[cp1,cp2])
        x1=(A[i,cp2]-A[cp2,cp2]*x2)/A[cp2,cp1]
        A[i,k]=x1
        A[i,n-1-k]=x2
        for j1=k+1 to n-k do
            { update to form z's }
            A[i,j1]=A[i,j1]-(x1*A[cp1,j1]+x2*A[cp2,j1])
        end for i
        cp1=cp1+1
        cp2=cp2-1
    end for k

```

Algorithm 4.5.2.2: Sequential algorithm for bi-directional substitution

```

for k=0 to (n/2)-1
  for i=k+1 to n-k-2
    A[i,k]=A[k,n]*A[i,k]
    A[i,n-1-k]=A[n-1-k,n]*A[i,n-1-k]
    A[i,n]=A[i,n]-(A[i,k]+A[i,n-1-k])
  end for i
end for k

```

The detailed outline of the computational work involved in QIF is given as follows:

It can be seen that the elements of the first and last rows of the  $Z$  matrix are those of  $A$  and involve no arithmetic operations.

The  $2 \times (n-2)$  elements of the first and last columns of  $W$  are obtained from the solution of  $(2 \times 2)$  equations which involve a total of  $W_1 + W_2$  where  $W_1$  and  $W_2$  are given below:

$$W_1 = (n/2)m \quad (4.5.2.1)$$

$$\begin{aligned}
 W_2 &= \sum_{k=2(2)}^{n-2} k(5m+3a) \\
 &= \left( \frac{n^2}{4} - \frac{n}{2} \right) (5m+3a) \\
 &= \left( \frac{5}{4}n^2 - \frac{5}{2}n \right) m + \left( \frac{3}{4}n^2 - \frac{3}{2}n \right) a
 \end{aligned} \quad (4.5.2.2)$$

The total amount of work involved in evaluating the reduced system is

$$\begin{aligned}
 W_3 &= \sum_{k=2(2)}^{n-2} k^2 (2m+2a) \\
 &= \left( \frac{n^3}{6} - \frac{3}{2}n^2 + \frac{n}{3} \right) (2m+2a) \\
 &= \left( \frac{n^3}{3} - 3n^2 + \frac{2}{3}n \right) m + \left( \frac{n^3}{3} - 3n^2 + \frac{2}{3}n \right) a
 \end{aligned} \quad (4.5.2.3)$$

Therefore, the total amount of work involved in the factorisation process is

$$\begin{aligned}
 W_{WZfactor} &= W_1 + W_2 + W_3 \\
 &= \left( \frac{n^3}{3} - \frac{7}{4}n^2 - \frac{4}{3}n \right) m + \left( \frac{n^3}{3} - \frac{9}{4}n^2 - \frac{5}{6}n \right) a
 \end{aligned} \quad (4.5.2.4)$$

In the bi-directional substitution stage, the value of  $y_1$  and  $y_n$  is given in the right hand side vector and hence needs no arithmetic. The solution of the subsequent values of the vector  $y$  requires 2 mults and 2 adds yielding a total of

$$\begin{aligned}
W_4 &= \sum_{k=2(2)}^{n-2} k(2m+2a) \\
&= \left( \frac{n^2}{4} - \frac{n}{2} \right) (2m+2a) \\
&= \left( \frac{n^2}{2} - n \right) m + \left( \frac{n^2}{2} - n \right) a
\end{aligned} \tag{4.5.2.5}$$

Hence, the total amount of computational work for QIF is

$$\begin{aligned}
W_{\text{QIF}} &= W_{\text{WZfactor}} + W_4 + W_{\text{bi-soln}} \\
&= \left( \frac{n^3}{3} + \frac{n^2}{4} - \frac{4}{3}n \right) n + \left( \frac{n^3}{3} - \frac{3}{4}n^2 - \frac{7}{3}n \right) a
\end{aligned} \tag{4.5.2.6}$$

which, as expected, is the same as the computational work for PIE.

The details of the shared memory access count for the QIF is as follows:

The evaluation of the multiplier to solve the (2x2) linear equation requires 2 accesses to shared memory and this gives a total of

$$M_1 = \left( \frac{n}{2} \right) 2 = n \tag{4.5.2.7}$$

In solving the (2x2) linear equation, 9 shared memory accesses are required, giving a total of

$$\begin{aligned}
M_2 &= \sum_{k=2(2)}^{n-2} (k) 9 \\
&= \left( \frac{9}{4}n^2 - \frac{9}{2}n \right)
\end{aligned} \tag{4.5.2.8}$$

The evaluation of the reduced system requires 4 accesses to shared memory to give a total of

$$\begin{aligned}
M_3 &= \sum_{k=2(2)}^{n-2} (k^2) 4 \\
&= \left( \frac{2}{3}n^3 - 6n^2 + \frac{4}{3}n \right)
\end{aligned} \tag{4.5.2.9}$$

Thus, in the factorisation stage, the total amount of accesses to shared memory is

$$\begin{aligned}
M_{\text{WZfactor}} &= M_1 + M_2 + M_3 \\
&= \frac{2}{3}n^3 - 6n^2 + \frac{4}{3}n
\end{aligned} \tag{4.5.2.10}$$

The bi-directional substitution involves 6 shared memory accesses and this results in a total of

$$M_4 = \sum_{k=2(2)}^{n-2} (k)6 = \left( \frac{3}{2}n^2 - 3n \right) \quad (4.5.2.11)$$

So, the total number of accesses to shared memory involved in the QIF method is therefore

$$M_{QIF} = M_{WZfactor} + M_4 + M_{bi-soln} = \frac{2}{3}n^3 - \frac{1}{4}n^2 - \frac{17}{3}n. \quad (4.5.2.12)$$

This is slightly more to that of PIE due to the accesses required to retrieve and store the elements of the solution vector during the bidirectional substitution process.

### 4.5.3 Computational Complexity and Shared Memory Access Count for GE

The algorithms on which the computational count and shared memory access count for GE is based on are Algorithm 4.5.3.1 for the forward elimination process and Algorithm 4.5.3.2 for the back substitution stage.

#### Algorithm 4.5.3.1 Forward elimination algorithm of GE

```

for k=0 to n-2
    for i=k+1 to n-1
        m=a[i,k]/a[k,k]
        for jr=k+1 to n
            a[jr,jr]=a[i,jr]-m*a[k,jr]
        end for jr
    end for i
end for k

```

#### Algorithm 4.5.3.2 Backsubstitution algorithm of GE

```

for nv=n-1 downto 0
    a[nv,n]=a[nv,n]/a[nv,nv]
    for k= nv+1 to n
        a[nv,n+1]=a[nv,n+1]-a[nv,k]*a[k,n+1]
    end for k
end for nv

```

The detailed computational count for GE is as follows:

The computation of the multipliers in the forward elimination stage requires 1 mult, resulting in a total of

$$\begin{aligned} W_1 &= \sum_{k=1}^{n-1} (k)m \\ &= \left[ \frac{1}{2} n(n-1) \right] m \end{aligned} \quad (4.5.3.1)$$

The updating of the submatrix involves 1 mult and 1 add, giving a total computational count of

$$\begin{aligned} W_2 &= \sum_{k=1}^{n-1} k^2 (m+a) \\ &= \left[ \frac{1}{6} (n-1)n(2n-1) \right] (m+a) \\ &= \left[ \frac{1}{3} n^3 - \frac{1}{2} n^2 + \frac{1}{6} n \right] (m+a) \end{aligned} \quad (4.5.3.2)$$

and the updating of the right hand side vector also requires 1 mult and 1 add.

$$\begin{aligned} W_3 &= \sum_{k=1}^{n-1} k(m+a) \\ &= \left[ \frac{1}{2} n(n-1) \right] (m+a) \end{aligned} \quad (4.5.3.3)$$

Hence, the total amount of computational work involved in the forward elimination is

$$\begin{aligned} W_{\text{elim}} &= W_1 + W_2 + W_3 \\ &= \left( \frac{1}{3} n^3 + \frac{1}{2} n^2 - \frac{5}{6} n \right) m + \left( \frac{1}{3} n^3 - \frac{1}{3} n \right) a \end{aligned} \quad (4.5.3.4)$$

The backsubstitution process requires a total of  $W_4 + W_5$  mults and adds, where  $W_4$  and  $W_5$  are given below:

$$W_4 = n(m) \quad (4.5.3.5)$$

$$\begin{aligned} W_5 &= \sum_{k=1}^{n-1} k(m+a) \\ &= \left( \frac{1}{2} n^2 - \frac{1}{2} n \right) (m+a) \end{aligned} \quad (4.5.3.6)$$

$$\begin{aligned} W_{\text{bs}} &= W_4 + W_5 \\ &= \left( \frac{1}{2} n^2 + \frac{1}{2} n \right) m + \left( \frac{1}{2} n^2 - \frac{1}{2} n \right) a \end{aligned} \quad (4.5.3.7)$$

Therefore, the total amount of computational work involved in GE is

$$\begin{aligned} W_{GE} &= W_{gelim} + W_{bs} \\ &= \left( \frac{1}{3}n^3 + n^2 - \frac{1}{3}n \right)m + \left( \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n \right)a \end{aligned} \quad (4.5.3.8)$$

The detailed shared memory access count for GE is as follows:

The evaluation of the multipliers involve 2 accesses to shared memory and this gives a total of

$$\begin{aligned} M_1 &= \sum_{k=1}^{n-1} k(2) = 2 \left( \frac{1}{2}n^2 - \frac{1}{2}n \right) \\ &= (n^2 - n) \end{aligned} \quad (4.5.3.9)$$

The updating of the submatrix requires 3 accesses to shared memory resulting in a total of

$$\begin{aligned} M_2 &= \sum_{k=1}^{n-1} k^2(3) = 3 \left( \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n \right) \\ &= n^3 - \frac{3}{2}n^2 + \frac{1}{2}n \end{aligned} \quad (4.5.3.10)$$

and similarly, the updating of the right hand side vector also requires 3 accesses to shared memory giving a total of

$$\begin{aligned} M_3 &= \sum_{k=1}^{n-1} k(3) \\ &= \left( \frac{3}{2}n^2 - \frac{3}{2}n \right) \end{aligned} \quad (4.5.3.11)$$

Hence, the total shared memory access count for the forward elimination stage is

$$\begin{aligned} M_{gelim} &= M_1 + M_2 + M_3 \\ &= n^3 + n^2 - 2n \end{aligned} \quad (4.5.3.12)$$

The back substitution stage requires a total of  $M_4 + M_5$  accesses to shared memory where  $M_4$  and  $M_5$  is as given below:

$$M_4 = n(3) \quad (4.5.3.13)$$

$$\begin{aligned} M_5 &= \sum_{k=1}^{n-1} k(4) \\ &= 2n^2 - 2n \end{aligned} \quad (4.5.3.14)$$

$$\begin{aligned} M_{bs} &= M_4 + M_5 \\ &= 2n^2 + n \end{aligned} \quad (4.5.3.15)$$

Thus, the total count of shared memory accesses involved in GE is

$$\begin{aligned} M_{GE} &= M_{gelim} + M_{bs} \\ &= n^3 + 3n^2 - n \end{aligned} \quad (4.5.3.16)$$

#### 4.5.4 Computational Complexity and Shared Memory Access Count for LU

The algorithms on which the computational count and shared memory access count are based on are Algorithm 4.5.4.1 for the factorisation process, Algorithm 4.5.4.2 for the forward substitution process and Algorithm 4.5.3.2 for the backsubstitution process.

##### Algorithm 4.5.4.1 Factorisation stage in LU

```

for k = 0 to n-2
    for i=k+1 to n-1
        m=a[i,k]/a[k,k]
        a[i,k] = m
        for jr=k+1 to n-1
            a[i,jr] = a[i,jr] - m * a[k,jr]
        end for jr
    end for i
end for k

```

##### Algorithm 4.5.4.2 Forward substitution

```

for nv = 1 to n-1
    for j = 0 to nv -1
        a[nv,n] = a[nv,n] - a[nv,j] * a[j,n]
    end for j
end for nv

```

The formation of the  $L$  matrix elements involve a total of

$$\begin{aligned} W_1 &= \sum_{k=1}^{n-1} (k)m \\ &= \left[ \frac{1}{2}n(n-1) \right] m \end{aligned} \quad (4.5.4.1)$$

while the formation of the  $U$  matrix elements involve a total of

$$\begin{aligned} W_2 &= \sum_{k=1}^{n-1} k^2(m+a) \\ &= \left[ \frac{1}{6}(n-1)n(2n-1) \right] (m+a) \\ &= \left[ \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n \right] (m+a) \end{aligned} \quad (4.5.4.2)$$



Therefore, the amount of computational work involved in the factorisation phase is

$$\begin{aligned} W_{LUfactor} &= W_1 + W_2 \\ &= \left(\frac{1}{3}n^3 - \frac{1}{3}n\right)m + \left(\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n\right)a \end{aligned} \quad (4.5.4.3)$$

The forward substitution stage requires 1 mult and 1 add giving a total of

$$\begin{aligned} W_3 &= \sum_{k=1}^{n-1} k(m+a) \\ &= \left[\frac{1}{2}n(n-1)\right](m+a) \end{aligned} \quad (4.5.4.4)$$

Therefore, the total computational work required to perform LU factorisation is

$$\begin{aligned} W_{LU} &= W_{LUfactor} + W_3 + W_{bs} \\ &= \left(\frac{1}{3}n^3 + n^2 - \frac{1}{3}n\right)m + \left(\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n\right)a \end{aligned} \quad (4.5.4.5)$$

which, as expected, is the same as the computational work of GE.

The detailed shared-memory access count for LU factorisation is as follows:

The total amount of shared memory accesses required in forming the elements of the  $L$  matrix is

$$\begin{aligned} M_1 &= \sum_{k=1}^{n-1} k(3) \\ &= \frac{3}{2}n^2 - \frac{3}{2}n \end{aligned} \quad (4.5.4.6)$$

while the formation of the  $U$  matrix needs a total of

$$\begin{aligned} M_2 &= \sum_{k=1}^{n-1} k^2(3) \\ &= n^3 - \frac{3}{2}n^2 + \frac{1}{2}n \end{aligned} \quad (4.5.4.7)$$

accesses to shared memory. Therefore, the total count of shared memory accesses required to perform the factorisation phase is

$$\begin{aligned} M_{LUfactor} &= M_1 + M_2 \\ &= n^3 - n \end{aligned} \quad (4.5.4.8)$$

The forward substitution requires a total of

$$M_3 = \sum_{k=1}^{n-2} k(4) \quad (4.5.4.9)$$

$$= 2n^2 - 2n$$

accesses to shared memory. Hence, the total amount of shared memory accesses involved in LU factorisation is

$$M_{LU} = M_{LUfactor} + M_3 + M_{bs}$$

$$= n^3 + 4n^2 - 2n. \quad (4.5.4.10)$$

This is slightly more than GE due to the accesses to memory required during the forward substitution stage.

#### 4.5.5 A Summary

A summary of the operation count and shared memory access count of all the four methods, GE, LU, PIE and QIF are given in the following Table 4.5.5.1.

| Method | Computational Complexity   | Shared Memory Access Count                        |
|--------|--|---|
| PIE    | $\left(\frac{n^3}{3} + \frac{n^2}{4} - \frac{4}{3}n\right)m + \left(\frac{n^3}{3} - \frac{3}{4}n^2 - \frac{7}{3}n\right)a$ | $\frac{2}{3}n^3 - \frac{5}{4}n^2 - \frac{8}{3}n$  |
| GE     | $\left(\frac{1}{3}n^3 + n^2 - \frac{1}{3}n\right)m + \left(\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n\right)a$         | $n^3 + 3n^2 - n$                                  |
| QIF    | $\left(\frac{n^3}{3} + \frac{n^2}{4} - \frac{4}{3}n\right)m + \left(\frac{n^3}{3} - \frac{3}{4}n^2 - \frac{7}{3}n\right)a$ | $\frac{2}{3}n^3 - \frac{1}{4}n^2 - \frac{17}{3}n$ |
| LU     | $\left(\frac{1}{3}n^3 + n^2 - \frac{1}{3}n\right)m + \left(\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n\right)a$         | $n^3 + 4n^2 - 2n$                                 |

Table 4.5.5.1 : A Summary of the computational complexity and shared memory access count of PIE, GE, QIF and LU

From Table 4.5.5.1, it can be seen that there is a difference in the shared memory access count of PIE and QIF as compared to GE and LU. This difference will have a significant influence on the performance of the algorithms as shown in section 4.6 where the numerical results of the implementation of the algorithms are presented.

#### 4.6 Numerical Results

In this section the performance results obtained from the implementation of PIE and QIF are presented. These results are also compared to their equivalent counterparts, that is, PIE is compared to GE and QIF is compared with LU factorisation.

The numerical tests were performed on the Sequent Balance system which has 10 processors. All the processors are plugged into a single bus and share a common memory. Each processor has 8 Kb of cache memory.

The algorithms were implemented in single precision using the C language. The parallel constructs were supported by the Sequent C library. The programs employed a static scheduling of tasks, i.e. distributing the computation load amongst the processors before execution of the program commences.

In order to avoid the overhead of switching from one process to another, only a single process was assigned to each processor and the tests were completed while no other users' tasks were using the Balance. All the programs were written with the same accuracy to obtain a meaningful comparison.

For methods without pivoting, the matrix  $A$  is,

$$A = (a_{ij}) \text{ for } i=1,2,\dots,n; j=1,2,\dots,n,$$

$$a_{i,i}=n; a_{i,j}=1.0$$

for  $n=600,800$  and  $1000$ .

For methods involving partial pivoting, the matrix  $A$  is,

$$A=(a_{ij}), \text{ for } i=1,2,\dots,n; j=1,2,\dots,n,$$

where  $a_{ij}$  is a random number uniformly distributed between 0 and 1 and  $n=600,800$  and  $1000$ .

The execution times of the algorithms are shown in section 4.6.1. The speedup figures of the algorithms are given in section 4.6.2 and the efficiency figures are in section 4.6.3.

As the aim of this work is to compare different algorithms to solve the same problem, the temporal performance metric is used to aid the comparison. Temporal performance is defined as the inverse of the execution time where the unit is solutions per second or timesteps per second [Hockney 96]. The algorithm with the highest performance executes in the least time and therefore is the better algorithm. The temporal performance results of the algorithms are shown in section 4.6.4.

#### 4.6.1 Execution time of PIE, QIF, GE and LU

The execution time of the algorithms were measured in seconds and input/output was not included in the timing. Table 4.6.1.1 shows the execution times of PIE and GE without partial pivoting while Table 4.6.1.2 shows the execution times of QIF and LU without partial pivoting.

It can be seen from the tables that the gains in execution time of PIE over GE ranges from 21% up to 25%. The gains of QIF over LU ranges from 19% up to 24%.

| n    | Method  | Number of processors |          |         |         |         |         |
|------|---------|----------------------|----------|---------|---------|---------|---------|
|      |         | 1                    | 2        | 4       | 6       | 8       | 10      |
| 600  | GE      | 6042.31              | 3075.69  | 1554.68 | 1083.34 | 789.95  | 634.43  |
|      | PIE     | 4723.65              | 2349.42  | 1170.46 | 785.62  | 590.76  | 476.54  |
|      | Gain(%) | 21.82                | 23.61    | 24.71   | 27.48   | 25.22   | 24.89   |
| 800  | GE      | 14343.94             | 7236.38  | 3615.13 | 2433.25 | 1840.41 | 1481.6  |
|      | PIE     | 11103.08             | 5587.16  | 2786.84 | 1858.75 | 1399.07 | 1126.6  |
|      | Gain(%) | 22.59                | 22.79    | 22.91   | 23.61   | 23.98   | 23.96   |
| 1000 | GE      | 27985.72             | 13994.19 | 7094.67 | 4751.07 | 3573.04 | 2877.23 |
|      | PIE     | 21610.31             | 10891.04 | 5439.09 | 3622.95 | 2708.0  | 2216.1  |
|      | Gain(%) | 22.78                | 22.17    | 23.33   | 23.74   | 24.18   | 22.98   |

Table 4.6.1.1 - Timings of PIE and GE for the non-pivoting case

| n    | Method  | Number of processors |          |         |         |         |         |
|------|---------|----------------------|----------|---------|---------|---------|---------|
|      |         | 1                    | 2        | 4       | 6       | 8       | 10      |
| 600  | LU      | 5991.74              | 3023.66  | 1534.35 | 1031.03 | 779.76  | 627.30  |
|      | QIF     | 4697.27              | 2367.95  | 1203.03 | 800.55  | 606.45  | 488.44  |
|      | Gain(%) | 21.60                | 21.69    | 21.59   | 22.35   | 22.22   | 22.14   |
| 800  | LU      | 14165.8              | 7160.04  | 3610.41 | 2416.75 | 1876.83 | 1465.49 |
|      | QIF     | 11037.56             | 5553.72  | 2815.15 | 1892.31 | 1425.41 | 1146.78 |
|      | Gain(%) | 22.08                | 22.43    | 22.03   | 21.7    | 24.05   | 21.75   |
| 1000 | LU      | 27828.56             | 13872.24 | 6964.23 | 4710.75 | 3542.68 | 2857.4  |
|      | QIF     | 21611.26             | 10901.26 | 5631.56 | 3677.82 | 2764.56 | 2221.7  |
|      | Gain(%) | 22.34                | 21.42    | 19.14   | 21.93   | 21.96   | 22.25   |

Table 4.6.1.2 - Timings for QIF and LU for the non-pivoting case

The execution times of the pivoting cases are shown in Table 4.6.1.3 for PIE and GE and Table 4.6.1.4 for QIF and LU. The gains of PIE over GE ranges from 18% up to 22% . The gains of QIF over LU were less encouraging, ranging from 13% up to 18%.

| n    | Method  | Number of processors |          |         |         |         |         |
|------|---------|----------------------|----------|---------|---------|---------|---------|
|      |         | 1                    | 2        | 4       | 6       | 8       | 10      |
| 600  | GE      | 6192.65              | 3050.30  | 1553.36 | 1045.99 | 794.88  | 641.83  |
|      | PIE     | 4907.95              | 2474.03  | 1227.94 | 816.02  | 616.07  | 499.72  |
|      | Gain(%) | 20.75                | 18.89    | 20.95   | 21.99   | 22.5    | 22.14   |
| 800  | GE      | 14435.24             | 7219.87  | 3643.9  | 2443.15 | 1836.82 | 1488.04 |
|      | PIE     | 11442.58             | 5744.95  | 2874.23 | 1915.88 | 1444.18 | 1158.25 |
|      | Gain(%) | 20.73                | 20.43    | 21.12   | 21.58   | 21.38   | 22.16   |
| 1000 | GE      | 27958.15             | 14073.87 | 7057.90 | 4799.10 | 3567.92 | 2874.78 |
|      | PIE     | 22169.05             | 11082.29 | 5567.83 | 3698.81 | 2794.91 | 2243.72 |
|      | Gain(%) | 20.71                | 21.26    | 21.11   | 22.93   | 21.67   | 21.95   |

Table 4.6.1.3 - Timings for PIE and GE for the case of partial pivoting

| n    | Method  | Number of processors |          |         |         |         |         |
|------|---------|----------------------|----------|---------|---------|---------|---------|
|      |         | 1                    | 2        | 4       | 6       | 8       | 10      |
| 600  | LU      | 6809.24              | 3465.62  | 1760.04 | 1212.01 | 934.75  | 773.38  |
|      | QIF     | 5858.32              | 2906.02  | 1472.86 | 996.98  | 760.0   | 620.42  |
|      | Gain(%) | 13.97                | 16.15    | 16.32   | 17.74   | 18.69   | 19.78   |
| 800  | LU      | 16191.19             | 8120.19  | 4119.85 | 2791.57 | 2114.54 | 1720.77 |
|      | QIF     | 13783.3              | 6835.15  | 3451.94 | 2335.13 | 1766.19 | 1427.23 |
|      | Gain(%) | 14.87                | 15.83    | 16.21   | 16.35   | 16.47   | 17.06   |
| 1000 | LU      | 31744.92             | 15931.08 | 8086.32 | 5419.52 | 4106.13 | 3323.11 |
|      | QIF     | 26960.81             | 13393.84 | 6754.35 | 4555.96 | 3443.06 | 2769.18 |
|      | Gain(%) | 15.07                | 15.93    | 16.47   | 15.93   | 16.15   | 16.67   |

Table 4.6.1.4 - Timings for QIF and LU for the case of partial pivoting

## 4.6.2 Speedup of PIE, QIF, GE and LU

The speedup of the algorithms were measured by using the formula  $T_1/T_p$  where  $T_1$  is the execution time of the algorithm executed using 1 processor. Table 4.6.2.1 shows the speedup of PIE and GE without partial pivoting while Table 4.6.2.2 shows the speedup of QIF and LU without partial pivoting. It can be seen from the speedup figures that all the methods have a linear speedup, although in some cases superlinear speedups were obtained.

| n    | Method | Number of processors |      |      |      |      |
|------|--------|----------------------|------|------|------|------|
|      |        | 2                    | 4    | 6    | 8    | 10   |
| 600  | GE     | 1.96                 | 3.89 | 5.58 | 7.65 | 9.52 |
| 800  |        | 1.98                 | 3.97 | 5.89 | 7.93 | 9.68 |
| 1000 |        | 1.99                 | 3.94 | 5.89 | 7.83 | 9.73 |
| 600  | PIE    | 2.01                 | 4.04 | 6.01 | 7.99 | 9.91 |
| 800  |        | 1.99                 | 3.98 | 5.97 | 7.94 | 9.86 |
| 1000 |        | 1.98                 | 3.97 | 5.96 | 7.98 | 9.75 |

Table 4.6.2.1- Speedup of PIE and GE for the non-pivoting case

| $n$  | Method | Number of processors |      |      |      |      |
|------|--------|----------------------|------|------|------|------|
|      |        | 2                    | 4    | 6    | 8    | 10   |
| 600  | LU     | 1.98                 | 3.91 | 5.81 | 7.68 | 9.55 |
| 800  |        | 1.98                 | 3.92 | 5.86 | 7.55 | 9.67 |
| 1000 |        | 2.0                  | 3.99 | 5.91 | 7.86 | 9.74 |
| 600  | QIF    | 1.98                 | 3.9  | 5.87 | 7.75 | 9.62 |
| 800  |        | 1.99                 | 3.92 | 5.83 | 7.74 | 9.62 |
| 1000 |        | 1.98                 | 3.84 | 5.88 | 7.82 | 9.72 |

Table 4.6.2.2 - Speedup of QIF and LU for the non-pivoting case

| $n$  | Method | Number of processors |      |      |      |      |
|------|--------|----------------------|------|------|------|------|
|      |        | 2                    | 4    | 6    | 8    | 10   |
| 600  | GE     | 2.03                 | 3.99 | 5.92 | 7.79 | 9.65 |
| 800  |        | 1.99                 | 3.96 | 5.91 | 7.86 | 9.7  |
| 1000 |        | 1.99                 | 3.96 | 5.83 | 7.84 | 9.73 |
| 600  | PIE    | 1.98                 | 3.99 | 6.01 | 7.97 | 9.82 |
| 800  |        | 1.99                 | 3.98 | 5.97 | 7.92 | 9.88 |
| 1000 |        | 2.0                  | 3.98 | 5.99 | 7.93 | 9.88 |

Table 4.6.2.3 - Speedup of PIE and GE for the case of partial pivoting

| $n$  | Method | Number of processors |      |      |      |      |
|------|--------|----------------------|------|------|------|------|
|      |        | 2                    | 4    | 6    | 8    | 10   |
| 600  | LU     | 1.96                 | 3.87 | 5.62 | 7.28 | 8.8  |
| 800  |        | 1.99                 | 3.93 | 5.8  | 7.66 | 9.41 |
| 1000 |        | 1.99                 | 3.93 | 5.86 | 7.73 | 9.55 |
| 600  | QIF    | 2.01                 | 3.98 | 5.88 | 7.7  | 9.44 |
| 800  |        | 2.01                 | 3.99 | 5.9  | 7.8  | 9.66 |
| 1000 |        | 2.01                 | 3.99 | 5.92 | 7.83 | 9.64 |

Table 4.6.2.4 - Speedup of QIF and LU for the case of partial pivoting

### 4.6.3 Efficiency of PIE, QIF, GE and LU

The efficiency figures of PIE and GE without partial pivoting are shown in Table 4.6.3.1 while that of QIF and LU without partial pivoting are shown in Table 4.6.3.2. Table 4.6.3.3 shows the efficiency figures for PIE and GE with partial pivoting and Table 4.6.3.4 shows the efficiency figures for QIF and LU with partial pivoting.

| $n$  | Method | Number of processors |        |       |        |       |
|------|--------|----------------------|--------|-------|--------|-------|
|      |        | 2                    | 4      | 6     | 8      | 10    |
| 600  | GE     | 0.98                 | 0.9725 | 0.93  | 0.9563 | 0.952 |
| 800  |        | 0.99                 | 0.9925 | 0.982 | 0.9913 | 0.968 |
| 1000 |        | 0.995                | 0.985  | 0.982 | 0.9788 | 0.973 |
| 600  | PIE    | 1.005                | 1.01   | 1.001 | 0.9988 | 0.991 |
| 800  |        | 0.995                | 0.995  | 0.995 | 0.9925 | 0.986 |
| 1000 |        | 0.99                 | 0.9925 | 0.993 | 0.9975 | 0.975 |

Table 4.6.3.1 - Efficiency of PIE and GE for the non-pivoting case

| n    | Method | Number of processors |        |        |        |       |
|------|--------|----------------------|--------|--------|--------|-------|
|      |        | 2                    | 4      | 6      | 8      | 10    |
| 600  | LU     | 0.99                 | 0.9775 | 0.9683 | 0.96   | 0.955 |
| 800  |        | 0.99                 | 0.98   | 0.9766 | 0.944  | 0.967 |
| 1000 |        | 1.0                  | 0.9975 | 0.985  | 0.9825 | 0.974 |
| 600  | QIF    | 0.99                 | 0.975  | 0.978  | 0.968  | 0.962 |
| 800  |        | 0.995                | 0.98   | 0.972  | 0.9675 | 0.962 |
| 1000 |        | 0.99                 | 0.96   | 0.98   | 0.9775 | 0.973 |

Table 4.6.3.2 - Efficiency of QIF and LU for the non-pivoting case

| n    | Method | Number of processors |        |       |       |       |
|------|--------|----------------------|--------|-------|-------|-------|
|      |        | 2                    | 4      | 6     | 8     | 10    |
| 600  | GE     | 1.01                 | 0.9975 | 0.987 | 0.974 | 0.965 |
| 800  |        | 0.995                | 0.99   | 0.985 | 0.983 | 0.97  |
| 1000 |        | 0.995                | 0.99   | 0.972 | 0.98  | 0.973 |
| 600  | PIE    | 0.99                 | 0.9975 | 1.0   | 0.996 | 0.982 |
| 800  |        | 0.995                | 0.995  | 0.995 | 0.99  | 0.988 |
| 1000 |        | 1.0                  | 0.995  | 0.998 | 0.991 | 0.988 |

Table 4.6.3.3 - Efficiency of PIE and GE for the case of partial pivoting

| n    | Method | Number of processors |       |       |       |       |
|------|--------|----------------------|-------|-------|-------|-------|
|      |        | 2                    | 4     | 6     | 8     | 10    |
| 600  | LU     | 0.98                 | 0.968 | 0.937 | 0.91  | 0.88  |
| 800  |        | 0.995                | 0.983 | 0.967 | 0.958 | 0.941 |
| 1000 |        | 0.995                | 0.983 | 0.977 | 0.966 | 0.955 |
| 600  | QIF    | 1.005                | 0.995 | 0.98  | 0.963 | 0.944 |
| 800  |        | 1.005                | 0.998 | 0.983 | 0.975 | 0.966 |
| 1000 |        | 1.005                | 0.998 | 0.987 | 0.979 | 0.964 |

Table 4.6.3.4 -Efficiency of QIF and LU for the case of partial pivoting

It can be seen that the effectiveness with which the processors are utilised for all the methods is roughly 90%.

#### 4.6.4 Temporal Performance of PIE, QIF, GE and LU

The temporal performance of the algorithms are shown graphically for the case of  $n=600$ . Figure 4.6.4.1 shows the temporal performance of PIE vs. GE without partial pivoting and Figure 4.6.4.2 shows the temporal performance of QIF vs. LU without partial pivoting. For cases with partial pivoting, the temporal performance of PIE vs. GE is shown in Figure 4.6.4.3 while that of QIF vs. LU is shown in Figure 4.6.4.4.

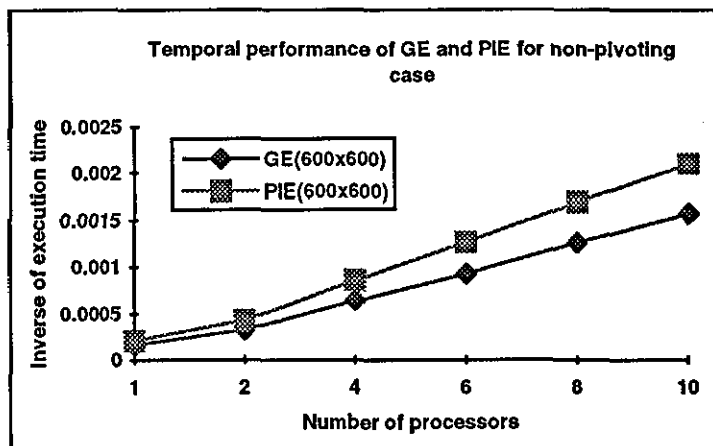


Figure 4.6.4.1 - Temporal performance of GE and PIE without pivoting

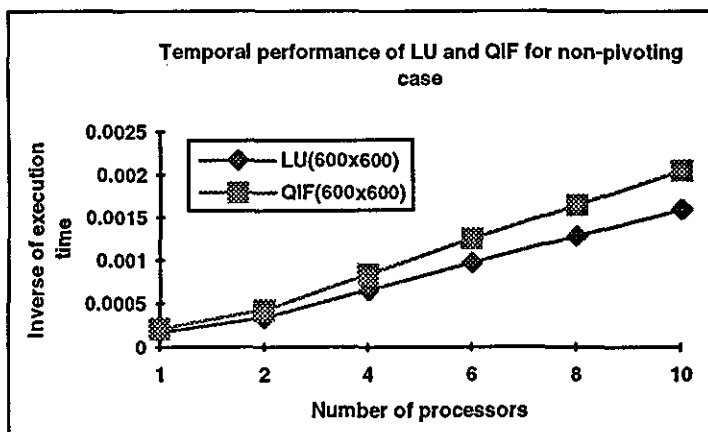


Figure 4.6.4.2 - Temporal performance of LU and QIF without pivoting

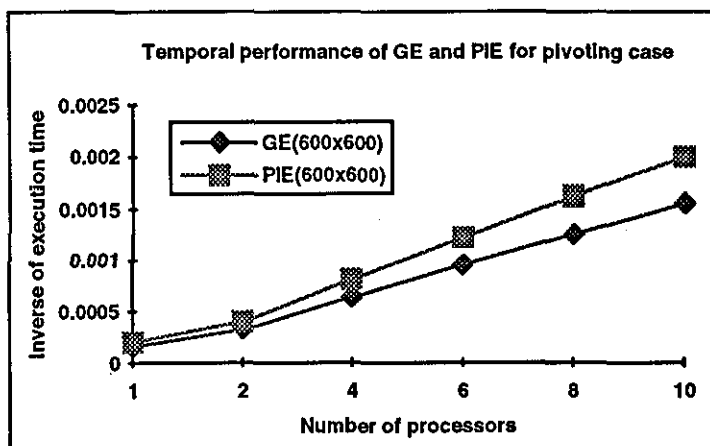


Figure 4.6.4.3 - Temporal performance of GE and PIE with pivoting



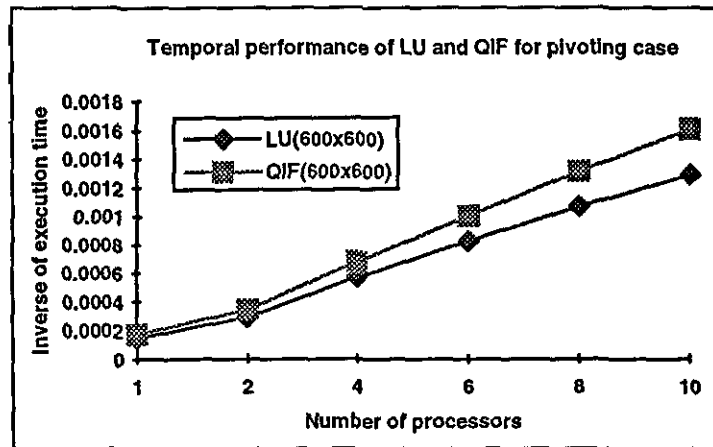


Figure 4.6.4.4 - Temporal performance of LU and QIF with pivoting

#### 4.7 Summary

In this chapter, the Parallel Implicit Elimination (PIE) method and the Quadrant Interlocking Factorisation (QIF) method have been discussed from the formulation of the method, the sequential and parallel algorithms for a shared memory parallel computer and their numerical results obtained from the implementation on the Sequent Balance, a shared memory parallel computer.

As the main aim of this research is to compare the performance of PIE and QIF with that of GE and LU respectively, the Sequent Balance, although an old machine, is sufficient for this purpose. Since all the algorithms are tested on the same machine, they should yield consistent results.

In general, it can be seen from the performance results of the algorithms implemented on the Sequent Balance that the parallel algorithms PIE and QIF are superior than their counterparts GE and LU respectively.

For the non-pivoting cases, the timing of PIE is between 21%-23% better than GE, while the timing of QIF is 20%-22% better than LU. For the pivoting cases, PIE is between 20%-22% better than GE while QIF is between 13%-17% faster than LU. Again, even with pivoting, the timings of PIE and QIF maintain the superiority as was in the non-pivoting case. However, the gains obtained from QIF over LU in the pivoting cases are less encouraging.

Although the computational complexity for all the algorithms were of order  $O(n^3)$ , and hence the timings too were expected to be roughly the same, there seemed to be a significant difference of roughly 20% between the timings of GE and PIE and LU and

QIF. These gains could have been contributed by the fact that in PIE and QIF, there were less accesses to shared memory as there were in the case of GE and LU as was shown in Table 4.5.5.1, i.e. PIE and QIF only required  $2/3n^3$  accesses to shared memory while GE and LU required  $n^3$  accesses to shared memory.

In terms of the temporal performance metric of the algorithms, PIE and QIF have also demonstrated to be superior than GE and LU respectively for both the pivoting and non-pivoting cases. It is apparent then that PIE and QIF performs more computational work in a unit of time as compared to GE and LU.

PIE and QIF are in fact 2x2 block forms expressed in explicit point form/notation and should be compared with BLAS. Since BLAS was not available at the start of this work, the experimental comparison was not considered then. However, the BLAS overheads would need to be considered and even the pivoting by blocks is not satisfactorily resolved in the literature. These disadvantages might outweigh the advantage of easier programming that BLAS has to offer. Moreover, it is the purpose of PIE and QIF to express a block algorithm in explicit point form, thus saving the BLAS overheads and consequently obtaining a more efficient algorithm.

## Chapter 5

### Parallel Implicit Elimination (PIE) and Quadrant Interlocking Factorisation (QIF) on Distributed Memory Architecture

The design and implementation of PIE and QIF on a shared memory architecture has been discussed in Chapter 4. One important objective that parallel computing has yet to achieve is the portability of parallel programs. At the moment, even programs written for a particular shared memory machine cannot be run on another shared memory machine without some modifications. Porting a parallel program from a shared memory machine to a distributed memory machine needs a major rethinking. The algorithm design is slightly different to that of a shared-memory programming model. The major difference lies in the sharing of data where it is achieved via message-passing while shared data is placed in the shared memory area of the shared memory parallel computer.

There has been some work done on the development of QIF on distributed memory parallel computers. Asenjo et. al [Asenjo 93] has developed the parallel QIF factorisation on mesh multiprocessors. They have achieved half the messages of the equivalent parallel LU algorithm, but no numerical results of this comparison was given. Their aim was to compare the implementation of QIF on multiprocessors with mesh topology and hypercube topology.

Garcia et. al [Garcia 90] investigated the parallelisation of the QIF algorithm on a SIMD hypercube computer and compared the algorithm with the parallel algorithm based on the LU factorisation. The results they obtained showed that the execution time is reduced by a factor of two, approximately. [Saeed 92] has also implemented the QIF on a hypercube multiprocessor.

Pivoting is a widely used technique for improving the numerical stability of matrix factorisation. While pivoting is almost a trivial operation on sequential machines, it can require a fairly complicated algorithm in the concurrent implementation [Fox 88]. In particular, this is true in a distributed memory computer where the matrix decomposition has a relevant impact on the performance of the pivot search. In almost all the decompositions partial pivoting does not perform well due to inherent load unbalancing which is due to an unfortunate sequence of pivot choices and/or from the

inactivity of the processors during the pivoting stage. Angelaccio et al [Angelaccio 94] have introduced a new pivoting strategy (row/column pivoting) that assigns extra work to idle processors thus reducing load unbalancing and assuring a better stability.

This chapter discusses the design and implementation of PIE and QIF on distributed memory architectures. Section 5.1 discusses cluster computing and one of the many software simulation systems employed in cluster computing today i.e. Parallel Virtual Machine (PVM). In section 5.2 the design of PIE and QIF for a distributed-memory architecture is presented. Two very important factors affecting performance in message-passing concurrent systems are communication patterns and computation-to-communication ratios. In section 5.3 a theoretical analysis of the communication involved in PIE, QIF, GE and LU are shown followed by the predicted communication times for the algorithms in section 5.4. The numerical results which include the measured communication time to be compared against the predicted communication time is given in section 5.5. Lastly, the discussion and summary of the chapter follows in section 5.6.

The aim of the work in this chapter is to show that parallel implicit solvers, PIE and QIF, are better suited for message-passing systems as compared to the classical linear solvers, GE and LU respectively, in terms of amount of communication involved. This is achieved by firstly performing a theoretical analysis of communication within each algorithm. Next, the measured communication times are taken by using the wall clock times of the computer. Lastly, the trend of the measured communication time should conform to the trend of the theoretical prediction.

## **5.1 Cluster Computing and PVM**

Computer clusters may be defined as systems of networked computers, each with its own memory and may be composed of collections of heterogeneous or homogeneous computers of various sizes and types.

Concurrent processing is the use of several working entities (either identical or heterogeneous) working together toward a common goal [Fox 88]. Concurrency is achieved by domain decomposition where data is divided into parts and one part is assigned to each computer.

Cluster computing or concurrent computing on a collection of loosely coupled computer systems, is a rapidly evolving technology with tremendous potential for high performance applications. Such systems allow a collection of networked machines to be used as a unified, general purpose, concurrent computing resource. They are effective platforms for highly compute-intensive applications and can complement or augment the processing capabilities of traditional parallel machines. Most contemporary computing environments are networked based. A common configuration is a network of stand-alone general purpose workstations, different kinds of multiprocessors and a few special purpose machines such as a graphics processor or vector computers.

Cluster computing is expected to gain widespread attention in the future because it is a cost-effective way to use the collective power of systems which would otherwise be used in a single-processor mode. Furthermore, they are widely available and provide straight forward access interfaces as well as development monitoring tools.

Many applications show significant parallel speedups on clusters, though overall performance is still well below that of the more traditional vector supercomputer. One distinct advantage of clusters is the availability of considerable memory. Furthermore, clusters can be used as a platform to develop applications for massively parallel processors having a distributed memory architecture. Message-passing libraries, such as the PVM package, that enable passing of data between processors, are available on MPP's as well as on clusters.

There are several systems available that support cluster-based concurrent computing. Among these are the PVM system, MPI, Network Linda, and the Express environment. In this work, PVM has been used as it was more widely used at the start of this research.

The PVM system emulates a general-purpose concurrent computing framework on a networked collection of independent computer systems. A system level process which executes on each machine in the user-configurable network of processing elements and through co-operative distributed algorithms, allows this collection of machines to be used as a coherent concurrent computing resource. Applications access this virtual machine via library routines embedded in imperative procedural languages, such as C or Fortran. PVM provides primitives for concurrent process management, and for

communication and synchronisation of processes in a heterogeneous distributed computing environment.

Processes of parallel application distributed over a collection of processors must communicate problem parameters and results. In distributed memory multiprocessors or workstations on a network, the information is typically communicated with explicit message-passing subroutine calls. To send data to another process, a subroutine usually requires a destination address, message and message length. The receiving process usually provides a buffer, a maximum length and the sender's address. The PVM communication model provides asynchronous blocking send, asynchronous blocking receive and non-blocking receive functions. A blocking send returns as soon as the send buffer is free for reuse regardless of the state of the receiver. A non-blocking receive immediately returns with either a data or a flag that the data has not arrived. A blocking receive returns only when the data is in the receive buffer. In addition to these point-to-point communication functions, the model also supports multicast to a set of tasks and broadcast to a user-defined group of tasks. The PVM model guarantees that message order is preserved between any pair of communicating entities.

PVM supports two kinds of message-passing, one using datagram (connectionless) protocols, where data is routed via system daemons while the other is based on stream (connection-oriented) transport which is direct communication between the processes.

PVM and similar systems normally operate in general purpose networked environments, where neither the CPU's of the individual machines nor the interconnection network is dedicated. As a result, raw performance or speedup of a given application is hard to measure [Schmidt 95]. Even in a dedicated networked environment, with no external use, the above is true since operating system activity, window and filesystem overheads and administrative network traffic can contribute to deviated measurements.

PVM is not merely a software framework for network-based concurrent computing; it is an integrated methodology for concurrent, distributed and parallel processing, and more importantly, it is an interface definition for portable application development [Sunderam 94a]. From the portability point of view, PVM applications may be

migrated not just from one machine (parallel or serial) to another but across different collections of machines.

## 5.2 Implementation of PIE and QIF in PVM

Both PIE and QIF have also been designed and implemented on a message-passing model using PVM on workstation clusters. In this section the design of the PIE and QIF algorithms for message-passing architectures is discussed. The master-slave paradigm has been chosen as the programming model.

In order to attain a good load balancing, data is partitioned by rows in a wrap around/cyclic fashion. In order for a slave program to perform elimination or factorisation with minimal communication, it is advisable that rows  $i$  and row  $n-1+i$  reside in the same slave program. For this reason, the data is partitioned such that for every value of  $i$  from 0 to  $n/2-1$ , row  $i$  and the corresponding symmetric row  $n-1+i$  is distributed to each slave in a cyclic manner. In other words, each slave will store rows which are symmetric with respect to the central axis of the matrix. The partitioning of data for a 6x6 matrix to 2 slaves is illustrated in Figure 5.2.1. Rows of the same shade are symmetric rows that are assigned to the same slave program.

|          |          |          |          |          |          |       |                        |
|----------|----------|----------|----------|----------|----------|-------|------------------------|
| $a_{11}$ | $a_{12}$ | $a_{13}$ | $a_{14}$ | $a_{15}$ | $a_{16}$ | row 0 | -> assigned to slave 0 |
| $a_{21}$ | $a_{22}$ | $a_{23}$ | $a_{24}$ | $a_{25}$ | $a_{26}$ | row 1 | -> assigned to slave 1 |
| $a_{31}$ | $a_{32}$ | $a_{33}$ | $a_{34}$ | $a_{35}$ | $a_{36}$ | row 2 | -> assigned to slave 0 |
| $a_{41}$ | $a_{42}$ | $a_{43}$ | $a_{44}$ | $a_{45}$ | $a_{46}$ | row 3 | -> assigned to slave 0 |
| $a_{51}$ | $a_{52}$ | $a_{53}$ | $a_{54}$ | $a_{55}$ | $a_{56}$ | row 4 | -> assigned to slave 1 |
| $a_{61}$ | $a_{62}$ | $a_{63}$ | $a_{64}$ | $a_{65}$ | $a_{66}$ | row 5 | -> assigned to slave 0 |

Figure 5.2.1 Partitioning of 6x6 matrix to 2 slaves

In this implementation the matrix is generated within each slave, so only the index of the rows of the matrix  $A$  that each slave is responsible for is sent by the master process to each slave process. Each slave has 2 two-dimensional arrays used to store the upper half of the matrix and the corresponding symmetric row (i.e. the lower half of the matrix). The right-hand side vector is augmented to each of these two arrays.

For notation purposes, the array storing rows from the top half of the coefficient matrix  $A$  is called  $tp$  and the array storing the rows from the bottom half of the coefficient matrix  $A$  is called  $bt$ . The one dimensional array storing the middle row for

an odd-sized coefficient matrix  $A$  is called  $md$ . These arrays  $tp$ ,  $bt$ , and  $md$  which originally store rows from the coefficient matrix  $A$  will be used to keep the  $Z$  matrix for PIE and the  $W$  and  $Z$  factors for QIF. The solution vector  $x$  is overwritten on the right-hand side vector  $b$ . The indices of the rows of the coefficient matrix  $A$ , when sent to the slaves, are kept in single dimensional arrays.

The algorithm for the master program of PIE and QIF to partition the matrix and send them to the appropriate slaves is shown in Algorithm 5.2.1.

The algorithm to factorise the coefficient matrix  $A$  into  $W$  and  $Z$ , performed by each slave program of QIF, is shown in Algorithm 5.2.2. (Let depth of matrix =  $n/2$  if  $n$  is odd and  $n/2-1$  if  $n$  is even.)

The solution stage of QIF consists of the bi-directional substitution process and the algorithm is given in Algorithm 5.2.3. This is then followed by the bi-directional solution process and the algorithm is shown in Algorithm 5.2.4. The algorithm for the elimination stage of PIE is given in Algorithm 5.2.5.

Since the performance of GE and LU will be compared to that of PIE and QIF respectively, the algorithms for GE and LU for message-passing systems are also given. The LU factorisation algorithm is given in Algorithm 5.2.6. The solution stage of LU factorisation consists of the forward substitution algorithm given in Algorithm 5.2.7 followed by the back substitution algorithm given in Algorithm 5.2.8. The forward elimination algorithm of GE is given in Algorithm 5.2.9.

#### Algorithm 5.2.1 Partitioning of data to slave programs

```

nextslave=0
for i=0 to n/2-1 (upper half of matrix)
  send row index i to nextslave
  if nextslave = number of slaves
    nextslave=0
  else nextslave=nextslave+1
if n is odd
  send middle row index to nextslave
  
```



### Algorithm 5.2.2 WZ factorisation algorithm

```

for k=0 to depth of matrix
  if k is an index in this slave
    Broadcast row k and n-1-k to other slaves
    Update index of matrix in this slave
  else Receive row k and n-1-k
  for j = 0 to number of rows in a slave
    let jindex=index of row j
    if jindex > k (Top half of matrix)
      calculate  $w_{jindex,k}$  and  $w_{jindex,n-1-k}$ . store in  $tp_{j,k}$  and  $tp_{j,n-1-k}$ 
      calculate  $z_{jindex,k+1}, z_{jindex,k+2}, \dots, z_{jindex,n-2-k}$ . store in  $tp_{j,k+1}, tp_{j,k+2}, \dots, tp_{j,n-2-k}$ 
    end if
  for j = 0 to number of rows in a slave
    let jindex=index of row j
    if jindex < n-1-k (bottom half of matrix)
      calculate  $w_{jindex,k}$  and  $w_{jindex,n-1-k}$ . store in  $bt_{j,k}$  and  $bt_{j,n-1-k}$ 
      calculate  $z_{jindex,k+1}, z_{jindex,k+2}, \dots, z_{jindex,n-2-k}$ . store in  $bt_{j,k+1}, bt_{j,k+2}, \dots, bt_{j,n-2-k}$ 
    end if
  end for j
  if n is odd
    if (index of middle row > k)
      calculate  $w_{jindex,k}$  and  $w_{jindex,n-1-k}$ . store in  $md_{j,k}$  and  $md_{j,n-1-k}$ 
      calculate  $z_{jindex,k+1}, z_{jindex,k+2}, \dots, z_{jindex,n-2-k}$ . store in  $tp_{j,k+1}, tp_{j,k+2}, \dots, tp_{j,n-2-k}$ 
    end if
  end if
end for k

```

### Algorithm 5.2.3 Bi-directional substitution algorithm

```

for k = 0 to depth of matrix
  if k is an index on this slave
    send  $b_k$  and  $b_{n-1-k}$  to other slaves
    update index of matrix on this slave
  else
    receive  $b_k$  and  $b_{n-1-k}$ 
  for j= 0 to number of rows in slave
    let jindex = index of row j
    if jindex > k (top half of matrix)
       $b_{jindex} = b_{jindex} - w_{jindex,k} b_k - w_{jindex,n-1-k} b_{n-1-k}$ 
       $tp_{j,n} = b_{jindex}$ 
    end for j
  for j= 0 to number of rows in slave
    let jindex = index of row j
    if jindex < n-1-k (lower half of matrix)
       $b_{jindex} = b_{jindex} - w_{jindex,k} b_k - w_{jindex,n-1-k} b_{n-1-k}$ 
       $bt_{j,n} = b_{jindex}$ 
    end for j
  if n is odd
    if index of middle row > k
       $b_{jindex} = b_{jindex} - w_{jindex,k} b_k - w_{jindex,n-1-k} b_{n-1-k}$ 
       $md_{j,n} = b_{jindex}$ 
    end if
  end for k

```

#### Algorithm 5.2.4 Bi-directional Solution algorithm

```

if n is odd, slave having the middle row
  calculate  $x_{n/2}$  and send to other slaves
  update  $tp_{i,n} = tp_{i,n} - tp_{i,n/2} * x_{n/2}$  (upper matrix)
   $bt_{i,n} = bt_{i,n} - bt_{i,n/2} * x_{n/2}$  (lower matrix)
for k =  $n/2 - 1$  downto 0
  if k is an index on this slave
    solve for  $x_k$  and  $x_{n-1-k}$ 
    send  $x_k$  and  $x_{n-1-k}$  to other slaves
    update index of matrix on this slave
  else
    receive  $x_k$  and  $x_{n-1-k}$ 
  for j = 0 to number of rows in slave
    let jindex = index of row j
    if jindex < k
       $tp_{j,n} = tp_{j,n} - tp_{j,k} * x_k - tp_{j,n-1-k} * x_{n-1-k}$ 
       $bt_{j,n} = bt_{j,n} - bt_{j,k} * x_k - bt_{j,n-1-k} * x_{n-1-k}$ 
    end for j
  end for k

```

#### Algorithm 5.2.5 Parallel Implicit Elimination algorithm

```

for k = 0 to depth of matrix
  if k is an index in this slave
    Broadcast row k and n-1-k to others
    Update index of matrix in this slave
  else
    Receive row k and n-1-k
  for j = 0 to number of rows in a slave
    let jindex = index of row j
    if jindex > k (Upper half of matrix)
      (use array tp here)
      eliminate  $w_{jindex,k}$  and  $w_{jindex,n-1-k}$ 
      update  $a_{jindex,k+1}, a_{jindex,k+2}, \dots, a_{jindex,n-2-k}$ 
    end if
  end for j
  for j = 0 to number of rows in a slave
    let jindex = index of row j
    if jindex < n-1-k (lower half of matrix)
      (use array bt here)
      eliminate  $w_{jindex,k}$  and  $w_{jindex,n-1-k}$ 
      update  $a_{jindex,k+1}, a_{jindex,k+2}, \dots, a_{jindex,n-2-k}$ 
    end if
  end for j
  if n is odd let jindex = index of middle row
  if jindex > k (use array md here)
    eliminate  $w_{jindex,k}$  and  $w_{jindex,n-1-k}$ 
    update  $a_{jindex,k+1}, a_{jindex,k+2}, \dots, a_{jindex,n-2-k}$ 
  end if n odd
end for k

```

#### Algorithm 5.2.6 LU factorisation algorithm

```
for k = 0 to n-1 do
  if k is an index in this slave
    (pivot row is in this slave process)
      Broadcast pivot row to other slaves
      Update index of matrix on this slave
  else
    Receive pivot row
  for i = 0 to number of rows in the slave process
    Provided k is less than the highest index held on this slave,
    perform factorisation process
end for k
```

#### Algorithm 5.2.7 Forward Substitution Algorithm

```
for k = 0 to n-1
  if k is an index on this slave
    Calculate value of  $y[k]$ 
    Broadcast value to other slaves
    Update index of matrix on this slave
  else
    Receive solution  $y[k]$ 
  for i = 0 to number of rows in slave process
    Provided that k is less than the highest index
    held on this slave, then perform substitution.
end for k
```

#### Algorithm 5.2.8 Back Substitution Algorithm

```
for k = n-1 to 0
  if k is an index on this slave
    Calculate value of  $x[k]$ 
    Broadcast value to other slaves
    Update index of matrix on this slave
  else
    Receive solution  $x[k]$ 
  for i = 0 to number of rows in slave process
    Provided that k is greater than the highest index
    held on this slave, then perform substitution.
end for k
```

#### Algorithm 5.2.9 Forward Elimination Algorithm

```
for k = 0 to n-1 do
  if k is an index in this slave
    (pivot row is in this slave process)
      Broadcast pivot row to other slaves
      Update index of matrix on this slave
  else
    Receive pivot row
  for i = 0 to number of rows in the slave process
    Provided k is less than the highest index held on this
    slave,
    perform elimination process
end for k
```

### 5.3 Theoretical analysis of communication for PIE, GE, QIF and LU

Before proceeding with the analysis, some comments and assumptions need to be stated. The comments and assumptions will apply to all the algorithms discussed in this chapter.

- All algorithms employ the master and slave programming model.
- The matrix used in all the algorithms is:

$$a_{ij} = 1 \text{ for } i \neq j$$

$$\text{and } a_{ii} = n \text{ for } i = j.$$

- The coefficient matrix is generated in the slave program. Therefore, when the master program partitions the matrix to the slave programs, only the index of the row is sent instead of the entire row.
- For simplicity, all the slave programs have been spawned on a single workstation and the master program on the other.
- The theoretical analysis of the communication has been done in two ways. First is by disregarding the length of message sent or received and second is by taking into account the length of messages.
- The number of slaves is denoted by  $p$ .
- An integer value is assumed to occupy one byte of storage while a floating point value is assumed to occupy two bytes of storage.
- Row subscripts and pivot rows are integer values while the results are floating point values.
- Throughout the analysis, the value  $n/p$  is assumed to be an integer.

#### 5.3.1 Theoretical analysis of communication for GE algorithm

The master program of GE distributes subscripts of rows of the coefficient matrix to the slaves  $n$  times and likewise receives results from all the slaves  $n$  times. Hence the message passing involved in the GE master is  $2n$ .

The slave program receives subscripts from the master program  $n/p$  times. During the elimination stage, each slave communicates the pivot rows to other slaves  $n/p$  times and receives pivot rows from other slaves  $(n-n/p)$  times. In the solution stage, each slave sends its solution  $x$  to other slaves  $n/p$  times and receives solutions from other

slaves  $(n-n/p)$  times. Lastly, each slave must send its results to the master program  $n/p$  times. Therefore, the total message passing within the slave program is

$$4(n/p) + 2(n-n/p). \quad (5.3.1.1)$$

The total message passing involved in the GE algorithm is

$$\begin{aligned} & 2n + 4(n/p) + 2n - 2(n/p) \\ & = 4n + 2(n/p). \end{aligned} \quad (5.3.1.2)$$

When the message length is considered, the GE master program involves  $n$  one-byte messages and  $n$  two-byte messages.

The slave program requires

|                   |                          |   |
|-------------------|--------------------------|---|
| $n/p$             | one-byte messages        | for receiving row numbers from the master program |
| $n/p + (n - n/p)$ | $(n+1)/2$ -byte messages | during the elimination stage                      |
| $n/p + (n - n/p)$ | two-byte messages        | during the solution stage                         |
| $n/p$             | two-byte messages        | in sending the results back to the master program |

In summary, the GE algorithm requires

$$\begin{aligned} & n + n/p \quad \text{one-byte messages,} \\ & 2n + n/p \quad \text{two byte messages and} \\ & n \quad \text{ $(n+1)/2$ -byte messages.} \end{aligned} \quad (5.3.1.3)$$

### 5.3.2 Theoretical analysis of communication for PIE algorithm

During the elimination stage of PIE, two elements are eliminated simultaneously instead of one as in GE. If elements  $a_{ij}$  and  $a_{n-i+1,j}$  are eliminated, then rows  $i$  and  $n-i+1$  must be available in the slave for the elimination to take place. Thus, in partitioning the coefficient matrix to the slave programs, the PIE master program sends the subscripts of two symmetric rows at a time.

The PIE master program then sends out messages  $n/2$  times to the slave programs and receives results  $n/2$  times from the slaves. Therefore, the message-passing involved in the PIE master is  $n$  times.

Each PIE slave receives data from the master program  $n/2p$  times. During the elimination process, each slave sends pivot rows to other slaves  $n/2p$  times and receives pivot rows from other slaves  $(n/2 - n/2p)$  times. During the solution stage,

each slave sends solution to other slaves  $n/2p$  times and receives solutions from other slaves  $(n/2 - n/2p)$  times. Results are then sent to the master by each slave  $n/2p$  times.

In total, the amount of message-passing involved in PIE is

$$\begin{aligned}
 & n + 4(n/2p) + 2(n/2 - n/2p) \\
 &= n + 2(n/p) + n - n/p \\
 &= 2n + n/p
 \end{aligned}
 \tag{5.3.2.1}$$

When message length is considered for PIE, the master program which sends two row subscripts at a time and receives two results at a time, will then require

$$\begin{aligned}
 & n/2 \quad \text{two-byte messages and} \\
 & n/2 \quad \text{four-byte messages.}
 \end{aligned}$$

The slave program for PIE requires

|                       |                        |  |
|-----------------------|------------------------|--|
| $n/2p$                | two-byte messages      | for receiving row numbers from the master program    |
| $n/2p + (n/2 - n/2p)$ | $(n+1)$ -byte messages | during the elimination stage                         |
| $n/2p + (n/2 - n/2p)$ | four-byte messages     | during the solution stage                            |
| $n/2p$                | four-byte messages     | when the results are sent back to the master program |

In total, the PIE algorithm requires

$$\begin{aligned}
 & n/2 + n/2p \quad \text{two-byte messages} \\
 & n + n/2p \quad \text{four-byte messages and} \\
 & n/2 \quad \text{ $(n+1)$ -byte messages.}
 \end{aligned}
 \tag{5.3.2.2}$$

### 5.3.3 Theoretical analysis of communication for LU algorithm

The master program of LU distributes subscripts of rows of the coefficient matrix to the slaves  $n$  times and receives results from the slaves  $n$  times. Hence, the message-passing involved in the LU master program is  $2n$ .

The slave program receives data from the master  $n/p$  times. During the factorisation stage, each slave sends pivot rows to the other slaves  $n/p$  times and receives pivot rows from other slaves  $(n - n/p)$  times. In the forward substitution stage, each slave sends solutions to other slaves  $n/p$  times and receives solutions from other slaves  $(n - n/p)$  times. During the back substitution process, each slave once again sends

solutions to other slaves  $n/p$  times and receives solutions from other slaves  $(n-n/p)$  times. Lastly, each slave must send results back to the master program  $n/p$  times.

In total, the message-passing within the slave program is

$$\begin{aligned} & 5n/p + 3(n-n/p) \\ & = 3n + 2n/p. \end{aligned} \quad (5.3.3.1)$$

The total message-passing within the LU algorithm is

$$5n + 2n/p. \quad (5.3.3.2)$$

When the message length is taken into consideration, the LU master program requires  $n$  one-byte messages and  $n$  two-byte messages. The slave program requires

|                   |                        |   |
|-------------------|------------------------|---|
| $n/p$             | one-byte messages      | for receiving row numbers from the master program |
| $n/p + (n - n/p)$ | $(n/2)$ -byte messages | during the factorisation stage                    |
| $n/p + (n - n/p)$ | two-byte messages      | during the forward substitution stage             |
| $n/p + (n - n/p)$ | two-byte messages      | during the back substitution stage                |
| $n/p$             | two-byte messages      | in sending the results back to the master program |

In total, the LU algorithm requires

$$\begin{aligned} & n + n/p \quad \text{one-byte messages,} \\ & 3n + n/p \quad \text{two-byte messages and} \\ & n \quad \text{ } (n/2)\text{-byte messages.} \end{aligned} \quad (5.3.3.3)$$

#### 5.3.4 Theoretical analysis of communication for QIF algorithm

Just as in PIE, the QIF master program also sends subscripts of two symmetric rows to the slave programs. The QIF master program sends out messages  $n/2$  times and receives results  $n/2$  times giving a total of  $n$  messages.

Each QIF slave program receives data  $n/2p$  times from the master program. During the factorisation stage, each slave program sends pivot rows to other slaves  $n/2p$  times and receives pivot rows from other slaves  $(n/2 - n/2p)$  times. In the bi-directional substitution stage, each slave program sends solutions to other slave programs  $n/2p$  times and receives solution from other slaves  $(n/2 - n/2p)$  times. Likewise in the bi-directional solution process where each slave program sends solutions to other slave

program  $n/2p$  times and receives solutions from other slave program  $(n/2 - n/2p)$  times. Finally, each slave program returns results to the master program  $n/2p$  times.

In total, the amount of message-passing involved in QIF is

$$\begin{aligned} & n + 5(n/2p) + 3(n/2 - n/2p) \\ & = 5/2n + n/p. \end{aligned} \tag{5.3.4.1}$$

In terms of message length, the QIF master program requires

$n/2$  two-byte messages and  
 $n/2$  four-byte messages.

The QIF slave program requires

|                       |                    |   |
|-----------------------|--------------------|---|
| $n/2p$                | two-byte messages  | for receiving row numbers from the master program |
| $n/2p + (n/2 - n/2p)$ | n-byte messages    | during the factorisation stage                    |
| $n/2p + (n/2 - n/2p)$ | four-byte messages | during the bi-directional substitution stage      |
| $n/2p + (n/2 - n/2p)$ | four-byte messages | during the bi-directional solution stage          |
| $n/2p$                | four-byte messages | in sending the results back to the master program |

In total, the QIF algorithm requires

$$\begin{aligned} & n/2 + n/2p \quad \text{two-byte messages} \\ & 3n/2 + n/2p \quad \text{four-byte messages and} \\ & n/2 \quad \text{n-byte messages.} \end{aligned} \tag{5.3.4.2}$$



#### 5.4 Prediction of communication times

The objective in parallel processing is to obtain faster execution by using multiple processing elements that work co-operatively on a single problem. The level of efficiency in speeding up computation is dependent on several factors ranging from inherent non-parallelism in the algorithm to the overheads of communication and synchronisation among the multiple processors. In cluster-based environments, there are also external influences as both the networks and the processors may be in use by other applications. Therefore, analysing communication must take into account extraneous traffic and the variable nature of network characteristics in terms of delay and throughput.

In this work, the experimental work done to investigate the amount of communication in the algorithms have been performed on a lightly loaded LAN where the computing elements are similar workstations which are not dedicated and do not execute external computer intensive applications. A "lightly loaded network" mentioned above refers to one which operates at 15-20% of capacity and on which the traffic is lightly loaded. Under these assumptions, an attempt is made to parametrise the communication overheads involved in the classical as well as the implicit algorithms. The conventional approach in analysing communications for most message passing, distributed memory multiprocessors [Schmidt 95] is adopted. This approach defines communication time as a simple linear function of the number of bytes transmitted, with a constant additive factor representing "start-up" overheads.

Hence,

$$T_{\text{comm}} = \alpha + \beta N$$

where  $\alpha$  is the start-up time,  $\beta$  is the cost per byte and  $N$  is the number of bytes transmitted.

In order to estimate the coefficients  $\alpha$  and  $\beta$ , a number of experiments were conducted under varying network and host load conditions, for different message sizes. The results of these experiments are shown in Table 5.4.1.

| Size (bytes) | Transmission time (secs) |
|--------------|--------------------------|
| 64           | 0.022448                 |
| 1024         | 0.038064                 |
| 2048         | 0.044496                 |
| 4096         | 0.037664                 |

Table 5.4.1: Measured time (in seconds) for PVM communication

From the experimental results, the coefficients were determined by curve fitting. The equations to be solved are:

$$\alpha + 64\beta = 0.022448$$

$$\alpha + 1024\beta = 0.038064$$

$$\alpha + 2048\beta = 0.044496$$

$$\alpha + 4096\beta = 0.037664$$

which is an overdetermined linear system.

In matrix form,

$$Ax=b$$

$$\begin{bmatrix} 1 & 64 \\ 1 & 1024 \\ 1 & 2048 \\ 1 & 4096 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0.022448 \\ 0.038064 \\ 0.044496 \\ 0.037664 \end{bmatrix} \quad (5.4.1)$$

Applying the least squares method and multiplying both sides by  $A^T$  (5.4.2) yields the 2x2 linear system shown in (5.4.3).

$$A^T Ax = A^T b \quad (5.4.2)$$

$$\begin{bmatrix} 4 & 7232 \\ 7232 & 22024192 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0.142672 \\ 285.81376 \end{bmatrix} \quad (5.4.3)$$

Solving the linear system in (5.4.3) results in the following values for  $\alpha$  and  $\beta$ .

$$\alpha = 0.0076584$$

$$(5.4.4)$$

$$\beta = 0.00001549 \quad (5.4.5)$$

Once the values of  $\alpha$  and  $\beta$  have been determined, they can be used to predict the communication times required by each algorithm using the formula for communication time for each algorithm obtained earlier.

The number of times each algorithm sends or receives messages is summarised in Table 5.4.2.

| Algorithm | Number of messages |
|-----------|--------------------|
| GE        | $4n + 2n/p$        |
| PIE       | $2n + n/p$         |
| LU        | $5n + 2n/p$        |
| QIF       | $5/2n + n/p$       |

Table 5.4.2 Frequency of messages for each algorithm

Each time a message is sent, there is an initial start-up time of  $\alpha$  incurred, regardless of the message length. Hence, Table 5.4.2 also reflects the amount of start-up time incurred in each algorithm. The predicted start-up time can be calculated by multiplying the number of messages by the value  $\alpha$ . For example, for  $n=100$  and  $p=1,2$  and  $4$ , the predicted start-up time for the algorithms are shown in Table 5.4.3.

| $n$ | $p$ | Algorithm | Predicted start-up time |
|-----|-----|-----------|-------------------------|
| 100 | 1   | GE        | 11.4567                 |
|     |     | PIE       | 5.72835                 |
|     | 2   | GE        | 9.54725                 |
|     |     | PIE       | 4.773625                |
|     | 4   | GE        | 8.592525                |
|     |     | PIE       | 4.2962625               |
| 100 | 1   | LU        | 5.36088                 |
|     |     | QIF       | 2.68044                 |
|     | 2   | LU        | 4.59504                 |
|     |     | QIF       | 2.29752                 |
|     | 4   | LU        | 4.21212                 |
|     |     | QIF       | 2.10606                 |

Table 5.4.3 Predicted start-up times for algorithms when  $n=100$

The total predicted communication time required by each algorithm can be calculated by considering the length of messages transmitted. A sample of the total predicted times for the case of  $n=100$ ,  $p=1, 2$  and  $4$  for all the algorithms are shown in Table 5.4.4.

| $n$ | $p$ | Algorithm | Formula   | Predicted communication time |
|-----|-----|-----------|---|------------------------------|
| 100 | 1   | GE        | $200\alpha + \beta + 300\alpha + 2\beta + 100\alpha + 101\beta$ | 4.5966416                    |
|     |     | PIE       | $100\alpha + 2\beta + 150\alpha + 4\beta + 50\alpha + 202\beta$ | 2.3007231                    |
|     | 2   | GE        | $150\alpha + \beta + 250\alpha + 2\beta + 100\alpha + 101\beta$ | 3.8308016                    |
|     |     | PIE       | $75\alpha + 2\beta + 125\alpha + 4\beta + 50\alpha + 202\beta$  | 1.9178032                    |
|     | 4   | GE        | $125\alpha + \beta + 225\alpha + 2\beta + 100\alpha + 101\beta$ | 3.4478816                    |
|     |     | PIE       | $63\alpha + 2\beta + 113\alpha + 4\beta + 50\alpha + 202\beta$  | 1.7340016                    |
| 100 | 1   | LU        | $200\alpha + \beta + 400\alpha + 2\beta + 100\alpha + 101\beta$ | 5.3624816                    |
|     |     | QIF       | $100\alpha + 2\beta + 300\alpha + 4\beta + 50\alpha + 202\beta$ | 2.6836431                    |
|     | 2   | LU        | $150\alpha + \beta + 350\alpha + 2\beta + 100\alpha + 101\beta$ | 4.5966416                    |
|     |     | QIF       | $75\alpha + 2\beta + 175\alpha + 4\beta + 50\alpha + 202\beta$  | 2.3007232                    |
|     | 4   | LU        | $125\alpha + \beta + 325\alpha + 2\beta + 100\alpha + 101\beta$ | 4.2137216                    |
|     |     | QIF       | $63\alpha + 2\beta + 113\alpha + 4\beta + 50\alpha + 202\beta$  | 2.11686                      |

Table 5.4.4 Total predicted times for  $n=100$ ,  $p=1, 2$  an  $4$ .

## 5.5 Numerical results

The PVM programs for both PIE and QIF have been written in C using the master and slave programming model. They have been tested on a cluster of two Dec-Alphas connected via Internet.

The time taken for an application is dependent on the computation time, communication time and load imbalance. Load imbalance on workstation clusters can also be due to the effect of external CPU, memory or network loads. For example, in a cluster of identical workstations, equal amounts of work may well require different execution times, due to externally generated computation, swapping and network activity.

In order to measure the communication time of the algorithms, the time spent in various components of the algorithms had to be measured. All measurements are elapsed (or wall-clock) times. The total time was measured by the master process that waited for all the slave processes to finish execution. The master process was also responsible for gathering local timing results from all the slave processes; the computation, communication and idle times reported are the arithmetic means of corresponding individually measured timings.

Computation time is the time a process spends doing useful work which can be contributed to the final result. The communication time is the time spent in packing a message as well as transmission to the intended receiver. Idle time is the time during which a process was blocked awaiting message arrival or the elapsed time between the beginning and the end of a receive operation. The idle time is also a means of quantifying the degree of load imbalance in the algorithm. Total time is the sum of idle time, computation time and communication time

Table 5.5.1 shows the measured timings of GE vs PIE using 1 slave, Table 5.5.2 shows the measured timings of GE vs PIE using 2 slaves and Table 5.5.3 shows the measured timings of GE vs PIE for 4 slaves. Table 5.5.4 shows the measured timings of LU vs QIF using 1 slave, Table 5.5.5 shows the measured timings of LU vs QIF using 2 slaves and Table 5.5.6 shows the measured timings of LU vs QIF for 4 slaves.

| <i>n</i> | Algorithm | Idle time | Computation time | Communication time | Total Time |
|----------|-----------|-----------|------------------|--------------------|------------|
| 100      | GE        | 1.429440  | 0.111264         | 0.306464           | 1.847168   |
|          | PIE       | 0.464576  | 0.083936         | 0.154784           | 0.703296   |
|          | Gains(%)  |           | 24.56            | 49.49              |            |
| 200      | GE        | 2.251808  | 0.767712         | 0.551440           | 3.57096    |
|          | PIE       | 1.635952  | 0.525664         | 0.345504           | 2.50712    |
|          | Gains(%)  |           | 31.53            | 37.35              |            |
| 300      | GE        | 1.12688   | 2.670512         | 0.764784           | 4.562176   |
|          | PIE       | 1.896944  | 1.874096         | 0.474912           | 4.245952   |
|          | Gains(%)  |           | 29.82            | 37.9               |            |
| 400      | GE        | 0.966816  | 7.047824         | 1.044896           | 9.059536   |
|          | PIE       | 1.315824  | 4.690608         | 0.693936           | 6.700368   |
|          | Gains(%)  |           | 33.45            | 33.59              |            |
| 600      | GE        | 0.837984  | 25.515983        | 1.618785           | 27.972752  |
|          | PIE       | 0.815536  | 16.313297        | 1.013663           | 18.142496  |
|          | Gains(%)  |           | 36.07            | 37.38              |            |

Table 5.5.1 Measured Times for PIE and GE (1 slave)

| <i>n</i> | Algorithm | Idle time | Computation time | Communication time | Total Time |
|----------|-----------|-----------|------------------|--------------------|------------|
| 100      | GE        | 1.097112  | 0.333791         | 0.962425           | 2.393328   |
|          | PIE       | 0.791048  | 0.207            | 0.449936           | 1.447984   |
|          | Gains(%)  |           | 37.99            | 53.51              |            |
| 200      | GE        | 1.0732    | 0.573596         | 2.665544           | 4.31234    |
|          | PIE       | 5.449536  | 0.547137         | 1.178607           | 7.17528    |
|          | Gains(%)  |           | 4.6              | 55.78              |            |
| 300      | GE        | 1.202032  | 0.9598075        | 5.4096085          | 7.571448   |
|          | PIE       | 0.615856  | 0.620331         | 3.236592           | 4.472779   |
|          | Gains(%)  |           | 35.25            | 40.17              |            |
| 400      | GE        | 0.690572  | 2.80937          | 8.9944545          | 12.494396  |
|          | PIE       | 1.252096  | 2.1456           | 5.236504           | 8.6342     |
|          | Gains(%)  |           | 23.63            | 41.78              |            |
| 600      | GE        | 0.77308   | 12.898912        | 20.068391          | 33.740383  |
|          | PIE       | 1.408944  | 8.9895765        | 12.038727          | 22.437247  |
|          | Gains(%)  |           | 30.31            | 40.01              |            |

Table 5.5.2 Measured times for PIE and GE (2 slaves)

| $n$ | Algorithm | Idle time | Computation time | Communication time | Total Time |
|-----|-----------|-----------|------------------|--------------------|------------|
| 100 | GE        | 1.1950045 | 0.5629072        | 1.0700287          | 2.8279404  |
|     | PIE       | 0.321104  | 0.3325983        | 0.480525           | 1.1342273  |
|     | Gains(%)  |           | 40.91            | 55.09              |            |
| 200 | GE        | 0.77226   | 1.577304         | 2.172596           | 4.52216    |
|     | PIE       | 0.32442   | 0.9426712        | 1.1950447          | 2.4621359  |
|     | Gains(%)  |           | 63.46            | 44.99              |            |
| 300 | GE        | 1.113948  | 3.114594         | 4.70265            | 8.931192   |
|     | PIE       | 2.009472  | 1.7486805        | 2.7808875          | 6.53904    |
|     | Gains(%)  |           | 43.86            | 40.87              |            |
| 400 | GE        | 1.561444  | 2.9547705        | 10.660929          | 15.177143  |
|     | PIE       | 1.29362   | 1.8911765        | 6.3325675          | 9.517364   |
|     | Gains(%)  |           | 35.99            | 40.6               |            |
| 600 | GE        | 1.41196   | 7.1106007        | 28.30615           | 36.82871   |
|     | PIE       | 0.542412  | 6.495452         | 16.594428          | 23.632292  |
|     | Gains(%)  |           | 8.65             | 41.38              |            |

Table 5.5.3 Measured times for PIE and GE (4 slaves)

| $n$ | Algorithm | Idle time | Computation time | Communication time | Total Time |
|-----|-----------|-----------|------------------|--------------------|------------|
| 100 | LU        | 1.176656  | 0.110288         | 0.301184           | 1.588128   |
|     | QIF       | 0.699392  | 0.0732           | 0.193248           | 0.965840   |
|     | Gains(%)  |           | 33.63            | 35.84              |            |
| 200 | LU        | 1.434896  | 0.787232         | 0.576816           | 2.798944   |
|     | QIF       | 1.019520  | 0.490928         | 0.382592           | 1.893040   |
|     | Gains(%)  |           | 37.64            | 33.67              |            |
| 300 | LU        | 1.792112  | 2.917038         | 0.923874           | 5.633024   |
|     | QIF       | 0.790160  | 1.788209         | 0.569983           | 3.148352   |
|     | Gains(%)  |           | 38.7             | 38.31              |            |
| 400 | LU        | 1.192272  | 7.678898         | 1.134686           | 10.005856  |
|     | QIF       | 0.904352  | 4.422208         | 0.788608           | 6.115168   |
|     | Gains(%)  |           | 42.41            | 30.5               |            |
| 600 | LU        | 3.632048  | 26.299633        | 2.004879           | 31.936560  |
|     | QIF       | 1.029280  | 15.482720        | 1.295728           | 17.807728  |
|     | Gains(%)  |           | 41.13            | 35.37              |            |

Table 5.5.4 Measured times for QIF and LU (1 slave)

| <i>n</i> | Algorithm | Idle time | Computation time | Communication time | Total Time |
|----------|-----------|-----------|------------------|--------------------|------------|
| 100      | LU        | 1.354288  | 0.4645765        | 1.3280235          | 3.146888   |
|          | QIF       | 1.079544  | 0.22692          | 0.615944           | 1.922408   |
|          | Gains(%)  |           | 51.16            | 53.62              |            |
| 200      | LU        | 1.415088  | 1.150503         | 3.092521           | 5.658112   |
|          | QIF       | 0.751032  | 0.4840965        | 1.7546235          | 2.989752   |
|          | Gains(%)  |           | 57.92            | 43.26              |            |
| 300      | LU        | 1.645136  | 0.94916          | 6.8981835          | 9.540216   |
|          | QIF       | 1.722816  | 0.3843675        | 4.1667185          | 6.16592    |
|          | Gains(%)  |           | 59.5             | 39.6               |            |
| 400      | LU        | 1.159576  | 3.484848         | 10.68604           | 15.330464  |
|          | QIF       | 1.519032  | 1.506921         | 6.501575           | 9.527528   |
|          | Gains(%)  |           | 56.76            | 39.16              |            |
| 600      | LU        | 0.991792  | 14.481692        | 22.314236          | 37.78772   |
|          | QIF       | 0.896056  | 8.374032         | 13.571312          | 22.8414    |
|          | Gains(%)  |           | 33.99            | 39.18              |            |

Table 5.5.5 Measured times for QIF and LU (2 slaves)

| <i>n</i> | Algorithm | Idle time | Computation time | Communication time | Total Time |
|----------|-----------|-----------|------------------|--------------------|------------|
| 100      | LU        | 0.310124  | 0.8169995        | 1.4387125          | 2.565836   |
|          | QIF       | 0.444168  | 0.440908         | 0.6771882          | 1.562264   |
|          | Gains(%)  |           | 46.03            | 52.93              |            |
| 200      | LU        | 2.110288  | 2.1526562        | 3.2575077          | 7.520452   |
|          | QIF       | 4.343452  | 1.1838207        | 1.7723592          | 7.299632   |
|          | Gains(%)  |           | 45.01            | 45.59              |            |
| 300      | LU        | 2.816756  | 2.6812612        | 7.5124747          | 13.010492  |
|          | QIF       | 1.628056  | 2.4405967        | 3.0901915          | 7.158844   |
|          | Gains(%)  |           | 8.98             | 58.87              |            |
| 400      | LU        | 1.486868  | 3.511868         | 13.761412          | 18.760148  |
|          | QIF       | 1.1525    | 2.1177387        | 7.6549647          | 10.732732  |
|          | Gains(%)  |           | 39.7             | 44.37              |            |
| 600      | LU        | 2.04514   | 8.4495777        | 32.933012          | 43.42773   |
|          | QIF       | 0.640288  | 6.4147236        | 18.903752          | 25.958762  |
|          | Gains(%)  |           | 24.08            | 42.6               |            |

Table 5.5.6 Measured times for QIF and LU (4 slaves)



## 5.6 Summary

The objective of this work was to investigate the performance of QIF and PIE algorithms as compared to LU and GE algorithms on a distributed memory parallel computer which employs message passing as a means of co-ordination amongst the processors.

The theoretical analysis of communication within the algorithms revealed that the communication involved in PIE and QIF is halved to that of GE and LU. Although the total amount of data moved for both algorithms are the same, the start-up cost associated with the movement of data in the implicit algorithms are greatly reduced because 50% less messages are needed to move the data as compared to the classical algorithms. This relationship can be seen in the measured communication times of the algorithms with the reduction of communication time for PIE and QIF being in the range of 30% - 50%. The theoretical difference of 50% could not be obtained in the numerical results due to several factors such as network load, delay and throughput.

The total time taken for an application is dependent on the computation time, communication time (including synchronisation) and load imbalance. Load imbalance is usually a measure of the equitable distribution of workload among the processing elements. However, in networked systems, where processors are typically not dedicated to the application under consideration, there is another side to load imbalance, that is the effect of external CPU, memory or network loads. For example, in a cluster of identical workstations, equal amounts of work may require different execution times, owing to externally generated computation, swapping and network activity. These influences vary dynamically and it is difficult to incorporate them accurately in the theoretical analysis.

For the case of one slave program, relationship in communication between PIE/QIF and GE/LU is not really halved because it is essentially a sequential program and the communication is just between the master program and the slave program in sending data and receiving results. However, for two and four slave programs the relationship begins to show and the inherent parallelism within PIE and QIF is fully exploited.

In a message-passing parallel computer, the speed of communication between different nodes of the network is of critical importance. Programs with a lot of data communication between nodes may find the overall performance limited more by the

performance of the communication network than by the computational performance of the nodes themselves. All the algorithms discussed in this chapter possess a communication pattern which is regular and symmetric (and at approximately equally spaced intervals). While regularity and symmetry may be a desirable property for parallel algorithms on true multiprocessors, the above experience indicates that they can be detrimental in cluster computing where communication costs factors occur so heavily in performance degradation.

There are two factors that must be considered in order to access communication. First is the start-up time which determines the short-message performance and the asymptotic bandwidth which determines the long-message performance. Since communication start-up times are comparatively high, best network utilisation is achieved when messages are large and less frequent. Compared to GE and LU, PIE and QIF send a larger volume of messages and the frequency of communication is halved.

The principal impediment to performance on workstation clusters was interprocessor communication overhead, as verified by our analytic performance model, suggesting enhanced scalability for PIE and QIF with the use of faster communication (both hardware and software). Further tests on larger clusters are necessary to confirm these tentative conclusions.

## Chapter 6

### QZ decomposition on a shared memory multiprocessor

In the solution of dense linear systems of equation

$$Ax = b, \quad (6.1)$$

it is usual for the matrix  $A$  to be converted to upper triangular (easily solvable) form by a series of elementary stabilised eliminations.

Recently, orthogonal transformations operating on the rows of the matrix have become popular for stability considerations. If Givens rotation matrices are used each rotation eliminates a single element. The complete decomposition process can be described as

$$Q^T A = Q_r^T Q_{r-1}^T \cdots Q_2^T Q_1^T A = R \quad (6.2)$$

where  $Q_1^T, Q_2^T, \dots, Q_r^T$  are  $(nxn)$  Givens transformations and  $R$  is the resulting upper triangular matrix.

Substituting for  $A$  from (6.2) in equation (6.1) and taking into account the orthogonality condition  $Q^T Q = I$  gives

$$Rx = Q^T b \quad (6.3)$$

which can be solved by the usual back substitution process. The Givens QR method has been discussed in chapter 2 and its parallelisation has been given in chapter 3.

In this chapter the performance of the Givens transformation will be compared to the implicit version of orthogonal transformation, known as QZ factorisation. In this factorisation  $Q$  is an orthogonal matrix and  $Z$  is similar to the  $Z$  matrix in the QIF method discussed in chapter 4.

QZ was first proposed by Evans and Yalamov [Evans 94a]. In their work, it was shown that for larger matrices the computational time for QZ decomposition is about 8% less than the time for Givens QR decomposition. Their comparison involved only the sequential implementation of QR and QZ. The numerical stability of QZ was also analysed and it was proved that the error growth is a linear function of the size of the matrix and does not depend on the growth of intermediate results.

The work in this chapter involves investigating the performance of parallel QZ on a shared memory parallel computer and comparing the performance with Givens parallel QR decomposition. The results obtained will also be justified by a theoretical analysis of the computational work and shared memory access count. The QR decomposition

method is presented in section 2.8.3 of chapter 2. Section 6.1 presents the QZ decomposition method. The sequential and parallel algorithms for QZ decomposition are given in section 6.2. In section 6.3, the computational complexity analysis and shared memory access count is discussed. The results obtained are presented in section 6.4. A greedy strategy for QZ decomposition is proposed in section 6.5 and the summary of the chapter is given in section 6.6.

### 6.1 QZ decomposition method

Given the linear system (6.1) where  $A$  is an  $n \times n$  non-singular matrix,  $\mathbf{x}$  is the unknown vector and  $\mathbf{b}$  is the right hand side vector and consider a concurrent transformation matrix  $T$  such that when the matrix product  $TA$  is formed the entries in the first and final columns of  $A$ , i.e.  $a_{21}, \dots, a_{n-1,1}$  and  $a_{2,n-1}, \dots, a_{n-1,n-1}$  are eliminated simultaneously.

Consider a matrix  $T$  to be a product form of plane rotations which can be viewed as two successive Givens rotations applied simultaneously, i.e.,

$$T = T_{23}T_{13} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sin \phi & \cos \phi \\ 0 & -\cos \phi & \sin \phi \end{bmatrix} \begin{bmatrix} \sin \theta & 0 & -\cos \theta \\ 0 & 1 & 0 \\ \cos \theta & 0 & \sin \theta \end{bmatrix} \quad (6.1.1)$$

$$= \begin{bmatrix} \sin \theta & 0 & -\cos \theta \\ \cos \theta \cos \phi & \sin \phi & \sin \theta \cos \phi \\ \cos \theta \sin \phi & -\cos \phi & \sin \theta \sin \phi \end{bmatrix} \quad (6.1.2)$$

applied to rows (1,3) and rows (2,3) of  $A$  for  $n=3$ .

Next,  $A$  and  $\mathbf{b}$  are transformed as,

$$\begin{aligned} TA_3 &= \begin{bmatrix} s\theta & 0 & -c\theta \\ c\theta c\phi & s\phi & s\theta c\phi \\ c\theta s\phi & -c\phi & s\theta s\phi \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\ &= \begin{bmatrix} a_{11}s\theta - a_{31}c\theta & a_{12}s\theta - a_{32}c\theta & a_{13}s\theta - a_{33}c\theta \\ a_{11}c\theta c\phi + a_{21}s\phi + a_{31}s\theta c\phi & a_{12}c\theta c\phi + a_{22}s\phi + a_{32}s\theta c\phi & a_{13}c\theta c\phi + a_{23}s\phi + a_{33}s\theta c\phi \\ a_{11}c\theta s\phi - a_{21}c\phi + a_{31}s\theta s\phi & a_{12}c\theta s\phi - a_{22}c\phi + a_{32}s\theta s\phi & a_{13}c\theta s\phi - a_{23}c\phi + a_{33}s\theta s\phi \end{bmatrix} \end{aligned} \quad (6.1.3)$$

$$T\mathbf{b} = \begin{bmatrix} b_1s\theta - b_3c\theta \\ b_1c\theta c\phi + b_2s\phi + b_3s\theta c\phi \\ b_1c\theta s\phi - b_2c\phi + b_3s\theta s\phi \end{bmatrix} \quad (6.1.4)$$

where  $c\theta$  denotes  $\cos \theta$  and  $s\theta c\phi$  denotes  $\sin\theta\cos\phi$ , etc.

The values of  $\theta$  and  $\phi$  are now chosen to annihilate the elements  $a_{21}$  and  $a_{23}$ . This is achieved when

$$\sin \theta = \frac{\Delta_3}{\sqrt{\Delta_1^2 + \Delta_3^2}}, \quad \cos \theta = \frac{\Delta_1}{\sqrt{\Delta_1^2 + \Delta_3^2}}, \quad (6.1.5)$$

$$\sin \phi = \frac{\Delta_2}{\sqrt{\Delta_1^2 + \Delta_2^2 + \Delta_3^2}}, \quad \cos \phi = \frac{\sqrt{\Delta_1^2 + \Delta_3^2}}{\sqrt{\Delta_1^2 + \Delta_2^2 + \Delta_3^2}},$$

where

$$\Delta_1 = a_{33}a_{21} - a_{31}a_{23}, \quad \Delta_2 = a_{13}a_{31} - a_{11}a_{33} \quad \text{and} \quad \Delta_3 = a_{11}a_{23} - a_{13}a_{21}. \quad (6.1.6)$$

The forward elimination process of QZ uses matrices of the form (6.1.2) for the concurrent elimination of the entries in the  $k$ th and  $(n-k+1)$ th columns of matrix  $A$  of order  $n$  where  $n$  is even.

The following is a description of the concurrent orthogonal decomposition process.

For  $k = 1$  to  $n/2-1$  perform the following steps:

For  $i=k+1, \dots, n-k$  form the matrices,

$$T_{ki} = \begin{bmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & \sin \theta_{ki} & \dots & 0 & \dots & -\cos \theta_{ki} & & \\ & & & & 1 & & & & & \\ & & & \vdots & \ddots & \vdots & & \vdots & & \\ & & & & & 1 & & & & \\ \cos \theta_{ki} \cos \phi_{ki} & \dots & \sin \phi_{ki} & \dots & \sin \theta_{ki} \cos \phi_{ki} & & & & & \\ & & & & & 1 & & & & \\ & & & \vdots & & \ddots & & \vdots & & \\ & & & & & & 1 & & & \\ \cos \theta_{ki} \sin \phi_{ki} & \dots & -\cos \phi_{ki} & \dots & \sin \theta_{ki} \sin \phi_{ki} & & & & & \\ & & & & & & & 1 & & \ddots \\ & & & & & & & & \ddots & 1 \end{bmatrix} \begin{matrix} k \\ \updownarrow \\ i \\ \updownarrow \\ n-k+1 \end{matrix}$$

(6.1.7)

and update  $A$  and  $b$  as follows,

$$A = T_{ki}A, \quad d = T_{ki}b, \quad (6.1.8)$$

where for  $k=1, 2, \dots, m$ , and  $i = k+1, \dots, n-k$ , where  $m = n/2 - 1$

$$\begin{aligned} \Delta_1 &= a_{n+1-k, n+1-k} a_{ik} - a_{n+1-k, k} a_{i, n+1-k}, \\ \Delta_2 &= a_{k, n+1-k} a_{n+1-k, k} - a_{k, k} a_{n+1-k, n+1-k}, \\ \Delta_3 &= a_{kk} a_{i, n+1-k} - a_{k, n+1-k} a_{ik}. \end{aligned} \quad (6.1.9)$$

After this procedure is completed, a system  $Zu=d$  is obtained, where the final matrix  $Z$  of the form,

$$Z = \begin{bmatrix} a_{11} & a_{12}^{(1)} & \dots & a_{1, n-1}^{(1)} & a_{1n} \\ 0 & a_{22} & \dots & a_{2, n-1} & 0 \\ 0 & 0 & \ddots & \frac{(n-1)}{2} a_{\frac{n+1}{2}, \frac{n+1}{2}} & 0 \\ 0 & a_{n-1, 2}^{(1)} & \ddots & a_{n-1, n-1}^{(1)} & 0 \\ a_{n1} & a_{n2} & \dots & a_{n, n-1} & a_{nn} \end{bmatrix}, \text{ for } n = \text{odd} \quad (6.1.10)$$

and

$$Z = \begin{bmatrix} a_{11} & a_{12}^{(1)} & \dots & \dots & \dots & a_{1, n-1}^{(1)} & a_{1n} \\ 0 & a_{22} & \dots & \dots & \dots & a_{2, n-1} & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & \frac{(n-1)}{2} a_{\frac{n}{2}, \frac{n}{2}} & \frac{(n-1)}{2} a_{\frac{n}{2}, \frac{n}{2}+1} & 0 & 0 \\ 0 & 0 & \frac{(n-1)}{2} a_{\frac{n}{2}+1, \frac{n}{2}} & \frac{(n-1)}{2} a_{\frac{n}{2}+1, \frac{n}{2}+1} & 0 & 0 \\ 0 & 0 & \ddots & \ddots & 0 & 0 \\ 0 & a_{n-1, 2}^{(1)} & \dots & \dots & a_{n-1, n-1}^{(1)} & 0 \\ a_{n1} & a_{n2} & \dots & \dots & a_{n, n-1} & a_{nn} \end{bmatrix}, \text{ for } n = \text{even}. \quad (6.1.11)$$

The solution of the reduced system,

$$Zu = d,$$

where  $Z$  is given by equations (6.1.10) and (6.1.11) requires the bi-directional solution process which has been described in section 4.1 of chapter 4.

### Parallel QZ decomposition

As in the parallel QR algorithm where several rotations can be performed concurrently, several QZ rotations can also be done concurrently. The order in which the elements of a matrix is eliminated is called the annihilation pattern. The sequence in which the

rotations are done for QZ is denoted by Figures 6.1.1 and 6.1.2 for even and odd sized matrices respectively. The integers in the annihilation pattern denote the steps at which the given elements are annihilated. The annihilation pattern for parallel QR has been shown in chapter 3. The annihilation by Sameh and Kuck [Sameh 77] has been chosen for the parallel QR implementation in this work.

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| xx |    |    |    |    |    |    |    |    | xx |
| 1  | xx |    |    |    |    |    |    | xx | 1  |
| 3  | 5  | xx |    |    |    |    | xx | 5  | 3  |
| 5  | 7  | 9  | xx |    |    | xx | 9  | 7  | 5  |
| 7  | 9  | 11 | 13 | xx | xx | 13 | 11 | 9  | 7  |
| 8  | 10 | 12 | 14 | xx | xx | 14 | 12 | 10 | 8  |
| 6  | 8  | 10 | xx |    |    | xx | 10 | 8  | 6  |
| 4  | 6  | xx |    |    |    |    | xx | 6  | 4  |
| 2  | xx |    |    |    |    |    |    | xx | 2  |
| xx |    |    |    |    |    |    |    |    | xx |

Figure 6.1.1 Annihilation pattern for parallel QZ when n=10

|    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|
| xx |    |    |    |    |    |    |    |    |    | xx |
| 1  | xx |    |    |    |    |    |    |    | xx | 1  |
| 3  | 5  | xx |    |    |    |    |    | xx | 5  | 3  |
| 5  | 7  | 9  | xx |    |    |    | xx | 9  | 7  | 5  |
| 7  | 9  | 11 | 13 | xx |    | xx | 13 | 11 | 9  | 7  |
| 9  | 11 | 13 | 15 | 16 | xx | 16 | 15 | 13 | 11 | 9  |
| 8  | 10 | 12 | 14 | xx |    | xx | 14 | 12 | 10 | 8  |
| 6  | 8  | 10 | xx |    |    |    | xx | 10 | 8  | 6  |
| 4  | 6  | xx |    |    |    |    |    | xx | 6  | 4  |
| 2  | xx |    |    |    |    |    |    |    | xx | 2  |
| xx |    |    |    |    |    |    |    |    |    | xx |

Figure 6.1.2 Annihilation pattern for parallel QZ when n=11

The QR decomposition using the standard Givens annihilation pattern of Sameh and Kuck on a (10x10) matrix will take 17 steps (Figure 6.1.3) as compared to the QZ decomposition method which takes 14 steps as shown in Figure 6.1.1.

|   |    |    |    |    |    |    |    |    |  |
|---|----|----|----|----|----|----|----|----|--|
|   |    |    |    |    |    |    |    |    |  |
| 9 |    |    |    |    |    |    |    |    |  |
| 8 | 10 |    |    |    |    |    |    |    |  |
| 7 | 9  | 11 |    |    |    |    |    |    |  |
| 6 | 8  | 10 | 12 |    |    |    |    |    |  |
| 5 | 7  | 9  | 11 | 13 |    |    |    |    |  |
| 4 | 6  | 8  | 10 | 12 | 14 |    |    |    |  |
| 3 | 5  | 7  | 9  | 11 | 13 | 15 |    |    |  |
| 2 | 4  | 6  | 8  | 10 | 12 | 14 | 16 |    |  |
| 1 | 3  | 5  | 7  | 9  | 11 | 13 | 15 | 17 |  |

Figure 6.1.3 Givens annihilation pattern for n=10

## 6.2 Sequential and Parallel Algorithms for QR and QZ

Both QR and QZ algorithms were developed for a shared-memory parallel computer and implemented on the Sequent Balance. The sequential algorithms are shown in Algorithm 6.2.1 for QR decomposition and Algorithm 6.2.2 for the QZ decomposition. When parallelising both the QR and QZ methods, the dynamic scheduling strategy was more feasible because the number of annihilations that can be performed concurrently varies at each step. For both the QR and QZ methods, the annihilations to be performed are generated and kept in a task queue and the annihilation is performed as the processors are available. Although more overheads are generated in keeping the jobs in the task queue, load balancing of the processors is ensured. Algorithm 6.2.3 shows the parallel QR decomposition while Algorithm 6.2.4 shows the parallel QZ algorithm.

The segments of the algorithm that is done in parallel is enclosed between the *par* and *end par* statements. The columns and rows of the matrix  $A$  are numbered from 0 to  $n-1$ . The right hand side vector  $b$  is augmented to the matrix  $A$  and becomes the  $n$ th column of the matrix. All algorithms assume that  $n$  is even. The algorithms are described in an algorithmic language and hence they cannot be directly converted to running programs without some slight modifications. Statements enclosed within { } are comments. The implementation of the parallel programs is dependent upon the compiler of the shared memory parallel computer in use.

### Algorithm 6.2.1 Sequential QR Decomposition

```
for k = 0 to n-2
  for i = k+1 to n-1
    { annihilate A(i,k) }
    d = sqrt(A(i,i)*A(i,i) + A(i-1,i)*A(i-1,i))
    cos_t = A(i,i)/d
    sin_t = A(i-1,i)/d
    A(i,i) = d
    for j = i+1 to n
      A(i,j) = cos_t * A(i,j) + sin_t * A(i-1,j)
      A(i-1,j) = cos_t * A(i-1,j) - sin_t * A(i,j)
    end for j
  end for i
end for k
```



### Algorithm 6.2.2 Sequential QZ Decomposition

```

for k = 0 to n/2-1
  nk = n-1-k
  for i = k+1 to nk-1
    { annihilate A(i,k) and A(i,n-1-k) }
    d1 = A(nk,nk) * A(i,k) - A(nk,k) * A(i,nk)
    d2 = A(k,nk) * A(nk,k) - A(k,k) * A(nk,nk)
    d3 = A(k,k) * A(i,nk) - A(k,nk) * A(i,k)
    sin_t = d3 / sqrt(d1*d1 + d3*d3)
    cos_t = d1 / sqrt(d1*d1 + d3*d3)
    sin_f = d2 / sqrt(d1*d1 + d2*d2 + d3*d3)
    cos_f = sqrt(d1*d1 + d3*d3) // sqrt(d1*d1 + d2*d2 + d3*d3)
    for j = k+1 to nk-1
      A(k,j) = A(k,j) * sin_t - A(nk,j) * cos_t
      A(i,j) = (A(k,j) * cos_t + A(nk,j) * sin_t) * cos_f + A(i,j) * sin_f
      A(nk,j) = sin_f * (A(k,j) * cos_t + A(nk,j) * sin_t) - A(i,j) * cos_f
    end for j
    /* update RHS */
    A(k,n) = A(k,n) * sin_t - A(nk,n) * cos_t
    A(i,n) = (A(k,n) * cos_t + A(nk,n) * sin_t) * cos_f + A(i,n) * sin_f
    A(nk,n) = (A(k,n) * cos_t + A(nk,n) * sin_t) * sin_f - A(i,n) * cos_f
    /* update top and bottom pivot rows */
    A(k,k) = A(nk,k) / cos_f
    A(k,nk) = A(nk,nk) / cos_f
    A(nk,k) = A(i,k) * cos_t - A(k,k) * sin_t
    A(nk,nk) = A(i,nk) * cos_t - A(k,nk) * sin_t
  end for i
end for k

```

When parallelising the GE, PIE, LU and QIF algorithms, static scheduling was employed because it was known in advance the number of loop iterations to be performed to complete the method. These loop iterations were partitioned and assigned to be executed by the available processors.

In the QR and QZ methods, the number of rotations that can be done in parallel varies at each step, hence dynamic scheduling seemed more appropriate. This was achieved by generating a list of tasks that can be performed in parallel and keeping them in a task queue. The tasks are then assigned to the available processors.

For the parallelisation of the QR method, the generation of tasks was done in two parts (refer to Figure 6.2.1 for a 10x10 example). This is due to the programming restriction on the Sequent Balance. The first **par** loop of Algorithm 6.2.3 was to generate the annihilation tasks of the shaded elements of the lower triangular matrix in Figure 6.2.1. The rest of the annihilation tasks in the lower triangular matrix of Figure 6.2.1 is generated by the second **par** loop in Algorithm 6.2.3.

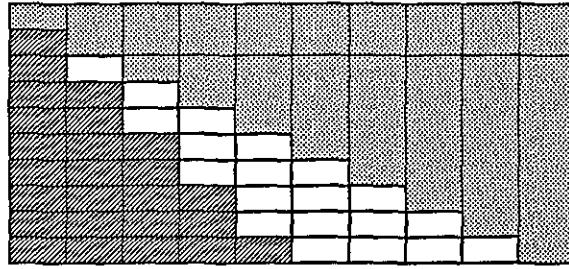


Figure 6.2.1 QR annihilation pattern for a 10x10 matrix

### Algorithm 6.2.3 Parallel QR Decomposition

```

jstart = 0
for k = n-1 downto 0
    i = k
    j = jstart
    par
        while i <= n-1
            annihilate A(i,j)
            j = j + 1
            i = i + 2
        end while
    end par
end for k
jstart = jstart + 1
for k = 2 to n - 1
    i = k
    j = jstart
    par
        while i <= n-1
            annihilate A(i,j)
            j = j + 1
            i = i + 2
        end while
    end par
    jstart = jstart + 1
end for k

```

In parallelising the QZ algorithm, the first  $k$  loops of Algorithm 6.2.4 generates the unshaded annihilation tasks list in Figure 6.2.2. The first **par** loop generates the upper half while the second **par** loop generates bottom half. Similarly, the second  $k$  loop of Algorithm 6.2.4 generates the annihilation tasks list for the striped elements in Figure 6.2.2. Once again, the first **par** loop generates the upper half of the tasks while the second **par** loop generates the bottom half. It is anticipated then that with four **par** loops for dynamic scheduling, there will be more overheads generated to maintain the task list as compared to the parallel QR algorithm. As in the parallel QR, these four loops are required due to the programming restriction on the Sequent Balance.

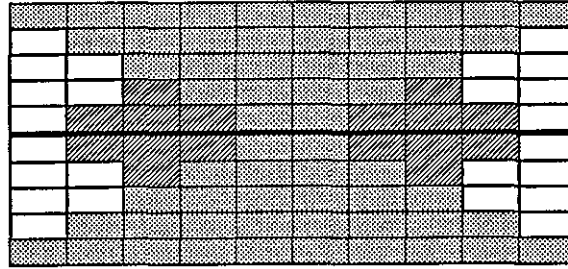


Figure 6.2.2 QZ annihilation pattern for a 10x10 matrix

#### Algorithm 6.2.4 Parallel QZ Decomposition

```

num_in_par = 1 { number of rotations to be done in parallel }
if n is even, m=n/2-1 else m=n/2
for k = 1 to n/2 -1
    if k is odd, add 1 to num_in_par
    i_up = k, j_up = 1
    par
        while j_up < num_in_par
            annihilate A(i_up,j_up-1) and A(i_up,n-1-(j_up-1))
            i_up = i_up -1, j_up = j_up + 1
        end while
    end par
    i_down = n-k-1, j_down = 1
    par
        while j_down < num_in_par
            annihilate A(i_down,j_down-1) and A(i_down,n-1-(j_down-1))
            i_down = i_down + 1, j_down = j_down + 1
        end while
    end par
end for k
if n is odd, num_in_par = (m-1)/2+1 else num_in_par = (m-1)/2
for k = 1 to m-1
    nk = m+1, i_up = m, j_up = k
    par
        while j_up < k + num_in_par
            annihilate A(i_up,j_up) and A(i_up,n-1-j_up)
            i_up = i_up -1, j_up = j_up + 1
        end while
    end par
    i_down = nk, j_down = k
    par
        while j_down < num_in_par
            annihilate A(i_down,j_down) and A(i_down,n-1-j_down)
            i_down = i_down + 1, j_down = j_down + 1
        end while
    end par
    if m-1 is even and k is even
        num_in_par = num_in_par -1
    if m-1 is odd and k is odd
        num_in_par = num_in_par -1
end for k

```

### 6.3 Computational Work and Memory Accesses of the QR and QZ methods

In this analysis, only the decomposition phase is considered as it is the most computationally expensive stage in both QR and QZ methods. Furthermore, the back substitution process of QR and the bi-directional solution process of QZ have been covered in depth in chapter 4. The computational work and shared memory access count of QR is covered in section 6.3.1 and QZ in section 6.3.2. A summary of the analysis is given in section 6.3.3. To simplify the analysis in this chapter, all operations are assumed to execute in the same amount of time and shall be termed as *ops*.

#### 6.3.1 QR method

The analysis in this section refers to Algorithm 6.2.1. In order to evaluate the values of  $\sin \theta$  and  $\cos \theta$  requires 2 mults, 2 div, 1 add and 1 sqrt. Assuming all operations take the same amount of time to execute, this is a total of

$$\begin{aligned}
 & \sum_{j=1}^{n-1} j \text{ (6 ops)} \\
 &= \left( \frac{1}{2}n^2 - \frac{1}{2}n \right) [6\text{ops}] \\
 &= 3n^2 - 3n \text{ ops}
 \end{aligned} \tag{6.3.1.1}$$

To calculate the new entries for the reduced matrix and the RHS is

$$\begin{aligned}
 & \sum_{j=1}^{n-1} (2m+1a)(2j+1)(j-1) \\
 &= \sum_{j=1}^{n-1} (3\text{ops})(2j^2 - j - 1) \\
 &= \left( \frac{2}{3}n^3 - \frac{3}{2}n^2 + \frac{5}{6}n - 1 \right) [3\text{ops}] \\
 &= 2n^3 - \frac{9}{2}n^2 + \frac{5}{2}n - 3 \text{ ops}
 \end{aligned} \tag{6.3.1.2}$$

Therefore, the total amount of computational work required by the QR algorithm is

$$2n^3 - \frac{3}{2}n^2 - \frac{1}{2}n - 3 \text{ ops} \tag{6.3.1.3}$$

The amount of shared memory accesses required during the calculation of  $\sin \theta$  and  $\cos \theta$  is

$$3 \sum_{j=1}^{n-1} j = \frac{3}{2}n^2 - \frac{3}{2}n \quad (6.3.1.4)$$

and the shared memory accesses for the calculation of the reduced matrix and the RHS is

$$\begin{aligned} 4 \sum_{j=1}^{n-1} (2j+1)(j-1) &= 4 \left( \frac{2}{3}n^3 - \frac{3}{2}n^2 + \frac{5}{6}n - 1 \right) \\ &= \frac{8}{3}n^3 - 6n^2 + \frac{10}{3}n - 4 \end{aligned} \quad (6.3.1.5)$$

Hence, the total sum of shared memory accesses required by the QR factorisation algorithm is

$$\frac{8}{3}n^3 - \frac{9}{2}n^2 + \frac{11}{6}n - 4 \quad (6.3.1.6)$$

### 6.3.2 QZ method

In a similar manner the amount of computational work required for the QZ algorithm (Algorithm 6.2.2) can be calculated.

To evaluate  $\Delta_1$ ,  $\Delta_2$  and  $\Delta_3$  involves  $3(2 \text{ mults} + 1a)$ .

Similarly, we have  $\sin \theta = (2 \text{ mults} + 1a + 1\text{div} + 1\text{sqrt})$ ,

$$\cos \theta = 1 \text{ div},$$

$$\sin \phi = (1 \text{ mult} + 1a + 1 \text{ div} + 1 \text{ sqrt}),$$

$$\text{and } \cos \phi = 1 \text{ div}.$$

Assuming all operations take the same time to execute, the above evaluation then requires 20 ops. In total, the complete evaluation process requires

$$\begin{aligned} & \sum_{j=2(2)}^n (10\text{ops})(j-2) \\ &= 10 \sum_{j=1}^{n/2} 2j - 20 \\ &= 10 * 2 \sum_{j=1}^{n/2} j - 20 \\ &= 20 \left( \frac{n^2}{4} + \frac{n}{2} \right) - 20 \\ &= 20 \left( \frac{n^2}{4} + \frac{n}{2} - 1 \right) \text{ ops.} \end{aligned} \tag{6.3.2.1}$$

To calculate the new entries for the reduced matrix requires

$$\begin{aligned} & \sum_{j=2(2)}^n (8m + 4a)(j-2)(j-2) \\ &= \sum_{j=2(2)}^n (12\text{ops})(j^2 - 4j + 4) \\ &= 12 \sum_{j=1}^{n/2} 2j^2 - 12 * 4 \sum_{j=1}^{n/2} 2j + 48 \\ &= 12 * 2 \left( \frac{1}{6} * \frac{n}{2} \left( \frac{n}{2} + 1 \right) (n+1) \right) - 48 * 2 \left( \frac{n}{2} \left( \frac{n}{2} + 1 \right) \right) + 48 \\ &= n^3 - 21n^2 - 46n + 48 \text{ ops.} \end{aligned} \tag{6.3.2.2}$$

The updating of the RHS requires

$$\begin{aligned}
 & \sum_{j=2(2)}^n (6m+3a)(j-2) \\
 &= \sum_{j=2(2)}^n (9\text{ops})(j-2) \\
 &= 18 \left( \frac{n^2}{4} + \frac{n}{2} - 1 \right) \text{ ops.}
 \end{aligned} \tag{6.3.2.3}$$

The updating of the top and bottom pivot row corners requires

$$\begin{aligned}
 & \sum_{j=2(2)}^n (4m+2a)(j-2) + \sum_{j=2(2)}^n (8m+4a)(j-2) \\
 &= \sum_{j=2(2)}^n (18\text{ops})(j-2) \\
 &= 36 \left( \frac{n^2}{4} + \frac{n}{2} - 1 \right) \text{ ops.}
 \end{aligned} \tag{6.3.2.4}$$

Thus the total amount of computational work required by QZ decomposition is

$$n^3 - \frac{5}{2}n^2 - 9n - 26 \text{ ops.} \tag{6.3.2.5}$$

For the shared memory accesses, the evaluation of  $\Delta_1$ ,  $\Delta_2$  and  $\Delta_3$ ,  $\sin \theta$ ,  $\cos \theta$ ,  $\sin \phi$  and  $\cos \phi$  requires

$$\begin{aligned}
 & \sum_{j=2(2)}^n 6(j-2) \\
 &= 12 \left( \frac{n^2}{4} + \frac{n}{2} - 1 \right) \text{ accesses.}
 \end{aligned} \tag{6.3.2.6}$$

The updating of new entries for the reduced matrix requires

$$\begin{aligned}
 & \sum_{j=2(2)}^n 9(j-2)(j-2) \\
 &= \sum_{j=2(2)}^n 9(j^2 - 4j + 4) \\
 &= \frac{3}{4}n^3 - \frac{63}{4}n^2 - \frac{69}{2}n + 36
 \end{aligned} \tag{6.3.2.7}$$

accesses to shared memory.

Also, the updating of the RHS requires

$$\sum_{j=2(2)}^n 6(j-2) = 12 \left( \frac{n^2}{4} + \frac{n}{2} - 1 \right) \tag{6.3.2.8}$$

shared memory accesses.

The updating of the top and bottom pivot row corners requires

$$\sum_{j=2(2)}^n 6(j-2) = 12 \left( \frac{n^2}{4} + \frac{n}{2} - 1 \right) \tag{6.3.2.9}$$

shared memory accesses.

Finally, the total amount of shared memory accesses required by QZ decomposition is

$$\frac{3}{4}n^3 - \frac{27}{4}n^2 - \frac{33}{2}n \tag{6.3.2.10}$$

### 6.3.3 A Summary

A summary of the computational work and the shared memory access count for QR and QZ methods is given in Table 6.3.3.1.

| Method | Computational Complexity                   | Shared Memory Access Count                            |
|--------|--|---|
| QR     | $2n^3 - \frac{3}{2}n^2 - \frac{1}{2}n - 3$ | $\frac{8}{3}n^3 - \frac{9}{2}n^2 + \frac{11}{6}n - 4$ |
| QZ     | $n^3 - \frac{5}{2}n^2 - 9n - 26$           | $\frac{3}{4}n^3 - \frac{27}{4}n^2 - \frac{33}{2}n$    |

Table 6.3.3.1 A summary of the computational complexity and shared memory access count of QR and QZ methods.

Thus from Table 6.3.3.1 it is clear that the QR and QZ factorisations take roughly the same amount of computational work but the shared memory access count for QR is roughly twice more than that for the QZ factorisation.



## 6.4 Numerical Results

The parallel programs for both QR and QZ methods have been implemented using dynamic scheduling on the Sequent Balance. This was considered to be the best strategy to implement them as the number of elements to be eliminated in parallel varies each time. Although dynamic scheduling will ensure a balanced load across the available processors, there are still some overheads incurred in maintaining the task queue.

The numerical tests were performed on the Sequent Balance system which has 10 processors. All the processors are plugged into a single bus and share a common memory. Each processor has 8 Kb of cache memory. The algorithms were implemented in single precision using the C language. The parallel constructs were supported by the Sequent C library.

In order to avoid the overhead of switching from one process to another, only a single process was assigned to each processor and the tests were completed while no other users' tasks were using the Balance. All the programs were written with the same accuracy to obtain a meaningful comparison.

The exact solution of all the systems is chosen to be  $x=(1, \dots, 1)^T$ . Matrix  $A$  is of the following kind in all the tests,

$$A = \{a_{ij} \mid a_{ij} = \text{abs}(i-j)/10.0, i \neq j, a_{ii} = 0.001\},$$

where  $n = 600, 800$  and  $1000$ .

Table 6.4.1 shows the timing (in seconds) of the parallel QR and parallel QZ decomposition programs. The speedup and efficiency of both the decomposition methods are shown in Tables 6.4.2 and 6.4.3 respectively.

Figure 6.4.1 shows the timing of QR and QZ decomposition methods in graphical form while Figure 6.4.2 shows the temporal performance of both methods.

| n    | Method  | Number of processors |          |          |          |         |         |
|------|---------|----------------------|----------|----------|----------|---------|---------|
|      |         | 1                    | 2        | 4        | 6        | 8       | 10      |
| 600  | QR      | 13532.9              | 6774.66  | 3411.89  | 2289.54  | 1729.17 | 1389.78 |
|      | QZ      | 12200.49             | 6070.89  | 3073.66  | 2072.86  | 1572.16 | 1280.96 |
|      | Gain(%) | 9.85                 | 10.39    | 9.91     | 9.46     | 9.08    | 7.83    |
| 800  | QR      | 32119.79             | 16027.8  | 8060.43  | 5388.72  | 4071.03 | 3268.78 |
|      | QZ      | 28670.8              | 14386.06 | 7260.72  | 4874.45  | 3684.87 | 2973.42 |
|      | Gain(%) | 10.74                | 10.24    | 9.92     | 9.54     | 9.49    | 9.04    |
| 1000 | QR      | 62887.9              | 31249.58 | 15760.28 | 10519.21 | 7915.58 | 6676.59 |
|      | QZ      | 55943.02             | 28020.09 | 14101.17 | 9469.6   | 7266.2  | 5757.75 |
|      | Gain(%) | 11.04                | 10.33    | 10.53    | 9.98     | 8.2     | 13.76   |

Table 6.4.1- Timings of parallel QR and parallel QZ decomposition

| n    | Method | Number of processors |      |      |      |      |
|------|--------|----------------------|------|------|------|------|
|      |        | 2                    | 4    | 6    | 8    | 10   |
| 600  | QR     | 1.99                 | 3.96 | 5.91 | 7.82 | 9.73 |
| 800  |        | 2.00                 | 3.98 | 5.96 | 7.89 | 9.83 |
| 1000 |        | 2.01                 | 3.99 | 5.97 | 7.94 | 9.41 |
| 600  | QZ     | 2.00                 | 3.97 | 5.89 | 7.76 | 9.52 |
| 800  |        | 1.99                 | 3.94 | 5.88 | 7.78 | 9.64 |
| 1000 |        | 1.99                 | 3.97 | 5.91 | 7.7  | 9.72 |

Table 6.4.2- Speedup of parallel QR and parallel QZ decomposition

| n    | Method | Number of processors |        |       |        |       |
|------|--------|----------------------|--------|-------|--------|-------|
|      |        | 2                    | 4      | 6     | 8      | 10    |
| 600  | QR     | 0.995                | 0.99   | 0.985 | 0.9775 | 0.973 |
| 800  |        | 1.0                  | 0.995  | 0.993 | 0.986  | 0.983 |
| 1000 |        | 1.005                | 0.9975 | 0.995 | 0.993  | 0.941 |
| 600  | QZ     | 1.0                  | 0.993  | 0.982 | 0.97   | 0.952 |
| 800  |        | 0.995                | 0.985  | 0.98  | 0.973  | 0.964 |
| 1000 |        | 0.995                | 0.993  | 0.985 | 0.963  | 0.972 |

Table 6.4.3- Efficiency of parallel QR and parallel QZ decomposition

It can be seen from the results that there is a gain of roughly 10% in execution time yielded by the QZ decomposition method. Both the QR and QZ methods have a linear speedup and the efficiency of both methods lie between 95-100%. The temporal performance graph also shows the QZ decomposition method to be superior to QR in permitting more parallelism to be achieved.

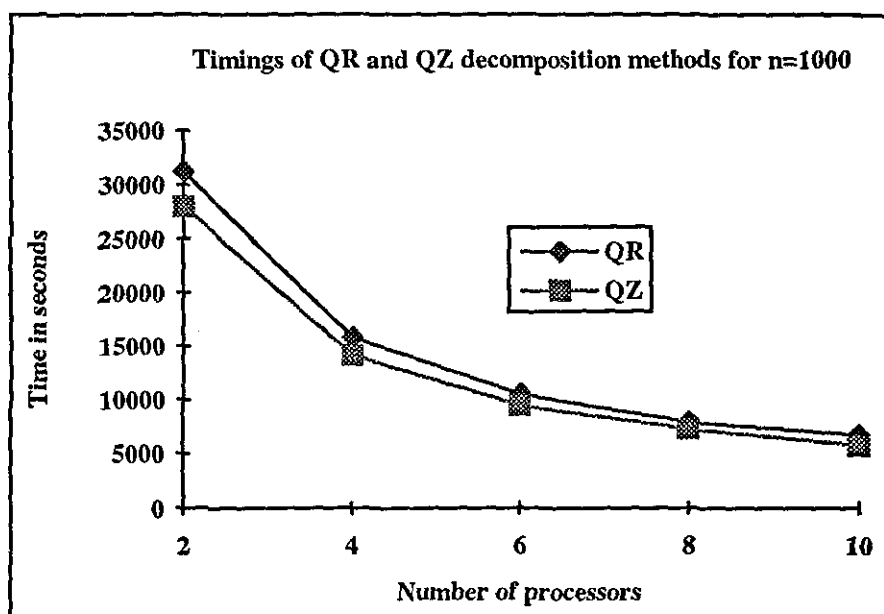


Figure 6.4.1 Timings of QR and QZ decomposition methods for  $n=1000$

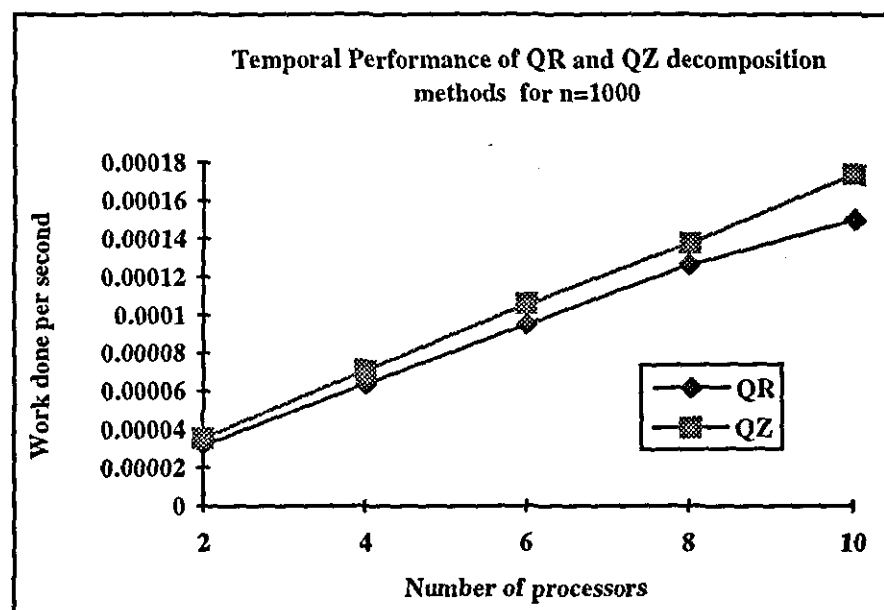


Figure 6.4.2 The temporal performance of the QR and QZ methods for  $n=1000$

### 6.5 The Greedy Approach

The parallelisation of the QZ method discussed in the earlier section was based on the standard Givens annihilation pattern. Another annihilation strategy for QR proposed by Modi and Clarke [Modi 84] was that based on the greedy strategy. In this section, the greedy strategy for the QZ algorithm is proposed.

Consider a 10x10 matrix example. The greedy QR annihilation pattern is shown in Figure 6.5.1 and the greedy QZ annihilation pattern is shown in Figure 6.5.2.

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| xx |    |    |    |    |    |    |    |    |    |
| 4  | xx |    |    |    |    |    |    |    |    |
| 3  | 6  | xx |    |    |    |    |    |    |    |
| 2  | 5  | 8  | xx |    |    |    |    |    |    |
| 2  | 4  | 7  | 10 | xx |    |    |    |    |    |
| 1  | 4  | 6  | 9  | 11 | xx |    |    |    |    |
| 1  | 3  | 5  | 8  | 10 | 12 | xx |    |    |    |
| 1  | 3  | 5  | 7  | 9  | 11 | 13 | xx |    |    |
| 1  | 2  | 4  | 6  | 8  | 10 | 12 | 14 | xx |    |
| 1  | 2  | 3  | 5  | 7  | 9  | 11 | 13 | 15 | xx |

Figure 6.5.1 The greedy QR annihilation pattern for a 10x10 matrix

In the greedy QR strategy, the maximum possible number of rotations is performed in each step, with the elements in one column being annihilated from bottom to top and those in each row from left to right. Execution is fast to start with, as 5 rotations are performed in step 1, 4 in steps 2, 3, 4 and 5, 3 in steps 6, 7, 8, 9, 10 and 11. But it ends slowly with two rotations in steps 12 and 13 and only 1 in the last two steps.

Similarly in the QZ greedy strategy, the maximum possible rotations is performed in each step too, with the elements being annihilated from left to right and right to left simultaneously. In steps 1 and 2, 4 rotations are performed simultaneously with each rotation zeroing 2 elements at a time. Three rotations are performed in steps 3 and 4, 2 in steps 5 and 6 and 1 in the last two steps.

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| xx |    |    |    |    |    |    |    |    | xx |
| 1  | xx |    |    |    |    |    |    | xx | 1  |
| 2  | 1  | xx |    |    |    |    | xx | 1  | 2  |
| 3  | 2  | 1  | xx |    |    | xx | 1  | 2  | 3  |
| 4  | 3  | 2  | 1  | xx | xx | 1  | 2  | 3  | 4  |
| 5  | 4  | 3  | 2  | xx | xx | 2  | 3  | 4  | 5  |
| 6  | 5  | 4  | xx |    |    | xx | 4  | 5  | 6  |
| 7  | 6  | xx |    |    |    |    | xx | 6  | 7  |
| 8  | xx |    |    |    |    |    |    | xx | 8  |
| xx |    |    |    |    |    |    |    |    | xx |

Figure 6.5.2 The greedy QZ annihilation pattern for a 10x10 matrix

To summarise, the standard Givens annihilation pattern would take 17 steps to factorise the matrix while the standard QZ takes 14 steps for the given 10x10 example. Similarly, the greedy QR method takes 15 steps while the greedy QZ method takes 8 steps, a reduction of almost 50%. In general, the greedy QZ method takes  $n-2$  steps to transform a matrix of size  $nxn$ .

The results in section 6.4 show a gain of 10% in execution time for the standard annihilation pattern employed in the parallelisation of QR and QZ. It is anticipated that using the greedy strategy in parallelising QR and QZ would result in larger gains.

## 6.6 Summary

In this chapter, the Givens QZ orthogonal method has been described. The design of the sequential and parallel algorithm was also presented together with that of QR method. An analysis of the computational work and shared memory access was also performed for both QR and QZ factorisation.

In terms of computational work, QR and QZ both share the same computational complexity of  $2n^3$ . However, there is a difference in the shared memory access count where QZ factorisation has a shared memory access of order  $(3/2)n^3$  while QR factorisation has a shared memory access of order  $(8/3)n^3$ .

Both algorithms were implemented on the Sequent Balance, a shared memory parallel computer, and it was shown that QZ factorisation is faster than QR factorisation by 10% on average.

Although there is a larger difference of shared memory accesses between QR and QZ as compared to the PIE and QIF methods, the amount of computational work for QZ is greater requiring more overheads than QR and which erodes the possible gains. Furthermore, there were also more overheads incurred in maintaining the task queue in the implementation of parallel QZ as it had twice more parallel loops than the QR factorisation.

## Chapter 7

### Quadrant Interlocking (QI) Iterative methods on Shared Memory Parallel Computer

Iterative methods are procedures to solve systems of linear equations of the form  $Ax=b$  by generating a sequence of approximations to the solution vector  $x$ . These methods start with an arbitrary first approximation to  $x$  and then improve this estimate in a convergent sequence of steps. Iterative algorithms are frequently used to solve the large sparse linear systems generated when working with partial differential equations using discrete methods.

The Jacobi and the Gauss-Seidel iterations are two well known iterative methods that solve linear systems of equations. Parallel versions of Jacobi and Gauss-Seidel methods have been developed and widely implemented on parallel computers. In this chapter, a new class of iterative methods are investigated and their performance on a shared memory parallel computer is compared to the classical iterative methods of Jacobi and Gauss-Seidel methods. Accelerated methods for these new parallel iterative schemes are also investigated and compared with accelerated Jacobi and Gauss-Seidel methods. Iterative methods are built around a partition (or splitting) of matrix  $A$ . The conventional diagonal splitting of  $A$  into  $A=D-L-U$  where  $D$  is the main diagonal of  $A$ ,  $-L$  and  $-U$  are strictly lower and upper triangular elements of  $A$  respectively is the basis of the Jacobi and Gauss-Seidel methods. The Quadrant Interlocking (QI) iterative schemes split  $A$  into  $A=X-W-Z$ . The QI iterative methods were first developed by Evans and Sojoodi-Haghighi [Evans 82]. They proved many convergence theorems for these methods. What was not clear then was the rate of convergence of the new methods because they were not implemented on a parallel computer and how they compared to the classical iterative methods.

The new splitting scheme for the QI iterative method is described in section 7.1. The model problem used in this investigation is presented in section 7.2. The sequential and parallel implementation of the new schemes are discussed in section 7.3. Section 7.4 contains the discussion on computational complexity and shared memory accesses for the classical and QI iterative methods. The results of the numerical experiments are given in section 7.5 and a summary of the chapter is given in section 7.6.

## 7.1 Quadrant Interlocking Matrix Splitting Strategy

Consider the linear system of equations

$$Ax=b \quad (7.1.1)$$

where  $A$  is a non-singular matrix of order  $n$ , with elements  $a_{ij}$ ;  $i,j=1(1)n$  and  $x$  and  $b$  are two  $n$ -dimensional vectors with  $x$  (unknown) and  $b$  (known) given by

$$\begin{aligned} x &= [x_1, x_2, \dots, x_n]^T, \\ b &= [b_1, b_2, \dots, b_n]^T. \end{aligned} \quad (7.1.2)$$

The matrix  $A$  is partitioned into the form

$$A = X - W - Z \quad (7.1.3)$$

where the structure of the matrices  $X$ ,  $-W$  and  $-Z$  are defined as follows:

If  $X_i$ ,  $W_i$ , and  $Z_i$ ,  $i=1,2,\dots,n$  are the column vectors of the matrices  $X$ ,  $-W$  and  $-Z$  we shall have,

$$X = [X_1, X_2, \dots, X_n], \quad (7.1.4)$$

$$-W = [W_1, W_2, \dots, W_n], \quad (7.1.5)$$

$$\text{and} \quad -Z = [Z_1, Z_2, \dots, Z_n]. \quad (7.1.6)$$

The column vectors  $X_i$  have the following general form,

$$X_i \equiv \begin{cases} \begin{bmatrix} 0, \dots, 0, \underbrace{a_{ii}}_{i-1}, 0, \dots, 0, a_{n-i+1,i}, 0, \dots, 0 \end{bmatrix}^T, & i = 1(1) \left[ \frac{n+1}{2} \right], \\ \begin{bmatrix} 0, \dots, 0, \underbrace{a_{n-i+1,i}}_{n-i}, 0, \dots, 0, a_{ij}, 0, \dots, 0 \end{bmatrix}^T, & i = \left[ \frac{n+3}{2} \right] (1)n. \end{cases} \quad (7.1.7)$$

where the symbol  $[\alpha]$  denotes the largest integer less than or equal to  $\alpha$ . The column vectors  $W_i$  and  $Z_i$  have the following general forms:

For  $n$  odd,

$$W_i \equiv \begin{cases} \left[ \underbrace{0, \dots, 0}_i a_{i+1, j}, \dots, a_{n-i, j}, 0, \dots, 0 \right]^T, & i = 1(1) \frac{n-1}{2}, \\ \left[ 0, \dots, 0 \right]^T, & i = \frac{n+1}{2} \\ \left[ \underbrace{0, \dots, 0}_{n-i+1} a_{n-i+2, j}, \dots, a_{i-1, j}, 0, \dots, 0 \right]^T, & i = \left[ \frac{n+3}{2} \right] (1)n. \end{cases} \quad (7.1.8)$$

and

$$Z_i \equiv \begin{cases} \left[ \underbrace{0, \dots, 0}_i a_{i, j+1}, \dots, a_{i, n-1}, 0, \dots, 0 \right]^T, & i = 1(1) \frac{n-1}{2}, \\ \left[ 0, \dots, 0 \right]^T, & i = \frac{n+1}{2} \\ \left[ \underbrace{0, \dots, 0}_{n-i+1} a_{i, n-i+2}, \dots, a_{i, j-1}, 0, \dots, 0 \right]^T, & i = \left[ \frac{n+3}{2} \right] (1)n. \end{cases} \quad (7.1.9)$$

For  $n$  even,

$$W_i \equiv \begin{cases} \left[ \underbrace{0, \dots, 0}_i a_{i+1, j}, \dots, a_{n-i, j}, 0, \dots, 0 \right]^T, & i = 1(1) \frac{n}{2} - 1, \\ \left[ 0, \dots, 0 \right]^T, & i = \frac{n}{2}, \frac{n}{2} + 1, \\ \left[ \underbrace{0, \dots, 0}_{n-i+1} a_{n-i+2, j}, \dots, a_{i-1, j}, 0, \dots, 0 \right]^T, & i = \frac{n}{2} + 2(1)n. \end{cases} \quad (7.1.10)$$

and

$$Z_i \equiv \begin{cases} \left[ \underbrace{0, \dots, 0}_i a_{i, j+1}, \dots, a_{i, n-1}, 0, \dots, 0 \right]^T, & i = 1(1) \frac{n}{2} - 1, \\ \left[ 0, \dots, 0 \right]^T, & i = \frac{n}{2}, \frac{n}{2} + 1 \\ \left[ \underbrace{0, \dots, 0}_{n-i+1} a_{i, n-i+2}, \dots, a_{i, j-1}, 0, \dots, 0 \right]^T, & i = \frac{n}{2} + 2(1)n. \end{cases} \quad (7.1.11)$$



The four basic Quadrant Interlocking (QI) iterative methods for the system (7.1.1) can now be defined. The elements of the solution vector  $x$  can be derived in  $[n+1/2]$  distinct steps where in each step the following (2x2) linear systems are being solved.

$$\left. \begin{aligned} a_{i,i} x_i + a_{i,n-i+1} x_{n-i+1} &= c_i \\ a_{n-i+1,i} x_i + a_{n-i+1,n-i+1} x_{n-i+1} &= c_{n-i+1} \end{aligned} \right\} i = 1(1) \left[ \frac{n+1}{2} \right] \quad (7.1.12)$$

where

$$\left. \begin{aligned} c_i &= - \sum_{j=1}^n a_{i,j} x_j + b_i \\ c_{n-i+1} &= - \sum_{j=1}^n a_{n-i+1,j} x_j + b_{n-i+1} \end{aligned} \right\} j \neq i \quad \& \quad n-i+1 \quad (7.1.13)$$

therefore, if

$$\Delta_i \equiv \det \begin{bmatrix} a_{i,i} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix} \neq 0 \quad (7.1.14)$$

we obtain the unknowns  $x_i$  and  $x_{n-i+1}$ ,  $i = 1, 2, \dots, [n+1/2]$  from the formulae

$$x_i = \frac{(c_i \times a_{n-i+1,n-i+1} - c_{n-i+1} \times a_{i,n-i+1})}{\Delta_i}, \quad (7.1.15)$$

$$x_{n-i+1} = \frac{(c_{n-i+1} \times a_{i,i} - c_i \times a_{n-i+1,i})}{\Delta_i} \quad (7.1.16)$$

The (2x2) linear systems (7.1.12) can be solved by using the symmetric elimination and Gauss elimination methods which were shown in chapter 2. In addition, Cramers Rule can be used if it is diagonally dominant.

Note that if  $n$  is odd, (7.1.12) is a single equation.

It can be easily established that the system (7.1.1) has been replaced by the equivalent system

$$x = Bx + c \quad (7.1.17)$$

$$\text{where} \quad B = X^{-1}(W + Z) \quad (7.1.18)$$

$$\text{and} \quad c = X^{-1}b \quad (7.1.19)$$

Clearly, if the  $\Delta_i \neq 0$ ,  $i = 1, 2, \dots, \left[ \frac{n+1}{2} \right]$ , then  $X^{-1}$  exists.

### 7.1.1 Simultaneous Quadrant Interlocking iterative method (J.Q.I.)

In this method we choose arbitrary starting values

$x_i^{(0)}, i = 1, 2, \dots, n$  and compute  $x_i^{(1)}$  from (7.1.15) and (7.1.16) in pairs, using  $x_i^{(0)}$  in the right - hand side vector (7.1.13).

Then  $x_i^{(2)}$  is determined from  $x_i^{(1)}$  in the usual manner.

In general, given  $x_i^{(k)}, x_{n-i+1}^{(k+1)}$  can be determined by

$$\left. \begin{aligned} a_{ii}x_i^{(k+1)} + a_{i,n-i+1}x_{n-i+1}^{(k+1)} &= c_i^{(k)} \\ a_{n-i+1,i}x_i^{(k+1)} + a_{n-i+1,n-i+1}x_{n-i+1}^{(k+1)} &= c_{n-i+1}^{(k)} \end{aligned} \right\} i = 1(1)\left[\frac{n+1}{2}\right] \quad (7.1.1.1)$$

where

$$\left. \begin{aligned} c_i^{(k)} &= - \sum_{j=1}^n a_{i,j}x_j^{(k)} + b_i \\ c_{n-i+1}^{(k)} &= - \sum_{j=1}^n a_{n-i+1,j}x_j^{(k)} + b_{n-i+1} \end{aligned} \right\} j \neq i \text{ \& } n-i+1 \quad (7.1.1.2)$$

Note that for  $n$  odd, the linear system in (7.1.1.1) is a single equation.

In matrix form, this can be expressed as

$$x^{(k+1)} = Bx^{(k)} + c \quad (7.1.1.3)$$

### 7.1.2 Simultaneous Overrelaxation iterative method (J.O.Q.I.)

In this method, a real parameter  $\omega$  is chosen and (7.1.1.1) is replaced by

$$\left. \begin{aligned} a_{ii}x_i^{(k+1)} + a_{i,n-i+1}x_{n-i+1}^{(k+1)} &= \omega c_i^{(k)} + (1-\omega)\left(a_{i,i}x_i^{(k)} + a_{i,n-i+1}x_{n-i+1}^{(k)}\right) \\ a_{n-i+1,i}x_i^{(k+1)} + a_{n-i+1,n-i+1}x_{n-i+1}^{(k+1)} &= \omega c_{n-i+1}^{(k)} + (1-\omega)\left(a_{n-i+1,i}x_i^{(k)} + a_{n-i+1,n-i+1}x_{n-i+1}^{(k)}\right) \end{aligned} \right\} \quad (7.1.2.1)$$

where  $c_i^{(k)}$  and  $c_{n-i+1}^{(k)}$  are defined in (7.1.1.2).

In matrix form, the J.O.Q.I. method is defined as,

$$x^{(k+1)} = B_{\omega}x^{(k)} + \omega c \quad (7.1.2.2)$$

$$\text{where } B_{\omega} = \omega B + (1-\omega)I \quad (7.1.2.3)$$

When  $\omega=1$  the JOQI method is equivalent to the J.Q.I. method.

### 7.1.3 Successive Quadrant Interlocking iterative method (S.Q.I.)

In this method at each step the most recent values of  $x_i^{(k+1)}$  is used when available.

Instead of (7.1.1.2) we now have

$$\left. \begin{aligned} c_i^{(k)} &= -\sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=n-i+2}^n a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n-i} a_{i,j} x_j^{(k)} + b_i \\ c_{n-i+1}^{(k)} &= -\sum_{j=1}^{i-1} a_{n-i+1,j} x_j^{(k+1)} - \sum_{j=n-i+2}^n a_{n-i+1,j} x_j^{(k+1)} - \sum_{j=i+1}^{n-i} a_{n-i+1,j} x_j^{(k)} + b_{n-i+1} \end{aligned} \right\} \quad (7.1.3.1)$$

$j \neq i \quad \& \quad j \neq n-i+1$

In matrix form, the S.Q.I. method can be defined as,

$$x^{(k+1)} = (X-W)^{-1} Z x^{(k)} + (X-W)^{-1} b \quad (7.1.3.2)$$

$$\text{or} \quad x^{(k+1)} = L_1 x^{(k)} + c \quad (7.1.3.3)$$

where  $L = (X-W)^{-1} Z$

and  $c = (X-W)^{-1} b$ .

### 7.1.4 Successive Overrelaxation iterative method (S.O.Q.I.)

In this method a real parameter  $\omega$  is introduced and the equation (7.1.3.2) is replaced by

$$x^{(k+1)} = (X-\omega W)^{-1} [\omega Z + (1-\omega)X] x^{(k+1)} + (X-\omega W)^{-1} b \quad (7.1.4.1)$$

$$\text{or} \quad x^{(k+1)} = L_\omega x^{(k)} + c_\omega, \quad (7.1.4.2)$$

where  $L_\omega = (X-\omega W)^{-1} (\omega Z + (1-\omega)X)$

and  $c_\omega = (X-\omega W)^{-1} b$

Again when  $\omega=1$  the SOQI method is equivalent to the S.Q.I. method.

## 7.2 The Model Problem

The model problem used to investigate the QI iterative methods is taken from a set of coupled ordinary differential equation (O.D.E.). The problem is described briefly in this section.

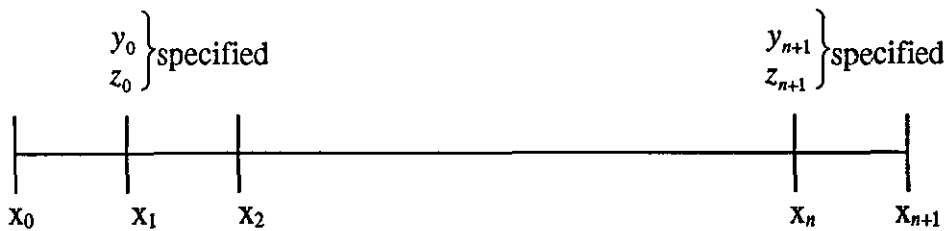
Given the problem,

$$-\frac{d^2 y}{dx^2} + py + qz = f_1(x), \quad (7.2.1)$$

$$-\frac{d^2 z}{dx^2} + qz + py = f_2(x), \quad (7.2.2)$$

with specified boundary values at the end points of the interval (0,1).

Discretisation of the equations (7.2.1) and (7.2.2) by central difference operators on a equally spaced grid of points of  $x_i, i=1,2,\dots,n$ ,



will result in the following set of finite difference equations,

$$\frac{(-y_{i-1} + 2y_i - y_{i+1}))}{h^2} + py_i + qz_i = f_1(x_i), i = 1, 2, \dots, n, \quad (7.2.3)$$

$$\frac{(-z_{i-1} + 2z_i - z_{i+1}))}{h^2} + qz_i + py_i = f_2(x_i), \text{ and } h = \frac{1}{n+1} \quad (7.2.4)$$

with the values  $y_0, y_{n+1}, z_0$ , and  $z_{n+1}$  specified as above and  $p = q = 1/h^2$ .

By setting up equations (7.2.3) and (7.2.4) in matrix form, and numbering the  $y_i, i=1,2,\dots,n$ , values in increasing order and the  $z_i$  values in reverse order, the following matrix equation is obtained.

$$\left[ \begin{array}{ccc|ccc} 3 & -1 & 0 & & & -1 \\ -1 & 3 & \ddots & & & \\ & \ddots & \ddots & & & \\ 0 & & -1 & 3 & \ddots & \\ \hline & & & \ddots & 3 & -1 & 0 \\ & & & & -1 & \ddots & \ddots \\ & & & & & \ddots & 3 & -1 \\ -1 & & & & 0 & & -1 & 3 \end{array} \right] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ z_n \\ z_{n-1} \\ \vdots \\ z_1 \end{bmatrix} = \begin{bmatrix} h^2 f_1(x_1) + h^2 y_0 \\ h^2 f_1(x_2) \\ \vdots \\ h^2 f_1(x_n) + h^2 y_{n+1} \\ h^2 f_2(x_n) + h^2 z_{n+1} \\ h^2 f_2(x_{n-1}) \\ \vdots \\ h^2 f_2(x_1) + h^2 z_0 \end{bmatrix} \quad (7.2.5)$$

The matrix  $A$  of (7.2.5) can be expressed as

$$A = 3(I - B) \quad (7.2.6)$$

where  $B$  is a  $n \times n$  real symmetric matrix given explicitly by

$$\frac{1}{3} \begin{bmatrix} 0 & 1 & & & & 1 \\ 1 & 0 & 1 & 1 & & \\ & 1 & \ddots & \ddots & & \\ & & \ddots & \ddots & 0 & 1 \\ 1 & & & & 1 & 0 \end{bmatrix} \quad (7.2.7)$$

### 7.3 Sequential and Parallel Algorithms for the Classical and QI iterative methods

The Jacobi, JOR, JQI and JOQI methods are well suited for implementation on parallel and distributed computers because the computation of each element of the vector  $\mathbf{x}$  may proceed simultaneously. The new estimate of the vector  $\mathbf{x}$  is computed from the old estimate of  $\mathbf{x}$  and the value of  $A$  and  $\mathbf{b}$ . On the other hand, the Gauss-Seidel, SOR, SQI and SOQI methods have a sequential nature and therefore require some special techniques such as reordering of equations (Red-Black or multi-colour) to improve the level of parallelism.

However, the performance of iterative methods as well as other parallel algorithms, is in general, degraded by synchronisation. Synchronisation ensures that the parallel implementation follows the exact execution flow of a sequential algorithm.

There are two classes of implementation for parallel algorithms, the synchronous algorithm and the asynchronous algorithm. For synchronous algorithms, the processors exchange all the necessary information regarding the results of the current iteration before a new iteration is initiated. Hence, the overall performance of a synchronous algorithm depends on the speed of the slowest processor. Asynchronous algorithms, on the other hand, carry out the computations totally independent of the execution of other processors. None of the processor has to wait for the slowest processor to complete its execution cycle. Research in this area was pioneered more than twenty years ago by Chazan and Miranker [Chazan 69] who studied the chaotic relaxation method. For several years, interest in this subject was rather limited. Dramatic progress in computer technology and the growing use of multiprocessor systems have

contributed to the renaissance of the asynchronous algorithms as demonstrated by their use in a variety of computational problems including parallel integration of differential equations [Mitra 87], discrete data problems [Uresin 90] and optimisation and network flow problems [Bertsekas 89].

In this thesis, both the synchronous and asynchronous iterative algorithms are investigated and the algorithms are presented in this section. The algorithms are organised as follows: First, a description of the sequential algorithms for the classical and QI iterative methods are given followed by the synchronous parallel versions of the classical and the QI iterative methods and lastly, the outline of the asynchronous algorithm employed for both the classical and the QI iterative methods.

In all the algorithms, only the iteration part of each iterative method is described. The initial step of setting a starting value for the  $x$  vector and the convergence test are omitted. A tolerance value of 0.0001 has been used in the convergence tests of all the algorithms. Vector **xold** is a vector that keeps the previous approximations for  $x$  while **xnew** retains the current approximation.

### *Sequential algorithms for classical and QI iterative methods*

For the model problem described in section 7.2, the sequential algorithm for the Jacobi iterative method and the Gauss-Seidel methods are given in Algorithm 7.3.1 and Algorithm 7.3.2 respectively. The Jacobi Overrelaxation (JOR) method and the Successive Overrelaxation (SOR) method are shown in Algorithms 7.3.3 and 7.3.4 respectively.

The sequential JQI and SQI algorithms are given in Algorithms 7.3.5 and 7.3.6 respectively. The JOQI and SOQI algorithms are given in Algorithms 7.3.7 and 7.3.8 respectively.

#### Algorithm 7.3.1 Sequential Jacobi Algorithm

```

for r=0 to n-1
  if r = 0
    xnew(r)=((xold(r+1)+xold(n-1))+1.0)/3.0
  else if r=n-1
    xnew(r)=((xold(n-2)+xold(0))+1.0)/3.0
  else
    xnew(r)=(xold(r+1)+xold(r-1)+xold(n-1-r))/3.0
  end for

```

### Algorithm 7.3.2 Sequential Gauss-Seidel Algorithm

```

for r=0 to n
  if r = 0
    xold(r) = ((xold(r+1) + xold(n-1)) + 1.0) / 3.0
  else if r = n-1
    xold(r) = ((xold(n-2) + xold(0)) + 1.0) / 3.0
  else
    xold(r) = (xold(r+1) + xold(r-1) + xold(n-1-r)) / 3.0
  end for

```

### Algorithm 7.3.3 Sequential JOR Algorithm

```

for r=0 to n-1
  if r = 0
    xnew(r) = xold(r) + w/3.0(1.0 - xold(r) * 3.0 - xold(r+1) - xold(n-1))
  else if r = n-1
    xnew(r) = xold(r) + w/3.0(1.0 - xold(r) * 3.0 - xold(r-1) - xold(0))
  else
    xnew(r) = xold(r) - w/3.0(xold(r) * 3.0 - xold(r+1) - xold(r-1) - xold(n-1-r))
  end for

```

### Algorithm 7.3.4 Sequential SOR Algorithm

```

for r=0 to n
  if r = 0
    xold(r) = (w + w*(xold(r+1) + xold(n-1))) / 3.0 - w*xold(r) + xold(r)
  else if r = n-1
    xold(r) = (w + w*(xold(r-1) + xold(0))) / 3.0 - w*xold(r) + xold(r)
  else
    xold(r) = (w*(xold(r+1) + xold(n-1) + xold(n-1-r))) / 3.0 - w*xold(r) + xold(r)
  end for

```

### Algorithm 7.3.5 Sequential JQI Algorithm

```

for r = 0 to n/2 - 1
  if r = 0
    rhs1 = 1.0 + xold(r+1)
    rhs2 = 1.0 + xold(n-r-2)
    det = 1.0/3.0
    xnew(r) = (rhs1 + det*rhs2) / (3.0 - det)
    xnew(n-1-r) = (rhs2 + det*rhs1) / (3.0 - det)
  else if r = n/2 - 1
    rhs1 = xold(r-1)
    rhs2 = xold(n-r)
    det = 2.0/3.0
    xnew(r) = (rhs1 + det*rhs2) / (3.0 - det*2.0)
    xnew(n-1-r) = (rhs2 + det*rhs1) / (3.0 - det*2.0)
  else
    rhs1 = xold(r+1) + xold(r-1)
    rhs2 = xold(n-r) + xold(n-r-2)
    det = 1.0/3.0
    xnew(r) = (rhs1 + det*rhs2) / (3.0 - det)
    xnew(n-1-r) = (rhs2 + det*rhs1) / (3.0 - det)
  end if
end for r

```

### Algorithm 7.3.6 Sequential SQI Algorithm

```

for r = 0 to n/2 - 1
  if r = 0
    rhs1 = 1.0 + xold(r+1)
    rhs2 = 1.0 + xold(n-r-2)
    det = 1.0/3.0
    xold(r) = (rhs1+det*rhs2)/(3.0-det)
    xold(n-1-r) = (rhs2+det*rhs1)/(3.0-det)
  else if r = n/2-1
    rhs1 = xold(r-1)
    rhs2 = xold(n-r)
    det = 2.0/3.0
    xold(r) = (rhs1+det*rhs2)/(3.0-det*2.0)
    xold(n-1-r) = (rhs2+det*rhs1)/(3.0-det*2.0)
  else
    rhs1 = xold(r+1)+xold(r-1)
    rhs2 = xold(n-r)+xold(n-r-2)
    det = 1.0/3.0
    xold(r) = (rhs1+det*rhs2)/(3.0-det)
    xold(n-1-r) = (rhs2+det*rhs1)/(3.0-det)
  end if
end for r

```

### Algorithm 7.3.7 Sequential JOQI Algorithm

```

for r = 0 to n/2 - 1
  if r = 0
    rhs1 = w+w*xold(r+1)+(1.0-w)*(3.0*xold(r)-xold(n-1-r))
    rhs2 = w+w*xold(n-r-2)+(1.0-w)*(3.0*xold(n-1-r)-xold(r))
    det = 1.0/3.0
    xnew(r) = (rhs1+det*rhs2)/(3.0-det)
    xnew(n-1-r) = (rhs2+det*rhs1)/(3.0-det)
  else if r = n/2-1
    rhs1 = w*xold(r-1)+(1.0-w)*(3.0*xold(r)-2.0*xold(n-1-r))
    rhs2 = w*xold(n-r)+(1.0-w)*(3.0*xold(n-1-r)-2.0*xold(r))
    det = 2.0/3.0
    xnew(r) = (rhs1+det*rhs2)/(3.0-det*2.0)
    xnew(n-1-r) = (rhs2+det*rhs1)/(3.0-det*2.0)
  else
    rhs1 = w*xold(r+1)+w*xold(r-1)+(1.0-w)*(3.0*xold(r)-xold(n-1-r))
    rhs2 = w*xold(n-r)+w*xold(n-r-2)+(1.0-w)*(3.0*xold(n-1-r)-xold(r))
    det = 1.0/3.0
    xnew(r) = (rhs1+det*rhs2)/(3.0-det)
    xnew(n-1-r) = (rhs2+det*rhs1)/(3.0-det)
  end if
end for r

```



### Algorithm 7.3.8 Sequential SOQI Algorithm

```

for r = 0 to n/2 - 1
  if r = 0
    rhs1 = w+w*xold(r+1)+(1.0-w)*(3.0*xold(r)-xold(n-1-r))
    rhs2 = w+w*xold(n-r-2)+(1.0-w)*(3.0*xold(n-1-r)-xold(r))
    det = 1.0/3.0
    xold(r) = (rhs1+det*rhs2)/(3.0-det)
    xold(n-1-r) = (rhs2+det*rhs1)/(3.0-det)
  else if r = n/2-1
    rhs1 = w*xold(r-1)+(1.0-w)*(3.0*xold(r)-2.0*xold(n-1-r))
    rhs2 = w*xold(n-r)+(1.0-w)*(3.0*xold(n-1-r)-2.0*xold(r))
    det = 2.0/3.0
    xold(r) = (rhs1+det*rhs2)/(3.0-det*2.0)
    xold(n-1-r) = (rhs2+det*rhs1)/(3.0-det*2.0)
  else
    rhs1 = w*xold(r+1)+w*xold(r-1)+(1.0-w)*(3.0*xold(r)-xold(n-1-r))
    rhs2 = w*xold(n-r)+w*xold(n-r-2)+(1.0-w)*(3.0*xold(n-1-r)-xold(r))
    det = 1.0/3.0
    xold(r) = (rhs1+det*rhs2)/(3.0-det)
    xold(n-1-r) = (rhs2+det*rhs1)/(3.0-det)
  end if
end for r

```

### *Synchronous Parallel Algorithms for classical and QI iterative methods*

In the synchronous parallel Jacobi, JQI, JOR and JOQI methods, parallelism is achieved by statically allocating the Jacobi iterations for the different grid points to the available processors. The outline of the synchronous parallel algorithms for Jacobi and JOR is shown in Algorithm 7.3.9 and the outline of the synchronous parallel algorithms for the JQI and JOQI is shown in Algorithm 7.3.10.

### Algorithm 7.3.9 Parallel Jacobi and JOR Algorithms

```

do
  par
    for r = 0 to n - 1
      perform Jacobi or JOR iteration
    end for r
  end par
  synchronise all processors
until convergence is reached

```

### Algorithm 7.3.10 Parallel JQI and JOQI Algorithms

```

do
  par
    for r = 0 to n/2 - 1
      perform JQI or JOQI iteration
    end for r
  end par
  synchronise all processors
until convergence is reached

```

Parallelising the Gauss-Seidel, SOR, SQI and SOQI methods are not as straight forward because these methods are not directly parallelisable. In order to parallelise these methods, the points need to be reordered such that the iterations are performed on the even points first, synchronise and then iterate on the odd points. This strategy is known as the red-black ordering. The outline of the synchronous parallel algorithm for Gauss-Seidel and SOR methods is shown in Algorithm 7.3.11 and the outline of the synchronous parallel algorithm for the SQI and SOQI methods is shown in Algorithm 7.3.12.

Algorithm 7.3.11 Parallel Gauss-Seidel/SOR Algorithms (Red-Black Ordering)

```

do
  par
    for all even indices between 0 to n-1
      partition points to available processor
      perform Gauss Seidel or SOR iteration
      synchronise processors
    for all odd indices between 0 to n-1
      partition points to available processor
      perform Gauss Seidel or SOR iteration
      synchronise processors
  end par
until convergence is reached

```

Algorithm 7.3.12 Parallel SQI/SOQI Algorithms (Red Black Ordering)

```

do
  par
    for all even indices between 0 to n/2
      partition points to available processor
      perform SQI or SOQI iteration
      synchronise processors
    for all odd indices between 0 to n/2
      partition points to available processor
      perform SQI or SOQI iteration
      synchronise processors
  end par
until convergence is reached

```

### *Asynchronous Algorithms for classical and QI iterative methods*

In the asynchronous algorithms, data is partitioned amongst the processors and each processor iterates on the data points within its local memory until local convergence is achieved without synchronising with other processors. The algorithm terminates when all processors have converged to the actual solution, i.e. upon global convergence. The outline of the asynchronous algorithms for all the iterative methods is shown in Algorithm 7.3.13.

Algorithm 7.3.13 Asynchronous Iterative Algorithm

```
do
  do
    local_flag = 1
    each processor perform Jacobi/GS/JOR/SOR/JQI/SQI/JOQI/SOQI iterations
    if no convergence within a processor,
      local_flag = 0
  while local_flag still 0
  Set global convergence flag for the particular processor that has converged.
while global convergence not achieved.
```

## 7.4 Computational Complexity and Shared Memory Access Analysis for the Classical and QI iterative methods

The computational complexity and shared memory access counts for the iterative methods discussed in this chapter have been analysed for a single iteration. The analysis is based on the sequential iterative programs. In the results section 7.5, a simple model is derived to relate the numerical results to the analysis completed in this section.

### 7.4.1 Jacobi and Gauss-Seidel iterative methods

In approximating the first and last values of the vector  $x$ , a total of  $2m+4a$  operations are required. The approximation of the remaining  $n-2$  values require  $(n-2)(1m+2a)$  operations. Hence a total of

$$n(m + 2a) \text{ operations} \quad (7.4.1.1)$$

are required for a single iteration of the Jacobi and Gauss-Seidel methods.

The number of shared memory accesses required in approximating the first and last values of vector  $x$  is 6 and the remaining  $n-2$  values require  $(n-2)4$ . Hence a total of

$$4n-2 \text{ shared memory accesses} \quad (7.4.1.2)$$

is required for a single iteration of the Jacobi and Gauss-Seidel algorithms.

A similar analysis can be performed on the JOR and SOR algorithms.

### 7.4.2 JQI and SQI methods

The number of operations required to approximate the first and last values of vector  $x$  for both JQI and SQI methods is  $10m + 10a$ . The approximation of the remaining values require  $(n/2-2)(5m+6a)$  operations. Hence, a total of

$$5/2n m + (3n-2)a \text{ operations} \quad (7.4.1.3)$$

are required for a single iteration of the JQI and SQI methods.

The number of shared memory accesses required to approximate the first and last values of the vector  $x$  is 8. The remaining values of vector  $x$  require  $6(n/2-2)$  shared memory accesses. In total, a single iteration of the JQI and SQI methods require

$$3n - 4 \text{ accesses to shared memory.} \quad (7.4.1.4)$$

A similar analysis can be performed on the JOQI and SOQI algorithms.

### 7.4.3 Summary

Table 7.4.3.1 provides a summary of the operational counts and the shared memory access counts for the classical and QI iterative methods discussed in this chapter. The figures shown in the summary reflect the computational work and shared memory access for a single iteration for each of the methods. The total work for the methods would then be the number of iterations times the computational work for a single iteration. Likewise the analysis for the shared memory access counts of the iterative methods is also based on a single iteration.

| Methods               | Computational Count | Shared Memory Access<br>Count |
|-----------------------|---------------------|-------------------------------|
| Jacobi & Gauss-Seidel | $n(m + 2a)$         | $4n - 2$                      |
| JQI and SQI           | $5/2n m + (3n-2)a$  | $3n - 4$                      |
| JOR                   | $2n(m+2a)$          | $6n-2$                        |
| SOR                   | $(5n-2)m + 4na$     | $6n-2$                        |
| JOQI                  | $13/2n m + 5n a$    | $5n-4$                        |
| SOQI                  | $13/2n m + 5n a$    | $5n-4$                        |

Table 7.4.3.1 A summary of the operational count and the shared memory access count for classical and QI iterative methods

The JQI and SQI methods have 25% less accesses to shared memory compared to the Jacobi and Gauss-Seidel methods while the JOQI and SOQI methods have 16% less accesses to shared memory when compared to the JOR and SOR methods. On the other hand, JQI and SQI have about 60% more computational work than Jacobi and Gauss-Seidel while JOQI and SOQI have about 23% more computational work than JOR and SOR for each iteration.

## 7.5 Numerical Results

Numerical experiments were carried out in the dedicated mode on the Sequent Balance. All timing results are given in CPU seconds. The asynchronous iterative methods and their synchronous counterparts were tested on  $p=1, 2, 4, 6, 8, 10$  processors. The model problem as stated earlier with 100, 200, and 400 equations were used. The convergence criteria is 0.0001.

The algorithms were implemented in single precision using the C language. The parallel constructs were supported by the Sequent C library. All the programs, except the synchronous parallel Gauss-Seidel, SQI, SOR and SOQI methods, employed a static scheduling of tasks, i.e. distributing the computation load amongst the processors before execution of the program commences. The synchronous parallel Gauss-Seidel, SQI, SOR and SOQI methods were parallelised using the red-black ordering strategy where the dynamic scheduling of tasks was employed.

In order to avoid the overhead of switching from one process to another, only a single process was assigned to each processor and the tests were completed while no other users' tasks were using the Balance. All the programs were written with the same accuracy to obtain a meaningful comparison.

| $n$ | Methods  | Iteration | $p=1$    | $p=2$    | $p=4$   | $p=6$   | $p=8$   | $p=10$  |
|-----|----------|-----------|----------|----------|---------|---------|---------|---------|
| 100 | Jacobi   | 28614     | 468.37   | 252.46   | 144.19  | 110.72  | 96.52   | 93.54   |
|     | JQI      | 18699     | 414.07   | 218.25   | 124.87  | 93.8    | 79.69   | 77.14   |
|     | Gain (%) |           | 11.5934  | 13.5507  | 13.399  | 15.2818 | 17.4368 | 17.5326 |
| 200 | Jacobi   | 113332    | 3679.17  | 1890.24  | 1048.18 | 755.85  | 623.7   | 560.48  |
|     | JQI      | 74803     | 3274.56  | 1698.31  | 903.78  | 647.81  | 528.46  | 455.11  |
|     | Gain (%) |           | 10.9973  | 10.1537  | 13.7763 | 14.2938 | 15.27   | 18.8    |
| 400 | Jacobi   | 451079    | 29045.68 | 14843.44 | 7824.25 | 5517.72 | 4481.35 | 3702.8  |
|     | JQI      | 299221    | 26172.77 | 13241.47 | 6927.87 | 4808.62 | 3750.8  | 3121.61 |
|     | Gain (%) |           | 9.891    | 10.7924  | 11.4564 | 12.8513 | 16.302  | 15.696  |

Table 7.5.1: Timings of the synchronous parallel Jacobi and JQI

Table 7.5.1 shows timings for the parallel synchronous Jacobi and JQI methods. As can be seen from the results, the gains in timings of the parallel synchronous JQI method over the Jacobi method varies between 10% to 18%. Table 7.5.2 shows the

timings of the asynchronous Jacobi and JQI methods while Table 7.5.3 shows the speedups obtained for the synchronous and asynchronous Jacobi and JQI methods. The gains in timings for the asynchronous JQI over the asynchronous Jacobi was greater, i.e. around about 20%.

| <i>n</i> | Methods  | <i>p</i> =1 | <i>p</i> =2 | <i>p</i> =4 | <i>p</i> =6 | <i>p</i> =8 | <i>p</i> =10 |
|----------|----------|-------------|-------------|-------------|-------------|-------------|--------------|
| 100      | Jacobi   | 413.35      | 209.27      | 105.95      | 72.07       | 55.42       | 59.94        |
|          | JQI      | 324.0       | 162.46      | 82.52       | 56.73       | 43.32       | 35.49        |
|          | Gain (%) | 21.6161     | 22.3682     | 22.1142     | 21.2849     | 21.8333     | 40.7908      |
| 200      | Jacobi   | 3257.36     | 1645.58     | 826.67      | 555.44      | 421.01      | 349.17       |
|          | JQI      | 2613.7      | 1290.21     | 649.79      | 438.35      | 331.91      | 268.71       |
|          | Gain (%) | 19.7602     | 21.5954     | 21.3967     | 21.0806     | 21.1634     | 23.0432      |
| 400      | Jacobi   | 26351.42    | 13263.54    | 6557.32     | 4369.83     | 3292.32     | 2646.4       |
|          | JQI      | 20776.98    | 10313.67    | 5143.32     | 3447.65     | 2599.29     | 2089.07      |
|          | Gain (%) | 21.1542     | 22.2404     | 21.5637     | 21.1033     | 21.0499     | 21.06        |

Table 7.5.2: Timings of the asynchronous parallel Jacobi and JQI

| <i>n</i> | <i>p</i> | Synchronous |      | Asynchronous |      |
|----------|----------|-------------|------|--------------|------|
|          |          | Jacobi      | JQI  | Jacobi       | JQI  |
| 100      | 2        | 1.86        | 1.9  | 1.98         | 1.99 |
|          | 4        | 3.25        | 3.32 | 3.9          | 3.93 |
|          | 6        | 4.23        | 4.41 | 5.74         | 5.71 |
|          | 8        | 4.85        | 5.2  | 7.46         | 7.48 |
|          | 10       | 5.01        | 5.37 | 6.9          | 9.13 |
| 200      | 2        | 1.95        | 1.93 | 1.98         | 2.02 |
|          | 4        | 3.51        | 3.62 | 3.94         | 4.02 |
|          | 6        | 4.87        | 5.05 | 5.86         | 6.1  |
|          | 8        | 5.9         | 6.2  | 7.74         | 7.87 |
|          | 10       | 6.56        | 7.2  | 9.33         | 9.73 |
| 400      | 2        | 1.96        | 1.98 | 1.99         | 2.01 |
|          | 4        | 3.71        | 3.78 | 4.01         | 4.04 |
|          | 6        | 5.26        | 5.44 | 6.03         | 6.02 |
|          | 8        | 6.48        | 6.98 | 8.0          | 7.99 |
|          | 10       | 7.84        | 8.38 | 9.96         | 9.95 |

Table 7.5.3 Speedup of synchronous and asynchronous Jacobi and JQI

The timing of an asynchronous program is the maximum execution time of the processors participating in the execution. Likewise, the number of iterations of an asynchronous program is the maximum number of iterations amongst the participating processors. The definition of speedup used in this chapter is  $S=t_1/t_p$  where  $t_p$  is the

execution time of the slowest processor and  $t_1$  is the time to execute the parallel program on a single processor. It can be seen that the asynchronous Jacobi and JQI programs gave better speedup as compared to the synchronous counterparts.

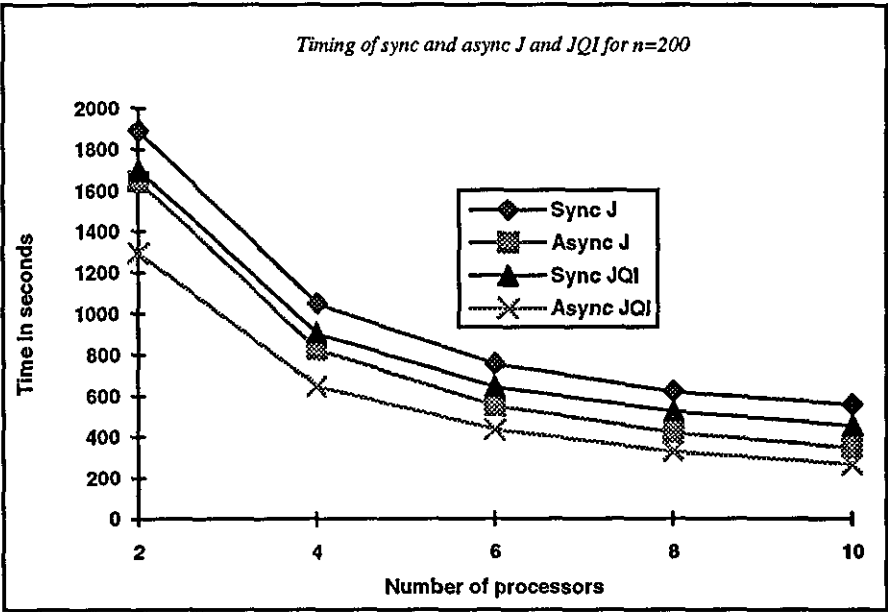


Figure 7.5.1 - Execution time in seconds for Jacobi and JQI

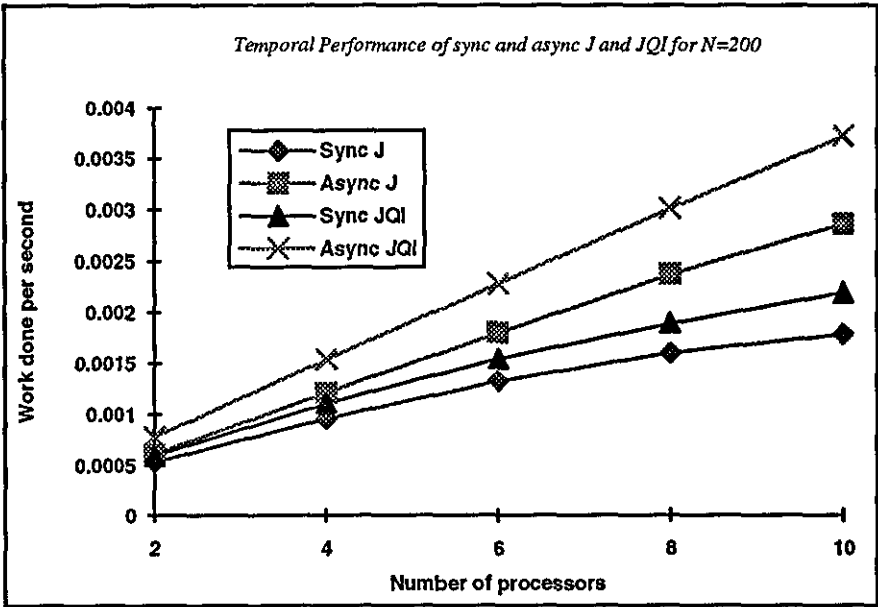


Figure 7.5.2 - Temporal performance of Jacobi and JQI

Figure 7.5.1 shows a graphical representation of the execution times of the synchronous and asynchronous Jacobi and JQI methods for  $n=200$ . The temporal



performance of the synchronous and asynchronous Jacobi and JQI methods for  $n=200$  is shown in the graph of figure 7.5.2. Temporal performance is defined as the inverse of the execution time where the units are solution per second or timesteps per second [Hockney 96]. It can be seen that the algorithm with the highest performance executes in the least time and therefore is the better algorithm. From figure 7.5.2 it can be seen that the asynchronous versions of both Jacobi and JQI has a better temporal performance than their synchronous counterparts. Again, the JQI method outperformed the Jacobi method in terms of timesteps per second for both the synchronous and asynchronous versions.

| $n$ | Methods      | Iterations | p=1      | p=2      | p=4     | p=6     | p=8     | p=10    |
|-----|--------------|------------|----------|----------|---------|---------|---------|---------|
| 100 | Gauss-Seidel | 14308      | 312.21   | 166.32   | 95.38   | 73.04   | 67.93   | 69.04   |
|     | SQI          | 9350       | 237.75   | 126.29   | 72.36   | 56.11   | 48.12   | 46.93   |
|     | Gain (%)     |            | 23.8493  | 24.0681  | 24.135  | 23.1791 | 29.1624 | 32.0249 |
| 200 | Gauss-Seidel | 56667      | 2440.96  | 1274.67  | 691.01  | 507.43  | 440.1   | 451.15  |
|     | SQI          | 37402      | 1871.55  | 971.91   | 524.47  | 375.5   | 312.78  | 361.75  |
|     | Gain (%)     |            | 23.273   | 23.752   | 24.101  | 25.9996 | 28.9298 | 19.816  |
| 400 | Gauss-Seidel | 225540     | 19467.68 | 10057.83 | 5266.87 | 3733.33 | 3125.6  | 2997.83 |
|     | SQI          | 149611     | 14888.78 | 7634.5   | 3972.83 | 2780.66 | 2185.95 | 1857.26 |
|     | Gain (%)     |            | 23.5205  | 24.0939  | 24.5694 | 25.5779 | 30.063  | 38.0465 |

Table 7.5.4: Timings of the synchronous parallel Gauss-Seidel and SQI (Red-Black ordering)

| $n$   | Methods      | p=1     | p=2     | p=4     | p=6     | p=8     | p=10   |
|-------|--------------|---------|---------|---------|---------|---------|--------|
| n=100 | Gauss-Seidel | 143.5   | 73.66   | 37.75   | 25.64   | 19.77   | 16.32  |
|       | SQI          | 129.93  | 64.95   | 33.1    | 22.85   | 17.77   | 14.63  |
|       | Gain (%)     | 9.46    | 11.82   | 12.32   | 10.88   | 10.12   | 10.36  |
| n=200 | Gauss-Seidel | 1132.61 | 578.2   | 290.42  | 195.34  | 148.82  | 120.44 |
|       | SQI          | 1042.66 | 515.22  | 260.02  | 175.89  | 133.12  | 108.27 |
|       | Gain (%)     | 7.94    | 10.89   | 10.47   | 9.96    | 10.55   | 10.10  |
| n=400 | Gauss-Seidel | 9068.81 | 4638.51 | 2287.83 | 1534.68 | 1158.57 | 932.86 |
|       | SQI          | 8406.84 | 4116.33 | 2058.89 | 1381.49 | 1041.72 | 838.22 |
|       | Gain (%)     | 7.29    | 11.26   | 10.0    | 9.98    | 10.09   | 10.15  |

Table 7.5.5: Timings of the asynchronous parallel Gauss-Seidel and SQI methods

Table 7.5.4 shows the timings for the parallel synchronous Gauss-Seidel and SQI methods, both employing the red-black strategy to achieve parallelism. The gain of the parallel synchronous SQI method over the Gauss-Seidel method ranges between 20-30%. Table 7.5.5 shows the timings of the asynchronous Gauss-Seidel and asynchronous SQI methods. The gain of the asynchronous SQI over Gauss-Seidel is between 7-10%. This is probably due to the natural ordering employed in the asynchronous versions as opposed to the red-black ordering in the synchronous versions. Table 7.5.6 shows the speedup of the synchronous and asynchronous Gauss-Seidel and SQI methods. The speedup of the asynchronous programs seemed better than the synchronous versions. Table 7.5.7 shows the number of iterations for the asynchronous Jacobi, JQI, Gauss-Seidel and SQI methods.

| $n$ | $p$ | Synchronous  |      | Asynchronous |       |
|-----|-----|--------------|------|--------------|-------|
|     |     | Gauss-Seidel | SQI  | Gauss-Seidel | SQI   |
| 100 | 2   | 1.88         | 1.88 | 1.95         | 2.0   |
|     | 4   | 3.27         | 3.29 | 3.8          | 3.93  |
|     | 6   | 4.27         | 4.24 | 5.59         | 5.69  |
|     | 8   | 4.6          | 4.94 | 7.26         | 7.31  |
|     | 10  | 4.52         | 5.07 | 8.79         | 8.88  |
| 200 | 2   | 1.91         | 1.93 | 1.96         | 2.02  |
|     | 4   | 3.53         | 3.57 | 3.89         | 4.01  |
|     | 6   | 4.81         | 4.98 | 5.79         | 5.93  |
|     | 8   | 5.55         | 5.98 | 7.61         | 7.83  |
|     | 10  | 5.41         | 5.17 | 9.4          | 9.63  |
| 400 | 2   | 1.94         | 1.95 | 1.96         | 2.04  |
|     | 4   | 3.7          | 3.75 | 3.96         | 4.08  |
|     | 6   | 5.21         | 5.35 | 5.91         | 6.08  |
|     | 8   | 6.23         | 6.81 | 7.83         | 8.07  |
|     | 10  | 6.49         | 8.02 | 9.72         | 10.02 |

Table 7.5.6 Speedup of the synchronous and asynchronous Gauss-Seidel and SQI methods

Figure 7.5.3 shows the graphical view of the execution time of the synchronous and asynchronous Gauss-Seidel and SQI programs. In Figure 7.5.4 the graph of the temporal performance of the synchronous and asynchronous Gauss-Seidel and SQI programs are shown. It can be seen that both the asynchronous programs have a better temporal performance than the synchronous counterparts but in both cases the SQI method showed to be better.

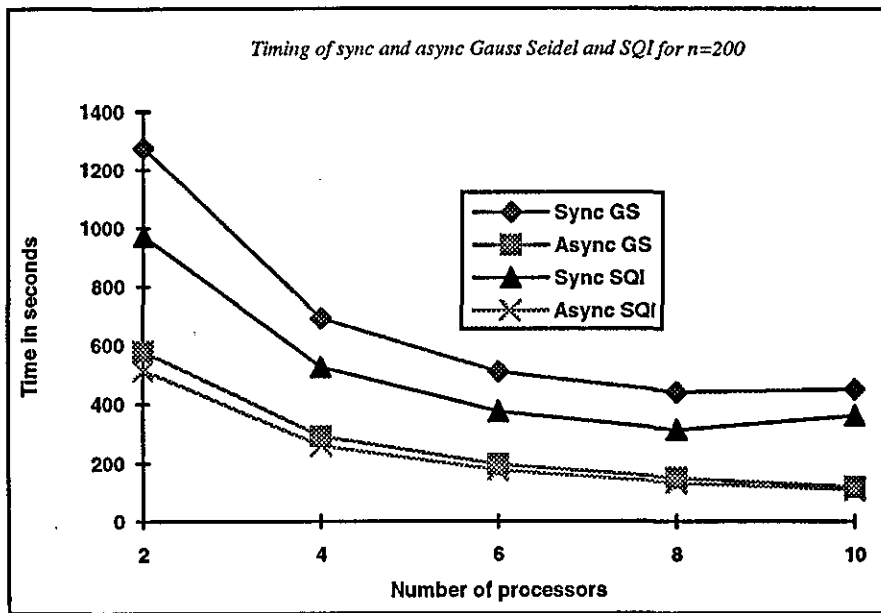


Figure 7.5.3 - Execution time in seconds for Gauss-Seidel and SQI methods

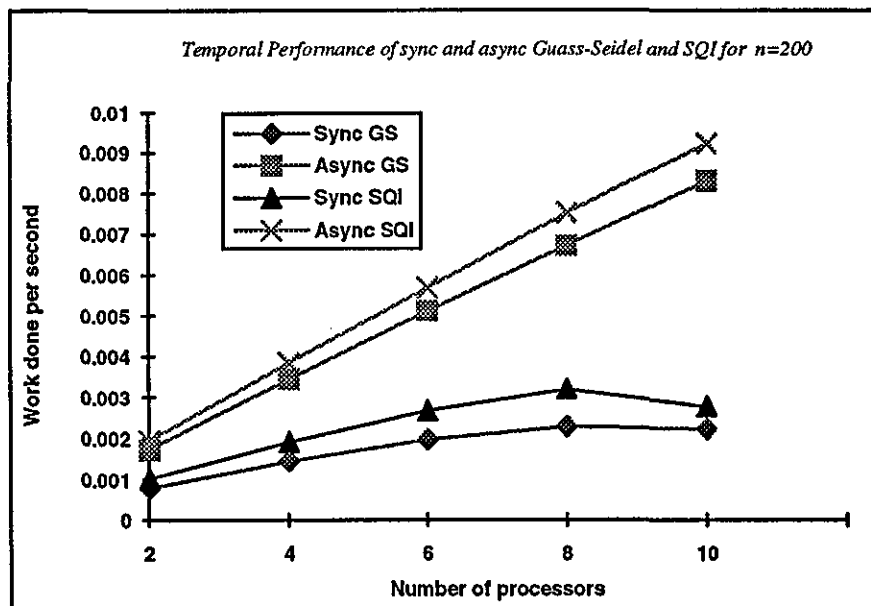


Figure 7.5.4 - Temporal performance of Gauss-Seidel and SQI methods

Table 7.5.8 shows the timings of the synchronous JOR and JOQI methods while the timings of their asynchronous counterparts is shown in Table 7.5.9. The speedup of both the synchronous and asynchronous JOR and JOQI methods are shown in Table 7.5.10. The number of iterations for the asynchronous JOR and JOQI methods are shown in Table 7.5.11.

The gain in execution time of the synchronous JOQI is between 20-25% while the gain in the asynchronous JOQI is between 33-34%. Again, the speedup of the asynchronous programs were much better than the synchronous versions.

The timings of the synchronous SOR and SOQI methods are shown in Table 7.5.12 while their speedup figures are shown in Table 7.5.13.

| n   | p  | Number of iterations for asynchronous methods |        |              |        |
|-----|----|---|--------|--------------|--------|
|     |    | Jacobi  | JQI    | Gauss-Seidel | SQI    |
| 100 | 1  | 28614   | 18700  | 14310        | 9326   |
|     | 2  | 28939   | 18790  | 14612        | 9397   |
|     | 4  | 28839   | 19500  | 14612        | 9786   |
|     | 6  | 29833   | 19681  | 15115        | 9936   |
|     | 8  | 30033   | 19656  | 15328        | 10097  |
|     | 10 | 36836   | 18930  | 14911        | 9756   |
| 200 | 1  | 113332  | 74804  | 56668        | 37353  |
|     | 2  | 114271  | 75273  | 57792        | 37627  |
|     | 4  | 113752  | 75337  | 57642        | 37700  |
|     | 6  | 114950  | 78070  | 58347        | 38931  |
|     | 8  | 113777  | 78049  | 57638        | 39359  |
|     | 10 | 115637  | 75421  | 57877        | 38112  |
| 400 | 1  | 451079  | 299222 | 225542       | 149512 |
|     | 2  | 453926  | 301009 | 228822       | 150566 |
|     | 4  | 454870  | 301140 | 228588       | 150892 |
|     | 6  | 457480  | 304587 | 228966       | 152648 |
|     | 8  | 453986  | 301351 | 228831       | 150725 |
|     | 10 | 453734  | 301556 | 229211       | 150713 |

Table 7.5.7 Number of iterations for the asynchronous Jacobi, JQI, Gauss-Seidel and SQI methods.

| n   | Methods  | Iterations | p=1      | p=2      | p=4      | p=6     | p=8     | p=10    |
|-----|----------|------------|----------|----------|----------|---------|---------|---------|
| 100 | JOR      | 28614      | 815.44   | 420.61   | 229.12   | 167.42  | 135.18  | 119.4   |
|     | JOQI     | 18699      | 606.98   | 318.12   | 175.46   | 129.5   | 106.51  | 93.84   |
|     | Gain (%) |            | 25.5641  | 24.3669  | 23.42    | 22.6496 | 21.2088 | 25.56   |
| 200 | JOR      | 113332     | 6513.23  | 3291.68  | 1710.57  | 1206.12 | 946.39  | 864.16  |
|     | JOQI     | 74803      | 4815.7   | 2467.75  | 1309.32  | 921.36  | 748.91  | 671.94  |
|     | Gain(%)  |            | 26.0628  | 25.0307  | 23.4571  | 23.6096 | 20.8667 | 22.2436 |
| 400 | JOR      | 451079     | 51492.32 | 26118.05 | 13449.19 | 9180.03 | 7041.79 | 5765.34 |
|     | JOQI     | 299221     | 38583.89 | 19686.06 | 10069.14 | 6957.85 | 5329.19 | 4375.28 |
|     | Gain(%)  |            | 25.0692  | 24.6226  | 25.1319  | 24.2067 | 24.3205 | 24.1106 |

Table 7.5.8: Timings of the synchronous parallel JOR and JOQI methods (w=1.0)

| $n$ | Methods | $p=1$    | $p=2$    | $p=4$    | $p=6$   | $p=8$   | $p=10$  |
|-----|---------|----------|----------|----------|---------|---------|---------|
| 100 | JOR     | 795.13   | 403.57   | 202.45   | 137.09  | 104.18  | 83.82   |
|     | JOQI    | 521.63   | 263.01   | 131.62   | 90.1    | 68.24   | 55.18   |
|     | Gain(%) | 34.3969  | 34.8291  | 34.9864  | 34.2768 | 34.4979 | 34.1685 |
| 200 | JOR     | 6321.06  | 3193.25  | 1590.87  | 1067.92 | 805.8   | 648.3   |
|     | JOQI    | 4861.11  | 2091.84  | 1048.12  | 702.79  | 529.7   | 426.46  |
|     | Gain(%) | 33.6841  | 34.4918  | 34.1166  | 34.1908 | 34.2641 | 34.2187 |
| 400 | JOR     | 50598.29 | 25455.11 | 12648.76 | 8432.29 | 6356.57 | 5104.95 |
|     | JOQI    | 33648.62 | 16644.91 | 8329.14  | 5558.11 | 4184.12 | 3361.14 |
|     | Gain(%) | 33.49    | 34.6107  | 34.1505  | 34.0854 | 34.1765 | 34.1592 |

Table 7.5.9: Timings of the asynchronous parallel JOR and JOQI methods ( $w=1.0$ )

| $n$ | $p$ | Synchronous |      | Asynchronous |       |
|-----|-----|-------------|------|--------------|-------|
|     |     | JOR         | JOQI | JOR          | JOQI  |
| 100 | 2   | 1.94        | 1.91 | 1.97         | 1.98  |
|     | 4   | 3.56        | 3.46 | 3.93         | 3.96  |
|     | 6   | 4.87        | 4.69 | 5.8          | 5.79  |
|     | 8   | 6.03        | 5.7  | 7.63         | 7.64  |
|     | 10  | 6.83        | 6.47 | 9.49         | 9.45  |
| 200 | 2   | 1.98        | 1.95 | 1.98         | 2.0   |
|     | 4   | 3.81        | 3.68 | 3.97         | 3.99  |
|     | 6   | 5.4         | 5.23 | 5.92         | 5.96  |
|     | 8   | 6.88        | 6.43 | 7.84         | 7.91  |
|     | 10  | 7.54        | 7.17 | 9.75         | 9.83  |
| 400 | 2   | 1.97        | 1.96 | 1.99         | 2.02  |
|     | 4   | 3.83        | 3.83 | 4.0          | 4.03  |
|     | 6   | 5.61        | 5.55 | 6.0          | 6.05  |
|     | 8   | 7.31        | 7.24 | 7.96         | 8.04  |
|     | 10  | 8.93        | 8.82 | 9.91         | 10.01 |

Table 7.5.10 Speedup of the synchronous and asynchronous JOR and JOQI methods

| $n=100$ | $p=1$  | $p=2$  | $p=4$  | $p=6$  | $p=8$  | $p=10$ |
|---------|--------|--------|--------|--------|--------|--------|
| JOR     | 28615  | 28948  | 28756  | 29611  | 30012  | 28731  |
| JOQI    | 18699  | 18710  | 19430  | 19548  | 19560  | 18867  |
| $n=200$ | $p=1$  | $p=2$  | $p=4$  | $p=6$  | $p=8$  | $p=10$ |
| JOR     | 113334 | 114451 | 113834 | 115487 | 113848 | 114099 |
| JOQI    | 74803  | 74885  | 75163  | 78254  | 78078  | 75216  |
| $n=400$ | $p=1$  | $p=2$  | $p=4$  | $p=6$  | $p=8$  | $p=10$ |
| JOR     | 451087 | 454800 | 453129 | 455454 | 453121 | 453393 |
| JOQI    | 299221 | 300744 | 301199 | 303795 | 300202 | 300963 |

Table 7.5.11 Number of iterations for the asynchronous JOR and JOQI methods.

| $n$ | Methods  | $w$  | Iteration | $p=1$  | $p=2$ | $p=4$ | $p=6$ | $p=8$ | $p=10$ |
|-----|----------|------|-----------|--------|-------|-------|-------|-------|--------|
| 100 | SOR      | 1.95 | 242       | 7.83   | 4.08  | 2.29  | 1.7   | 1.45  | 1.31   |
|     | SOQI     | 1.94 | 176       | 5.34   | 2.84  | 1.77  | 1.52  | 1.45  | 1.49   |
|     | Gain (%) |      |           | 31.8   | 30.39 | 22.7  | 10.59 | -     | -      |
| 200 | SOR      | 1.98 | 486       | 31.35  | 16.11 | 8.66  | 6.13  | 4.94  | 4.35   |
|     | SOQI     | 1.97 | 327       | 19.56  | 10.08 | 5.55  | 4.16  | 3.56  | 3.2    |
|     | Gain (%) |      |           | 37.61  | 37.43 | 35.91 | 32.14 | 27.94 | 26.44  |
| 400 | SOR      | 1.99 | 969       | 124.85 | 63.7  | 33.14 | 22.94 | 17.98 | 15.71  |
|     | SOQI     | 1.99 | 913       | 109.82 | 55.34 | 28.11 | 19.85 | 14.1  | 11.55  |
|     | Gain (%) |      |           | 12.04  | 13.12 | 15.18 | 13.47 | 21.58 | 26.48  |

Table 7.5.12: Timings of the synchronous parallel SOR and SOQI (Red-Black)

| $n=100$ | $p=2$ | $p=4$ | $p=6$ | $p=8$ | $p=10$ |
|---------|-------|-------|-------|-------|--------|
| SOR     | 1.92  | 3.42  | 4.61  | 5.4   | 5.98   |
| SOQI    | 1.88  | 3.01  | 3.51  | 3.68  | 3.58   |
| $n=200$ | $p=2$ | $p=4$ | $p=6$ | $p=8$ | $p=10$ |
| SOR     | 1.95  | 3.62  | 5.11  | 6.35  | 7.21   |
| SOQI    | 1.94  | 3.52  | 4.7   | 5.49  | 6.11   |
| $n=400$ | $p=2$ | $p=4$ | $p=6$ | $p=8$ | $p=10$ |
| SOR     | 1.96  | 3.78  | 5.44  | 6.94  | 7.95   |
| SOQI    | 1.98  | 3.91  | 5.53  | 7.79  | 9.5    |

Table 7.5.13 Speedup of the synchronous SOR and SOQI methods

Finally, the results of the asynchronous SOR and SOQI methods are shown in Tables 7.5.14 and 7.5.15 respectively. The optimum values of  $\omega$  in the asynchronous SOR and SOQI appears to be a function of the number of processors, as would be expected since each processor has only  $n/p$  equations to solve, which decreases as more processors are utilised. In the case of the asynchronous SOR, this results in an increase in the number of iterations required to reach convergence and seriously degrades the performance of the parallel implementation. The same kind of results was obtained in the asynchronous SOR implementation of [Nieplocha 92a] and [Nieplocha 92b]. However, in the case of the asynchronous SOQI implementation, the phenomenon of decreasing  $\omega$  is not so obvious.

|            |        |        |       |        |        |
|------------|--------|--------|-------|--------|--------|
| $n=100$    | $p=2$  | $p=4$  | $p=6$ | $p=8$  | $p=10$ |
| $\omega$   | 1.87   | 1.84   | 1.81  | 1.88   | 1.78   |
| iterations | 1088   | 1464   | 1895  | 1598   | 1799   |
| time       | 12.51  | 8.54   | 5.98  | 3.81   | 3.61   |
| $n=200$    | $p=2$  | $p=4$  | $p=6$ | $p=8$  | $p=10$ |
| $\omega$   | 1.89   | 1.84   | 1.89  | 1.79   | 1.78   |
| iterations | 3721   | 4140   | 3548  | 7559   | 8209   |
| time       | 83.93  | 47.02  | 22.52 | 37.13  | 32.6   |
| $n=400$    | $p=2$  | $p=4$  | $p=6$ | $p=8$  | $p=10$ |
| $\omega$   | 1.95   | 1.9    | 1.89  | 1.85   | 1.82   |
| iterations | 6328   | 10951  | 14309 | 19688  | 21584  |
| time       | 286.42 | 210.18 | 181.9 | 189.81 | 167.66 |

Table 7.5.14: Results of the asynchronous parallel SOR method

|            |       |       |       |       |        |
|------------|-------|-------|-------|-------|--------|
| $n=100$    | $p=2$ | $p=4$ | $p=6$ | $p=8$ | $p=10$ |
| $\omega$   | 1.95  | 1.95  | 1.95  | 1.91  | 1.75   |
| iterations | 155   | 188   | 323   | 583   | 2494   |
| time       | 2.04  | 1.15  | 1.31  | 1.81  | 7.07   |
| $n=200$    | $p=2$ | $p=4$ | $p=6$ | $p=8$ | $p=10$ |
| $\omega$   | 1.97  | 1.97  | 1.97  | 1.97  | 1.91   |
| iterations | 300   | 382   | 469   | 618   | 1868   |
| time       | 7.53  | 4.96  | 4.1   | 3.75  | 9.49   |
| $n=400$    | $p=2$ | $p=4$ | $p=6$ | $p=8$ | $p=10$ |
| $\omega$   | 1.99  | 1.99  | 1.99  | 1.98  | 1.97   |
| iterations | 835   | 833   | 903   | 1399  | 2280   |
| time       | 41.92 | 21.23 | 15.06 | 17.4  | 22.79  |

Table 7.5.15: Results of the asynchronous parallel SQI method

In general, it can be seen that in terms of the number of iterations and execution times, the QI iterative methods have shown to be more superior than the classical iterative methods. The asynchronous versions of both classes of iterative methods have shown to be more superior than the synchronous ones. The lack of synchronisation has obviously led to better execution times.

## 7.6 A Simplified Model Study

In section 7.4, the computational work and shared memory access analyses for a single iteration of the iterative methods have been shown. The actual timings of the iterative methods as well as the number of iterations for convergence to the exact solution have been shown in section 7.5. In this section, a simple model is derived to show that although the computational work for a single iteration of the QI iterative methods exceeds that of the classical methods, the gain from shared memory access incorporating the total number of iterations will reveal reduced computational work. However, this simplified model does not take into account the synchronisation points and hence is comparable to the asynchronous iterative methods.

Let  $W_C$  and  $W_{QI}$  be the computational work for a single iteration of the classical and QI iterative methods respectively. In the computational work, an add and a multiplication operation (flop) is assumed to take the same amount of time to execute and shall be termed a flop. Let  $SMA_C$  and  $SMA_{QI}$  be the shared memory access count for a single iteration of the classical and QI iterative methods respectively. Let  $I_C$  and  $I_{QI}$  be the number of iterations it takes to converge by the classical and QI iterative methods respectively.

Then, the total amount of time taken to execute the classical and QI iterative methods can be defined as

$$T_C = I_C W_C + I_C SMA_C \quad (7.6.1)$$

for the classical iterative methods and

$$T_{QI} = I_{QI} W_{QI} + I_{QI} SMA_{QI} \quad (7.6.2)$$

for the QI iterative methods.

For example, for the asynchronous Jacobi and JQI methods when  $n=100$  and  $p=1$ , we have

$$W_{Jacobi} = n(m+2a) = 3n \text{ flops}$$

$$SMA_{Jacobi} = 4n-2$$

$$I_{Jacobi} = 28614$$

$$\begin{aligned} T_{Jacobi} &= I_{Jacobi} W_{Jacobi} + I_{Jacobi} SMA_{Jacobi} \\ &= 28614 * 3(100) + 28614 * (400-2) \\ &= 19972174 \end{aligned}$$

$$W_{JQI} = 5/2nm + (3n-2)a = 11/2n-2 \text{ flops}$$

$$SMA_{JQI} = 3n-4$$

$$I_{JQI} = 18699$$

$$\begin{aligned} T_{JQI} &= I_{JQI} W_{JQI} + I_{JQI} SMA_{JQI} \\ &= 18699 * 548 + 18699 * 296 \\ &= 15781956 \end{aligned}$$



Therefore since  $T_{JQI} < T_{\text{jacobi}}$  we can verify that the JQI method takes less time than the Jacobi method to converge.

Similarly, for the case of asynchronous JOR and JOQI methods when  $n=100$  and  $p=1$ ,

$$W_{\text{JOR}} = 2n(m+2a) = 6n \text{ flops}$$

$$W_{\text{JOQI}} = 13/2nm + 5n a = 23/2n \text{ flops}$$

$$\text{SMA}_{\text{JOR}} = 6n-2$$

$$\text{SMA}_{\text{JOQI}} = 5n-4$$

$$I_{\text{JOR}} = 28615$$

$$I_{\text{JOQI}} = 18699$$

$$\begin{aligned} T_{\text{Jacobi}} &= I_{\text{JOR}}W_{\text{JOR}} + I_{\text{JOR}}\text{SMA}_{\text{JOR}} \\ &= 28615*600 + 28615*598 \\ &= 34280770 \end{aligned}$$

$$\begin{aligned} T_{\text{JOQI}} &= I_{\text{JOQI}}W_{\text{JOQI}} + I_{\text{JOQI}}\text{SMA}_{\text{JOQI}} \\ &= 18699*1150 + 18699*496 \\ &= 30778554 \end{aligned}$$

Hence,  $T_{\text{JOQI}} < T_{\text{JOR}}$

Finally, for the same case of asynchronous JOR and JOQI methods when  $n=100$  but this time  $p=2$ , therefore each processor has 50 equations to solve.

$$W_{\text{JOR}} = 2n(m+2a) = 6n \text{ flops}$$

$$W_{\text{JOQI}} = 13/2nm + 5n a = 23/2n \text{ flops}$$

$$\text{SMA}_{\text{JOR}} = 6n-2$$

$$\text{SMA}_{\text{JOQI}} = 5n-4$$

$$I_{\text{JOR}} = 28948$$

$$I_{\text{JOQI}} = 18710$$

$$\begin{aligned} T_{\text{JOR}} &= I_{\text{JOR}}W_{\text{JOR}} + I_{\text{JOR}}\text{SMA}_{\text{JOR}} \\ &= 28948*300 + 28948*298 \\ &= 17310904 \end{aligned}$$

$$\begin{aligned} T_{\text{JOQI}} &= I_{\text{JOQI}}W_{\text{JOQI}} + I_{\text{JOQI}}\text{SMA}_{\text{JOQI}} \\ &= 18710*575 + 18710*246 \\ &= 15360910 \end{aligned}$$

Hence,  $T_{\text{JOQI}} < T_{\text{JOR}}$

Therefore the theoretical model results shown above agree qualitatively with the experimental results shown in section 7.5.

## 7.7 Summary

In this chapter, the QI iterative methods have been presented. The computational complexity and the shared memory access counts have been analysed for both the classical and QI iterative methods for a single iteration. Clearly, there was more work incurred in the QI iterative algorithms. However, using a simplified model study, it has been shown that the fewer iterations of the QI iterative methods has led to better timings for the QI iterative methods.

In general, in terms of the number of iterations and execution times, the QI iterative methods have shown to be better than the classical iterative methods. The asynchronous versions of both classes of iterative methods have shown to be more superior than the synchronous ones. The lack of synchronisation has obviously led to better execution times. The gains were much more for the JQI and JOQI classes of methods, probably due to the fact that these methods parallelise better than the SQI and SOQI classes.

The results of the numerical experiments obtained in this chapter confirms the superiority of the QI iterative methods over the classical iterative methods for the model problem discussed in section 7.2. The asynchronous methods have also shown very promising results. It was the purpose of this chapter to apply the parallel strategies used in the direct methods of chapters 4, 5 and 6 to iterative methods and to derive experimental results as a basis for developing theoretical justification later.

## Chapter 8

### Summary and Future Work

The objectives of the work presented in this thesis was to investigate the performance of a class of numerical algorithms for the solution of linear systems. This class of numerical algorithms is based on the splitting strategy i.e. the Quadrant Interlocking (QI) structure of a matrix and encompasses both direct and iterative methods of solution. The QI methods are (2x2) block algorithms expressed in explicit point form. It is not the intention of this work to study the architecture of a particular parallel machine or software language. The parallel machine and software language were tools in helping to investigate the performance of the algorithms.

The introductory chapter described applications in which linear systems of equation occur. In Chapter 2, some basic mathematical concepts relevant to the work in this thesis were discussed. Chapter 3 includes a brief overview of parallel processing and a survey of parallel algorithms for the solution of linear systems.

In Chapter 4, the QI direct methods were investigated on the Sequent Balance, a shared memory parallel computer. The Parallel Implicit Elimination method (PIE) was compared with the classical Gaussian Elimination method (GE) while the Quadrant Interlocking Factorisation method (QIF) was compared with the classical LU factorisation method. Numerical results revealed a 20% gain in execution time by the PIE and QIF methods for both the pivoting and non-pivoting cases. The analysis of the computational count of the methods showed that the amount of computational work for the methods were the same. Hence the gain emanated from another dimension of the execution, which is accesses to shared memory. With the direct methods, the updating of the matrix is the most expensive part of the algorithm. Hence reference to shared memory at this stage of the algorithm would also be a crucial factor. Analysis of the shared memory accesses showed that both PIE and QIF have 33% less accesses than GE and LU respectively.

In Chapter 5, the QI direct methods were implemented on a distributed memory architecture. The aim of this chapter was to investigate the communication complexity of the direct methods on a distributed memory architecture. The communication

complexity of the QI algorithms were analysed and shown to be 50% less than that of the classical methods which was supported by the numerical results obtained.

The orthogonal decomposition method QZ was investigated on the Sequent Balance. The results are presented in Chapter 6 and show QZ to be 10% faster than QR, less than that of PIE and QIF over GE and LU respectively. This is probably due to the extra overheads incurred in the complex computation involved in QZ and in maintaining the task queue during dynamic scheduling.

The QI iterative methods JQI, SQI, JOQI and SOQI were investigated on a shared memory parallel computer and compared with the classical iterative methods i.e. Jacobi, Gauss-Seidel, JOR and SOR. Both methods were implemented as synchronous and asynchronous algorithms. A simplified model study was derived to show that although the computational work for the QI methods exceeded that of the classical methods, the gain from the smaller number of iterations and shared memory accesses revealed a reduction in computation time.

There are other promising areas of further study involving the QI method. The Gauss-Jordan method is another linear solver which requires more computation but is used because of its conceptual simplicity. The Gauss-Jordan method transforms the coefficient matrix  $A$  into a diagonal matrix. An implicit version of the Gauss-Jordan method also exists. In this method the  $W$  coefficients are computed as in PIE, two at a time solving  $n-2$  ( $2 \times 2$ ) systems at each of the  $n/2$  steps. The solution phase consists of solving the resulting bi-diagonal system by means of solving  $n/2$  ( $2 \times 2$ ) systems. Barulli and Evans in [Barulli 96] have shown that the implicit Gauss-Jordan yielded 20% gains over the classical Gauss-Jordan method.

There are also the Choleski form and the square root free form of the QI method that needs further investigation. Again, these methods can be compared with the equivalent classical methods.

Apart from solving linear systems which is the focus of this thesis the QI methods can also be used to solve other related problems such as the inverse problem of  $AX=I$ , problems with many right-hand sides  $AX=B$  and evaluating the determinant of  $A$ .

The QI iterative methods can also be applied to the preconditioned conjugate gradient method. Due to time shortage, a detailed theoretical investigation has been omitted in

the study of the QI iterative methods while the experimental results obtained require additional theoretical analysis.

The bi-directional solver of QIF can also be implemented in either Crout or Doolittle form. The QZ orthogonal decomposition method can also be implemented as a column orthogonal decomposition method. Further, there is also the Householder form of QZ that can be investigated.

The asynchronous QI iterative methods could also be implemented on a distributed memory architecture. In addition, it would also be interesting to investigate the performance of the orthogonal QZ on a distributed memory architecture. An implementation of the QI methods on a SIMD architecture would also be an interesting investigation for the future.

Finally, implementing the QI methods as a set of routines available as part of a numerical library would be a worthwhile task.

The QI methods investigated in this thesis has shown very promising results. It has shown that parallel algorithms should exploit not only the inherent parallelism of the problem but also attempt to reformulate the problem to introduce more parallelism.

## References

- [Abdullah 95] Abdullah, R. and Evans, D.J., *Parallel Algorithms for the Solution of Linear Systems in PVM* in Proceedings of Parallel and Distributed Processing, Techniques and Applications 95 (PDPTA 95), H.Arabnia (ed.), Athens, Georgia, Nov 4-5, 1995, pp101-110.
- [Akl 89] Akl, S.G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.
- [Almasi 94] Almasi and Gottlieb, *Highly Parallel Computing Second Edition*, Benjamin Cummings, 1994.
- [Angelaccio 94] Angelaccio, M., Colajanni, M., *The Row/Column Pivoting Strategy on Multicomputers*, Parallel Computing 20 (1994) pp. 197-213.
- [Asenjo 93] Asenjo, R., Ujaldon, M. and Zapata, E.L., *Parallel WZ Factorisation on Mesh Multiprocessors*, Microprocessing and Microprogramming 38 (1993) pp 319-326.
- [Barnett 90] Barnett, S., *Matrices Methods and Applications*, Oxford Press, 1990.
- [Barulli 96] Barulli, M. and Evans, D.J., *Implicit Gauss-Jordan Scheme for the Solution of Linear Systems*, Internal Report 1013, Department of Computer Studies, Loughborough University, April 1996.
- [Baudet 78] Baudet, G.M., *Asynchronous Iterative Methods for Multiprocessors*, Journal of the ACM, Vol 25, No2, April 1978, pp. 226-244.
- [Beguelin 94] Beguelin, A., Dongarra, J., Geist, A., Manchek, R. & Sunderam, V., *Recent Enhancements to PVM*, June 17, 1994.  
<http://www.netlib.org/utk/papers/pvm-ijsa/ijsa.html>
- [Benaini 94] Benaini, A. and Laiymani, D., *Generalized WZ Factorization on A Reconfigurable Machine*, Parallel Algorithms and Applications, Vol 3, pp. 261-269, 1994.
- [Bertsekas 89] Bertsekas, D.P. and Tsitsiklis, J.N., *Parallel and Distributed Computations*, Prentice Hall, 1989
- [Bonomo 89] Bonomo, J.P. and Dyksen, W.R., *Pipelined Iterative Methods for Shared Memory Machines*, Parallel Computing 11(1989) pp.187-199.

- [Brawer 89] Brawer, S., *Introduction to Parallel Programming*, Academic Press, 1989.
- [Carriero 90] Carriero, N. and Gelernter, D., *How To Write Parallel Programs A First Course*, The MIT Press, 1990.
- [Carriero 94] Carriero, N.J., Gelernter, D., Mattson, T.G., Sherman, A.H., *The Linda Alternative to Message-Passing Systems*, *Parallel Computing* 20 (1994) pp 633-655.
- [Casanova 95] Casanova, H., Dongarra, J. and Jiang, W., *The Performance of PVM on MPP Systems*, Univ. of Tennessee T.R. CS-95-301, August 1995. <http://www.netlib.org/utk/papers/pvmmpp.ps>
- [Casvant 96] Casvant, T.L., Tvrdik, P. and Plasil, F., *Parallel Computing Theory and Practice*, IEEE Computer Society Press, 1996.
- [Chamberlain 87] Chamberlain, R.M., *An Alternative View of LU Factorization with Partial Pivoting on a Hypercube Multiprocessor*, *Hypercube Multiprocessor 1987*, M.T.Heath (Ed) SIAM, 1987.
- [Chazan 69] Chazan, D. and Miranker, W., *Chaotic Relaxation*, *Lin. Alg. Appl*, Vol 2, pp. 199-222, 1969.
- [Chikohora 91] Chikohora, S., *Parallel Algorithms for The Solution of Elliptic and Parabolic Problems on Transputer Networks*, Thesis, Loughborough University of Technology, 1991.
- [Chorafas 90] Chorafas, D.N. and Steinmann, H., *Supercomputers*, McGraw-Hill, 1990.
- [Chu 87] Chu, E. and George, A., *Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor*, *Parallel Computing* 5, 1987, pp. 65-74.
- [Coffin 92] Coffin, M.H., *Parallel Programming A New Approach*, Prentice Hall, 1992.
- [Cohen 73] Cohen, A.M., *Numerical Analysis*, McGraw-Hill, 1973.
- [Cook 83] Cook, S.A., *An Overview of Computational Complexity*, *Comm. of the ACM*, June 1983, Vol 26, Number 6, pp. 400-408.
- [Cosnard 86] Cosnard, M. and Robert, Y., *Complexity of Parallel QR Decomposition*, *J. ACM* 33 (Oct) pp. 712-723, 1986.

- [Cosnard 94] Cosnard, M. and Daoudi, E.M., *Optimal Algorithms for Parallel Givens Factorization on a Coarse-Grained PRAM*, J. ACM, Vol 41, No 2, March 1994, pp. 399-421.
- [Cosnard 95] Cosnard, M. and Trystram, D., *Parallel Algorithms and Architectures*, International Thomson Computer Press, 1995.
- [Cunha 91] Cunha, R.D. and Hopkins, T., *Parallel Overrelaxation Algorithms for Systems of Linear Equations*, Transputing '91, P.Welch et al. (Eds), IOS Press, 1991, pp. 159-169.
- [D'Ambra 95] D'Ambra, P. and Giunta, G., *Concurrent Banded Cholesky Factorisation on Workstation Networks using PVM*, Parallel Computing 21 (1995) pp 487-494.
- [Darmohray 87] Darmohray, G.A. and Brooks III, E.D., *Gaussian Techniques on Shared Memory Multiprocessor Computers* from Proceedings of the Third SIAM conference on Parallel Processing for Scientific Computing, Rodrigue, G.(ed.), California, Dec.1-4, 1987, pp.20-26.
- [Dekker 94] Dekker, T.J., Hoffman, W., and Potma, K., *Parallel Algorithms for Solving Linear Systems*, Journal of Computational and Applied Mathematics 50 (1994), pp. 221-232.
- [Dongarra 84] Dongarra, J.J., Gustavson, F.G. and Karp, A., *Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine*, SIAM Review, Vol 26, 1984, pp 91-112.
- [Dongarra 91] Dongarra, J.J., Duff, I.S., Sorenson, D.C. and Van der Vorst, H.A., *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Press, 1991.
- [Dongarra 95a] Dongarra, J.J., Dunigan, T., *Message-Passing Performance of Various Computers*, University of Tennessee, 1995.  
<http://www.netlib.org/utk/papers/commperf.ps>
- [Dongarra 95b] Dongarra, J. and Walker, D., *Libraries for Linear Algebra in High Performance Computing* in High Performance Computing, Sabot, G.W. (ed) Addison Wesley 1995.
- [Douglas 93] Douglas, C.C., Mattson, T.G. and Schultz, M.H., *Parallel Programming Systems for Workstation Clusters*, Yale University Dept. of CS Research Report YALEU/DCS/TR-975, August 93.
- [Duncan 92] Duncan, R., *Parallel Computer Architectures*, Advances in Computers, Vol 34, M.C.Yovits (ed.), Academic Press, 1992, pp. 113-157.



- [East 95] East, I., *Parallel Processing with Communicating Process Architecture*, UCL Press Limited, 1995.
- [Evans 79] Evans, D.J. and Hatzopoulos, A *Parallel Linear System Solver*, Intern. J. Computer Math, 1979, Section B, Vol 7, pp 227-238.
- [Evans 82] Evans, D.J. and Sojoodi Haghighi, R., *Parallel Iterative Methods for Solving Linear Equations*, Inten. J. Computer Math, 1982, Vol 11, pp. 247-284.
- [Evans 84] Evans, D.J., *Parallel S.O.R. Iterative Methods*, Parallel Computing 1 (1984), pp. 3-18.
- [Evans 88] Evans, D.J. and Bekakos, M.P., *The Solution of The QIF Algorithm on a Wavefront Array Processor*, J.Parallel Computing, Vol 7, 1988, pp. 111-130.
- [Evans 93a] Evans, D.J. and Saeed, M.A., *Parallel Direct Methods For Solving Linear Systems*, Tech. Report, Computer Studies 843, Loughborough University of Technology, October 1993.
- [Evans 93b] Evans, D.J., *Implicit Matrix Elimination (IME) Schemes*, International J. of Computer Math, Vol 48, 1993, pp. 229-237.
- [Evans 94a] Evans, D.J. and Yalamov, P., *The QZ Orthogonal Decomposition Method*, Parallel Algorithms and Applications, Vol 2, pp. 263-276, 1994.
- [Evans 94b] Evans, D.J. and Abdullah, R., *The Parallel Implicit Elimination (PIE) Method for The Solution of Linear Systems*, Parallel Algorithms and Applications, Vol 4, pp. 153-162, 1994.
- [Evans 94c] Evans, D.J. and Abdullah, R., *LU and WZ Matrix Factorisation Methods with Partial Pivoting on Parallel Computers*, Report 941, Department of CS, Loughborough University, Nov 1994.
- [Evans 95a] Evans, D.J. and Abdullah, R., *Design and Analysis of Matrix Elimination and Matrix Factorisation Methods for the Solution of Linear Systems*, Report 976, Department of CS, Loughborough University, May 1995.
- [Evans 95b] Evans, D.J. and Abdullah, R., *A Comparison of the QR and QZ Matrix Factorisation Methods on Parallel Computers*, Parallel Algorithms and Applications, Vol 7, pp. 43-52, 1995.
- [Foster 95] Foster, I, *Designing and Building Parallel Programs* (online), Addison Wesley, 1995. <http://www.mc.anl.gov/dbpp/>

- [Fountain 94] Fountain, T.J., *Parallel Computing Principles and Practice*, Cambridge University Press, 1994.
- [Fox 64] Fox, L., *An Introduction to Numerical Linear Algebra*, Clarendon Press, 1964.
- [Fox 88] Fox, G.C., Johnson, M., Lyzenga, G. and Otto, S.W., *Solving Problems on Concurrent Processors, Vol. 1*, Prentice Hall, 1988.
- [Fox 94] Fox, G.C., *Parallel Computing Works!* Morgan Kaufman, 1994.
- [Freeman 92] Freeman, T.L. and Phillips, C., *Parallel Numerical Algorithms*, Prentice-Hall Int. (UK) Ltd., 1992.
- [Gallivan 90] Gallivan, K.A., Plemmons, R.J. and Sameh, A.H., *Parallel Algorithms for Dense Linear Algebra Computations*, SIAM Reveiw, Vol 32, March 1990, pp 54-135.
- [Garcia 90] Garcia, I., Merelo, J.J., Bruguera, J. D., and Zapata, E.L., *Parallel Quadrant Interlocking Factorisation on Hypercube Computers*, *Parallel Comptuing* 15 (1990) pp. 87-100.
- [Geist 85] Geist, G.A., *Efficient Parallel LU Factorization With Pivoting on a Hypercube Multiprocessor*, T. R. ORNL-6211, Oct 1985.
- [Geist 87] Geist, G.A. and Heath, M.T., *Matrix Factorization on a Hypercube Multiprocessor*, Hypercube Multiprocessor 1987, M.T.Heath (Ed) SIAM, 1987.
- [Geist 94a] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V., *PVM: Parallel Virtual Machine, A Users Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.
- [Geist 94b] Geist, G.A., *Cluster Computing: The Wave of the Future?* *Parallel Scientific Computing*, 1st Int Workshop 1994, pp. 236-246.
- [Gentleman 75] Gentleman, W.M., *Error Analysis of QR decomposition by Givens transformation*, *Linear Algebra and it's Applications* 10, 1975, pp. 189-197.
- [Gentleman 78] Gentleman, W.M., *Some Complexity Results for Matrix Computations on Parallel Processors*, J.ACM Vol 25, No 1, January 1978, pp. 112-115.
- [Hageman 81] Hageman, L.A. and Young, D.M., *Applied Iterative Methods*, Academic Press, 1981.

- [Heller 78] Heller, D., *A Survey of Parallel Algorithms in Numerical Linear Algebra*, SIAM Review, Vol 20, #4, Oct 78, pp 740-777.
- [Hey 94] Hey, T and Ferrante, J., (eds), *Portability and Performance for Parallel Processing*, Wiley, 1994.
- [Hockney 88] Hockney, R.W. and Jesshope, C.R., *Parallel Computers 2 Architecture, Programming and Algorithms*, Adam Hilger, 1988.
- [Hockney 94] Hockney, R.W., *The Communication Challenge for MPP: Intel Paragon and Meiko CS-2*, Parallel Computing 20 (1994) pp.389-398.
- [Hockney 96] Hockney, R.W., *The Science of Computer Benchmarking*, SIAM, 1996.
- [Hultquist 88] Hultquist, P.F., *Numerical Methods for Engineers and Computer Science*, Benjamin Cummings, 1988.
- [Ibarra 94] Ibarra, O.H. and Kim, M.H., *Fast Parallel Algorithms for Solving Triangular Systems of Linear Equations on the Hypercube*, J. of Parallel and Distributed Computing 20, pp 303-316 (1994).
- [Jamieson 87] Jamieson, L.H., Gannon, D. and Douglass R.J., (eds), *The Characteristics of Parallel Algorithms*, The MIT Press, 1987.
- [Johnston 66] Johnston, J.B., Price, G.B. and Van Vleck, F.S., *Linear Equations and Matrices*, Addison Wesley, 1966.
- [Kronsjo 85] Kronsjo, L., *Computational Complexity of Sequential and Parallel Algorithms*, John Wiley, 1985.
- [Kumar 94] Kumar, V., Grama, A., Gupta, A. and Karypis, G., *Introduction to Parallel Computing Design and Analysis of Algorithms*, Benjamin Cummings, 1994.
- [Kung 76] Kung, H.T., *Synchronized and Asynchronous Parallel Algorithms for Multiprocessors* in Algorithms and Complexity: New Directions and Recent Results, J.F. Traub, ed. pp.153-200, Academic Press, New York, 1976.
- [Kung 80] Kung, H.T., *The Structure of Parallel Algorithms*, in M.Yovits, ed, Advances in Computers, Vol 19, Academic Press, New York, pp. 65-111

- [Lester 93] Lester, B.P., *The Art of Parallel Programming*, Prentice Hall, 1993.
- [Levin 90] Levin, M.D., *Parallel Algorithms for SIMD and MIMD Computers*, Thesis, Loughborough University of Technology, 1990.
- [Lewis 92] Lewis, T.G. and El-Rewini, H., *Introduction to Parallel Computing*, Prentice Hall, 1992.
- [Lewis 93] Lewis, T.G., *Foundations of Parallel Programming A Machine-Independent Approach*, IEEE Comp. Soc. Press., 1993.
- [Lilja 91] Lilja, D.J., *Architectural Alternatives for Exploiting Parallelism*, IEEE Computer Society Press, 1991.
- [Lord 83] Lord, R.E., Kowalik, J.S. and Kumar, S.P., *Solving Linear Algebraic Equations on an MIMD Computer*, Journal of the ACM, Vol 30, No 1, January 1983, pp. 103-117.
- [Margulis 64] Margulis, B.E., *Systems of Linear Equations*, Pergamon Press, 1964.
- [Mathews 87] Mathews, J.H., *Numerical Methods for Computer Science, Engineering and Mathematics*, Prentice Hall, 1987.
- [McBryan 94] McBryan, O.A., *An Overview of Message Pasing Environments*, Parallel Computing 20 (1994) pp 417-444.
- [Miranker 71] Miranker, W.L., *A Survey of Parallelism in Numerical Analysis*, SIAM Review, Vol 13, No 4, Oct 71, pp. 54.
- [Mitra 87] Mitra, D., *Asynchronous Relaxations for the Numerical Solution of Differential Equation by Parallel Processors*, SIAM J. Sci. Stat. Comp. 8, No.1, 1987.
- [Modi 84]] Modi, J.J. and Clarke, M.R.S., *An Alternative Givens Ordering*, Numer. Math. 43, pp. 83-90, 1984.
- [Modi 88] Modi, J.J., *Parallel Algorithms and Matrix Computation*, Oxford University Press, 1988.
- [Moldovan 93] Moldovan, D.I., *Parallel Processing from Applications to Systems*, Morgan Kaufmann Publishers, 1993.
- [Morse 94] Morse, H.S., *Practical Parallel Computing*, Academic Press, 1994.

- [Nieplocha 92a] Nieplocha, J., *Solving Large Systems of Linear Equations Using Asynchronous Iterations on Parallel and Distributed Computers*, Dissertation, Univ of Alabama, Tuscaloosa, 1992.
- [Nieplocha 92b] Nieplocha, J., Mai, T. and Carroll, C.C., *Asynchronous Algorithms for Solving Large Systems of Linear Equations on Parallel Computers* in *Parallel Computing: From Theory to Sound Practice*, W.Joosen and E.Milgrom, Eds., pp. 76-79, IOS Press, 1992.
- [Ortega 85] Ortega, J.M and Voigt, R.G., *Solution of Partial Differential Equations on Vector and Parallel Computers*, SIAM Review, Vol 27, No 2, June 1985, pp. 149
- [Ortega 88] Ortega, J.M., *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1988.
- [Osterhaug 89] Osterhaug, A., *Guide to Parallel Programming on Sequent Computer Systems*, Second Edition, Prentice Hall, 1989.
- [Patel 84] Patel, N.R. and Jordan, H.F., *A Parallelized Point Rowwise Successive Over-relaxation Method on a Multiprocessor*, *Parallel Computing* 1 (1984) pp. 207-222.
- [Quinn 87] Quinn, M.J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.
- [Quinn 94] Quinn, M.J., *Parallel Computing Theory and Practice*, McGraw-Hill, 1994.
- [Rice 83] Rice, J.R., *Matrix Computations and Mathematical Software*, McGraw-Hill, 1983.
- [Sabot 95] Sabot, G.W., (ed), *High Performance Computing*, Addison Wesley, 1995.
- [Saeed 92] Saeed, M.A., *A Study of Numerical Algorithms for Hypercube Multiprocessor Systems*, PhD. Thesis, Loughborough University of Technology, 1992.
- [Sameh 77] Sameh, A.H. and Kuck, D.J. *Parallel Direct Linear System Solvers*, *Parallel Computers - Parallel Mathematics*, M.Feilmeyer (ed.), International Association for Mathematics and Computers in Simulation, 1977, pp. 25-29.
- [Sameh 78] Sameh, A. and Kuck, D.J., *On Stable Linear System Solvers*, *J.Assoc. Comput. Mach.*, No 25, pp 81-89, 1978.

- [Schendel 84] Schendel, U., *Introduction to Numerical Methods for Parallel Computers*, Ellis Horwood Limited, 1984.
- [Schmidt 95] Schmidt, B.K., Sunderam, V.S., *Empirical Analysis Overheads in Cluster Environments*, Dept. of Math and CS, Emory University, Atlanta, 1995.  
ftp://ftp.mathcs.em.ub/vss/empanal.ps.Z
- [Shanechi 80] Shanechi, J., *The Determination of Sparse Eigensystems and Parallel Linear System Solver*, Ph.d. Thesis, Loughborough University of Technology, 1980.
- [Shanechi 82] Shanechi, J. and Evans, D.J., *Further Analysis of the Quadrant Interlocking Factorisation (QIF) Method*, Intern. J. Computer Math, 1982, Vol 11, pp. 49-72.
- [Smith 78] Smith, G.D., *Numerical Solution of Partial Differential Equations: Finite difference methods Second Edition*, Oxford Applied Mathematics and Computing Science Series, 1978.
- [Smith 93] Smith, J.R., *The Design and Analysis of Parallel Algorithms*, Oxford University Press, 1993.
- [Sunderam 94a] Sunderam, V., *Methodologies and Systems for Heterogeneous Concurrent Computing*, Parallel Computing: Trends and Applications, Joubert, G.R., Trystram, D., Peters, F.J. and Evans, D.J. (eds), Elsevier Science, 1994.
- [Sunderam 94b] Sunderam, V.S., Geist, G.A., Dongarra, J and Manchek, R., *The PVM concurrent computing system: Evolution, experiences and trends*, Parallel Computing 20 (1994) pp 531-545.
- [Tanenbaum 90] Tanenbaum, A.S., *Structured Computer Organization Third Edition*, Prentice-Hall, 1990.
- [Traub 73] Traub, J.F.[ed], *Complexity of Sequential and Parallel Numerical Algorithms*, Academic Press, 1973.
- [Uresin 89] Uresin, A. and Dubois, M., *Sufficient Conditions for Convergence of Asynchronous Iterations*, Parallel Computing, Vol 10, pp. 83-92, 1989.
- [Uresin 90] Uresin, A. and Dubois, M., *Parallel Asynchronous Algorithms for Discrete Data*, J.ACM, Vol 37, No 3, pp. 588-606, 1990.
- [Varga 62] Varga, R.S., *Matrix Iterative Analysis*, Prentice Hall, N.J., 1962.

- [Walker 94] Walker, D.W., *The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers*, Parallel Computing 20 (1994) pp 657-673.
- [Watkins 91] Watkins, D.S., *Fundamentals of Matrix Computations*, John Wiley, 1991.
- [Wright 91] Wright, K., *Parallel Algorithms for QR Decomposition on a Shared Memory Multiprocessor*, Parallel Computing 17 (1991) pp. 779-790.
- [Yalamov 95] Yalamov, P. and Evans, D.J., *The WZ Matrix Factorisation Method*, Parallel Computing 21, 1995, pp 1111-1120.
- [Young 71] Young, D.M., *Iterative Solution of Large Linear Systems*, Academic Press, 1971.
- [Zomaya 96] Zomaya, A.Y.H., [ed], *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996.

# ***APPENDIX***



```

/*      Rosni Abdullah
        PARC, Computer Studies.
        Parallel Implicit Elimination (Without Pivot)
        This program generates input matrix where elements are:
            a[i][j]=1 and a[i][i]=n
        Filename: new_np_pie.c */

#include <stdio.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>
#include <math.h>

/* Global memory shared data */
shared float **a;
shared int const_posn, cp_loop, m1, m2;
shared int col_posn1, col_posn2, const_col, num_row;
FILE *fp_in, *fp_out;
FILE *fopen();
int time1, time2, time3;

main()
{
    float ** setup_matrix();
    char *shmalloc();
    int i, j, k, l, np, num, size_a;
    extern CLOCK();
    void setup(), init_a(), gauss(), set_solve(), print_results(), pie();

    if ((fp_in=fopen("data_size", "r"))==NULL)
        printf("Can't open file size\n");
    fscanf(fp_in, "%d", &size_a);
    while (size_a != 0) {
        a=setup_matrix(size_a, size_a+1);
        if (size_a % 2 == 0)
            num = size_a/2-1;
        else
            num = size_a/2;
        for (np=1; np<=10; np++) {
            m_set_procs(np);
            init_a(a, size_a);
            m_fork(pie, a, size_a, num);
            print_results(a, size_a, np);
            m_kill_procs();
        } /* for the np-th processor */
        fscanf(fp_in, "%d", &size_a);
    } /* while matrix size still exists */
    m_kill_procs();
} /* main */

/* procedure to initialise the coefficient array a */
void init_a(a, size)
float **a;
int size;
{
    int i, j;
    float sum;

```

```

        for (i=0; i <= size-1; i++) {
            sum = 0.0;
            for (j=0; j<= size-1; j++) {
                if (i==j) a[i][j]=size;
                else a[i][j]=1;
                sum += a[i][j];
            }
            a[i][size]=sum;
        }
    }

/* function to setup matrix of variable size */
float **
setup_matrix(nrows,ncols)
int nrows, ncols;
{
    int i,j;
    float **new_matrix;
/* allocate pointer arrays; set new_matrix to address of newly allocated shared matrix */
new_matrix = (float **) shmalloc(nrows*(sizeof(float *)));

/* allocate data arrays : set first element of new_matrix to address of first element of newly allocated
array */
new_matrix[0] = (float *) shmalloc(nrows * ncols * (sizeof(float)));

/* initialise pointer arrays : set each element of new_matrix to address of corresponding element of
data array */
for (i=1; i < nrows;i++)
    new_matrix[i] = new_matrix[0] + (ncols*i);
return(new_matrix);
}

#include <sys/types.h>
#include <sys/timeb.h>
int CLOCK()
{
    struct timeb tp;
    ftime(&tp);
    return (tp.time*100 + tp.millitm/10);
}

void pie(a,size_a,num)
float **a;
int size_a,num;
{
    int k, np, num1, n1, i, j, jj;
    float p1, p2, x1val, x2val;

    np=m_get_numprocs();
    m_single();
    col_posn1 = 0;
    col_posn2 = size_a-1;
    const_col=0;
    num_row=(size_a-2);
    time1=CLOCK();
    for (k=1; k<=num; k++) {

```

```

const_posn = const_col;
cp_loop=const_posn+1;
m_multi();
for (i=m_get_myid()+cp_loop; i<=const_posn+num_row; i+=np) {
    p1=a[col_posn1][col_posn2]/a[col_posn1][col_posn1];
    x2val=(p1*a[i][col_posn1]-a[i][col_posn2])/
        (p1*a[col_posn2][col_posn1]-a[col_posn2][col_posn2]);
    p2=a[col_posn2][col_posn1]/a[col_posn2][col_posn2];
    x1val=(p2*a[i][col_posn2]-a[i][col_posn1])/
        (p2*a[col_posn1][col_posn2]-a[col_posn1][col_posn1]);
    for (j=col_posn1+1; j <= col_posn2-1; j++)
        a[i][j] = a[i][j]-(x1val*a[col_posn1][j]+x2val*a[col_posn2][j]);
    a[i][size_a]=a[i][size_a]-(x1val*a[col_posn1][size_a]
        +x2val*a[col_posn2][size_a]);
} /* for */
m_sync();
m_single();
col_posn1++;
col_posn2--;
num_row -= 2;
const_col++;
} /* for k */
time2=CLOCK();
/* Bidirectional Substitution */
n1 = size_a;
m1 = num;
m2 = num+1;
if (size_a%2 != 0) {
    a[m1][n1]=a[m1][n1]/a[m1][m1];
    m_multi();
    for (j=m_get_myid(); j<=m1-1; j+=np) {
        a[j][m1]=a[m1][size_a]*a[j][m1];
        a[j][size_a]=a[j][size_a]-a[j][m1];
        a[size_a-1-j][m1]=a[m1][size_a]*a[size_a-1-j][m1];
        a[size_a-1-j][size_a]=a[size_a-1-j][size_a]-a[size_a-1-j][m1];
    }
    m_sync();
    m_single();
    m1=num-1;
    m2=num+1;
    p1 = a[m2][m1]/a[m1][m1];
    p2 = a[m2][m2] - p1*a[m1][m2];
    a[m2][n1] = (a[m2][n1]-p1*a[m1][n1])/p2;
    a[m1][n1] = (a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
}
else {
    p1 = a[m2][m1]/a[m1][m1];
    p2 = a[m2][m2] - p1*a[m1][m2];
    a[m2][n1] = (a[m2][n1]-p1*a[m1][n1])/p2;
    a[m1][n1] = (a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
}
if (size_a%2 != 0) num1=size_a/2-1;
else num1=num;
for (k=1; k<=num1; k++) {
    m_multi();
    for (j=m_get_myid(); j<=m1-1; j+=np) {
        a[j][m1]=a[m1][size_a]*a[j][m1];
        a[j][m2]=a[m2][size_a]*a[j][m2];

```

```

        a[j][size_a]=a[j][size_a]-(a[j][m1]+a[j][m2]);
        a[size_a-1-j][m1]=a[m1][size_a]*a[size_a-1-j][m1];
        a[size_a-1-j][m2]=a[m2][size_a]*a[size_a-1-j][m2];
        a[size_a-1-j][size_a]=a[size_a-1-j][size_a]-(a[size_a-1-j][m1]+a[size_a-1-j][m2]);
    }
    m_sync();
    m_single();
    m1--;
    m2++;
    p1 = a[m2][m1]/a[m1][m1];
    p2 = a[m2][m2]-p1*a[m1][m2];
    a[m2][n1]=(a[m2][n1]-p1*a[m1][n1])/p2;
    a[m1][n1]=(a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
}
time3=CLOCK();
m_multi();
}

void print_results(a,size_a,np)
float **a;
int size_a,np;
{
    int itmp;
    float temp_sum;

    if ((fp_out=fopen("new_outnppie","a"))==NULL)
        printf("Can't open file outnppie\n");
    fprintf(fp_out,"\\nPIE (pivot) for %d by %d matrix using %d processors:
        \\n",size_a,size_a,np);
    temp_sum=0.0;
    for (itmp =0; itmp < size_a; itmp++) {
        temp_sum += a[itmp][size_a];
        /*fprintf(fp_out,"\\n x%d = %f", itmp, a[itmp][size_a]);*/
    }
    fprintf(fp_out,"\\n Elimination time:%f \\n",(float)(time2-time1)/100.0);
    fprintf(fp_out,"Bidirectional substitution time : %f \\n",
        (float)(time3-time2)/100.0);
    fprintf(fp_out,"Total time taken: %f\\n",
        (float)((time3-time2)+(time2-time1))/100.0);
    fprintf(fp_out,"Temp_sum = %f \\n",temp_sum);
    fclose(fp_out);
}

```

```

/*      Rosni Abdullah
        PARC, Computer Studies.
        Parallel WZ Factorization
        Filename: new_parwz.c */

#include <stdio.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>

/* Global memory shared data */
shared float **a;
private float ge[2][3];
shared int const_posn, cp_loop, col_posn1, col_posn2, const_col, num_row, m1, m2;
FILE *fp_in, *fp_out;
FILE *fopen();
float x1val, x2val;
int time1, time2, time3, time4, time5, time6;

main()
{
    float ** setup_matrix();
    char *shmalloc();
    int i, j, k, size_a, num, np;
    extern CLOCK();
    void decomp(), setup_w(), wz_factor(), init_a(), gauss(), print_mat(), print_results();

    if ((fp_in=fopen("data_size", "r"))==NULL)
        printf("Can't open file size\n");
    fscanf(fp_in, "%d", &size_a);
    while (size_a != 0) {
        a=setup_matrix(size_a, size_a+1);
        for (np=1; np<=10; np++) {
            init_a(a, size_a);
            m_set_procs(np);
            if (size_a % 2 == 0)
                num = size_a/2-1;
            else
                num = size_a/2;
            m_fork(wz_factor, a, size_a, num);
            print_results(a, size_a, np);
            m_kill_procs();
        }
        fscanf(fp_in, "%d", &size_a);
    }
} /* main */

/* procedure to initialise the coefficient array a */
void init_a(a, size)
float **a;
int size;
{
    int i, j;
    float sum;

    for (i=0; i<=size-1; i++) {
        sum=0.0;
        for (j=0; j<=size-1; j++) {

```

```

        if (i==j) a[i][j] = size;
        else a[i][j]=1.0;
        sum+=a[i][j];
    }
    a[i][size]=sum;
}
}

/* function to setup matrix of variable size */
float **
setup_matrix(nrows,ncols)
int nrows, ncols;
{
    int i,j;
    float **new_matrix;
/* allocate pointer arrays; set new_matrix to address of newly allocated shared matrix */
    new_matrix = (float **) shmalloc(nrows*(sizeof(float *)));

/* allocate data arrays : set first element of new_matrix to address of
    first element of newly allocated array */
    new_matrix[0] = (float *) shmalloc(nrows * ncols * (sizeof(float)));

/* initialise pointer arrays : set each element of new_matrix to address of
    corresponding element of data array */
    for (i=1; i < nrows;i++) {
        new_matrix[i] = new_matrix[0] + (ncols*i); }
    return(new_matrix);
}

void gauss(ge)
float ge[][3];
{
    float factor;
    int j;

    factor = ge[1][0]/ge[0][0];
    for (j=0; j<=2; j++) {
        ge[0][j] = ge[0][j] * factor;
        ge[1][j] = ge[1][j] - ge[0][j];
    }
    x2val = ge[1][2] / ge[1][1];
    x1val = (ge[0][2] - (ge[0][1] * x2val)) / ge[0][0];
}

#include <sys/types.h>
#include <sys/timeb.h>
int CLOCK() {
    struct timeb tp;
    ftime(&tp);
    return (tp.time*100 + tp.millitm/10);
}

void
setup_w(ge,a,row_a,start_col,end_col)
float ge[][3], **a;
int row_a, start_col, end_col;
{
    ge[0][0] = a[start_col][start_col];

```

```

    ge[0][1] = a[end_col][start_col];
    ge[0][2] = a[row_a][start_col];
    ge[1][0] = a[start_col][end_col];
    ge[1][1] = a[end_col][end_col];
    ge[1][2] = a[row_a][end_col];
}

void wz_factor(a,size_a,num)
float **a;
int size_a, num;
{
    int n1, i, j, k, num1, np;
    float p1, p2;

    /* factorisation process */
    np=m_get_numprocs();
    m_single();
    col_posn1 = 0;
    col_posn2 = size_a-1;
    const_col=0;
    num_row=(size_a-2);
    time1=CLOCK();
    for (k=1; k<=num; k++) {
        const_posn = const_col;
        m_multi();
        cp_loop=const_posn+1;
        for (i=m_get_myid()+cp_loop; i<=const_posn+num_row; i+=np) {
            setup_w(ge,a,i,col_posn1,col_posn2);
            gauss(ge);
            a[i][col_posn1]=x1val;
            a[i][col_posn2]=x2val;
            for (j=col_posn1+1; j <= col_posn2-1; j++) {
                a[i][j] = a[i][j]-(x1val*a[col_posn1][j]+x2val*a[col_posn2][j]);
            }
            m_sync();
            m_single();
            col_posn1++;
            col_posn2--;
            num_row -= 2;
            const_col++;
        }
        m_single();
        time2=CLOCK();

        /* Inward substitution */
        col_posn1 = 0;
        col_posn2 = size_a - 1;
        const_col = 0;
        num_row = size_a - 2;
        time3=CLOCK();
        for (k=1; k<=num; k++) {
            const_posn = const_col;
            m_multi();
            cp_loop=const_posn+1;
            for (i=m_get_myid()+cp_loop; i<=const_posn+num_row; i+=np) {
                a[i][col_posn1]=a[col_posn1][size_a]*a[i][col_posn1];
                a[i][col_posn2]=a[col_posn2][size_a]*a[i][col_posn2];
            }
        }
    }
}

```

```

        a[j][size_a]=a[i][size_a]-(a[i][col_posn1]+a[i][col_posn2]);
    }
    m_sync();
    m_single();
    col_posn1++;
    col_posn2--;
    const_col++;
    num_row-=2;
}
time4=CLOCK();

/* Bidirectional Substitution */
time5=CLOCK();
n1 = size_a;
m1 = num;
m2 = num+1;
if (size_a%2 != 0) {
    a[m1][n1]=a[m1][n1]/a[m1][m1];
    m_multi();
    for (j=m_get_myid(); j<=m1-1; j+=np) {
        a[j][m1]=a[m1][size_a]*a[j][m1];
        a[j][size_a]=a[j][size_a]-a[j][m1];
        a[size_a-1-j][m1]=a[m1][size_a]*a[size_a-1-j][m1];
        a[size_a-1-j][size_a]=a[size_a-1-j][size_a]-a[size_a-1-j][m1];
    }
    m_sync();
    m_single();
    m2=num+1;
    p1 = a[m2][m1]/a[m1][m1];
    p2 = a[m2][m2] - p1*a[m1][m2];
    a[m2][n1] = (a[m2][n1]-p1*a[m1][n1])/p2;
    a[m1][n1] = (a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
}
else {
    p1 = a[m2][m1]/a[m1][m1];
    p2 = a[m2][m2] - p1*a[m1][m2];
    a[m2][n1] = (a[m2][n1]-p1*a[m1][n1])/p2;
    a[m1][n1] = (a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
}
if (size_a%2 != 0) num1=size_a/2-1;
else num1=num;
for (k=1; k<=num1; k++) {
    m_multi();
    for (j=m_get_myid(); j<=m1-1; j+=np) {
        a[j][m1]=a[m1][size_a]*a[j][m1];
        a[j][m2]=a[m2][size_a]*a[j][m2];
        a[j][size_a]=a[j][size_a]-(a[j][m1]+a[j][m2]);
        a[size_a-1-j][m1]=a[m1][size_a]*a[size_a-1-j][m1];
        a[size_a-1-j][m2]=a[m2][size_a]*a[size_a-1-j][m2];
        a[size_a-1-j][size_a]=a[size_a-1-j][size_a]-
            (a[size_a-1-j][m1]+a[size_a-1-j][m2]);
    }
    m_sync();
    m_single();
    m1--;
    m2++;
    p1 = a[m2][m1]/a[m1][m1];
    p2 = a[m2][m2]-p1*a[m1][m2];

```



```

        a[m2][n1]=(a[m2][n1]-p1*a[m1][n1])/p2;
        a[m1][n1]=(a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
    }
    time6=CLOCK();
    m_multi();

}

void print_results(a,size_a,np)
float **a;
int size_a, np;
{
    int itmp;
    float tot;

    if ((fp_out=fopen("new_outnpwz","a"))==NULL)
        printf("Can't open file outnpwz\n");
    fprintf (fp_out,"\nParallel WZ pivot for %d by %d matrix using %d
        processors: \n",size_a,size_a,np);

    tot=0.0;
    for (itmp =0; itmp < size_a; itmp++) {
        tot+=a[itmp][size_a];
        /*fprintf(fp_out,"\n x%d = %f", itmp, a[itmp][size_a]);*/
    }
    fprintf(fp_out,"\n The factorisation time :%f\n ", (float)(time2-time1)/100.0);
    fprintf(fp_out,"Bidirectional time is : %f\n", (float)(time6-time5)/100.0);
    fprintf(fp_out,"Total time : %f\n",
        (float)((time2-time1)+(time4-time3)+(time6-time5))/100.0);
    fprintf(fp_out,"Total xi's = %f\n",tot);
    fclose(fp_out);
}

```

```

/*      Rosni Abdullah
        PARC, Computer Studies.
        Parallel QZ Decomposition
        Filename: parqz.c */

#include <stdio.h>
#include <math.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>
#define BITE 1
    shared float **a;
    shared int num_in_par, mem_cnt;
    int time1, time2, time3;
    FILE *fp_in, *fp_out;

main()
{
    int i, j, k, n, np, CLOCK();
    float tt, b;
    float **setup_matrix();
    char *shmalloc();
    double sqrt(), pow();
    void decomp1(), decomp2(), decomp3(), dec_up(), dec_down(), dec_down_last();
    void bi_sub(), sub_odd(), sub_up(), print_mat(), print_results(), qz(), init_a(), elim();

    if ((fp_in=fopen("data_size", "r"))==NULL)
        printf("Can't open file size\n");
    b=0.001;
    fscanf(fp_in, "%d", &n);
    while (n != 0) {
        a=setup_matrix(n, n+1);
        for (np=1; np<=10; np++) {
            m_set_procs(np);
            init_a(a, b, n);
            qz(n, a);
            print_results(a, n, np);
            m_kill_procs();
        } /* for np-th processor */
        fscanf(fp_in, "%d", &n);
    } /* while size exists */
}

void qz(n, a)
int n;
float **a;
{
    int i, j, k, m, nk;
    int i_down, j_down, i_up, j_up, num;
    float p1, p2;
    int j_up_prime, j_down_prime;

    m=n/2-1;
    num_in_par=1;
    time1=CLOCK();
    for (k=1; k<=m; k++) {
        if (k%2 != 0) num_in_par++;

```

```

        m_fork(decomp1,n,a,k,num_in_par,m);
        m_fork(decomp2,n,a,k,num_in_par,m);
    }
    if (n%2 != 0) {
        if ((m+1) %2 !=0) num_in_par++;
        m_fork(decomp3,n,a,k,num_in_par,m);
    }
    /*printf("Begin second phase of elimination \n");*/
    if (n%2 == 0) {
        if ((m-1) % 2 != 0) num_in_par=(m-1)/2 +1;
        else num_in_par=(m-1)/2;
        for (k=1; k<m; k++) {
            m_fork(dec_up,n,a,k,num_in_par,m);
            m_fork(dec_down,n,a,k,num_in_par,m);
            if (((m-1)%2 == 0) && (k%2 == 0))
                num_in_par--;
            if (((m-1) %2 != 0) && (k%2 != 0))
                num_in_par--;
        }
    }
    else {
        m++; /* to get center value for odd sized matrices */
        if ((m-1) % 2 != 0) {
            num_in_par=(m-1)/2+1;
            for (k=1; k<m; k++) {
                if (k%2 != 0) {
                    m_fork(dec_up,n,a,k,num_in_par,m);
                    num_in_par--;
                    m_fork(dec_down,n,a,k,num_in_par,m);
                }
                else {
                    m_fork(dec_up,n,a,k,num_in_par,m);
                    m_fork(dec_down,n,a,k,num_in_par,m);
                }
            }
        }
        else {
            num_in_par=(m-1)/2+1;
            for (k=1; k<m; k++) {
                if (k%2 != 0) {
                    num_in_par--;
                    m_fork(dec_up,n,a,k,num_in_par,m);
                    m_fork(dec_down,n,a,k,num_in_par,m);
                }
                else {
                    m_fork(dec_up,n,a,k,num_in_par,m);
                    m_fork(dec_down_last,n,a,k,num_in_par,m);
                }
            }
        }
    }
    time2=CLOCK();

    /* Bidirectional Substitution */
    bi_sub(a,n,m);
    time3=CLOCK();
    return;
}

```

```

void elim(a,n,e_row,top_piv,bottom_piv,left_col,right_col)
float **a;
int n, e_row,top_piv,bottom_piv,left_col,right_col;
{
    int i,j;
    float delta1, delta2, delta3, sqd1d3, sqd1d2d3, d1d3;
    float sin_t,cos_t, sin_f, cos_f, tmp, tmp_up,tmp_down, tmp_piv, tmp1, tmp2;

    tmp=a[bottom_piv][left_col]; tmp_down=a[bottom_piv][right_col];
    tmp1=a[e_row][left_col]; tmp_piv=a[e_row][right_col];
    tmp2=a[top_piv][left_col]; tmp_up=a[top_piv][right_col];
    delta1=tmp_down*tmp1-tmp*tmp_piv;
    delta2=tmp_up*tmp-tmp2*tmp_down;
    delta3=tmp2*tmp_piv-tmp_up*tmp1;
    d1d3=(delta1*delta1)+(delta3*delta3);
    sqd1d3=sqrt((double) d1d3);
    sqd1d2d3=sqrt((double)(delta2*delta2)+d1d3);

    sin_t=delta3/sqd1d3;
    cos_t=delta1/sqd1d3;
    sin_f=delta2/sqd1d2d3;
    cos_f=sqd1d3/sqd1d2d3;

    /*      UPDATE      */
    /* update eliminated row */
    for (i=left_col+1; i<right_col; i++) {
        tmp2=a[top_piv][i]*cos_t+a[bottom_piv][i]*sin_t;
        tmp=a[e_row][i];
        a[e_row][i]=cos_f*tmp2+(a[e_row][i]*sin_f);
        tmp1=a[bottom_piv][i];
        a[bottom_piv][i]=sin_f*tmp2-tmp*cos_f;
        a[top_piv][i]=(a[top_piv][i]*sin_t)-(tmp1*cos_t);
    }

    /* update corners of top and bottom pivot row */
    tmp=a[top_piv][left_col];
    tmp1=a[top_piv][right_col];
    /*tmp_down=a[bottom_piv][left_col];
    tmp2=a[bottom_piv][right_col];*/
    a[top_piv][left_col]=(tmp*sin_t)-(a[bottom_piv][left_col]*cos_t);
    a[top_piv][right_col]=(tmp1*sin_t)-(a[bottom_piv][right_col]*cos_t);

    a[bottom_piv][left_col]=sin_f*(tmp*cos_t+a[bottom_piv][left_col]*sin_t)-
        a[e_row][left_col]*cos_f;
    a[bottom_piv][right_col]=sin_f*(tmp1*cos_t+a[bottom_piv][right_col]*
        sin_t)-a[e_row][right_col]*cos_f;

    /* update the right hand sides */
    tmp_up=a[top_piv][n];
    tmp_down=a[bottom_piv][n];
    tmp_piv=a[e_row][n];
    tmp=tmp_up*cos_t+tmp_down*sin_t;
    a[top_piv][n]=(tmp_up*sin_t)-(tmp_down*cos_t);
    a[e_row][n]=(cos_f*tmp)+(tmp_piv*sin_f);
    a[bottom_piv][n]=(sin_f*tmp)-(tmp_piv*cos_f);
}

```

```

#include <sys/types.h>
#include <sys/timeb.h>
int CLOCK() {
    struct timeb tp;
    ftime(&tp);
    return (tp.time*100 + tp.millitm/10);
}

/* procedure to initialise the coefficient array a */
void init_a(a,b,size)
float **a,b;
int size;
{
    int i,j;
    float sum;

    for (i=0; i<=size-1; i++) {
        sum = 0.0;
        for (j=0; j<=size-1; j++) {
            if (i==j) a[i][j] = b;
            else a[i][j] = fabs((double)((i-j)/10.0));
            sum += a[i][j];
        }
        a[i][size]=sum;
    }
}

/* function to setup matrix of variable size */
float **
setup_matrix(nrows,ncols)
int nrows, ncols;
{
    int i,j;
    float **new_matrix;

    /* allocate pointer arrays; set new_matrix to address of newly allocated shared matrix */
    new_matrix = (float **) shmalloc(nrows*(sizeof(float *)));

    /* allocate data arrays : set first element of new_matrix to address of first
    element of newly allocated array */
    new_matrix[0] = (float *) shmalloc(nrows * ncols * (sizeof(float)));

    /* initialise pointer arrays : set each element of new_matrix to address of
    corresponding element of data array */
    for (i=1; i < nrows;i++)
        new_matrix[i] = new_matrix[0] + (ncols*i);
    return(new_matrix);
}

void decomp1(n,a,k,num_in_par,m)
float **a;
int n, k,num_in_par,m;
{
    int i,j;
    while (((j=BITE*(m_next()-1)+1)<num_in_par) && ((i=k-j+1) <= n-1))
        elim(a,n,i,j-1,((n-1)-(j-1)),j-1,((n-1)-(j-1)));
}

```

```

void decomp2(n,a,k,num_in_par,m)
float **a;
int n, k,num_in_par,m;
{
    int i,j;
    while (((j=BITE*(m_next()-1)+1)<num_in_par) && ((i=(n-k-1)+(j-1)) <= n-1))
        elim(a,n,i,j-1,((n-1)-(j-1)),j-1,((n-1)-(j-1)));
}

void decomp3(n,a,k,num_in_par,m)
float **a;
int n, k,num_in_par,m;
{
    int i,j;
    while (((j=BITE*(m_next()-1)+1)<num_in_par) && ((i=(m+1)-(j-1)) <= n-1))
        elim(a,n,i,j-1,((n-1)-(j-1)),j-1,((n-1)-(j-1)));
}

void dec_up(n,a,k,num_in_par,m)
float **a;
int n, k,num_in_par,m;
{
    int i,j;
    while (((j=BITE*(m_next()-1)+k)<k+num_in_par) && ((i=(k-j)+m) <= n-1))
        elim(a,n,i,j,((n-1)-j),j,((n-1)-j));
}

void dec_down(n,a,k,num_in_par,m)
float **a;
int n, k,num_in_par,m;
{
    int i,j;
    while (((j=BITE*(m_next()-1)+k)<k+num_in_par) && ((i=(m+1)+(j-k)) <= n-1))
        elim(a,n,i,j,((n-1)-j),j,((n-1)-j));
}

void dec_down_last(n,a,k,num_in_par,m)
float **a;
int n, k,num_in_par,m;
{
    int i,j;
    while (((j=BITE*(m_next()-1)+k)<k+num_in_par-1) && ((i=(m+1)+(j-k)) <= n-1))
        elim(a,n,i,j,((n-1)-j),j,((n-1)-j));
}

void bi_sub(a,n,m)
float **a;
int n,m;
{
    int n1, k, m1, m2, num;
    float p1,p2;

    /* Bidirectional Substitution */
    n1 = n;
    m1 = m;
    m2 = m+1;
    if (n%2 != 0) {
        a[m1][n1]=a[m1][n1]/a[m1][m1];
    }
}

```

```

        m_fork(sub_odd,a,n,m1);
        m1=m-1;
        m2=m+1;
        p1 = a[m2][m1]/a[m1][m1];
        p2 = a[m2][m2] - p1*a[m1][m2];
        a[m2][n1] = (a[m2][n1]-p1*a[m1][n1])/p2;
        a[m1][n1] = (a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
    }
    else {
        p1 = a[m2][m1]/a[m1][m1];
        p2 = a[m2][m2] - p1*a[m1][m2];
        a[m2][n1] = (a[m2][n1]-p1*a[m1][n1])/p2;
        a[m1][n1] = (a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
    }
    if (n%2 != 0) num=n/2-1;
    else num=m;
    for (k=1; k<=num; k++) {
        m_fork(sub_up,a,n,m1,m2);
        m1--;
        m2++;
        p1 = a[m2][m1]/a[m1][m1];
        p2 = a[m2][m2]-p1*a[m1][m2];
        a[m2][n1]=(a[m2][n1]-p1*a[m1][n1])/p2;
        a[m1][n1]=(a[m1][n1]-a[m1][m2]*a[m2][n1])/a[m1][m1];
    }
}

void sub_odd(a,n,middle)
float **a;
int n, middle;
{
    int j,np;

    np=m_get_numprocs();
    for (j=m_get_myid(); j<=middle-1; j+=np) {
        a[j][middle]=a[middle][n]*a[j][middle];
        a[j][n]=a[j][n]-a[j][middle];
        a[n-1-j][middle]=a[middle][n]*a[n-1-j][middle];
        a[n-1-j][n]=a[n-1-j][n]-a[n-1-j][middle];
    }
}

void sub_up(a,n,mid1,mid2)
float **a;
int n,mid1,mid2;
{
    int j, np;

    np=m_get_numprocs();
    for (j=m_get_myid(); j<=mid1-1; j+=np) {
        a[j][mid1]=a[mid1][n]*a[j][mid1];
        a[j][mid2]=a[mid2][n]*a[j][mid2];
        a[j][n]=a[j][n]-(a[j][mid1]+a[j][mid2]);
        a[n-1-j][mid1]=a[mid1][n]*a[n-1-j][mid1];
        a[n-1-j][mid2]=a[mid2][n]*a[n-1-j][mid2];
        a[n-1-j][n]=a[n-1-j][n]-(a[n-1-j][mid1]+a[n-1-j][mid2]);
    }
}

```

```

void print_results(a,n,np)
float **a;
int n, np;
{
    int i;
    float tot;

    if ((fp_out=fopen("outqz","a"))==NULL)
        printf("Can't open file outqz\n");
    fprintf(fp_out, "\nParallel QZ for %d by %d matrix using %d processors\n",n,n,np);
    tot=0.0;
    for (i=0; i <= n-1; i++){
/*          fprintf(fp_out, "%5d    %16.6f\n", i, a[i][n]);*/
        tot+=a[i][n];
    }
    fprintf(fp_out, "Time taken for decomposition was: %f\n", (float)(time2-time1)/100.0);
    fprintf(fp_out, "Bidirectional Substitution time: %f\n", (float)(time2-time1)/100.0);
    fprintf(fp_out, "Time taken for QZ was: %f\n", (float)(time3-time1)/100.0);
    fprintf(fp_out, "Total xi's = %f\n", tot);
    fclose(fp_out);
}

```



```

/* PVM program pie_master.c */

#include <pvm3.h>
#define SLAVENAME "pie_slave"
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

main()
{
    int mytid,cnt;          /* my task id */
    int tids[32];          /* slave task ids */
    int n, nproc, i, j, msgtype, bufid;
    float **a, *xk, *xnk, middle, sum;
    int nextslave, numrows, pvminfo, *awhere, *bwhere, midindex, num;
    double time1, time2, secs(), idle1, idle2;

    /* enroll in pvm */
    mytid = pvm_mytid();

    cnt=0;
    /* start up slave tasks */
    puts("How many slave programs (1-32)?");
    scanf("%d", &nproc);
    /*nproc=2;*/
    pvm_spawn(SLAVENAME,(char**)0,0,"",nproc,tids);
    for (i=0; i< nproc; i++)
        printf("\n The tids[%d] = %d \n",i,tids[i]);

    /* Begin user program */
    puts("size of matrix is : \n");
    scanf("%d",&n);

    xk=((float *) malloc((n/2)*sizeof(float)));
    xnk=((float *) malloc((n/2)*sizeof(float)));
    awhere=((int *) malloc((n/2)*sizeof(int)));
    bwhere=((int *) malloc((n/2)*sizeof(int)));

    time1=secs();
    /* partition and distribute data to slaves */
    puts("sending data to slaves\n");
    pvm_initsend(PvmDataRaw);
    pvm_pkint(&nproc,1,1);
    pvm_pkint(tids,nproc,1);
    pvm_pkint(&n,1,1);
    pvm_mcast(tids,nproc,0);
    cnt++;
    puts("Data sent to slaves...\n");

    /* The next n messages send, two rows at a time, the row index, the corresponding row of a
    and the corresponding elements of b to the slave processes. The rows are distributed
    cyclically amongst the slave processes. The k-th and n-1-kth rows are sent to the slaves. */

    num=n/2-1;
    /*puts("Sending index of rows of matrix a to slaves \n");    */
    nextslave=0;
    msgtype=1;
    for (i=0; i<=num; i++) {

```

```

        pvm_itsend(PvmDataRow);
        pvm_pkint(&i,1,1);
        /*printf("Sending rows %d and %d to slave %d\n",i,n-1-i,nextslave);*/
        pvminfo=pvm_send(tids[nextslave],msgtype);
        cnt++;
        if (pvminfo < 0)
            printf("\n Error in send");
        if (nextslave == nproc-1)
            nextslave = 0;
        else
            nextslave++;
    }

/* send the middle row of the odd sized matrix */
if ((n%2) !=0) {
    i=n/2;
    pvm_itsend(PvmDataRow);
    pvm_pkint(&i,1,1);
    pvminfo=pvm_send(tids[nextslave],msgtype);
    cnt++;
}

/* Recieve results from slaves */
msgtype = 5;
puts("Waiting for results from slaves\n");
for (i=0; i<nproc; i++) {
    idle1=secs();
    bufid=pvm_recv(-1,msgtype);
    idle2=secs();
    cnt++;
    if (bufid <0) {
        printf("Error on pvm_recv! Bailing out...\n");
        pvm_exit();
        exit(-1);
    }
    pvm_upkint(&numrows,1,1);
    pvm_upkint(awhere,numrows,1);
    pvm_upkint(bwhere,numrows,1);
    pvm_upkint(&midindex,1,1);
    pvm_upkfloat(xk,numrows,1);
    sum=0.0;
    for (j=0;j<numrows;j++) {
        /* printf("x[%d]= %f\n",awhere[j],xk[j]);*/
        sum+=xk[j];
    }
    pvm_upkfloat(xnk,numrows,1);
    for (j=0;j<numrows;j++) {
        /* printf("x[%d]= %f\n",bwhere[j],xnk[j]);*/
        sum+=xnk[j];
    }
    pvm_upkfloat(&middle,1,1);
    /*printf("x[%d]= %f\n",midindex,middle);*/
    sum+=middle;
}
time2=secs();
puts("Finished getting results from slaves\n");
printf("Number of sends/receive is %d\n",cnt);
printf("Time for pie_master=%f\n",time2-time1);

```

```

    printf("Time (minus idle time) for pie_master=%f\n", (time2-time1)-(idle2-idle1));
    printf("Sum of xi's = %f\n", sum);
    puts("Exiting...\n");
    free(awhere);
    free(bwhere);
    free(xk);
    free(xnk);
    pvm_exit();
    exit();
}

```

```

#include <sys/time.h>
double secs()
{
    struct timeval ru;
    gettimeofday(&ru, (struct timezone *)0);
    return(ru.tv_sec + ((double)ru.tv_usec)/1000000);
}

```

```

/* Rosni Abdullah */
/* PIE slave program, began on 1/2/95 */

#include <stdio.h>
#include <pvm3.h>
#include <stdlib.h>
#include <time.h>

main()
{
    int mytid; /* my task id */
    int tids[32], others[31], mygid, info, index1, pvminfo, num, midindex;
    int n, me, i, j, j1, k, nproc, master, msgtype;
    float **arow, /* keeps rows 0 to n/2-1 of matrix */
          *middle,
          **brow; /* keeps row n/2 to n-2 of matrix */
    int *aindex, /* keeps index 0 to n/2-1 of matrix */
        *bindex; /* keeps index n/2 to n-1 of matrix */
    int numrows, nexta, nextslave, nextindex, ipointer, cnt;
    float *xk, /* in factorisation: keeps row k in solution process: keeps solution */
          *xnk, /* in factorisation: keeps row nk in solution process: keeps solution */
          sum, p1, p2, x1, x2;
    double time1, time2, secs(), idle1, idle2, idle3, idle4, idle5, idle6, idle7;
    double idle8, idle9, idle10, idle11, idle12, idle13, idle14, tmp1, tmp2,
          comp_t, comp1, comp2, pack_t, idle, pack1, pack2, pack3,
          pack4, pack5, pack6, pack7, pack8, pack_elim, pack_soln;

    idle1=idle2=idle3=idle4=idle5=idle6=idle7=0.0;
    idle8=idle9=idle10=idle11=idle12=idle13=idle14=0.0;
    pack1=pack2=pack3=pack4=pack5=pack6=pack7=pack8=0.0;

    /* enroll in PVM */
    mytid = pvm_mytid();
    master = pvm_parent();
    printf("I am %d\n", mytid);
    mygid = pvm_joining("workers");
    if (mygid < 0) {
        printf("Error in joining\n");
        pvm_exit();
        return;
    }
    cnt=0;
    time1=secs();
    /* receive data from master */
    msgtype = 0;
    idle1=secs();
    pvm_rcv(master, msgtype);
    idle2=secs();
    cnt++;
    pvm_upkint(&nproc, 1, 1);
    pvm_upkint(tids, nproc, 1);
    pvm_upkint(&n, 1, 1);

    /*determine which slave I am */
    j = 0;
    for ( i=0; i<nproc; i++) {
        if (mytid==tids[i])
            me=i;

```

```

        else
            others[j++] = tids[i];
    }
    printf("PIE with %d slaves for n= %d\n ", nproc, n);
    arow = (float **) malloc((n/nproc)*sizeof(float *));
    arow[0] = (float *) malloc((n/nproc) * (n+1) * (sizeof(float)));
    for (i=1; i < (n/nproc); i++)
        arow[i] = arow[0] + ((n+1)*i);
    brow = (float **) malloc((n/nproc)*sizeof(float *));
    brow[0] = (float *) malloc((n/nproc) * (n+1) * (sizeof(float)));
    for (i=1; i < (n/nproc); i++)
        brow[i] = brow[0] + ((n+1)*i);
    middle = (float *) malloc((n+1)*sizeof(float));
    xk = (float *) malloc((n+2)*sizeof(float));
    xnk = (float *) malloc((n+2)*sizeof(float));
    aindex = (int *) malloc((n/nproc+2)*sizeof(int));
    bindex = (int *) malloc((n/nproc+2)*sizeof(int));
    num = n/2-1;
    numrows = 0;

    /* unpacks the row index and the corresponding row of a */
    for (i=0; i < (num+1)/nproc; i++) {
        idle3 = secs();
        pvm_recv(master, 1);
        idle4 = secs();
        cnt++;
        pvm_upkint(&index1, 1, 1);
        aindex[numrows] = index1;
        bindex[numrows] = n-1-index1;
        numrows++;
    }
    if (((num+1) % nproc) != 0) {
        for (j=0; j < ((num+1) % nproc); j++) {
            if (mytid == tids[j]) {
                idle5 = secs();
                pvm_recv(master, 1);
                idle6 = secs();
                cnt++;
                pvm_upkint(&index1, 1, 1);
                aindex[numrows] = index1;
                bindex[numrows] = n-1-index1;
                numrows++;
            }
        }
    }

    if ((n%2) != 0) {
        if (mytid == tids[((num+1)%nproc)]) {
            idle7 = secs();
            pvm_recv(master, 1);
            idle8 = secs();
            cnt++;
            pvm_upkint(&index1, 1, 1);
            midindex = index1;
        }
        else midindex = 0;
    }
}

```

```

/* generates input for submatrix that each slave has to handle */
for (i=0; i<numrows; i++) {
    sum=0.0;
    for (j=0; j<n; j++) {
        if (aindex[i]!=j) arow[i][j]=1.0;
        else arow[i][j]=(float)n;
        sum+=arow[i][j];
    }
    arow[i][n]=sum;
}
for (i=0; i<numrows; i++) {
    sum=0.0;
    for (j=0; j<n; j++) {
        if (bindex[i]!=j) brow[i][j]=1.0;
        else brow[i][j]=(float)n;
        sum+=brow[i][j];
    }
    brow[i][n]=sum;
}
if ((n%2) !=0) {
    sum=0.0;
    for (j=0; j<n; j++) {
        if (midindex != j) middle[j]=1.0;
        else middle[j]=(float)n;
        sum+=middle[j];
    }
    middle[n]=sum;
}
/* factorisation process */
/* nextindex points to the next row on this slave process. After a row is being used, nextindex
is incremented to point to the next row on this slave process */
pack_elim=0.0;
comp1=secs();
ipointer = 0;
nextindex = aindex[ipointer];
for (k=0; k<=num; k++){
    if (k==nextindex) {
        xk[0]=aindex[ipointer];
        for (i=1; i<=n+1; i++)
            xk[i]=arow[ipointer][i-1];
        xnk[0]=bindex[ipointer];
        for (i=1; i<=n+1; i++)
            xnk[i]=brow[ipointer][i-1];
        pack1=secs();
        pvm_initsend(PvmDataRaw);
        pvm_pkfloat(xk,n+2,1);
        pvm_pkfloat(xnk,n+2,1);
        pvm_mcast(others,nproc-1,100);
        pack2=secs();
        cnt++;
        if (ipointer < numrows){
            ipointer++;
            nextindex=aindex[ipointer];
        }
    }
    else {
        idle9=secs();
        pvm_recv(-1,100);
    }
}

```

```

        idle10=secs();
        cnt++;
        pack3=secs();
        pvm_upkfloat(xk,n+2,1);
        pack4=secs();
        pvm_upkfloat(xnk,n+2,1);
    }
    for (j=0; j<numrows; j++) {
        if (aindex[j] > k) {
            p1 = (xk[n-k]/xk[k+1]);
            x2 = (arow[j][n-1-k] - p1 * arow[j][k])/(xnk[n-k] - p1 * xnk[k+1]);
            x1 = (arow[j][k]-xnk[k+1]*x2)/xk[k+1];
            arow[j][n-1-k]=x2;
            arow[j][k]=x1;
            for (j1=k+1; j1<=n-2-k; j1++)
                arow[j][j1]=x1*-(xk[j1+1])+x2*-(xnk[j1+1])+arow[j][j1];
            arow[j][n]=x1*-(xk[n+1])+x2*-(xnk[n+1])+arow[j][n];
        }
    }
    for (j=0; j<numrows; j++) {
        if (bindex[j] < n-1-k) {
            p1 = (xk[n-k]/xk[k+1]);
            x2 = (brow[j][n-1-k] - p1 * brow[j][k])
                /(xnk[n-k] - p1 * xnk[k+1]);
            x1 = (brow[j][k]-xnk[k+1]*x2)/xk[k+1];
            brow[j][n-1-k] = x2;
            brow[j][k] = x1;
            for (j1=k+1; j1<=n-2-k; j1++)
                brow[j][j1]=x1*-(xk[j1+1])+x2*-(xnk[j1+1])+brow[j][j1];
            brow[j][n]=x1*-(xk[n+1])+x2*-(xnk[n+1])+brow[j][n];
        }
    }
}

if ((n%2) != 0) {
    if (midindex > k) {
        p1=(xk[n-k]/xk[k+1]);
        x2=(middle[n-1-k]-p1*middle[k])/(xnk[n-k] - p1 * xnk[k+1]);
        x1=(middle[k]-xnk[k+1]*x2)/xk[k+1];
        middle[n-1-k] = x2;
        middle[k] = x1;
        for (j1=k+1; j1<=n-2-k; j1++)
            middle[j1]=x1*-(xk[j1+1])+x2*-(xnk[j1+1])+middle[j1];
        middle[n]=x1*-(xk[n+1])+x2*-(xnk[n+1])+middle[n];
    }
}
pack_elim+=(pack2-pack1)+(pack4-pack3);
}

/* Bidirectional Substitution */
pvm_barrier("workers",nproc);
if (n % 2 !=0) {
    if (midindex!=0) {
        middle[n]=middle[n]/middle[n/2];
        x1=middle[n];
        pvm_initsend(PvmDataRaw);
        pvm_pkfloat(&x1,1,1);
        pvm_mcast(others,nproc-1,160);
        cnt++;
    }
}

```

```

        else {
            idle11=secs();
            pvm_recv(-1,160);
            idle12=secs();
            cnt++;
            pvm_upkfloat(&x1,1,1);
        }
        for (j=0; j<numrows; j++) {
            if (aindex[j] < n/2) {
                arow[j][n]=arow[j][n]-arow[j][n/2]*x1;
                brow[j][n]=brow[j][n]-brow[j][n/2]*x1;
            }
        }
    }
    /*even matrices */
    pack_soln=0.0;
    ipointer=numrows-1;
    nextindex=aindex[ipointer];
    for (k=n/2-1; k>=0; k--) {
        if (k == nextindex) {
            p1=brow[ipointer][k]/arow[ipointer][k];
            x2=(brow[ipointer][n]-p1*arow[ipointer][n])/
                (brow[ipointer][n-1-k]-p1*arow[ipointer][n-1-k]);
            x1=(arow[ipointer][n]-arow[ipointer][n-1-k]*x2)/ [ipointer][k];
            arow[ipointer][n]=x1;
            brow[ipointer][n]=x2;
            pack5=secs();
            pvm_itsend(PvmDataRaw);
            pvm_pkfloat(&x1,1,1);
            pvm_pkfloat(&x2,1,1);
            pvm_mcast(others,nproc-1,200);
            pack6=secs();
            cnt++;
            if (ipointer > 0) {
                ipointer--;
                nextindex=aindex[ipointer];
            }
        }
        else {
            idle13=secs();
            pvm_recv(-1,200);
            idle14=secs();
            cnt++;
            pack7=secs();
            pvm_upkfloat(&x1,1,1);
            pvm_upkfloat(&x2,1,1);
            pack8=secs();
        }
        for (j=0; j<numrows; j++) {
            if (aindex[j]<k) {
                arow[j][n]=arow[j][n]-arow[j][k]*x1-arow[j][n-1-k]*x2;
                brow[j][n]=brow[j][n]-brow[j][k]*x1-brow[j][n-1-k]*x2;
            }
        }
        pack_soln+=(pack6-pack5)+(pack8-pack7);
    }
    comp2=secs();
    pack_t=pack_elim+pack_soln;

```



```

    comp_t=comp2-comp1-pack_t;
    /*printf("After bi_sub\n");*/
    for (i=0; i<numrows; i++) {
        xk[i]=arow[i][n];
        xnk[i]=brow[i][n];
    }
    /* send results to master */
    msgtype=5;
    pvm_initsend(PvmDataRow);
    pvm_pkint(&numrows,1,1);
    pvm_pkint(aindex,numrows,1);
    pvm_pkint(bindex,numrows,1);
    pvm_pkint(&midindex,1,1);
    pvm_pkfloat(xk,numrows,1);
    pvm_pkfloat(xnk,numrows,1);
    pvm_pkfloat(&middle[n],1,1);
    pvminfo=pvm_send(master,msgtype);
    if (pvminfo <0)
        printf("error in send \n");
    cnt++;
    printf("Number of sends/receive is %d\n",cnt);
    time2=secs();
    printf("Time taken by pie_slave=%f\n",time2-time1);
    tmp1=(idle2-idle1)+(idle4-idle3)+(idle6-idle5)+(idle8-idle7);
    tmp2=(idle10-idle9)+(idle12-idle11)+(idle14-idle13);
    printf("Finished.....\n");
    printf("Time(minus idle time) taken by pie_slave=%f\n", (time2-time1)-tmp1-tmp2);
    printf("Idle time of pie_slave= %f \n",tmp1+tmp2);
    printf("Computation time of pie_slave= %f \n",comp_t);
    printf("Communication time of pie_slave= %f \n",time2-time1-tmp1-tmp2-comp_t);
    free(arow);
    free(brow);
    free(aindex);
    free(bindex);
    free(xk);
    free(xnk);
    free(middle);
    pvm_barrier("workers",nproc);
    pvm_lvgroup("workers");
    pvm_exit();
}

#include <sys/time.h>
double secs()
{
    struct timeval ru;
    gettimeofday(&ru, (struct timezone *)0);
    return(ru.tv_sec + ((double)ru.tv_usec)/1000000);
}

```

```

/* PVM QIF master program : wz_master.c */

#include <pvm3.h>
#define SLAVENAME "wz_slave"
#include <stdio.h>
#include <stdlib.h>

main()
{
    int mytid, cnt, tids[32], n, nproc, i, j, msgtype, bufid;
    float **a, *xk, *xnk, middle, sum;
    int nextslave, numrows, pvminfo, *awhere, *bwhere, midindex, num;
    double time1, time2, secs(), idle1, idle2;

    /* enroll in pvm */
    mytid = pvm_mytid();

    /* start up slave tasks */
    puts("How many slave programs (1-32)?");
    scanf("%d", &nproc);
    pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
    for (i=0; i<nproc; i++)
        printf("\n The tids[%d] = %d \n", i, tids[i]);

    /* Begin user program */
    puts("size of matrix is : \n");
    scanf("%d", &n);
    xk=((float *) malloc((n/2)*sizeof(float)));
    xnk=((float *) malloc((n/2)*sizeof(float)));
    awhere=((int *) malloc((n/2)*sizeof(int)));
    bwhere=((int *) malloc((n/2)*sizeof(int)));
    cnt=0;
    time1=secs();
    /* partition and distribute data to slaves */
    puts("sending data to slaves\n");
    pvm_initsend(PvmDataRaw);
    pvm_pkint(&nproc, 1, 1);
    pvm_pkint(tids, nproc, 1);
    pvm_pkint(&n, 1, 1);
    pvm_mcast(tids, nproc, 0);
    cnt++;
    puts("Data sent to slaves...\n");

    /* The next n messages send, two rows at a time, the row index, the corresponding row of a
    and the corresponding elements of b to the slave processes. The rows are distributed
    cyclically amongst the slave processes. The k-th and n-1-kth rows are sent to the slaves. */
    num=n/2-1;
    puts("Sending index of rows of matrix a to slaves \n");
    nextslave=0;
    msgtype=1;
    for (i=0; i<=num; i++) {
        pvm_initsend(PvmDataRaw);
        pvm_pkint(&i, 1, 1);
        pvminfo=pvm_send(tids[nextslave], msgtype);
        cnt++;
        if (pvminfo < 0)
            printf("\n Error in send");
        if (nextslave == nproc-1)

```

```

        nextslave = 0;
    else
        nextslave++;
}
/* send the middle row of the odd sized matrix */
if ((n%2) != 0) {
    i=n/2;
    pvm_initsend(PvmDataRow);
    pvm_pkint(&i,1,1);
    pvminfo=pvm_send(tids[nextslave],msgtype);
}
/* Recieve results from slaves */
msgtype = 5;
puts("Waiting for results from slaves\n");
for (i=0; i<nproc; i++) {
    idle1=secs();
    bufid=pvm_recv(-1,msgtype);
    idle2=secs();
    cnt++;
    if (bufid < 0) {
        printf("Error on pvm_recv! Bailing out...\n");
        pvm_exit();
        exit(-1);
    }
    pvm_upkint(&numrows,1,1);
    pvm_upkint(awhere,numrows,1);
    pvm_upkint(bwhere,numrows,1);
    pvm_upkint(&midindex,1,1);
    pvm_upkfloat(xk,numrows,1);
    sum=0.0;
    for (j=0;j<numrows;j++)
        sum+=xk[j];
    pvm_upkfloat(xnk,numrows,1);
    for (j=0;j<numrows;j++)
        sum+=xnk[j];
    pvm_upkfloat(&middle,1,1);
}
time2=secs();
puts("Finished getting results from slaves \n");
printf("Number of sends/receive is %d\n",cnt);
printf("Time for wz_master=%f\n",time2-time1);
printf("Time (minus idle time) for wz_master=%f\n",(time2-time1)-(idle2-idle1));
printf("Total xi's = %f \n", sum);
puts("Exiting...\n");
pvm_exit();
exit();
}

#include <sys/time.h>
double secs()
{
    struct timeval ru;
    gettimeofday(&ru, (struct timezone *)0);
    return(ru.tv_sec + ((double)ru.tv_usec)/1000000);
}

```

```

/* Rosni Abdullah */
/* WZ slave program, began on 24/12/94 */

#include <stdio.h>
#include <pvm3.h>
#include <stdlib.h>
#include <time.h>

main()
{
    int mytid, cnt, tids[32], others[31], mygid, info;
    int n, me, i, j, j1, k, nproc, master, msgtype, index1, pvminfo, num, midindex;
    float **arow, /* keeps rows 0 to n/2-1 of matrix */
          *middle,
          **brow; /* keeps row n/2 to n-2 of matrix*/
    int *aindex, /* keeps index 0 to n/2-1 of matrix */
        *bindex; /* keeps index n/2 to n-1 of matrix */
    int numrows, nexta, nextslave, nextindex, ipointer;
    float *xk, /* in factorisation: keeps row k in solution process: keeps solution*/
          *xnk, /* in factorisation: keeps row nk in solution process: keeps solution*/
          sum, p1, p2, x1, x2;
    double secs(), time1, time2, pack_t, pack_elim, pack_soln, pack1, pack2,
          pack3, pack4, pack5, pack6, pack7, pack8, pack9, pack10, pack11,
          pack12, idle, idle1, idle2, idle3, idle4, idle5, idle6, idle7,
          idle8, idle9, idle10, idle11, idle12, idle13, idle14, idle15,
          idle16, comp_t, comp1, comp2, ump1, tmp2, pack_soln1;

    idle1=idle2=idle3=idle4=idle5=idle6=idle7=pack11=pack12=0.0;
    idle8=idle9=idle10=idle11=idle12=idle13=idle14=idle15=idle16=0.0;
    pack1=pack2=pack3=pack4=pack5=pack6=pack7=pack8=pack9=pack10=0.0;

    /* enroll in PVM */
    mytid = pvm_mytid();
    master = pvm_parent();
    printf("I am %d \n", mytid);
    mygid = pvm_joingroup("workers");
    if (mygid < 0) {
        printf("Error in joingroup\n");
        pvm_exit();
        return;
    }
    cnt=0;
    time1=secs();
    /* receive data from master */
    msgtype = 0;
    idle1=secs();
    pvm_rcv(master,msgtype);
    idle2=secs();
    cnt++;
    pvm_upkint(&nproc,1,1);
    pvm_upkint(&tids,nproc,1);
    pvm_upkint(&n,1,1);

    /*determine which slave I am */
    j = 0;
    for ( i=0; i<nproc; i++) {
        if (mytid==tids[i])
            me=i;

```

```

        else
            others[j++] = tids[i];
    }

    printf("QIF with %d slaves for n= %d\n ", nproc, n);
    arow = (float **) malloc((n/nproc)*sizeof(float *));
    arow[0] = (float *) malloc((n/nproc) * (n+1) * (sizeof(float)));
    for (i=1; i < (n/nproc); i++)
        arow[i] = arow[0] + ((n+1)*i);
    brow = (float **) malloc((n/nproc)*sizeof(float *));
    brow[0] = (float *) malloc((n/nproc) * (n+1) * (sizeof(float)));
    for (i=1; i < (n/nproc); i++)
        brow[i] = brow[0] + ((n+1)*i);
    middle = (float *) malloc((n+1)*sizeof(float));
    xk = (float *) malloc((n+2)*sizeof(float));
    xnk = (float *) malloc((n+2)*sizeof(float));
    aindex = (int *) malloc((n/nproc+2)*sizeof(int));
    bindex = (int *) malloc((n/nproc+2)*sizeof(int));
    num = n/2-1;
    numrows = 0;

    /* unpacks the row index and the corresponding row of a */
    for (i=0; i < (num+1)/nproc; i++) {
        idle3 = secs();
        pvm_recv(master, 1);
        idle4 = secs();
        cnt++;
        pvm_upkint(&index1, 1, 1);
        aindex[numrows] = index1;
        bindex[numrows] = n-1-index1;
        numrows++;
    }
    if (((num+1) % nproc) != 0) {
        for (j=0; j < ((num+1) % nproc); j++) {
            if (mytid == tids[j]) {
                idle5 = secs();
                pvm_recv(master, 1);
                idle6 = secs();
                cnt++;
                pvm_upkint(&index1, 1, 1);
                aindex[numrows] = index1;
                bindex[numrows] = n-1-index1;
                numrows++;
            }
        }
    }
    if ((n%2) != 0) {
        if (mytid == tids[((num+1)%nproc)]) {
            idle7 = secs();
            pvm_recv(master, 1);
            idle8 = secs();
            cnt++;
            pvm_upkint(&index1, 1, 1);
            midindex = index1;
        }
        else midindex = 0;
    }
}

```

```

/* generates input for submatrix that each slave has to handle */
for (i=0; i<numrows; i++) {
    sum=0.0;
    for (j=0; j<n; j++) {
        if (aindex[i]!=j) arow[i][j]=1.0;
        else arow[i][j]=(float)n;
        sum+=arow[i][j];
    }
    arow[i][n]=sum;
}
for (i=0; i<numrows; i++) {
    sum=0.0;
    for (j=0; j<n; j++) {
        if (bindex[i]!=j) brow[i][j]=1.0;
        else brow[i][j]=(float)n;
        sum+=brow[i][j];
    }
    brow[i][n]=sum;
}
if ((n%2) !=0) {
    sum=0.0;
    for (j=0; j<n; j++) {
        if (midindex != j) middle[j]=1.0;
        else middle[j]=(float)n;
        sum+=middle[j];
    }
    middle[n]=sum;
}
/* factorisation process */
/* nextindex points to the next row on this slave process. After a row is being used, nextindex
is incremented to point to the next row on this slave process */
pack_elim=0.0;
comp1=secs();
ipointer = 0;
nextindex = aindex[ipointer];
for (k=0; k<=num; k++){
    if (k==nextindex) {
        xk[0]=aindex[ipointer];
        for (i=1; i<=n+1; i++)
            xk[i]=arow[ipointer][i-1];
        xnk[0]=bindex[ipointer];
        for (i=1; i<=n+1; i++)
            xnk[i]=brow[ipointer][i-1];
        pack1=secs();
        pvm_initsend(PvmDataDefault);
        pvm_pkfloat(xk,n+2,1);
        pvm_pkfloat(xnk,n+2,1);
        pvm_mcast(others,nproc-1,100);
        pack2=secs();
        cnt++;
        if (ipointer < numrows){
            ipointer++;
            nextindex=aindex[ipointer];
        }
    }
    else {
        idle9=secs();
        pvm_recv(-1,100);
    }
}

```

```

        idle10=secs();
        cnt++;
        pack3=secs();
        pvm_upkfloat(xk,n+2,1);
        pack4=secs();
        pvm_upkfloat(xnk,n+2,1);
    }
    for (j=0; j<numrows; j++) {
        if (aindex[j] > k) {
            p1 = (xk[n-k]/xk[k+1]);
            x2 = (arow[j][n-1-k] - p1 * arow[j][k])/(xnk[n-k] - p1 * xnk[k+1]);
            x1 = (arow[j][k]-xnk[k+1]*x2)/xk[k+1];
            arow[j][n-1-k]=x2;
            arow[j][k]=x1;
            for (j1=k+1; j1<=n-2-k; j1++)
                arow[j][j1]=arow[j][j1]-arow[j][k]*xk[j1+1]-
                    arow[j][n-1-k]* xnk[j1+1];
        }
    }
    for (j=0; j<numrows; j++) {
        if (bindex[j] < n-1-k) {
            p1 = (xk[n-k]/xk[k+1]);
            x2 = (brow[j][n-1-k] -p1 * brow[j][k])/(xnk[n-k]- p1* xnk[k+1]);
            x1 =(brow[j][k]-xnk[k+1]*x2)/xk[k+1];
            brow[j][n-1-k] = x2;
            brow[j][k] = x1;
            for (j1=k+1; j1<=n-2-k; j1++)
                brow[j][j1]=brow[j][j1]-brow[j][k]*xk[j1+1]-brow[j][n-1-k]*
                    xnk[j1+1];
        }
    }
    if ((n%2) != 0) {
        if (midindex > k) {
            p1=(xk[n-k]/xk[k+1]);
            x2=(middle[n-1-k]-p1*middle[k])/(xnk[n-k] - p1 * xnk[k+1]);
            x1=(middle[k]-xnk[k+1]*x2)/xk[k+1];
            middle[n-1-k] = x2;
            middle[k] = x1;
            for (j1=k+1; j1<=n-2-k; j1++)
                middle[j1]=middle[j1]-middle[k]*xk[j1+1]-
                    middle[n-1-k]*xnk[j1+1];
        }
    }
    pack_elim+=(pack2-pack1)+(pack4-pack3);
}
pack_soln=0.0;
/* Inward Substitution */
ipointer = 0;
nextindex = aindex[ipointer];
for(k=0; k<=num; k++) {
    if (k == nextindex) {
        x1=arow[ipointer][n];
        x2 = arow[ipointer][n];
        pack5=secs();
        pvm_initsend(PvmDataDefault);
        pvm_pkfloat(&x1,1,1);
        pvm_pkfloat(&x2,1,1);
        pvm_mcast(others,nproc-1,150);
    }
}

```

```

        pack6=secs();
        cnt++;
        if (ipointer < numrows) {
            ipointer++;
            nextindex=aindex[ipointer];
        }
    }
    else {
        idle11=secs();
        pvm_recv(-1,150);
        idle12=secs();
        cnt++;
        pack7=secs();
        pvm_upkfloat(&x1,1,1);
        pvm_upkfloat(&x2,1,1);
        pack8=secs();
    }
    for (j=0; j<numrows; j++) {
        if (aindex[j]>k)
            arow[j][n]=arow[j][n]-arow[j][k]*x1-
                arow[j][n-1-k]*x2;
    }
    for (j=0; j<numrows; j++) {
        if (aindex[j]>k)
            brow[j][n]=brow[j][n]-brow[j][k]*x1-brow[j][n-1-k]*x2;
    }
    if ((n%2)!=0) {
        if (midindex > k)
            middle[n]=middle[n]-middle[k]*x1-middle[n-1-k]*x2;
    }
    pack_soln+=(pack6-pack5)+(pack8-pack7);
}
/* Bidirectional Substitution */
pack_soln1=0.0;
if (n % 2 !=0) {
    if (midindex!=0) {
        middle[n]=middle[n]/middle[n/2];
        x1=middle[n];
        pvm_initsend(PvmDataDefault);
        pvm_pkfloat(&x1,1,1);
        pvm_mcast(others,nproc-1,160);
    }
    else {
        pvm_recv(-1,160);
        pvm_upkfloat(&x1,1,1);
    }
    for (j=0; j<numrows; j++) {
        if (aindex[j] < n/2) {
            arow[j][n]=arow[j][n]-arow[j][n/2]*x1;
            brow[j][n]=brow[j][n]-brow[j][n/2]*x1;
        }
    }
}
/*even matrices */
ipointer=numrows-1;
nextindex=aindex[ipointer];
for (k=n/2-1; k>=0; k--) {
    if (k == nextindex) {

```



```

        p1=brow[ipointer][k]/arow[ipointer][k];
        x2=(brow[ipointer][n]-p1*arow[ipointer][n])/
            (brow[ipointer][n-1-k]-p1*arow[ipointer][n-1-k]);
        x1=(arow[ipointer][n]-arow[ipointer][n-1-k]*x2)/arow[ipointer][k];
        arow[ipointer][n]=x1;
        brow[ipointer][n]=x2;
        pack9=secs();
        pvm_initsend(PvmDataRow);
        pvm_pkfloat(&x1,1,1);
        pvm_pkfloat(&x2,1,1);
        pvm_mcast(others,nproc-1,200);
        pack10=secs();
        cnt++;
        if (ipointer > 0) {
            ipointer--;
            nextindex=aindex[ipointer];
        }
    }
    else {
        idle13=secs();
        pvm_recv(-1,200);
        idle14=secs();
        cnt++;
        pack11=secs();
        pvm_upkfloat(&x1,1,1);
        pvm_upkfloat(&x2,1,1);
        pack12=secs();
    }
    for (j=0; j<numrows; j++) {
        if (aindex[j]<k) {
            arow[j][n]=arow[j][n]-arow[j][k]*x1-arow[j][n-1-k]*x2;
            brow[j][n]=brow[j][n]-brow[j][k]*x1-brow[j][n-1-k]*x2;
        }
    }
    pack_soln1+=(pack10-pack9)+(pack12-pack11);
}

for (i=0; i<numrows; i++) {
    xk[i]=arow[i][n];
    xnk[i]=brow[i][n];
}
comp2=secs();
pack_t=pack_elim+pack_soln+pack_soln1;
comp_t=comp2-comp1-pack_t;
/* send results to master */
msgtype=5;
pvm_initsend(PvmDataDefault);
pvm_pkint(&numrows,1,1);
pvm_pkint(aindex,numrows,1);
pvm_pkint(bindex,numrows,1);
pvm_pkint(&midindex,1,1);
pvm_pkfloat(xk,numrows,1);
pvm_pkfloat(xnk,numrows,1);
pvm_pkfloat(&middle[n],1,1);
pvminfo=pvm_send(master,msgtype);
cnt++;
printf("Number of sends/receive is %d\n",cnt);
time2=secs();

```

```

printf("Time taken by wz_slave=%f\n",time2-time1);
tmp1=(idle2-idle1)+(idle4-idle3)+(idle6-idle5)+(idle8-idle7);
tmp2=(idle10-idle9)+(idle12-idle11)+(idle14-idle13);
printf("Finished.....\n");
printf("Time(minus idle time) taken by wz_slave=%f\n",(time2-time1)-tmp1-tmp2);
printf("Idle time of wz_slave= %f \n",tmp1+tmp2);
printf("Computation time of wz_slave= %f \n",comp_t);
printf("Communication time of wz_slave= %f \n",time2-time1-
tmp1-tmp2-comp_t);

if (pvminfo <0)
    printf("error in send \n");
pvm_barrier("workers",nproc);
pvm_lvgroup("workers");
pvm_exit();

}

#include <sys/time.h>
double secs()
{
    struct timeval ru;
    gettimeofday(&ru, (struct timezone *)0);
    return(ru.tv_sec + (((double)ru.tv_usec)/1000000));
}

```

