INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Parallel and fault tolerant algorithms for hypercube multiprocessors

Aykanat, Cevdet, Ph.D.
The Ohio State University, 1988
PARALLEL AND FAULT TOLERANT ALGORITHMS FOR HYPERCUBE MULTIPROCESSORS

A Dissertation
Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of the Ohio State University

by

Cevdet Aykanat, M.Sc.

The Ohio State University
1988

Dissertation Committee:
Professor Füsun Ö zgüner
Professor P. Sadayappan
Professor Stanley Ahalt

Approved by:
F. Özgüner
Adviser
Department of Electrical Engineering
To My Family.

Ahmet, Melek, and Esen
ACKNOWLEDGMENTS

I would like to acknowledge the encouragement and advice that I have received throughout my graduate studies from my adviser, Professor Füsun Özgüner. I am especially grateful for her support and friendship. I would also like to thank the members of the dissertation committee, P. Sadayappan and Stanley Ahalt, for their suggestions and comments.

I express my appreciation to the Fulbright Commission for giving me the opportunity to begin my graduate program at The Ohio State University.

Special thanks go to my research project colleagues, Dr. Sokka Doraivelu, Steven Martin, Win Bo, S. Mohanachandran, and Fikret Ercal for their cooperation and support.

My final appreciation and thanks go to my fellow graduate students in the Department of Electrical Engineering for their help and the good times.

This work was supported by the SBIR Program Phase II (F33615-85-C-5198) of Universal Energy Systems Inc. with the Air Force Materials Laboratory, WPAFB, Ohio 45433. Special thanks to Dr. Harold L. Gegel, senior scientist, and Mr. James C. Malas, project manager of this SBIR Program at the analytical process modeling WUD, Metals and Ceramics Division, WPAFB, Ohio 45433.

iii
VITA

1977 ............................................................ B.Sc. Electrical Engineering
The Middle East Technical University,
Ankara, Turkey
1977 — 1980 ............................................ Graduate Research Assistant,
Department of Electrical Engineering,
The Middle East Technical University,
Ankara, Turkey
1980 ............................................................ M.Sc. Electrical Engineering
The Middle East Technical University,
Ankara, Turkey
1980 — 1982 ............................................. Teaching Assistant,
Department of Electrical Engineering,
The Middle East Technical University,
Ankara, Turkey
1983 — 1986 ............................................. Recipient of a Fulbright Scholarship
for Graduate Studies at
The Ohio State University,
Columbus, Ohio
1987 ............................................................ Summer Research Associate,
INTEL Scientific Computers,
Portland, Oregon
1986 — 1988 ............................................. Graduate Research Associate,
Department of Electrical Engineering,
The Ohio State University,
Columbus, Ohio


FIELDS OF STUDY

Major Field:

Studies in Computer Engineering: Professor F. Ö zgüner
Professor K.J. Breeding

Minor Fields:

Studies in Computer and Information Science: Professor T.H. Lai
Studies in Control Systems: Professor H. Hemami
Studies in Power Systems: Professor A. Keyhani
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** iii  
**VITA** iv  
**LIST OF FIGURES** xi  
**LIST OF TABLES** xiv  

I. **Introduction** 1  
II. **Background** 8  
   2.1 Introduction ................................................. 8  
   2.2 The Finite Element Method .................................. 9  
   2.3 Parallel Architectures for FE Modeling ....................... 10  
   2.4 Techniques for Parallel Solution of Linear System of Equations 13  
      2.4.1 The Conjugate Gradient Algorithm ..................... 15  
      2.4.2 Scaled CG (SCG) Algorithm ............................. 16  
      2.4.3 Parallel SCG Algorithm ................................. 17  
      2.4.4 Mapping SCG Computations onto a Hypercube .......... 23  
   2.5 Fault Tolerance in Parallel Architectures .................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>III.</td>
<td>Coarse Grain Scaled Conjugate Gradient Algorithms for Distributed Memory Multiprocessors</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Data Dependencies in the SCG Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>General Formulation of a Coarse Grain SCG Algorithm</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Performance Analysis of the Algorithm</td>
<td>41</td>
</tr>
<tr>
<td>3.3.2</td>
<td>An s-Step Coarse Grain Partial SCG Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>s-Step Coarse Grain SCG Algorithms</td>
<td>44</td>
</tr>
<tr>
<td>3.4.1</td>
<td>General Formulation for s-Step CG–SCG Algorithms</td>
<td>44</td>
</tr>
<tr>
<td>3.4.2</td>
<td>A CG-SCG/s1 Algorithm</td>
<td>48</td>
</tr>
<tr>
<td>3.4.3</td>
<td>A CG-SCG/s2 Algorithm</td>
<td>51</td>
</tr>
<tr>
<td>IV.</td>
<td>Parallelization of the CG-SCG/s1 Algorithm on a Hypercube Multiprocessor</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Overlapping Communication with Computation</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>Overlapping Communications</td>
<td>61</td>
</tr>
<tr>
<td>4.3.1</td>
<td>A Distributed Exchange-Add Algorithm</td>
<td>61</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Communication Protocols for Strip Partitioning Schemes</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>Implementation of the CG-SCG/s1 Algorithm on the iPSC2/d4</td>
<td>71</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Distributed Sparse Matrix Vector Product</td>
<td>71</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Distributed Scaling</td>
<td>73</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Distributed Inner Products</td>
<td>73</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Distributed Vector Updates</td>
<td>74</td>
</tr>
<tr>
<td>4.5</td>
<td>Experimental Results and Discussions</td>
<td>74</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Comparison of the B–SCG and the CG–SCG Algorithms</td>
<td>75</td>
</tr>
</tbody>
</table>
4.5.2 Comparison of GS-GB and EA .................................................. 78
4.5.3 Experimental Performance .................................................... 84

V. Vectorization and Parallelization of the CG-SCG/s1 Algorithm
on a Vector Hypercube Multiprocessor ........................................... 92
5.1 Introduction ............................................................................... 92
5.2 Vectorization of the CG-SCG/s1 Algorithm on the iPSC-VX . 93
  5.2.1 Microprogramming Techniques on the iPSC-VX VP . 96
  5.2.2 Vectorization of the CG-SCG/s1 Computations on the VP . . 97
5.3 Parallel Implementation of the Vectorized CG-SCG on iPSC-
VX ...................................................................................................... 110
  5.3.1 A Communication Protocol for 1-D Strip Partitioning . 110
  5.3.2 An Efficient Local Ordering Scheme for 1-D Strip Parti-
tioning ......................................................................................... 114
  5.3.3 Overlapping Communication and Computation ......... 117
  5.3.4 Distributed Inner Product Computations ................. 118
5.4 Experimental Results ............................................................... 121
  5.4.1 Performance of the Microcoded Sparse Matrix Vector
      Product .................................................................................. 121
  5.4.2 Comparison of the B-SCG and the CG-SCG/s1 Algo-
      rithms .................................................................................... 122
  5.4.3 Comparison of Overlapped and Non-Overlapped Schemes 127
  5.4.4 Experimental Performance ............................................... 130

VI. A Concurrent Error Detecting SCG (CED SCG) Algorithm on
a Hypercube Multiprocessor .......................................................... 137
6.1 Introduction ................................................................. 137
6.2 Error Detection Conditions ........................................ 137
6.3 Distribution of Computation Steps for Error Detection  . 143
6.4 Time Redundancy for the CED SCG Algorithm ............. 147
6.5 Error Recovery ............................................................. 149
6.6 Graceful Degradation .................................................... 151
6.7 Implementation and Experimental Results .................... 155

VII. Reconfiguration Algorithms for a Hypercube Multiprocessor 161
7.1 Introduction ................................................................. 161
7.2 Finding the Maximum Dimension of the Fault-Free Subcubes . 162
  7.2.1 Faulty Processors ................................................... 163
  7.2.2 Faulty Links ........................................................... 171
  7.2.3 Faulty Subcubes ...................................................... 173
7.3 Finding the Maximum Dimensional Fault-Free Subcubes . . 179
  7.3.1 The Brute Force Approach ....................................... 179
  7.3.2 Missing Combinations Approach .............................. 181

VIII. Conclusion ................................................................. 185

A. Architectural Features of the iPSC-VX Vector Processor 189
  A.1 Multiple Functional Units ......................................... 189
  A.2 Multiple Interconnect Buses ....................................... 191
  A.3 Multioperation Microinstructions ............................... 192

REFERENCES ................................................................. 193
LIST OF FIGURES

1 4-dimensional Hypercube .............................................................. 4
2 Strip mapping of a FE mesh and the corresponding A–matrix onto a 2-dimensional hypercube ................................................. 11
3 Global Sum (GS) algorithm on a 4-dimensional hypercube .......... 20
4 Global Broadcast (GB) algorithm on a 4-dimensional hypercube ... 21
5 Strip mapping of a regular \( m \times m \) FE mesh onto \( P \) processors .. 28
6 Distributed Exchange-Add algorithm on a 4-dimensional hypercube 63
7 Communication protocol for 1-D strip partitioning on iPSC2/d3 ...... 66
8 Communication protocol for 2-D strip partitioning on iPSC2/d4 ...... 68
9 Communication protocol for 1.5-D strip partitioning on iPSC2/d4 ... 69
10 Percent performance improvement of \( CG-SCG/s1 \) over \( B-SCG \) ... 77
11 Performance of \( GS-GB \) and \( EA \) algorithms on iPSC2/d4 ....... 81
12 Percent performance improvement of \( EA \) over \( GS-GB \) in the \( CG-SCG/s1 \) Algorithm .................................................. 83
13 Percent performance improvement of \( CG-SCG/s1 (EA) \) over \( B-SCG (GS-GB) \) ................................................................. 85
14 Speed-up curves ........................................................................ 88
15 Efficiency curves ........................................................................ 89
<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Performance of the sequential $B$-$SCG$ ($\mu$VAX II, $iPSC2/d0$) and the parallel $CG$-$SCG/s1$ ($iPSC2/d1-d4$)</td>
</tr>
<tr>
<td>17</td>
<td>$iPSC$-$VX$ computational node</td>
</tr>
<tr>
<td>18</td>
<td>$iPSC$-$VX$ Vector Processor (VP) Architecture</td>
</tr>
<tr>
<td>19</td>
<td>The microcode ($S = 5$) in VP assembler for the inner loop of sparse matrix vector product</td>
</tr>
<tr>
<td>20</td>
<td>The microcode ($S = 4$) in VP assembler for the inner loop of sparse matrix vector product</td>
</tr>
<tr>
<td>21</td>
<td>Pipelined loop microcode structure for sparse matrix vector product</td>
</tr>
<tr>
<td>22</td>
<td>Communication protocol for 1-D strip partitioning on $iPSC/d3$</td>
</tr>
<tr>
<td>23</td>
<td>Vector scatter/gather operations required for communication in 1-D strip partitioning scheme</td>
</tr>
<tr>
<td>24</td>
<td>An efficient local ordering scheme for 1-D strip partitioning</td>
</tr>
<tr>
<td>25</td>
<td>Implementation of the Exchange-Add algorithm on the $iPSC/d4$</td>
</tr>
<tr>
<td>26</td>
<td>Performance of the microcoded sparse matrix vector product</td>
</tr>
<tr>
<td>27</td>
<td>Percent performance improvement of $CG$-$SCG/s1$ over $B$-$SCG$</td>
</tr>
<tr>
<td>28</td>
<td>Percent performance improvement obtained by overlapping communication with computation</td>
</tr>
<tr>
<td>29</td>
<td>Speed-up curves</td>
</tr>
<tr>
<td>30</td>
<td>Efficiency curves</td>
</tr>
<tr>
<td>31</td>
<td>Performance of the sequential $B$-$SCG$ ($\mu$VAX II, $iPSC$-$VX/d0$) and the parallel $CG$-$SCG/s1$ ($iPSC$-$VX/d1-d2$)</td>
</tr>
<tr>
<td>32</td>
<td>4-dimensional hypercube with faulty elements</td>
</tr>
<tr>
<td>33</td>
<td>Venn diagram showing the subcubes in $S_{F1}^2, S_{F2}^2, S_{F3}^2, S_{F4}^2$ and their intersections</td>
</tr>
</tbody>
</table>
LIST OF TABLES

1 Solution times (per iteration) of $B$-$SCG$ and $CG$-$SCG/s1$ algorithms with $GS$-$GB$ on $iPSC2/d4$ ........................................ 76

2 Distributed global sum computation times for $GS$-$GB$ and $EA$ algorithms ........................................................................................................... 79

3 Solution times (per iteration) of $B$-$SCG$ algorithm with $GS$-$GB$ and $CG$-$SCG/s1$ algorithm with $EA$ on $iPSC2/d4$ .......................... 82

4 Solution times (per iteration) of the sequential $B$-$SCG$ ($\mu$VAX II, $iPSC2/d0$), and the parallel $CG$-$SCG/s1$ ($iPSC2/d1\cdot d4$) ............. 87

5 Performance of the sequential $B$-$SCG$ ($\mu$VAX II, $iPSC2/d0$) and the parallel $CG$-$SCG/s1$ ($iPSC2/d1\cdot d4$) ........................................ 90

6 Measured performance of the microcoded sparse matrix vector product .......................... 123

7 Solution times (per iteration) of $B$-$SCG$ and $CG$-$SCG/s1$ algorithms on $iPSC$-$VX/d1\cdot d2$ ......................................................... 125

8 Solution times (per iteration) of overlapped and non-overlapped $CG$-$SCG/s1$ algorithms on $iPSC$-$VX/d1\cdot d2$ ........................ 128

9 Solution times (per iteration) for the $B$-$SCG$ ($\mu$VAX II, $iPSC$-$VX/d0$), and the $CG$-$SCG/s1$ ($iPSC2/d1\cdot d4$) ............................. 131

10 Performance for the $B$-$SCG$ ($\mu$VAX II, $iPSC$-$VX/d0$), and the $CG$-$SCG/s1$ ($iPSC$-$VX/d0\cdot d2$) .................................................. 134
Experimental results showing error detection by the CED SCG algorithm
CHAPTER I

Introduction

Solution of many scientific and engineering problems require large amounts of computing power. With advances in VLSI and parallel processing technology, it is now feasible to achieve high performance and even reach interactive or real-time speeds in solving complex problems. The drastic reduction in hardware costs has made parallel computers available to many users at affordable prices. However, in order to use these general purpose computers in a specific application, algorithms need to be developed and existing algorithms restructured for the architecture. The Finite Element method is a powerful numerical technique for solving boundary value problems involving partial differential equations, in engineering fields such as heat flow analysis, metalforming, fluid dynamics, and many others [1]. As a result of finite element discretization, linear equations of the form $Ax = b$ are obtained where $A$ is large, sparse, and banded with proper ordering of the variables $x$. Computational power demands of the solution of these equations cannot be met satisfactorily by conventional sequential computers and thus parallelism must be exploited. The problem has been recognized and addressed by other researchers [1,2,3,4]. Two approaches have been taken. NASA–Langley has designed and built a special–purpose Finite Element Machine (FEM) [5]. Others have investigated the implementation of Finite Element algorithms on existing or proposed general purpose multiprocessors: CHiP [6], Cedar [7], Butterfly [8], and Cosmic
Cube [1]. Distributed memory multiprocessors implementing mesh or hypercube topologies are suitable for these problems, as a regular domain can be mapped to these topologies requiring only nearest neighbor communication [1,9]. However, a closer look at message-passing multiprocessors reveals that speed-up cannot be easily achieved because of the communication overhead.

Methods for solving such equations on sequential computers can be classified as direct methods [10] and iterative methods [11]. Considerable amount of research has been done in recent years on the parallel solution of sparse linear systems of equations on message-passing and vector computers especially for the solution of the finite element problems [1,2,4,12]. Both direct and iterative methods can be parallelized, although the parallelization of iterative methods is more efficient.

Since the coefficient matrix A is very large in these applications, parallelization by distributing both data and computations has been of interest. The Scaled Conjugate Gradient (SCG) algorithm is an iterative method for solving sparse matrix equations and is being widely used because of its convergence properties. The sparsity of the matrix is preserved throughout the iterations and the SCG algorithm can be efficiently parallelized on distributed memory multiprocessors [1].

In this dissertation, solution of such equations on distributed memory message-passing multiprocessors implementing the hypercube topology [13,14], is addressed. In such an architecture, communication and coordination between processors is achieved through the exchange of messages. A d-dimensional hypercube consists of \( P = 2^d \) processors (nodes) with each processor being directly connected to \( d \) other processors [14]. A 4—dimensional hypercube, with binary encoding of the nodes is shown in Fig. 1. The binary encoding of a processor differs from that of its neighbors in one bit. Processors that are not directly connected can communicate through other processors by software or hardware routing. The maximum
distance between any two processors in a $d$-dimensional hypercube is $d$ [14]. It has been shown that many other topologies such as meshes, trees and rings can be embedded in a hypercube [14,15]. There are already several commercially available hypercube multiprocessors: Intel's iPSC series, Ncube's machine, Ametek's Hypernet and FPS's T series. Some vendors have recently added vectorization capabilities to their respective products (Intel's iPSC-VX and FPS's T series). These architectures combine two techniques for high speed computing: processor level parallelism based on hypercube interconnection, and low level parallelism based on pipelined vector arithmetic.

Achieving speed-up through parallelism on such an architecture is not straightforward. The algorithm must be designed so that both computations and data can be distributed to the processors with local memories in a such a way that computational tasks can be run in parallel, balancing the computational loads of the processors as much as possible [16]. Communication between processors in order to exchange data must also be considered as part of the algorithm design. One important factor in designing parallel algorithms is granularity [17]. Granularity depends on both the application and the parallel machine. The commercially available hypercube multiprocessors (Intel’s iPSC series, Ncube, and Ametek’s Hypernet) are designed for medium-to-coarse grain parallelism. In a parallel machine with a high communication latency, the algorithm designer must structure the algorithm so that large amounts of computation are done between communication steps. Another factor affecting parallel algorithms, is the ability of the parallel system to overlap communication with computation. An efficient parallel algorithm should also exploit the topology and the node-to-node communication architecture of a parallel machine to overlap individual inter-processor communications.

The SCG method requires the computation and broadcast of two inner prod-
Figure 1: 4-dimensional Hypercube
uct terms at each iteration step. These calculations require global information and introduce a large amount of inter-processor communication overhead, due to the large set-up time for communication. In this dissertation, coarse grain parallel SCG algorithms are developed by restructuring the computations of the SCG algorithms so that more than one inner product can be computed and broadcasted in one global communication step. The distributed communication protocols given in this dissertation exploit the hypercube topology and the node-to-node communication architecture of the system to overlap individual inter-processor communications. A local ordering scheme given for Finite Element problems prevents the indexing overhead required to align local data for communication. This local ordering scheme also achieves the overlap of local communications with computations. Computations in the SCG algorithm consist mainly of matrix operations that can be vectorized efficiently. An implementation of the SCG algorithm, for Finite Element problems, which exploits both vectorization and parallelization on a vector hypercube multiprocessor, is also described.

Fault tolerance is important in a parallel processing environment in order to ensure the correctness of the distributed computations and also for continued operation after a failure of one of the processing elements. Numerous techniques for on-line detection and correction of failures in digital computer systems have been proposed and implemented in the last two decades [18,19,20,21,22,23,24]. The degree and method of fault tolerance is highly dependent on the requirements of the application. However, the most efficient use of processing power can be attained by considering fault tolerance and parallelism at the same time. In this dissertation, the goal has been to achieve parallelism and fault tolerance on a commercially available architecture rather than by designing special purpose hardware. A low cost concurrent error detecting SCG algorithm is developed and implemented on a
hypercube multiprocessor. Concurrent error detection is achieved by checking the algorithm-specific properties at the end of each SCG iteration.

Another important factor in a real time fault tolerant parallel architecture is the reconfiguration capability of the system after the location of the faulty elements [25,26,27,28]. Most of the parallel algorithms can be formulated with the dimension(s) of the topology of the parallel architecture being the parameter of the algorithm. Hence, the reconfiguration should preserve the original parallel computer topology by utilizing the hardware redundancy. Thus, the original parallel algorithm mapping is preserved. In this dissertation, the issue of reconfiguration for fault tolerance is addressed for a commercially available parallel architecture which has no dynamic redundancy. Thus, the parallel system can be reconfigured to operate without faulty units in a degraded mode using these algorithms.

The dissertation consists of eight chapters and one appendix. Chapter II reviews the Finite Element method, and evaluates the parallel architectures and parallel solution techniques for the solution of Finite Element problems. A brief survey of on-line fault tolerance techniques for parallel architectures concludes this chapter.

Chapter III presents the formulation of coarse grain SCG algorithms with reduced inner product dependencies. In these algorithms, the basic SCG algorithm is restructured in order to increase the number of inner products which can be accumulated in a single global sum communication step. In Chapter IV, the implementation of a coarse grain SCG (CG–SCG/sl) algorithm on a 16 node Intel 386-based iPSC2/d4 hypercube and experimental results for the Finite Element simulation of metalforming problems are discussed. Communication protocols that overlap individual inter-processor communications are also presented.

In Chapter V, the vectorization and the parallel implementation of CG–
SCG/s1 algorithm on a 4 node Intel iPSC-VX/d2 vector hypercube and experimental results are discussed. A local ordering scheme presented in this chapter avoids the indexing overhead introduced by the communication requirement and achieves the overlap of communication with computation during the distributed sparse matrix vector product computations.

Chapter VI, presents a low redundancy concurrent error detecting parallel SCG algorithm. Error detection is achieved by checking algorithm–specific properties such as orthogonality of certain vectors at the end of each SCG iteration as described in this chapter. The CED SCG algorithm is implemented on a 32 node Intel 286–based iPSC hypercube.

In Chapter VII, reconfiguration algorithms for a hypercube multiprocessor in the presence of faulty processors and faulty links are presented. The reconfiguration problem for a hypercube multiprocessor is defined as finding the maximum dimensional fault–free subcube(s).

Finally, conclusions are presented in Chapter VIII.
CHAPTER II

Background

2.1 Introduction

The solution of large, sparse, linear systems of equations of the form $Ax = b$ constitutes a significant amount of computations in simulation of physical phenomena [1,4,29]. Finite Element (FE) simulation problems require the repeated generation and solution of such matrix equations, as described in Section 2.2. The hypercube topology is well suited for FE modeling problems, as discussed in Section 2.3. Section 2.4 reevaluates the algorithms for solving linear systems of equations with respect to parallel architectures. The Conjugate Gradient (CG) method is particularly suitable for solving large sparse linear systems of equations, arising in FE analysis, on parallel architectures. The computational steps of the CG algorithm and the preconditioning by diagonal scaling are described in Sections 2.4.1 and 2.4.2 respectively. The scaled CG (SCG) algorithm can be efficiently parallelized on a hypercube multiprocessor, as described in Section 2.4.3. Section 2.4.4 discusses the distribution of the SCG computations amongst the processors of the hypercube. A brief survey of on-line fault tolerance techniques for parallel architectures is given in Section 2.5.
2.2 The Finite Element Method

The \textit{FE} method is an important technique for constructing approximate solutions to boundary value problems \cite{30,31}. \textit{FE} method has been used to solve problems in diverse fields as fluid dynamics, metalforming, heat flow analysis, structural analysis and many others \cite{1}. In \textit{FE} analysis, a continuous structure is subdivided into elements connected at nodes \cite{32,33} as shown in Fig. 2(a). The field variables are expressed in terms of assumed approximating functions (\textit{basis} or \textit{interpolating} functions) within each element. The response of the complete structure to the boundary (loading) conditions is viewed as the calculation of the responses of the individual elements. The responses of the elements are formulated by elemental stiffness equations. The elemental stiffness equations for each element are determined by integrating the basis functions over each element. The contributions of elemental stiffness equations are assembled into a single system

\[
A x = b .
\]  

Here, the \textit{b}–vector represents the boundary (loading) conditions, and the \textit{x}–vector represents the unknown field variables. Such matrix equations are repeatedly formed and solved forward in time starting from an initial condition or state in order to simulate the response of the structure in space and time for the given loading conditions. A complete description of the \textit{FE} method may be found in a number of references \cite{30,31}.

Fig. 2(b) displays the structure of a sparse matrix resulting from the \textit{FE} discretization of a simple rectangular region shown in Fig. 2(a). The discretization uses four–node rectilinear elements. In Fig. 2(a), the diagonals of the finite–elements are joined by edges to give a \textit{FE} interaction-graph, whose structure bears a direct relation to the zero-nonzero structure of the sparse system of equations that
characterizes the discretization. Each node in a 2-dimensional (2-D) FE graph is associated with 2 variables, corresponding to two degrees of freedom. Each nodal degree of freedom has a corresponding row in the matrix A and is associated with a component in the x-vector and b-vector. Furthermore, it can be seen that the nonzeros in that row (column) of A occur only in positions corresponding to FE nodes directly connected to that node in Fig. 2(a). The figure shows only a single point corresponding to the two degrees of freedom of a node.

In general, elemental stiffness matrix generation and the solution of the global stiffness equations are the computationally intensive components of FE simulation [32,33]. The global stiffness matrix is merely a superposition of the individual elemental stiffness matrices [32,33]. Thus, the elemental stiffness matrices can be constructed independently and therefore in parallel. The contributions of the elemental stiffness matrices can be also added completely in parallel to generate the global stiffness matrix A. Hence, efficient parallelization of the solution phase is crucial to achieve speed-up in the overall FE simulation.

2.3 Parallel Architectures for FE Modeling

A survey for the various parallel computer architectures that are commercially available and the suitability of each for FE simulation problems is given in [9]. Broadly, parallel architectures are classified as SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) architectures. A further classification of MIMD architectures according to the inter-processor communication mechanism is given as: bus based, indirect multistage interconnection network (MIN) based, and direct interconnection (DIN) based architectures. SIMD architectures are not suitable due to the complex operations during the stiffness matrix generation phase. The advantage of the bus-based MIMD approach over
Figure 2: Strip mapping of (a) $FE$ domain and (b) the corresponding $A$-matrix onto a 2-dimensional hypercube
the distributed memory approach is that communication between processors is very fast in the former. The disadvantage is that only one access to the global shared memory is possible at any time. As will be discussed in Section 2.4.4, the solution phase of 2-D FE problems can be viewed as a collection of pairs of processes communicating with each other. The performance of such architectures degrades drastically as the number of processors increases due to the global bus contention. In MIN architectures, simultaneous memory accesses by several processors are possible by use of a high bandwidth multi-stage interconnection network (IN). However, the latency time for a memory access is longer compared to the bus-based approach. This is due to the need to traverse multiple stages of the IN. MIN based architectures require very fast IN switching elements that are quite expensive.

In DIN based MIMD architectures, processors only have local memory and inter-processor communication is by explicit message passing. Each processor is connected to a small, fixed number of processors in some regular geometry. The commercially available hypercube multiprocessors (Intel's iPSC series, Ncube) are typical examples of such architectures. The advantages of such architectures over the other MIMD architectures are:

1. they provide a larger communication bandwidth in the system due to the large number of simultaneous communications possible on the independent inter-processor links,

2. algorithms are scalable when complete locality of communication is achievable

The first feature makes it possible for many pairs of processors to communicate concurrently with each other. As will be discussed in Section 2.4.4, the solution
phase of the $FE$ problems can be viewed as a collection of processes, regularly arranged on a $1-D$ or $2-D$ grid requiring to communicate with neighbors only. The hypercube topology is a superset of $1-D$ and $2-D$ array topologies [14,15]. Hence, it can be concluded that hypercube architectures are well suited for $FE$ simulation problems.

2.4 Techniques for the Solution of Linear System of Equations

Methods for solving linear systems of equations on sequential computers can be classified as

1. Direct methods such as Gaussian Elimination, and Cholesky Factorization[10].

2. Iterative methods such as Jacobi, Gauss-Seidel, and Conjugate Gradient [11].

With the introduction of a parallel architecture, all previous algorithmic decisions need to be reexamined, since the best sequential algorithm may not perform well on a parallel architecture. Algorithms that were dismissed as inferior may turn out to be superior on a parallel architecture. Entirely new algorithms may even be better than any now known.

Direct methods introduce fill-in terms during elimination of sparse matrices. Existing parallel direct methods [4,34] for solving $FE$ problems try to minimize the number of inter-processor communications by partitioning the $FE$ graph into blocks that are isolated from each other by separators. These blocks can be eliminated independently and therefore in parallel. Communication is necessary during the computation of the contributions of the domain eliminations to the separators and during the parallel elimination of the resulting dense matrices corresponding to the separators. Nested dissection [35,36] techniques are used to find the minimal separators. In general, the computational cost of the Gaussian Elimina-
tion is proportional to $Nw^2$, where $N$ is the total number of degrees of freedom and $w$ is the main diagonal bandwidth of non-zero entries in matrix A. Although proper FE node numbering can minimize $w$, FE grids in higher dimensions ($3-D$) will unavoidably have very large $w$. However, these schemes remain as viable approaches for the solution of $2-D$ FE simulation problems on distributed memory multiprocessors.

Iterative methods have the property of preserving the original sparsity of the coefficient matrix A during the solution in contrast to direct methods. Thus, iterative methods parallelize better than direct methods and are therefore good candidates for solving large sparse linear systems of equations on parallel machines. Another advantage of iterative methods is that the process can converge in very few steps if a good initial guess is known. The traditional iterative methods are Jacobi ($J$), Jacobi-Overrelaxation $JOR$, Gauss-Seidel ($GS$), and Successive-Overrelaxation $SOR$. The $J$ and $JOR$ are very simple methods and can be parallelized efficiently. Many physical problems give rise to symmetric and positive definite coefficient matrices [1]. However, the $J$ and $JOR$ method are not guaranteed to converge for all symmetric and positive definite matrices [11].

The $GS$ and $SOR$ methods are also simple to implement and have a greater convergence rate than the $J$ and $JOR$ methods [9]. The $GS$ and $SOR$ methods (with a properly chosen relaxation parameter) are guaranteed to converge for all symmetric and positive definite matrices [11]. These methods are sequential in nature. However, the operations can be ordered according to the multi-color partitioning of the grid points such that one $SOR$ sweep can be implemented by a number of Jacobi sweeps, one for each set of equations associated with a given color [2,32]. Thus, $GS$ and $SOR$ algorithms can be effectively implemented on parallel computers. However, the granularity of the problem is reduced by a factor
equal to the number of colors required for partitioning. In general, the number of colors necessary will depend on the connectivity of the grid points. For example, four colors are needed for rectangular finite elements. This implies that four Jacobi sweeps are needed to complete one iteration of SOR.

The symmetric and positive definite nature of the coefficient matrices arising in FE analysis makes the CG method a good choice for implementation. In general, the CG method is superior to the classical iterative methods such as J, JOR, GS and SOR [32,37].

Iterative methods can be vectorized more effectively than direct methods since they consist mainly of repeated matrix operations. In GS and SOR methods, the effective vector lengths are reduced by a factor of the number of colors required for partitioning. The CG method is very suitable for implementation on vector hypercube architectures with nodes having vector processing capability.

2.4.1 The Conjugate Gradient Algorithm

The CG method is an optimization technique, iteratively searching the space of vectors \( \mathbf{x} \) in such a way to minimize an objective function

\[
\begin{align*}
  f(\mathbf{x}) &= \frac{1}{2} < \mathbf{x}, A\mathbf{x} > - < \mathbf{b}, \mathbf{x} > \\
  \mathbf{x} &= \begin{bmatrix} x_1, \ldots, x_N \end{bmatrix}^t
\end{align*}
\]  

(2.2)

where \( \mathbf{x} = [x_1, \ldots, x_N]^t \) and \( f : \mathbb{R}^N \to \mathbb{R} \). If the coefficient matrix \( A \) is positive definite, the objective function defined above is a convex function and has a global minimum where its gradient vector vanishes[38]. That is,

\[
\nabla f(\mathbf{x}) = A\mathbf{x} - \mathbf{b} = 0
\]  

(2.3)

which is also the solution to \( A\mathbf{x} = \mathbf{b} \). Here, \( A \) is a sparse, symmetric, and positive definite coefficient matrix of order \( N \) by \( N \) and \( \mathbf{x} \) and \( \mathbf{b} \) are the vectors of the unknown variables and right-hand sides respectively. The CG algorithm seeks this
global minimum by finding in turn the local minima along a series of lines, the directions of which are given by vectors \( p_0, p_1, p_2, \ldots \) in \( N \)-dimensional space[37].

The basic steps of the CG algorithm can be given as follows:

Initially, choose \( x_0 \) and let \( r_0 = p_0 = b - Ax_0 \), and then compute \( < r_0, r_0 > \).

Then,

for \( k = 0, 1, 2, \ldots \)

1. form \( q_k = Ap_k \)
2. a. form \( < p_k, q_k > \)
   b. \( \alpha_k = \frac{< r_k, r_k >}{< p_k, Ap_k >} \)
3. \( r_{k+1} = r_k - \alpha_k Ap_k \)
4. \( x_{k+1} = x_k + \alpha_k p_k \)
5. a. form \( < r_{k+1}, r_{k+1} > \)
   b. \( \beta_k = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >} \)
6. \( p_{k+1} = r_{k+1} + \beta_k p_k \) \hspace{1cm} (2.4)

Here, \( r_k \) is the residual error associated with the trial vector \( x_k \). That is

\[
    r_k = -\nabla f(x) = b - Ax_k \hspace{1cm} (2.5)
\]

which must be null when \( x_k \) is coincident with the solution vector. \( p_k \) is the direction vector at the \( k \)-th iteration. A suitable criterion for halting the iterations is \( [< r_k, r_k > / < b, b >]^{\frac{1}{2}} < \) tolerance.

2.4.2 Scaled CG (SCG) Algorithm

The convergence rate of the CG algorithm is improved if the rows and columns of matrix \( A \) are individually scaled by its diagonal, \( D = \text{diag}[a_{11}, a_{22}, \ldots, a_{NN}] \)[37]. Hence,
\[ \tilde{A}\tilde{x} = \tilde{b} \]  
(2.6)

where,

\[ \tilde{A} = D^{-1/2}AD^{-1/2} \]  
(2.7a)

\[ \tilde{x} = D^{1/2}x \]  
(2.7b)

\[ \tilde{b} = D^{-1/2}b \]  
(2.7c)

Note that, \( \tilde{A} \) will have unit diagonal entries. The eigenvalues of the scaled matrix \( \tilde{A} \) are more likely to be grouped together than those of the unscaled matrix \( A \), thus resulting in a better condition number[37]. Hence, in the SCG algorithm, the CG method is applied to Equation (2.6) obtained after scaling. The scaling process during the initialization phase requires only \( \sim 2zN \) multiplications, where \( z \) is the average number of nonzero entries per row of the \( N \times N \) matrix. Numerical results show that symmetric scaling increases the convergence rate of the basic CG algorithm by approximately 50% for a wide range of sample problems.

2.4.3 Parallel SCG Algorithm

As it is seen from Equation (2.4), the CG algorithm has three basic kind of operations [1]:

1. matrix vector product \( Ap_k \)
2. inner products \( <r_{k+1},r_{k+1}> \) and \( <p_k,Ap_k> \)

Distributing these computations among the processors of a distributed memory multiprocessor may be understood with reference to Fig. 2. The rows of the sparse coefficient matrix \( A \) and the vectors \( x, b, r, p, q \) are shown partitioned and
assigned to the processors of a 2-dimensional hypercube. For the sake of simplicity, assume that the total number of variables (degrees of freedom) is a perfect multiple of the number of processors $P$, and each processor is assigned $n = \frac{N}{P}$ variables. Let $M_\ell = \{M_1^\ell, \ldots, M_P^\ell\}$ denote the index set for the rows of the distributed sparse $A$ matrix and for the components of the distributed vectors mapped to processor $P_\ell$. Let $E_i$ denote the set of column indices for the nonzero entries in row $i$ of the coefficient matrix $A$. That is, $E_i = \{j \mid a^i_j \neq 0\}$. Let $E_{P_\ell}$ denote the set of column indices for the nonzero entries in the rows of the matrix $A$ mapped to processor $P_\ell$. That is, $E_{P_\ell} = \{j \mid a^i_j \neq 0 \text{ for } i \in M_\ell\}$. For example, considering the partitioning given in Fig. 2(b), the index sets for processor $P_1$ are $M_1 = \{17, 18, \ldots, 32\}$ and $E_{P_1} = \{13, 14, \ldots, 36\}$. Processor $P_\ell$ holds the nonzero entries $\{a^i_j\}_{i \in M_\ell, j \in E_i}$ of the coefficient matrix $A$ and the components $\{b^i\}_{i \in M_\ell}$ of the right hand side vector $b$ for $\ell = 0, 1, \ldots, P - 1$. Also, processor $P_\ell$ holds and is responsible from updating its local components $\{x^i\}_{i \in M_\ell}$ of the distributed $x$-vector, $\{r^i\}_{i \in M_\ell}$ of the distributed $r$-vector, $\{p^i\}_{i \in M_\ell}$ of the distributed $p$-vector, and $\{q^i\}_{i \in M_\ell}$ of the distributed $q$-vector, for $\ell = 0, 1, \ldots, P - 1$.

In order to perform the distributed sparse matrix vector product $q = Ap$, required in step 1, each processor $P_\ell$ for $\ell = 0, 1, \ldots, P - 1$ in the hypercube computes concurrently its local components of the distributed $q$-vector by calculating

$$q^i = \sum_{j \in E_i} a^{i,j} p^j = \sum_{j \in E_i, j \in M_\ell} a^{i,j} p^j + \sum_{j \in E_i, j \notin M_\ell} a^{i,j} p^j \quad (2.8)$$

for $i \in M_\ell$. The first summation term in this equation involves no communication since processor $P_\ell$ contains the pair of elements to be multiplied. However, the multiplications inside the second summation term require communication for the
non-local components \( \{ p_j \}_{j \in E_{P_{\ell}}, j \notin M_{\ell}} \) of the distributed \( p \)-vector. In general, the protocol for this communication step strongly depends on the partitioning scheme employed and will be discussed in Section 2.4.4. After this communication step, the multiplications inside the second summation term and the summation of these product terms can be performed concurrently.

Inner product terms \( < p, q > \) and \( < r, r > \) that should be computed in steps 2.a and 5.a respectively can be written as

\[
< p, q > = \sum_{i=1}^{N} p_i q^i = \sum_{\ell=0}^{P-1} \sum_{i \in M_{\ell}} p_i q^i \quad (2.9a)
\]

\[
< r, r > = \sum_{i=1}^{N} r_i r^i = \sum_{\ell=0}^{P-1} \sum_{i \in M_{\ell}} r_i r^i \quad (2.9b)
\]

As it is seen from these equations each processor can multiply its pair of components \( p^i \) and \( q^i \) (\( r^i \) and \( r^i \)) concurrently with all other processors. The summation over the index \( \ell \) adds the partial results from each processor to obtain the final result and requires global communication. A global sum algorithm (GS)[15], for a \( d = 4 \) dimensional hypercube is given in Figure 3. The computations and nearest neighbor communications (shown by solid lines) at the same step of the algorithm are performed concurrently. Hence, the distributed GS operation requires \( d \) concurrent communication steps. At the end of this algorithm, the root processor \( (P_0) \) calculates and broadcasts the updated values for the global scalars variables \( \alpha \) or \( \beta \) to all the other processors by using the global broadcast (GB) algorithm [39,40] shown in Figure 4. After \( d \) concurrent nearest neighbor communication steps of the GB, each processor receives the updated scalars \( \alpha \) and \( \beta \).

The processors of the hypercube can perform the vector updates in Steps 3, 4, and 6 only after receiving the updated scalar values. These equations can be
Figure 3: Global Sum (GS) algorithm on a 4-dimensional hypercube
Figure 4: Global Broadcast (GB) algorithm on a 4-dimensional hypercube
written as

\[ r_{k+1}^i = r_k^i - \alpha_k q_k^i \quad \text{for } i \in M_\ell \quad (2.10a) \]
\[ x_{k+1}^i = x_k^i + \alpha_k p_k^i \quad \text{for } i \in M_\ell \quad (2.10b) \]
\[ p_{k+1}^i = r_{k+1}^i + \beta_k p_k^i \quad \text{for } i \in M_\ell \quad (2.10c) \]

for \( \ell = 0, 1, \ldots, P - 1 \). These computation steps involve no communication. Each processor can perform the scalar vector multiplication and the vector addition steps given in the above equations concurrently with all the other processors. Thus, steps of the concurrent \( SCG \) algorithm after the initialization phase can be given as follows:

for \( k = 0, 1, 2, \ldots \)

1. a. \( \text{processor} P_\ell \) gets \( \{p_{k}^{j}\}_{j \in E_{P_{\ell}}, j \neq M_\ell} \) from neighbor processors.
   b. \( \text{processor} P_\ell \) for \( \ell = 0,1,\ldots,P - 1 \) computes
      \[ q_k^i = \sum_{j \in E_{P_\ell}, j \neq M_\ell} a_{i,j} p_k^j + \sum_{j \in E_{P_\ell}, j \neq M_\ell} a_{i,j} p_k^j \quad \text{for } i \in M_\ell. \]
2. a. \( \text{processor} P_\ell \) for \( \ell = 0,1,\ldots,P - 1 \) computes
      \[ R_\ell = \sum_{i \in M_\ell} p_k^i q_k^i \quad \text{for } i \in M_\ell. \]
   b. \( < p_k, A p_k > = \sum_{i=0}^{P-1} R_\ell \) is accumulated in \( P_0 \) using GS.
   c. \( \text{processor} P_0 \) computes \( \alpha_k = \frac{< r_k, r_k >}{< p_k, A p_k >} \)
   d. \( \text{processor} P_0 \) broadcasts the scalar \( \alpha_k \) using GB.
3. \( \text{processor} P_\ell \) for \( \ell = 0,1,\ldots,P - 1 \) computes
   \[ r_{k+1}^i = r_k^i - \alpha_k q_k^i \quad \text{for } i \in M_\ell \]
4. \( \text{processor} P_\ell \) for \( \ell = 0,1,\ldots,P - 1 \) computes
   \[ x_{k+1}^i = x_k^i + \alpha_k p_k^i \quad \text{for } i \in M_\ell \]
5. a. \( \text{processor} P_\ell \) for \( \ell = 0,1,\ldots,P - 1 \) computes
      \[ R_\ell = \sum_{i \in M_\ell} r_{k+1}^i r_{k+1}^i \quad \text{for } i \in M_\ell. \]
   b. \( < r_{k+1}, r_{k+1} > = \sum_{i=0}^{P-1} R_\ell \) is accumulated in \( P_0 \) using GS.
   c. \( \text{processor} P_0 \) computes \( \beta_k = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >} \)
   d. \( \text{processor} P_0 \) broadcasts the scalar \( \beta_k \) using GB.
6. \( \text{processor} P_\ell \) for \( \ell = 0,1,\ldots,P - 1 \) computes
   \[ p_{k+1}^i = r_{k+1}^i + \beta_k p_k^i \quad \text{for } i \in M_\ell \]
2.4.4 Mapping SCG Computations onto a Hypercube

The effective parallel implementation of the SCG algorithm on a hypercube parallel computer requires the partitioning and mapping of the computation among the processors in a manner that results in low inter-processor communication overhead. Sadayappan and Ercal's work [41,42] will be summarized in this section for the sake of completeness.

Considering the distributed computations and communications required in steps 2–6, if the rows of $A$ are evenly distributed among the processors, each processor will perform exactly the same amount of computation and communication per iteration of the SCG algorithm. This however is not the case for the distributed sparse matrix vector product in step 1. As mentioned earlier, during distributed sparse matrix vector product computation inter-processor communication is necessary to obtain non-local components of the distributed $p$-vector. Due to the relation between the non-zero structure of $A$ and the interconnection structure of the $FE$ interaction–graph, the inter-processor communication required is more easily seen from Fig. 1(a) than directly from Fig. 1(b). Two processors need to communicate if any node mapped onto one of them shares an edge with any node mapped onto the other. Thus, the inter-processor communication incurred with a given partitioning of the matrix $A$ and the vectors $x, b, r, p, q$ can be determined by looking at the edges of the $FE$ interaction graph that go across between processors.

The time taken to perform an inter-processor communication on the commercially available hypercube multiprocessors is an additive sum of two components:

1. set-up cost, $S_C$, that is relatively independent of the size of the message transmitted
2. transmission cost, $T_C$, that is linearly proportional to the number of bytes $\ell$ transmitted.

Thus,

$$T_{comm} = S_C + \ell \times T_C.$$  

(2.11)

The two bounding cases to be examined are when the communication time is dominated by either the set-up time or the transmission time. If $S_C$ is dominating it is then desirable to minimize the number of communications per processor. If $\ell T_C$ is dominant then it is preferable to minimize the volume of data to be communicated. In the present implementations of the hypercube (Intel's iPSC and iPSC2 and Ncube), the communications are best approximated by the set-up domination model. Since an additional set-up cost has to be paid for each processor communicated with, in attempting a mapping that minimizes communication costs, it is important to minimize not only the total number of bytes communicated, but also the number of distinct processors communicated with. The first of the two mapping schemes described, the 1-D strip-mapping approach [41,42], minimizes the number of processors that each processor needs to communicate to, while simultaneously keeping the volume of communication moderately low. The second scheme, the 2-D mapping approach [41,42], lowers the volume of communication, but requires more processor pairs to communicate.

### 1-D Strip Partitioning

The 1-D strip-mapping scheme attempts to partition the $FE$ graph into strips, in such a way that the nodes in any strip are connected to nodes only in the two neighbor strips. By assigning a strip-partition to each processor, the maximum number of processors that any processor will need to communicate with is limited
to two. For the sake of simplicity, assume that the total number of FE nodes \( N \) is a perfect multiple of the number of processors. A load-balanced mapping of the mesh onto a \( d \)-dimensional hypercube with \( P \) processors is therefore should assign \( n = \frac{N}{P} \) FE nodes per processor. The hypercube multiprocessor is viewed as a linear array of processors. Starting at the top of the leftmost column of the mesh, nodes in that column are counted off, till \( n \) FE nodes are visited. If the number of nodes in the leftmost column is less than \( n \), the column immediately to its right is next visited, starting again at the top. By so scanning columns from left to right, \( n \) FE nodes are picked off and assigned to the first processor \( P_0 \) in the linear array. Continuing similarly, another strip of \( n \) FE nodes is formed and assigned to the next processor \( P_1 \) in the linear array. The next processors in the linear array are assigned the next such strips. Thus, by only using a subset of the links of the hypercube, a linear array of the processors is formed and adjacent strips generated by the strip mapping are allocated to adjacent processors in the linear array. If the FE mesh is large enough, such a load-balanced 1-D strip mapping is generally feasible [41,42].

2-D Orthogonal Strip Partitioning

The partitions produced by 1-D strip mapping tend to require a relatively high volume of communication between processors due to the narrow but long shape of typical strips. The 2-D orthogonal-partitioning method attempts to create partitions with a smaller number of boundary nodes, thereby reducing the volume of communication required. It involves the generation of two orthogonal 1-D strips. The hypercube parallel computer is now viewed as a \( P_H \times P_V \) 2-D processor array. A \( P_H \)-way 1-D strip and a \( P_V \)-way 1-D strip in two orthogonal directions are generated. Partitions are now formed from the intersection regions
of the strips from the two orthogonal 1-D strips, and can be expected to be more "square" (and consequently have a lower perimeter/area) than those generated by a \( P = P_H \times P_V \)-way 1-D strip-mapping. It can be easily shown that the generated partition satisfies the "nearest-neighbor" property \([41,42]\). That is, each such partition can have connections to at most eight surrounding partitions.

While each of the two orthogonal 1-D strip-partitions is clearly load-balanced, the intersection partitions are definitely not. Such a load-imbalance among the intersection partitions can in general be expected. Consequently, the 2-D strip partitioning approach consequently employs a second boundary-refinement phase following the initial generation of the 2-D orthogonal strip partition. The boundary-refinement \([41,42]\) procedure attempts to perform node transfers at the boundaries of partitions in such a way that the nearest-neighbor property of the initial orthogonal partition is retained.

By performing all horizontal communications before vertical communications, the values to be transferred diagonally can be transmitted in a store-and-forward fashion, without incurring an additional set-up cost for the diagonal communications. Thus the number of transfers required between diagonally related processors in the mesh is added on to the volume of the intermediate processor's communication. The maximum number of processors that any processor will need to communicate with is limited to four by this way.

A special case of 2-D orthogonal strip-partitioning, where a 2-way partition is made along one dimension is also treated (Fig. 5(c)). This special case is interesting since it requires a maximum of three communications by any processor in any iteration, in contrast to two and four respectively for the 1-D and general 2-D case. Thus the set-up cost incurred with this special 2-D mapping is in between that of the other two, and the communication volume is also somewhere in between.
This special case of 2-D orthogonal partitioning is hence referred to as a 1.5-D partition.

To facilitate a comparison, the case of a square mesh FE graph with "m" nodes on a side is considered here. The load-balanced partitioning of an \(m \times m\) node FE mesh is shown in Fig. 5(a), 5(b) and 5(c) for the 1-D, 2-D and 1.5-D cases respectively. The number of variables is twice the number of nodes in the sample FE problems used here.

2.5 Fault Tolerance in Parallel Architectures

Basic sequence of events that occur in a fault tolerant parallel system in response to errors due to a fault can be listed as follows:

1. detection of errors
2. location of the faulty processor
3. reconfiguration
4. restoration of the programs and data

Detection of errors is accomplished by use of either hardware redundancy or time redundancy or both. For example, error detection can be performed by the duplication of the hardware [43] or the reexecution of the algorithm [22,44]. Error correction can be performed by triplicating the hardware [43] or by using error correcting codes [45]. The detection information also frequently provides some help in fault location. Once the fault location information is available, the system restructures to avoid the identified faulty processor being further used, by utilizing the inherent hardware redundancy [26,27,46,47]. The original parallel computer structure and algorithm mapping are preserved by this way. Several reconfiguration strategies for some regular architectures have been proposed in recent
Figure 5: Strip mapping of a regular $m \times m$ FE mesh onto $P$ processors
papers [28,47,48]. If however, the system does not have the dynamic redundancy capability to automatically replace the faulty unit(s) with good unit(s), then the system can be reconfigured to operate without faulty units in a degraded mode. This action is referred to as graceful degradation [49].

Real-time fault tolerance techniques can be divided into two categories: concurrent error correction and concurrent error detection. Concurrent error correcting techniques, which tolerate failure and provide continuous system operation, include triple modular redundancy (TMR) [43]. TMR requires 200% redundancy to tolerate a single module failure.

Concurrent error detection techniques, which are designed to detect errors but not mask them, include totally self-checking (TSC) circuits, alternating logic, computing with shifted operands (RESO). TSC [18] methods require less than 100% hardware redundancy in a processor for detecting single bit errors. TSC is based on the single stuck-at fault model. The alternating logic [21] technique achieves fault detection by utilizing redundancy in time and is based on the successive execution of the required function and its dual. Alternating logic requires 100% time redundancy plus an average value of 85% redundancy in hardware to detect a single stuck-at fault. RESO [22,44] can detect errors in arithmetic and logic operations utilizing time redundancy and using the already existing replicated hardware in the form of identical bit slices. RESO uses 100% time redundancy for recomputation and comparison of results to detect all errors caused by faults in a single bit slice.

The redundancy, either in space or time, for these techniques is generally high (between 73% and 200%) [50]. Also, TSC and alternating logic methods are based on classical gate level stuck-at fault models which cannot cover all physical failures in VLSI chips built with MOS technology. Hence, cost effective fault tolerance
techniques which cover a broad set of failures with low redundancy are required.

Recently, a new approach, namely algorithm based fault tolerance has been proposed [23,24,51,52,53]. An algorithm is defined in [50] as a *Concurrent Error Detecting (CED)* algorithm if it detects any errors caused by any fault from a well defined class of faults in a processor architecture during the normal operation of the algorithm on the processor architecture. The central idea proposed in this approach is that, algorithms operate on encoded input data and produce encoded output data such that the overhead in time and space is not high. The three major steps involved in this technique are: the encoding of the data used by the algorithm, the redesign of the algorithm to operate on the encoded data, and the distribution of the computation steps in the algorithm among the computing units of the parallel architecture.
CHAPTER III

Coarse Grain Scaled Conjugate Gradient Algorithms for Distributed Memory Multiprocessors

3.1 Introduction

Granularity is an important factor in designing parallel algorithms. Distributed computation and broadcast of two global scalars at each iteration reduces the granularity of the SCG algorithm as described in Section 3.2. Section 3.3 presents a general formulation for a coarse grain SCG algorithm that drastically reduces the number of global communication steps. The crucial limitations encountered in the implementation of this algorithm are discussed in Section 3.3.1. Section 3.3.2 describes the formulation of a coarse grain partial SCG algorithm using this general algorithm as an attempt to avoid these limitations. A general formulation for s-step coarse grain SCG algorithms, that avoids these limitations for low s values, is presented in Section 3.4.1. These algorithms reduce the number of global communication steps by a factor of 2s. However, as described in Section 3.4.1, this general formulation requires s-1 extra matrix vector products per s SCG iterations. Two special cases for s=1 and s=2 that avoid this matrix vector product overhead, are derived in Sections 3.4.2 and 3.4.3 using the given general formulation.
3.2 Data Dependencies in the SCG Algorithm

Strong data dependencies exist in the basic SCG algorithm which limit the available concurrency. The distributed inner product computation \( \langle p_k, A p_k \rangle \) which is required for the computation of the global scalar \( \alpha_k \) cannot be initiated until the global scalar \( \beta_{k-1} \) is computed. Similarly, the distributed inner product \( \langle r_{k+1}, r_{k+1} \rangle \) which is required for the computation of the global scalar \( \beta_k \) cannot be computed until the global scalar \( \alpha_k \) is computed. During each SCG iteration, three distributed vector updates which involve no communication and one distributed sparse matrix vector product which involves only local communication, cannot be initiated until the updating of these global scalars are completed. These inner product computations require global information and thus are inherently sequential and introduce a large amount of interprocessor communication overhead. Hence, these data dependencies due to the inner product computations introduce a fine grain parallelism which degrades the performance of the algorithm on the hypercube.

The idea of restructuring the basic SCG algorithm to minimize data dependencies was first proposed in [54]. However, the restructuring given in [54], is proposed for a hypothetical multiprocessor with negligible communication cost. This is not a reasonable assumption for message passing multiprocessors such as hypercubes.

As was previously pointed out, set-up time is the dominating factor in the overall communication cost in message-passing architectures. Due to the data dependencies in inner product computations only one inner product term can be accumulated during one global sum (GS) communication step and hence only one global scalar can be broadcasted during one global broadcast (GB) communication.
step. Hence, only one floating point word is transferred between nearest neighbor processors during the $2d = 2\log_2 P$ concurrent nearest neighbor communication steps of the $GS-GB$ operation. The frequency rather than the volume of the communication degrades the performance of the algorithm during the computation of the global scalars. In the formulation presented here, the basic $SCG$ algorithm is restructured in such a way as to increase the number of inner products that are accumulated and broadcast during one $GS-GB$ communication step.

3.3 General Formulation of a Coarse Grain SCG Algorithm

The $CG$ method recursively generates the sequence of conjugate direction ($A$-orthogonal) vectors $p_k$. It makes use of this sequence of direction vectors to generate the sequence $x_k$ which minimizes the the quadratic objective function $f(x) = \frac{1}{2} < x, Ax > - < b, x >$ over the linear variety $E_k = x_0 + Sp\{p_0, p_1, \ldots, p_{k-1}\}$. Here, $Sp\{p_i\}_{i=0}^{i=k-1}$ represents the subspace spanned by the vectors $\{p_i\}_{i=0}^{i=k-1}$. Hence, for a positive definite $A$ matrix, the sequence $x_k$ approaches the unique minimum of the above objective function which is also the unique solution to the linear system of equations $Ax = b$.

The $CG$ theorem is given below without proof. The proof can be found in [38].

**Theorem 3.1 (CG Theorem)** If the CG algorithm does not terminate at $x_k$ (i.e. $< r_k, r_k > \neq 0$) then

a: $E_k = Sp\{p_i\}_{i=0}^{i=k-1} = Sp\{r_i\}_{i=0}^{i=k-1} = Sp\{A^i r_0\}_{i=0}^{i=k-1} = Sp\{A^i p_0\}_{i=0}^{i=k-1}$,

b: $p_k$ is $A$-orthogonal to $E_k$, i.e., $< p_k, A p_i > = < p_k, A r_i > = 0$ for $i = 0, 1, \ldots, k - 1$,

c: $r_k$ is orthogonal to $E_k$, i.e., $< r_k, p_i > = < r_k, r_i > = 0$ for $i = 0, \ldots, k - 1$. 

33
Hence, due to a) of the CG theorem the sequence \( x_{k+1} \) generated according to

\[
x_{k+1} = x_0 + \sum_{i=0}^{k} \alpha_i p_i \quad (3.1)
\]
can be also generated using

\[
x_{k+1} = x_0 + \sum_{i=0}^{k} \eta_i q_i \quad (3.2)
\]

where \( q_i = A^i r_0 = A^i p_0 \) for \( i = 0, 1, \ldots, k \).

The desirable property of this alternative formulation is that the generation of the sequence \( q_k \) does not depend on the result of any inner product terms in contrast to the generation of the sequence \( p_k \) in the basic CG algorithm. The sequence of matrix vector products \( q_k \) can be generated using the recurrence relation

\[
q_i = A q_{i-1} \quad (3.3)
\]

for \( i = 1, 2, \ldots, k \), where \( q_0 = r_0 = p_0 \).

By equating (3.1) to (3.2) one can obtain

\[
\sum_{i=0}^{k} \alpha_i p_i = \sum_{i=0}^{k} \eta_i q_i \quad (3.4)
\]

Due to a) of the CG theorem, the sequence of direction vectors \( p_i \) in (3.4) can be expressed in terms of the sequence of vectors \( q_i \) as

\[
p_0 = d_{0,0} q_0 \\
p_1 = d_{1,0} q_0 + d_{1,1} q_1 \\
p_2 = d_{2,0} q_0 + d_{2,1} q_1 + d_{2,2} q_2 \\
\vdots \\
p_k = d_{k,0} q_0 + d_{k,1} q_1 + d_{k,2} q_2 + \cdots + d_{k,k} q_k
\]

that is,

\[
p_i = \sum_{j=0}^{i} d_{i,j} q_j \quad (3.6)
\]
Hence, inserting (3.6) into (3.4) and after some manipulations

\[
\sum_{i=0}^{k} \alpha_i \sum_{j=0}^{i} d_{ij} q_j = \sum_{i=0}^{k} \eta_i q_i
\]

\[
\sum_{i=0}^{k} \sum_{j=i}^{k} \alpha_j d_{ij} q_i = \sum_{i=0}^{k} \eta_i q_i
\]  

(3.7)

from which,

\[
\eta_i = \sum_{j=i}^{k} \alpha_j d_{ji}
\]

(3.8)

can be obtained for \( i = 0, 1, \ldots, k \). Now, the problem is to find the recurrence relations for the scalars \( \{\alpha_i\}_{i=0}^{k} \) and \( \{d_{ij}\}_{j=0}^{i} \). It is clear from the recurrence relations in the basic CG algorithm that, for the vectors \( r_i \) and \( p_i \)

\[
r_i = r_i(A)q_0
\]

(3.9a)

\[
p_i = p_i(A)q_0
\]

(3.9b)

holds where \( r_i(t) \) and \( p_i(t) \) are matrix polynomials defined by the recurrence formulas

\[
r_i(t) = r_{i-1}(t) - \alpha_{i-1} t p_{i-1}(t)
\]

(3.10a)

\[
p_i(t) = r_i(t) + \beta_{i-1} p_{i-1}(t)
\]

(3.10b)

where \( r_0(t) = p_0(t) = 1 \). Hence the coefficients of the polynomials

\[
p_i(t) = \sum_{j=0}^{i} d_{ij} t^j
\]

(3.11a)

\[
r_i(t) = \sum_{j=0}^{i} f_{ij} t^j
\]

(3.11b)

can be generated using the recurrence relations

\[
f_{i+1,j} = f_{i,j} - \alpha_i d_{ij-1} \quad \text{for } j = 0, \ldots, i + 1
\]

(3.12a)

\[
d_{i+1,j} = f_{i+1,j} + \beta_i d_{ij} \quad \text{for } j = 0, \ldots, i
\]

(3.12b)
where, \( f_{0,0} = d_{0,0} = 1 \).

Now, the problem is to find the recursive expressions for the scalars \( \alpha_i \) and \( \beta_i \) in the above recurrence relation. A corollary of the CG theorem, results of which will be used for the derivation of these expressions, is stated below.

**Corollary 3.1**

\begin{align*}
\text{a: } < p_i, A^j p_i > &= < p_i, A^j r_i > = 0 \text{ for } i + j \leq k \text{ and } j \neq 0 \\
\text{b: } < r_k, A^j p_i > &= < r_k, A^j r_i > = 0 \text{ for } i + j < k
\end{align*}

**PROOF**

\( a: \) If \( p_i \) and \( r_i \in Sp\{ r_0, A_0, \ldots, A^j r_0 \} \) due to a) of the CG theorem. Hence, \( A^{j-1} p_i \) and \( A^{j-1} r_i \in Sp\{ r_0, A_0, \ldots, A^{i+j} r_0 \} = Sp\{ p_0, p_1, \ldots, p_{i+j-1} \} \). Thus,

\begin{align*}
< p_k, A^j p_i > &= < p_k, A(A^{j-1})p_i > = 0 \\
< p_k, A^j r_i > &= < p_k, A(A^{j-1})r_i > = 0
\end{align*}

for \( i + j \leq k \) and \( j \neq 0 \) due to b) of the CG theorem. \( Q.E.D. \)

\( b: \) This part of the corollary can be similarly proven using the results a) and c) of the CG theorem. \( Q.E.D. \)

From the basic CG equations,

\begin{align*}
\alpha_i &= \frac{< r_i, r_i >}{< p_i, A p_i >} \quad (3.13a) \\
\beta_i &= \frac{< r_{i+1}, r_{i+1} >}{< r_i, r_i >} \quad (3.13b)
\end{align*}

Hence, expressions for the inner products \( < r_i, r_i > \) and \( < p_i, A p_i > \) are required. The recurrence relations defined for the sequence of residual and direction vectors in the basic SCG algorithm are rewritten below.

\begin{align*}
r_i &= r_{i-1} - \alpha_{i-1} A p_{i-1} \quad (3.14a) \\
p_i &= r_i + \beta_{i-1} p_{i-1} \quad (3.14b)
\end{align*}

36
Now, using the recurrence relation defined for \( r_i \) in (3.14a)

\[
< r_i, r_i > = < r_i, r_{i-1} > - \alpha_{i-1} < r_i, A p_{i-1} > \\
= (-\alpha_{i-1}) < r_i, A p_{i-1} > 
\]

(3.15)
since \( < r_i, r_{i-1} > = 0 \) by b) of Corollary 3.1. By use of the recurrence relation in (3.14b) for \( p_{i-1} \) (3.15) becomes

\[
< r_i, r_i > = (-\alpha_{i-1})[ < r_i, A r_{i-1} > + \beta_{i-2} < r_i, A p_{i-2} > ] \\
= (-\alpha_{i-1}) < r_i, A r_{i-1} > 
\]

(3.16)
since \( < r_i, A p_{i-2} > = 0 \) by b) of Corollary 3.1. Therefore, using the recurrence relation defined in (3.14a) for \( r_{i-1} \) (3.16) can be written as

\[
< r_i, r_i > = (-\alpha_{i-1})[ < r_i, A r_{i-2} > - \alpha_{i-2} < r_i, A^2 p_{i-2} > ] \\
= (- \alpha_{i-1})(-\alpha_{i-2}) < r_i, A^2 p_{i-2} > 
\]

(3.17)
since \( < r_i, A r_{i-2} > = 0 \) by b) of Corollary 3.1. Hence, repeating the above recursive procedure \( i \) times it can be shown that:

\[
< r_i, r_i > = (\prod_{j=0}^{i-1} (-\alpha_j)) < r_i, A^i q_0 > 
\]

(3.18)
where \( q_0 = r_0 = p_0 \).

Similarly, using the recurrence relation defined for \( p_i \) in (3.14b)

\[
< p_i, A p_i > = < p_i, A r_i > + \beta_{i-1} < p_i, A p_{i-1} > \\
= < p_i, A r_i > 
\]

(3.19)
since \( < p_i, A p_{i-1} > = 0 \) by a) of Corollary 3.1. Now using of the recurrence relation in (3.14a) for \( r_i \) (3.19) can be expressed as

\[
< p_i, A p_i > = < p_i, A r_{i-1} > - \alpha_{i-1} < p_i, A^2 p_{i-1} > \\
= (-\alpha_{i-1}) < p_i, A^2 p_{i-1} > 
\]

(3.20)
since \( \langle p_i, Ar_{i-1} \rangle = 0 \) by a) of Corollary 3.1. By use of the recurrence relation in (3.14b) for \( p_{i-1} \) (3.20) becomes

\[
\langle p_i, Ap_i \rangle = (-\alpha_i)\langle p_i, A^2r_{i-1} \rangle + \beta_{i-2} \langle p_i, A^2p_{i-2} \rangle
\]
\[
= (-\alpha_{i-1}) \langle p_i, A^2r_{i-1} \rangle
\]  

(3.21)
since \( \langle p_i, A^2p_{i-2} \rangle = 0 \) by a) of Corollary 3.1. Now using of the recurrence relation in (3.14a) for \( r_{i-1} \) (3.21) can be written as

\[
\langle p_i, Ap_i \rangle = (-\alpha_{i-1})\langle p_i, A^2r_{i-2} \rangle - \alpha_{i-2} \langle p_i, A^3p_{i-2} \rangle
\]
\[
= (-\alpha_{i-1})(-\alpha_{i-2})\langle p_i, A^3p_{i-2} \rangle
\]

(3.22)
since \( \langle p_i, A^2r_{i-2} \rangle = 0 \) by a) of Corollary 3.1. Hence, repeating the above recursive procedure \( i \) times it can be shown that:

\[
\langle p_i, Ap_i \rangle = \left( \prod_{j=0}^{i-1} (-\alpha_j) \right) \langle p_i, A^{i+1}q_0 \rangle
\]

(3.23)

Inserting \( r_i = r_i(A)q_0 = \sum_{j=0}^{i} f_{i,j}A^j q_0 \) and \( p_i = p_i(A)q_0 = \sum_{j=0}^{i} d_{i,j}A^j q_0 \) into (3.18) and (3.23) one can obtain

\[
\langle r_i, r_i \rangle = \left( \prod_{j=0}^{i-1} (-\alpha_j) \right) \langle \sum_{j=0}^{i} f_{i,j}A^j q_0, A^i q_0 \rangle
\]
\[
= \left( \prod_{j=0}^{i-1} (-\alpha_j) \right) \sum_{j=0}^{i} f_{i,j} \langle A^j q_0, A^i q_0 \rangle
\]

(3.24a)

\[
\langle p_i, Ap_i \rangle = \left( \prod_{j=0}^{i-1} (-\alpha_j) \right) \sum_{j=0}^{i} d_{i,j} \langle A^j q_0, A^{i+1} q_0 \rangle
\]

(3.24b)

Now, defining

\[
\ell_j = \langle q_0, A^j q_0 \rangle
\]

(3.25)

(3.24a) and (3.24b) become

\[
\langle r_i, r_i \rangle = \left( \prod_{j=0}^{i-1} (-\alpha_j) \right) \sum_{j=0}^{i} f_{i,j}\ell_{i+j}
\]

(3.26a)
\[ < p_i, A p_i > = (\prod_{j=0}^{i-1} (-\alpha_j)) \sum_{j=0}^{i} d_{i,j} \ell_{i+j+1}. \]  

(3.26b)

Hence, inserting these expressions for the inner products into (3.13a) and (3.13b)

\[
\alpha_i = \frac{\sum_{j=0}^{i} f_{i,j} \ell_{i+j}}{\sum_{j=0}^{i} d_{i,j} \ell_{i+j+1}} \]  

(3.27a)

\[
\beta_i = -\alpha_i \frac{\sum_{j=0}^{i+1} f_{i+1,j} \ell_{i+j+1}}{\sum_{j=0}^{i} f_{i,j} \ell_{i+j}} \]  

(3.27b)

are obtained. It can be deduced from (3.27a) and (3.27b) that the following sequence of inner products

\[ \ell_j = \langle q_0, A^j q_0 \rangle \quad \text{for } j = 0, 1, \ldots, 2s - 1 \]  

(3.28)

which can be computed using the sequence of vectors \( \{q_j\}_{j=0}^{s} \) from

\[ \ell_j = \langle A^\nu q_0, A^\omega q_0 \rangle = \langle q_\nu, q_\omega \rangle \quad \text{for } 0 \leq \nu \leq s - 1 \text{ and } \nu \leq \omega \leq \nu + 1 \]  

(3.29)

are needed to compute the sequence of scalars \( \{\eta_i\}_{i=0}^{s-1} \).

Hence, the steps for the general \( CG-SCG \) algorithm to compute the approximate solution vector \( x_s \) starting from the initial guess vector \( x_0 \) in a single \( GS-GB \) communication step can be given as follows:

**Initially** : Scale \( A \), let \( r_0 = b - A x_0 \) and let \( q_0 = r_0 \)

**Step 1** : Compute \( s \) distributed matrix vector products

\[ q_i = A q_{i-1} \]  

for \( i = 1, \ldots, s \).

**Step 2** : Compute \( 2s \) distributed inner products \( \{\ell_j\}_{j=0}^{2s-1} \) from

\[ \ell_j = \langle q_\nu, q_\omega \rangle \]  

for \( 0 \leq \nu \leq s - 1 \) and \( \nu \leq \omega \leq \nu + 1 \) in one \( GS-GB \) communication step.
Step 3.A: Compute the coefficients of the polynomials

\[ p_i(t) = \sum_{j=0}^{i} d_{ij} t^j \quad \text{and} \quad r_i(t) = \sum_{j=0}^{i} f_{ij} t^j \quad \text{from} \]

\[ \alpha_i = \frac{\sum_{j=0}^{i} f_{ij} \ell_{i+j}}{\sum_{j=0}^{i} d_{ij} \ell_{i+j+1}} \quad (3.32) \]

\[ f_{i+1,j} = f_{i,j} - \alpha_i d_{i,j-1} \quad \text{for} \quad j = 0, \ldots, i+1 \quad (3.33) \]

\[ \beta_i = \frac{\sum_{j=0}^{i+1} f_{i+1,j} \ell_{i+j+1}}{\sum_{j=0}^{i} f_{i,j} \ell_{i+j}} \quad (3.34) \]

\[ d_{i+1,j} = f_{i+1,j} + \beta_i d_{i,j} \quad \text{for} \quad j = 0, \ldots, i \quad (3.35) \]

for \( i = 0,1,\ldots,s-1 \), where \( d_{0,0} = f_{0,0} = 1 \).

Step 3.B: Compute \( s \) scalars

\[ \eta_i = \sum_{j=i}^{s-1} \alpha_j d_{j,i} \quad (3.36) \]

for \( i = 0,1,\ldots,s-1 \).

Step 4: Compute \( s \) distributed vector updates

\[ x_s = x_0 + \sum_{i=0}^{s-1} \eta_i q_i \quad (3.37) \]

In Step 1, \( s \) distributed sparse matrix vector products are computed to generate the sequence of vectors \( \{q_i\}_{i=0}^{s} \). In this step, \( s \) local interprocessor communications are required. At the end of Step 1 each processor contains its own slice of the sequence of vectors \( \{q_i\}_{i=0}^{s} \). In Step 2, each processor concurrently computes \( 2s \) partial sums. Then, in a single GS communication step, these \( 2s \) inner products \( \{\ell_i\}_{i=0}^{2s-1} \) are accumulated in the root processor. In Step 3.A, the root processor recursively computes the coefficients of the sequence of direction vector polynomials \( \{p_i(t)\}_{i=0}^{s-1} \) and the sequence of scalars \( \{\alpha_i\}_{i=0}^{s-1} \) in terms of the sequence of inner products \( \{\ell_i\}_{i=0}^{2s-1} \). Next, in Step 3.B the root processor computes the sequence of
scalars \( \{\eta_i\}_{i=0}^{s-1} \) in terms of the sequence of scalars \( \{\alpha_i\}_{i=0}^{s-1} \) and the coefficients of the sequence of direction vectors \( \{p_i(t)\}_{i=0}^{s-1} \) already computed in Step 3.A. Then, the root processor broadcasts this sequence of \( s \) scalars in a single GB communication step. In Step 4, after each processor receives these \( s \) scalars, a sequence of \( s \) distributed vector updates are performed without any communication.

### 3.3.1 Performance Analysis of the Algorithm

In the proposed algorithm, the approximate solution vector \( x_s \) is generated starting from \( x_0 \) in a single GS-GB communication step without generating the intermediate sequence of direction and residual vectors. Now, assume that the parameter \( s \) can be chosen such that the convergence is obtained in the first \( s \) iterations of the basic SCG algorithm (i.e. \( \|r_s, r_s\| < \|b, b\|^{\frac{1}{2}} \) < tolerance). Then, the performance model of the proposed algorithm can be given as follows:

\[
T = s(z + 3)N_P T_{\text{calc}} + 2.5s^2 T_{\text{calc}} + [2dS_C + 3sdT_C + sL_C]
\]

where \( N \) is the total number of variables, \( z \) is the average number of nonzero entries per row of the \( A \) matrix, \( P \) is the number of processors in the hypercube, \( d \) is the dimension of the hypercube, \( T_{\text{calc}} \) is the time taken per single floating point addition and multiplication, \( S_C \) is the set-up time per communication, \( T_C \) is the data transmission rate per single floating point word over the channel, and \( L_C \) is the total local communication time required during a single distributed sparse matrix vector product. Note that, this simplified analysis holds under perfectly load balanced conditions where equal number of variables are mapped to each processor, and each processor spends the same amount of time for the local communication steps required for the distributed sparse matrix vector product computation.
For the sake of comparison, the performance model of the basic \textit{SCG} algorithm for \( s \) iterations is also given below,

\[
T = s(z + 5)\frac{N}{P} T_{\text{calc}} + \left[ 4sd(S_C + T_C) + sL_C \right] = \text{parallel} + \text{communication}.
\]

The following can be observed from the comparison of these performance models:

1. the frequency of the global communication is reduced by a factor of \( 2s \) from \( 4sd \) to \( 2d \) for the first \( s \) iterations of the basic \textit{SCG} algorithm.

2. the volume of global communication is reduced from \( 4sd \) to \( 3sd \) for the first \( s \) iterations of the basic \textit{SCG} algorithm.

3. the parallel computation time is reduced from \( s(z + 5)\frac{N}{P} \) to \( s(z + 3)\frac{N}{P} \) for the first \( s \) iterations of the basic \textit{SCG} algorithm.

4. a sequential computational overhead proportional to \( s^2 (2.5s^2 T_{\text{calc}}) \) is introduced for the first \( s \) iterations of the basic \textit{SCG} algorithm.

5. memory requirement is increased from \( (z + 5)\frac{N}{P} \) to \( (z + 3 + s)\frac{N}{P} \) floating point words.

The performance comparison given above is made under the assumption that the parameter \( s \) can be chosen arbitrarily large. However, there are crucial limitations concerning the selection of the parameter \( s \). The first limitation is the fact that the new formulation is more sensitive to round-off errors than is the basic \textit{SCG} algorithm. For sufficiently large \( s \), the sequence of direction vectors generated in the proposed algorithm may begin to diverge from the true direction vectors. The second limitation is the fact that the sequential computational overhead increases with the square of the parameter \( s \). Hence, the step size parameter \( s \) cannot be
chosen arbitrarily large. Experimental results for a wide range of sample FE problems have shown that, \( s \) should be chosen less than 10 in order to sustain the numerical stability.

### 3.3.2 An \( s \)-Step Coarse Grain Partial SCG Algorithm

If the convergence is not obtained during the first pass of the algorithm for the chosen step size \( s \), Step 5 can be added to the given algorithm which computes the residual vector \( r_s \) using

\[
\mathbf{r}_s = \mathbf{b} - A\mathbf{x}_s .
\]

(3.38)

Then, the algorithm can be restarted by assigning

\[
\mathbf{q}_0 = \mathbf{r}_s .
\]

(3.39)

Hence, the algorithm effectively performs \( s \) iterations of the \( SCG \) procedure during the first pass and then rather than continuing the \( SCG \) iterations it restarts from the current solution vector \( \mathbf{x}_s \) and performs \( s \) more \( SCG \) iterations during the second pass. This procedure is repeated until convergence is obtained. Hence, the method becomes an \( s \)-step partial \( SCG \) method [55]. The special case of \( s = 1 \) corresponds to the standard method of steepest descent while \( s = N \) corresponds to the \( SCG \) method [55]. It has been shown in [55] that for sufficiently large \( y \) the result of applying \( \frac{y}{s} \) passes of the partial \( SCG \) with step size \( s \) gives a better approximation then \( \frac{y}{s-1} \) passes of the partial \( SCG \) with step size \( s - 1 \). Hence, in order to increase the convergence rate of the method, the step size \( s \) should be chosen as large as possible.

As is indicated above, the proposed algorithm may increase the total number of effective \( SCG \) iterations required for convergence, if convergence is not obtained
during the first pass of the algorithm. In this case, the performance comparison given previously does not hold since extra computation and communication steps have to be performed due to the increased number of iterations required for convergence. However, there are applications that exclusively require the use of partial SCG method[38]. Hence, this formulation is recommended only for such applications.

3.4 s-Step Coarse Grain SCG Algorithms

As is mentioned in the previous section, the partial SCG algorithm degrades the performance of the original SCG algorithm for low values of $s$. The solution to this problem is to continue the basic SCG iterations after each step instead of restarting the SCG algorithm. However, it will be shown in the next section that this formulation requires, in its general form, $(s-1)$ extra matrix vector products per $s$ SCG iterations, and therefore is computationally quite expensive. However, the formulation for two special cases ($s = 1$ and $s = 2$), with substantially low computational overhead, will be given using this general formulation.

3.4.1 General Formulation for s-Step CG–SCG Algorithms

The rationale behind this general formulation is to accumulate $2s$ inner products $\{\ell_j\}_{j=1}^{2s}$ using

$$\ell_1 = \langle p_{ks}, Ap_{ks} \rangle$$

$$\ell_j = \langle p_{ks}, A^j p_{ks} \rangle \quad \text{for } j = 2, \ldots, 2s - 1$$

$$= \langle A^\nu p_{ks}, A^\omega p_{ks} \rangle \quad \text{for } 1 \leq \nu \leq s - 1, \text{ and } \nu \leq \omega \leq \nu + 1$$

$$\ell_{2s} = \langle A^s p_{ks}, A^s p_{ks} \rangle$$

(3.40a) (3.40b) (3.40c)
in a single GS-GB communication step after computing \( s \) consecutive distributed sparse matrix vector products

\[
q_j = Aq_{j-1} = A^j p_{ks}
\]  

for \( j = 1, 2, \ldots, s \) with \( q_0 = p_{ks} \). Hence, the frequency of GS-GB communication steps will be reduced by a factor of \( 2^s \).

The matrix polynomial representation given in Section 3.3 for \( r \) and \( p \) can be rewritten for this case as follows:

\[
\begin{align*}
    r_{ks+i} &= d_i(A)r_{ks} + e_i(A)p_{ks} \\
    p_{ks+i} &= f_i(A)r_{ks} + g_i(A)p_{ks}
\end{align*}
\]  

where \( d_i(t) \), \( e_i(t) \), \( f_i(t) \) and \( g_i(t) \) are defined by the following recurrence relations

\[
\begin{align*}
    d_i(t) &= d_{i-1}(t) - \alpha_{ks+i-1}t f_{i-1}(t) \\
    e_i(t) &= e_{i-1}(t) - \alpha_{ks+i-1}t g_{i-1}(t) \\
    f_i(t) &= d_i(t) + \beta_{ks+i-1} f_{i-1}(t) \\
    g_i(t) &= e_i(t) + \beta_{ks+i-1} g_{i-1}(t)
\end{align*}
\]  

for \( i = 1, 2, \ldots, s \) with \( d_0(t) = 1 \), \( e_0(t) = 0 \), \( f_0(t) = 0 \), and \( g_0(t) = 1 \). Hence, the coefficients of the matrix polynomials

\[
\begin{align*}
    d_i(t) &= \sum_{j=0}^{i-1} d_{i,j} t^j \\
    e_i(t) &= \sum_{j=0}^{i} e_{i,j} t^j \\
    f_i(t) &= \sum_{j=0}^{i-1} f_{i,j} t^j \\
    g_i(t) &= \sum_{j=0}^{i} g_{i,j} t^j
\end{align*}
\]
can be generated using the recurrence relations

\[ d_{i+1,j} = d_{i,j} - \alpha_{ks+i} f_{i,j-1} \text{ for } j = 0, 1, \ldots, i \]  
(3.45a)

\[ e_{i+1,j} = e_{i,j} - \alpha_{ks+i} g_{i,j-1} \text{ for } j = 0, 1, \ldots, i + 1 \]  
(3.45b)

\[ f_{i+1,j} = d_{i+1,j} + \beta_{ks+i} f_{i,j} \text{ for } j = 0, 1, \ldots, i \]  
(3.45c)

\[ g_{i+1,j} = e_{i+1,j} + \beta_{ks+i} g_{i,j} \text{ for } j = 0, 1, \ldots, i + 1 \]  
(3.45d)

where, \( d_{0,0} = g_{0,0} = 1 \), and \( e_{0,0} = f_{0,0} = 0 \). The recursive expressions for the scalars \( \alpha_i \) and \( \beta_i \) can be derived by inserting the matrix polynomial representations for \( r_{ks+i} \) and \( p_{ks+i} \), given in (3.42a) and (3.42b), into (3.18) and (3.23) with \( q_0 = p_{ks} \) as follows:

\[
< r_{ks+i}, r_{ks+i} > = (\prod_{j=0}^{i-1} (-\alpha_{ks+j})) \left[ \sum_{j=0}^{i-1} d_{i,j} < A^j r_{ks}, A^i p_{ks} > + \sum_{j=0}^{i} e_{i,j} < A^j p_{ks}, A^i p_{ks} > \right] 
\]
(3.46a)

\[
< p_{ks+i}, A p_{ks+i} > = (\prod_{j=0}^{i-1} (-\alpha_{ks+j})) \left[ \sum_{j=0}^{i-1} f_{i,j} < A^j r_{ks}, A^{i+1} p_{ks} > + \sum_{j=0}^{i} g_{i,j} < A^j p_{ks}, A^{i+1} p_{ks} > \right] 
\]
(3.46b)

Now, defining

\[
\ell_j = < p_{ks}, A^j p_{ks} > 
\]
(3.47a)

\[
\phi_j = < r_{ks}, A^j p_{ks} > 
\]
(3.47b)

(3.46a) and (3.46b) can be written as

\[
< r_{ks+i}, r_{ks+i} > = (\prod_{j=0}^{i-1} (-\alpha_{ks+j})) \left[ d_{i,0} \phi_0 + \sum_{j=1}^{i-1} d_{i,j} \phi_{i+j} + \sum_{j=0}^{i} e_{i,j} \ell_{i+j} \right] 
\]
(3.48a)
\[ <\mathbf{p}_{k^s+i}, A^{j}\mathbf{p}_{k^s+i}> = (\prod_{j=0}^{i-1} (-\alpha_{k^s+j})) \left[ \sum_{j=0}^{i-1} f_{i,j}^{j}\phi_{i+j+1} \right. \]
\[ \left. + \sum_{j=0}^{i} g_{i,j}^{j}\ell_{i+j+1} \right] \] (3.48b)

Inserting (3.48a) and (3.48b) into (3.13a) and (3.13b)

\[ \alpha_{k^s+i} = \frac{d_{i,0}^{i,j}e_{i,j}^{j} + \sum_{j=1}^{i} \sum_{j=0}^{i} e_{i,j}^{j}\ell_{i+j+1}}{\sum_{j=0}^{i} f_{i,j}^{j}\phi_{i+j+1} + \sum_{j=0}^{i} g_{i,j}^{j}\ell_{i+j+1}} \] (3.49a)

\[ \beta_{k^s+i} = -\frac{\sum_{j=0}^{i} d_{i,0}^{i,j}e_{i,j}^{j} + \sum_{j=0}^{i} e_{i,j}^{j}\ell_{i+j+1}}{d_{i,0}^{i,j}\phi_{i+j+1} + \sum_{j=1}^{i} d_{i,j}^{i,j}\phi_{i+j+1} + \sum_{j=0}^{i} e_{i,j}^{j}\ell_{i+j+1}} \] (3.49b)

are obtained for \( i = 0, 1, \ldots, s \). It can be deduced from the above equations that the following sequence of inner products

\[ \ell_{j} = <\mathbf{p}_{k^s}, A^{j}\mathbf{p}_{k^s}> \text{ for } j = 1, \ldots, 2s \] (3.50a)

\[ \phi_{j} = <\mathbf{r}_{k^s}, A^{j}\mathbf{p}_{k^s}> \text{ for } j = 0, 1, \ldots, 2s - 1 \] (3.50b)

are required to compute the sequence of global scalars \( \{\alpha_{k^s+i}\}_{i=0}^{s-1} \) and \( \{\beta_{k^s+i}\}_{i=0}^{s-1} \).

The sequence of \( 2s \) inner products \( \{\ell_{j}\}_{j=1}^{2s} \) can be computed using the sequence of \( s + 1 \) vectors \( \{q_{j}\}_{j=0}^{s} \) with \( q_{0} = \mathbf{p}_{k^s} \) as shown in equations (3.40a)-(3.40c).

Recursive expressions for the sequence of inner products \( \{\phi_{j}\}_{j=0}^{2s-1} \) in terms of the sequence of inner products \( \{\ell_{j}\}_{j=1}^{2s} \) can be derived by making extensive use of the results of Corollary 3.1. The derivations for \( \phi_{0}, \phi_{1}, \phi_{2}, \) and \( \phi_{3} \) which are required for the special cases \( (s = 1) \) and \( (s = 2) \) will be given in the following sections.

Note that, the vector \( \mathbf{p}_{(k+1)^s} \), where

\[ \mathbf{p}_{(k+1)^s} = \sum_{j=0}^{s-1} f_{s,j}^{j}\mathbf{r}_{k^s} + \sum_{j=0}^{s} g_{s,j}^{j}A^{j}\mathbf{p}_{k^s} \] (3.51)

is required for the next step of the algorithm. As is seen from (3.51) \( (s - 1) \) extra matrix vector products, \( A^{j}\mathbf{r}_{k^s} \), for \( j = 1, 2, \ldots, s - 1 \) are required for \( s \)-step
CG-SCG algorithm. However, for $s=1$, equation (3.51) becomes

$$P_{k+1} = f_{1,0} r_{ks} + g_{1,0} p_{ks} + g_{1,1} A p_{ks}.$$  \hspace{1cm} (3.52)

Hence, CG-SCG/s1 algorithm requires no extra matrix vector product. The formulation for CG-SCG/s1, which has no computational overhead, is given in Section 3.4.2.

For $s = 2$, (3.51) becomes

$$P_{2k+1} = f_{2,0} r_{ks} + f_{2,1} A r_{ks} + g_{2,0} p_{ks} + g_{2,1} A p_{ks} + g_{2,2} A^2 p_{ks}.$$ \hspace{1cm} (3.53)

Hence, one extra matrix vector product computation, $A r_{ks}$, is required per two iterations of the basic SCG algorithm. A formulation for this special case which avoids this extra sparse matrix vector product by introducing an extra vector update per SCG iteration is presented in Section 3.4.3.

### 3.4.2 A CG-SCG/s1 Algorithm

The rationale behind this formulation is to accumulate two inner products

$$\ell_1 = < p_{ks}, A p_{ks} >$$ \hspace{1cm} (3.54a)

$$\ell_2 = < A p_{ks}, A p_{ks} >$$ \hspace{1cm} (3.54b)

in each GS-GB communication step after computing the distributed sparse matrix-vector product

$$q_k = A p_k.$$ \hspace{1cm} (3.55)

The problem is to find the expressions for the global scalars $\alpha_k$ and $\beta_k$ in terms of these inner products. The expressions for these global scalars can be obtained by evaluating (3.49a) and (3.49b) for $s = 1$ and $i = 0$. Hence, from (3.49a)

$$\alpha_k = \frac{d_{0,0} \phi_0 + e_{0,0} \ell_0}{f_{0,0} \phi_1 + g_{0,0} \ell_1} = \frac{\phi_0}{\ell_1}.$$ \hspace{1cm} (3.56)
since \( d_{0,0} = g_{0,0} = 1 \), and \( e_{0,0} = f_{0,0} = 0 \). The recurrence relation defined in (3.14b) for \( p_k \) can be used to find the expression for \( \phi_0 \) as follows:

\[
\phi_0 = \langle r_k, p_k \rangle \\
= \langle r_k, r_k \rangle + \beta_{k-1} \langle r_k, p_{k-1} \rangle \\
= \langle r_k, r_k \rangle
\]

(3.57)

since \( \langle r_k, p_{k-1} \rangle = 0 \) by b) of Corollary 3.1. Hence,

\[
\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle p_k, A p_k \rangle}
\]

(3.58)

which is the same expression given at Step 2.b of the basic SCG algorithm.

Similarly, evaluating (3.49b) for \( s = 1 \) and \( i = 0 \) yields

\[
\beta_k = -\alpha_k \frac{d_{1,0} \phi_1 + e_{1,0} \ell_1 + e_{1,1} \ell_2}{d_{0,0} \phi_0 + e_{0,0} \ell_1}
\]

\[
= -\alpha_k \frac{d_{1,0} \phi_1 + e_{1,0} \ell_1 + e_{1,1} \ell_2}{\phi_0}
\]

(3.59)

since \( d_{0,0} = 1 \) and \( e_{0,0} = 0 \). The coefficients of the matrix polynomials \( d_1(t) \), and \( e_1(t) \) can be computed from the recurrence relations given in (3.45a), and (3.45b) as follows:

\[
d_{1,0} = d_{0,0} - \alpha_k f_{0,-1} = 1
\]

(3.60a)

\[
e_{1,0} = e_{0,0} - \alpha_k g_{0,-1} = 0
\]

(3.60b)

\[
e_{1,1} = e_{0,1} - \alpha_k g_{0,0} = -\alpha_k
\]

(3.60c)

Hence, (3.59) becomes

\[
\beta_k = -\alpha_k \frac{\phi_1 - \alpha_k \ell_2}{\phi_0}
\]

(3.61)

Now the problem is to find the expression for \( \phi_1 \). Note that, from (3.14b) for \( i = k \)

\[
r_k = p_k - \beta_{k-1} p_{k-1}
\]

(3.62)
can be obtained. Hence,

\[ \phi_1 = \langle r_k, Ap_k \rangle \]
\[ = \langle p_k, Ap_k \rangle - \beta_{k-1} \langle p_{k-1}, Ap_k \rangle \]
\[ = \ell_1 - \beta_{k-1} \langle p_k, Ap_k \rangle \]
\[ = \ell_1 \]

(3.63)
since \( \langle p_{k-1}, Ap_{k-1} \rangle = 0 \) by a) of Corollary 3.1. Inserting (3.63) and \( \phi_0 = \alpha_k \ell_1 \) from (3.56) into (3.61)

\[ \beta_k = -\alpha_k \frac{\ell_1 - \alpha_k \ell_2}{\alpha_k \ell_1} = \alpha_k \frac{\ell_2}{\ell_1} - 1 \]  (3.64a)

\[ \beta_k = \alpha_k \frac{\langle Ap_k, Ap_k \rangle}{\langle p_k, Ap_k \rangle} - 1 \]  (3.64b)
is obtained.

Therefore, the inner products \( \langle p_k, Ap_k \rangle \) and \( \langle Ap_k, Ap_k \rangle \) are needed to compute the global scalars \( \alpha_k \) and \( \beta_k \). The inner product \( \langle r_{k+1}, r_{k+1} \rangle \) which is required for the computation of the global scalar \( \alpha_{k+1} \) on the next iteration can be computed in terms of the previous inner product \( \langle r_k, r_k \rangle \) using

\[ \langle r_{k+1}, r_{k+1} \rangle = \beta_k \langle r_k, r_k \rangle \]  (3.65)

The initial inner product, \( \langle r_0, r_0 \rangle \), is computed using the GS-GB algorithm. Hence, the steps of the CG-SCG/Sl algorithm can be given as follows:

Scale A, choose \( x_0 \), let \( r_0 = p_0 = b - Ax_0 \) and compute \( \langle r_0, r_0 \rangle \).

Then, for \( k = 0, 1, 2, \ldots \)
1. Form $q_k = A p_k$ (local communication)

   (in one GS–GB communication step)

3. a. $\alpha_k = \frac{< r_k, p_k >}{< p_k, A p_k >}$
   b. $\beta_k = \alpha_k \frac{< A p_k, A p_k >}{< p_k, A p_k >} - 1$
   c. $< r_{k+1}, r_{k+1} > = \beta_k < r_k, r_k >$

4. $r_{k+1} = r_k - \alpha_k A p_k$
   $x_{k+1} = x_k + \alpha_k p_k$
   $p_{k+1} = r_{k+1} + \beta_k p_k$

Hence, the number of GS–GB communication steps is reduced from two to one per iteration of the SCG algorithm. Note that, the volume of communication does not change when compared to the basic SCG (B–SCG) algorithm, since two innerproduct values are accumulated in a single GS–GB communication step. The computational overhead per iteration is only two scalar multiplications and one scalar subtraction which is negligible. Number of divisions is reduced from two to one ($\frac{1}{< p_k, q_k >}$ is computed once).

Numerical results for a wide range of sample problems have shown that the proposed algorithm introduces no numerical instability and it requires exactly the same number of iterations to converge as the (B–SCG) algorithm.

3.4.3 A CG-SCG/s2 Algorithm

The rationale behind this formulation is to accumulate four innerproducts

\begin{align*}
\ell_1 &= < p_{2k}, A p_{2k} > \\
\ell_2 &= < A p_{2k}, A p_{2k} > \\
\ell_3 &= < A p_{2k}, A^2 p_{2k} >
\end{align*}
\[ l_4 = \langle A^2 p_{2k}, A^2 p_{2k} \rangle \]  

(3.66d)

in a single GS-GB communication step after computing two consecutive distributed matrix-vector products

\[ q_{2k} = A p_{2k} \]  

(3.67a)

\[ v_{2k} = A q_{2k} = A^2 p_{2k} \]  

(3.67b)

and then compute four global scalars \( \alpha_{2k}, \beta_{2k}, \alpha_{2k+1}, \beta_{2k+1} \) using the four inner products \( \ell_1, \ell_2, \ell_3, \) and \( \ell_4 \). The expressions for these global scalars in terms of the inner products \( \ell_1, \ell_2, \ell_3, \) and \( \ell_4 \) can be derived by evaluating (3.49a) and (3.49b) for \( s = 2, i = 0 \) and \( s = 2, i = 1 \). It is clear that the expressions for \( \alpha_{2k} \) and \( \beta_{2k} \) will be similar to the expressions for \( \alpha_k \) and \( \beta_k \) obtained in (3.58) and (3.64a), respectively, since \( i = 0 \) in both cases. Hence,

\[ \alpha_{2k} = \frac{\langle r_{2k}, r_{2k} \rangle}{\ell_1} \]  

(3.68a)

\[ \beta_{2k} = \alpha_{2k} \frac{\ell_2}{\ell_1} - 1. \]  

(3.68b)

The expression for \( \alpha_{2k+1} \) can be derived by evaluating (3.49a) for \( s = 2, i = 1 \)

\[ \alpha_{2k+1} = \frac{d_{1,0} \phi_1 + e_{1,0} \ell_1 + e_{1,1} \ell_2}{f_{1,0} \phi_2 + g_{1,0} \ell_2 + g_{1,1} \ell_3}. \]  

(3.69)

The expressions for the coefficients of the matrix polynomials \( d_1(t) \) and \( e_1(t) \) are already derived in equations (3.60a)-(3.60c). Hence, (3.69) becomes

\[ \alpha_{2k+1} = \frac{\phi_1 - \alpha_{2k} \ell_2}{f_{1,0} \phi_2 + g_{1,0} \ell_2 + g_{1,1} \ell_3}. \]  

(3.70)

The expressions for the coefficients of the matrix polynomials \( f_1(t) \) and \( g_1(t) \) can be similarly found from the recurrence relations given in (3.45c) and (3.45d) as follows:

\[ f_{1,0} = d_{1,0} + \beta_{2k} f_{0,0} = 1 \]  

(3.71a)
\[ g_{1,0} = e_{1,0} + \beta_{2k}g_{0,0} = \beta_{2k} \quad (3.71b) \]
\[ g_{1,1} = e_{1,1} + \beta_{2k}g_{0,1} = -\alpha_k . \quad (3.71c) \]

Hence, (3.70) becomes
\[ \alpha_{2k+1} = \frac{\phi_1 - \alpha_{2k}\ell_2}{\phi_2 + \beta_{2k}\ell_2 - \alpha_{2k}\ell_3} . \quad (3.72) \]

Note that, \( \phi_1 = \ell_1 \) is already found in (3.63). The expression for the only unknown term \( \phi_2 \) in (3.72) can be derived as follows:

\[
\phi_2 = \langle r_{2k}, A^2p_{2k} \rangle \\
= \langle p_{2k}, A^2p_{2k} \rangle - \beta_{2k-1} \langle p_{2k}, A^2p_{2k-1} \rangle \\
= \ell_2 - \beta_{2k-1} \langle p_{2k}, A^2p_{2k-1} \rangle \quad (3.73)
\]

since \( r_{2k} = p_{2k} - \beta_{2k-1}p_{2k-1} \) as shown in (3.62). However, from (3.20)
\[
\langle p_{2k}, A^2p_{2k-1} \rangle = \frac{-1}{\alpha_{2k-1}} \langle p_{2k}, Ap_{2k} \rangle \\
= \frac{-1}{\alpha_{2k-1}} \ell_1 . \quad (3.74)
\]

Inserting (3.74) into (3.73) we obtain
\[
\phi_2 = \ell_2 + \frac{\beta_{2k-1}\ell_1}{\alpha_{2k-1}} . \quad (3.75)
\]

The expression for \( \beta_{2k+1} \) can be similarly found by first computing the coefficients of the matrix polynomials \( d_2(t) \) and \( e_2(t) \) from (3.45a) and (3.45b) as follows:

\[
d_{2,0} = d_{1,0} - \alpha_{2k+1}f_{1,-1} = 1 \quad (3.76a) \\
d_{2,1} = d_{1,1} - \alpha_{2k+1}f_{1,0} = -\alpha_{2k+1} \quad (3.76b) \\
e_{2,0} = e_{1,0} - \alpha_{2k+1}g_{1,-1} = 0 \quad (3.76c) \\
e_{2,1} = e_{1,1} - \alpha_{2k+1}g_{1,0} = -\alpha_{2k} + \alpha_{2k+1}\beta_{2k} \quad (3.76d) \\
e_{2,2} = e_{1,2} + \alpha_{2k+1}g_{1,1} = \alpha_{2k}\alpha_{2k+1} . \quad (3.76e)
\]
Then inserting these expressions into (3.49b) for \( s = 2, i = 1 \)

\[
\beta_{2k+1} = -\alpha_{2k+1} \frac{d_{2,0}\phi_2 + d_{2,1}\phi_3 + e_{2,0}\ell_2 + e_{2,1}\ell_3 + e_{2,2}\ell_4}{\phi_1 - \alpha_{2k}\ell_2} \\
= -\alpha_{2k+1} \frac{\phi_2 - \alpha_{2k}\ell_3 - \alpha_{2k+1}(\phi_3 + \beta_{2k}\ell_3 - \alpha_{2k}\ell_4)}{\ell_1 - \alpha_{2k}\ell_2} \tag{3.77}
\]

is obtained. Note that, the expression for \( \phi_2 \) is already found in (3.75). Now the problem is to find the expression for the only unknown term, \( \phi_3 \), in (3.77). Note that,

\[
\phi_3 = \langle r_{2k}, A^3p_{2k} > \\
= \langle p_{2k}, A^3p_{2k} > - \beta_{2k-1} \langle p_{2k}, A^3p_{2k-1} > \text{ from (3.62)} \\
= \ell_3 - \beta_{2k-1} \langle p_{2k}, A^3p_{2k-1} > . \tag{3.78}
\]

Therefore, the expression for \( \langle p_{2k}, A^3p_{2k-1} > \) is required. The known term \( \phi_2 \) can be expanded as

\[
\phi_2 = \langle r_{2k}, A^2p_{2k} > \\
= \langle r_{2k-1}, A^2p_{2k} > - \alpha_{2k-1} \langle Ap_{2k-1}, A^2p_{2k} > \text{ from (3.14a)} \\
= \langle p_{2k}, A^2r_{2k-1} > - \alpha_{2k-1} \langle p_{2k}, A^3p_{2k-1} > \tag{3.79}
\]

from which

\[
\langle p_{2k}, A^3p_{2k-1} > = \frac{-1}{\alpha_{2k-1}}(\phi_2 - \langle p_{2k}, A^2r_{2k-1} >) \tag{3.80}
\]

is obtained. Now, the expression for \( \langle p_{2k}, A^2r_{2k-1} > \) is required. Using (3.62) for \( r_{2k-1} \)

\[
\langle p_{2k}, A^2r_{2k-1} > = \langle p_{2k}, A^2p_{2k-1} > - \beta_{2k-2} \langle p_{2k}, A^2p_{2k-2} > \\
= \langle p_{2k}, A^2p_{2k-1} > \text{ and from (3.74)} \\
= \frac{-1}{\alpha_{2k-1}}\ell_1 \tag{3.81}
\]

54
can be obtained. Note that, \( < p_{2k}, A^2 p_{2k-2} > = 0 \) by a) of Corollary 3.1. Finally, substituting (3.81) into (3.80) and then (3.80) into (3.78)

\[
\phi_3 = \ell_3 + \frac{\beta_{2k-1}}{\alpha_{2k-1}} (\phi_2 + \frac{1}{\alpha_{2k-1}} \ell_1)
\]  

(3.82)
is obtained. As is seen in these derivations, the global scalars, \( \alpha_{2k-1} \) and \( \beta_{2k-1} \), that are computed in the \((k-1)\)-th step of the algorithm are utilized to compute \( \phi_2 \) and \( \phi_3 \) for the \( (k) \)-th step. It can be easily observed that \( \phi_i = \ell_i \) for \( i = 0, 1, 2, 3 \) in the first step of the algorithm since \( r_0 = p_0 \). Hence, it is sufficient to initialize \( \beta_{-1} = 0 \) for the first step \( (k = 0) \) of the algorithm.

Note that, the inner product \( < r_{2k+2}, r_{2k+2} > \) which is required for the computation of the global scalar \( \alpha_{2k+2} \) on the next iteration can be computed in terms of the previous inner product \( < r_{2k}, r_{2k} > \) using

\[
< r_{2k+1}, r_{2k+1} > = \beta_{2k} \beta_{2k+1} < r_{2k}, r_{2k} > .
\]  

(3.83)

As was mentioned earlier, an extra sparse matrix vector product, \( Ar_{2k} \), is required in the given formulation. This extra matrix vector product can be avoided by noting that

\[
Ar_{2k} = A(p_{2k} - \beta_{2k-1} p_{2k-1}) \quad \text{from (3.62)}
\]

\[
= A p_{2k} - \beta_{2k-1} A p_{2k-1} .
\]  

(3.84)

Therefore, the vector \( A p_{2k+1} \) should be computed at each step. The recursive expression for the vector \( A p_{2k+1} \) in terms of the previous vector \( A p_{2k-1} \) and the already computed vectors \( A p_{2k} \) and \( A^2 p_{2k} \) can be derived as follows:

\[
A p_{2k+1} = A(r_{2k+1} + \beta_{2k} p_{2k}) \quad \text{from (3.14b)}
\]

\[
= A r_{2k+1} + \beta_{2k} A p_{2k}
\]

\[
= A(r_{2k} - \alpha_{2k} A p_{2k}) + \beta_{2k} A p_{2k} \quad \text{from (3.14a)}
\]
\[ Ar_{2k} - \alpha_{2k}A^2p_{2k} + \beta_{2k}Ap_{2k} \text{ and using (3.62)} \]
\[ = A(p_{2k} - \beta_{2k-1}p_{2k-1}) - \alpha_{2k}A^2p_{2k} + \beta_{2k}Ap_{2k} \]
\[ Ap_{2k+1} = -\beta_{2k-1}Ap_{2k-1} + (1 + \beta_{2k})Ap_{2k} - \alpha_{2k}A^2p_{2k} . \] (3.85)

The expression for the first such vector \( Ap_1 \) that should be computed in the first step \( (k = 0) \) is
\[ Ap_1 = (1 + \beta_0)A_0 - \alpha_0A^2p_0 \] (3.86)

Hence, it is sufficient to initialize the vector \( Ap_{-1} \) to zero for the first step \( (k = 0) \) of the algorithm.

Therefore, the extra sparse matrix vector product is avoided by introducing this recursive expression for the vector \( Ap_{2k+1} \) which requires only one extra vector update per SCG iteration. The basic steps of the CG-SCG/s2 algorithm are given below.

Scale \( A \), choose \( x_0 \), let \( \beta_{-1} = 0, Ap_{-1} = 0 \), and then compute
\[ r_0 = p_0 = b - Ax_0 \text{ and } < r_0, r_0 > \] Then, for \( k = 0, 1, 2, \ldots \)

1. a) \( q_{2k} = Ap_{2k} \) (local communication)
   b) \( v_{2k} = Aq_{2k} = A^2p_{2k} \) (local communication)

2. a) \( \ell_1 = < p_{2k}, q_{2k} >, \ell_2 = < q_{2k}, q_{2k} >, \ell_3 = < q_{2k}, v_{2k} >, \ell_4 = < v_{2k}, v_{2k} > \)
   (in a single GS–GB communication step).

3. a) \( \alpha_{2k} = \frac{1}{\ell_1} < r_{2k}, r_{2k} > \)
   b) \( \beta_{2k} = \alpha_{2k}\ell_1^2 - 1 \)
   c) \( \phi_2 = \ell_2 + \frac{\beta_{2k-1}\ell_1}{\alpha_{2k-1}} \)

56
d) $\alpha_{2k+1} = \frac{\phi_1 - \alpha_{2k}\ell_2}{\phi_2 - \alpha_{2k}\ell_3} + \beta_{2k}\ell_2$

e) $\phi_3 = \ell_3 + \frac{\beta_{2k-1}}{\alpha_{2k-1}}(\phi_2 + \frac{1}{\alpha_{2k-1}}\ell_1)$

f) $\beta_{2k+1} = -\alpha_{2k+1}(\phi_2 - \alpha_{2k}\ell_3 - \alpha_{2k+1}\phi_3 + \beta_{2k}\ell_3 - \alpha_{2k}\ell_4)$

g) $\langle r_{2k+1}, r_{2k+1} \rangle = \beta_{2k}\beta_{2k+1} \langle r_{2k}, r_{2k} \rangle$

4. a) $r_{2k+1} = r_{2k} - \alpha_{2k}A\ell_{2k}$

b) $p_{2k+1} = r_{2k+1} + \beta_{2k}p_{2k}$

c) $x_{2k+2} = x_{2k} + \alpha_{2k}p_{2k} + \alpha_{2k+1}p_{2k+1}$

d) $A\ell_{2k+1} = -\beta_{2k-1}A\ell_{2k-1} + (1 + \beta_{2k})A\ell_{2k} - \alpha_{2k}A^2\ell_{2k}$

e) $r_{2k+2} = r_{2k+1} - \alpha_{2k+1}A\ell_{2k+1}$

f) $p_{2k+2} = r_{2k+2} + \beta_{2k+1}p_{2k+1}$

Note that, one iteration of the above algorithm corresponds to two iterations of the basic SCG algorithm. Hence, the number of global communication steps are reduced by a factor of four, that is, from twice per iteration to once every two iterations. The scalar computational overhead of this algorithm is 12 multiplications and 10 additions/subtractions per SCG iteration which can be neglected for sufficiently large $n$, where $n = \frac{N}{P}$. However, there is also a single vector update overhead per SCG iteration, resulting in a computational overhead of $\approx \frac{1}{z+5}100\%$, where $z$ is the average number of nonzero entries per row of the A matrix. For $z=18$, where each node interacts with 8 other nodes and there are two degrees of freedom per FE node, the computational overhead is $\approx 4.4\%$. In 3D FE simulation, each node interacts with 26 other nodes and there are 3 degrees of freedom per node, thus $z = 81$. Hence, the computational overhead reduces to 1.2% in this case. This algorithm is recommended for solving sparse linear system of equations.
with \( A \) matrices having large \( z \) and for large dimensional hypercubes. This approach can be generalized for larger \( s \) values. However, methods should be devised to avoid the extra sparse matrix vector product requirement inherent in the general formulation. Furthermore, the numerical stability of the algorithm decreases with increasing \( s \).
CHAPTER IV
Parallelization of the CG-SCG/s1 Algorithm on a Hypercube Multiprocessor

4.1 Introduction

The ability of a parallel system to overlap communication with computation is an important factor affecting parallel algorithms. Another important factor affecting the design of parallel algorithms is the ability of the parallel system to overlap individual inter-processor communications. The implementation described in this chapter, achieves efficient parallelization of the \( CG-SCG/s1 \) algorithm on a hypercube multiprocessor by considering both of these points. Section 4.2 describes a local ordering scheme which enables the overlap of computation and communication during the distributed sparse matrix vector product. In Section 4.3.1, a distributed global sum algorithm that overlaps individual communication steps is presented. This algorithm is incorporated into the distributed inner product computations in place of the \( GS-GB \) algorithm. Section 4.3.2 describes communication protocols developed to exploit the two bidirectional communication links connecting adjacent processors, to overlap individual communications during the distributed sparse matrix vector product. The implementation of the parallel \( CG-SCG/s1 \) algorithm on a 16 node Intel 386-based \( iPSC2/d4 \) hypercube is discussed in Section 4.4. Experimental results are presented and discussed in Section 4.5.
4.2 Overlapping Communication with Computation

As mentioned earlier, each processor has to communicate for the non-local components of the $p$-vector in order to compute its local components of the distributed $q = Ap$ vector at each iteration. The $FE$ nodes mapped to a processor can be classified as internal nodes and boundary nodes. Internal nodes do not interact with any $FE$ node mapped to another processor. Boundary nodes interact with at least one $FE$ node which is mapped to another processor. For example, in Fig. 2, $FE$ nodes 21–28 are the internal nodes, and $FE$ nodes 17–20 and 29–32 are the boundary nodes mapped to processor $P_1$. The local components (rows) of the distributed $q$-vector ($A$-matrix) can be classified accordingly as internal and boundary components (rows). The sparse matrix vector product performed for updating the internal components of the distributed $q$-vector does not require any non-local components of the distributed $p$-vector. For example, the column indices of the non-zero entries in internal rows 21–28 of the coefficient matrix given in Fig. 2(b) are between 17 and 32, which are also the indices of the local components of the distributed $p$-vector mapped to processor $P_1$. Hence, the internal sparse matrix-vector product computation on each processor can be initiated following the asynchronous local communication steps required for the computation of the boundary sparse matrix vector product. Each processor can initiate the boundary sparse matrix vector product only after the receive operations from its neighbors are completed. This ordering is chosen to overlap the local communications required for the boundary sparse matrix vector product with the computations during the internal sparse matrix vector product. Each processor can perform this ordering for its local $FE$ nodes in parallel before each solution step.
4.3 Overlapping Communications

In iPSC2 system, there are two bit-serial and full duplex links that connect nearest neighbor processors [56]. This configuration allows simultaneous bidirectional message traffic between two nearest neighbor processors. Communication protocols that make use of the two bidirectional communication links to overlap nearest neighbor communications required for the distributed inner product and the sparse matrix vector product computations are discussed in the following sections.

4.3.1 A Distributed Exchange-Add Algorithm

As already mentioned, in order to compute the inner products at Step 2 of the CG-SCG/sl algorithm, the partial sums computed by each processor must added to form the global sums \( \langle p_k, q_k \rangle \) and \( \langle q_k, q_k \rangle \). Furthermore, since these values are needed in the next step by all processors, they must be distributed to all processors.

In the GS algorithm[15] shown in Fig. 3, the nearest-neighbor communications and the accumulation operations at the same step of the algorithm are performed concurrently. Hence, the parallel complexity of the GS algorithm can be given as

\[
T_{GS} = dS_C + dwT_C + dwT_{add}
\]  

where, \( w \) is the number of inner products to be computed in a single GS communication step, \( T_{add} \) is the addition time per floating point word, \( S_C \) is the the set-up time for single communication, and \( T_C \) is the transmission time per floating point word. Note that, (4.1) is valid only if all the processors start the GS operation at the same time which can only happen under perfectly load balanced conditions.

In the GB algorithm[15,40] shown in Fig. 4, the nearest-neighbor communications at the same step of the algorithm are performed concurrently. Hence, the parallel
complexity of the $GB$ algorithm can be given as

$$T_{GB} = dS_C + dwT_C .$$

(4.2)

The total complexity for the $GS$-$GB$ algorithm becomes

$$T_{GSGB} = 2dS_C + 2dwT_C + dwT_{add} .$$

(4.3)

Note that, most of the processors and the communication links stay idle during the $GS$-$GB$ operation. Furthermore, only one of the two communication links between the neighbor processor are utilized at any instant of the $GS$-$GB$ algorithm.

An alternative algorithm, the $Exchange-Add$ algorithm, is illustrated in Fig. 6. The main idea in this algorithm is that, $P$ copies of the inner product are accumulated in $P$ processors by running $P$ concurrent $GS$ algorithms with $P$ different root processors instead of only one root processor accumulating the sum and then broadcasting. Let the processors of a $d$-dimensional hypercube be represented by a $d$-bit binary number ($b_{d-1}...b_0$). Also, define channel-$i$ as the set of $(2^{d-1})$ bidirectional communication links connecting two neighbor processors whose representation only differ in bit position-$i$. Then, the steps of the $Exchange-Add$ algorithm for $w = 1$ can be given as follows:

Initially, each processor has its own partial sum.

step-i : for $i = 0, \ldots, d - 1$

1. processors $P(b_{d-1}...b_i+1b_{i-1}...b_0)$ and $P(b_{d-1}...b_i+1b_{i-1}...b_0)$ concurrently exchange their most recent partial sum over channel-$i$.

2. each processor computes its new partial sum by adding the partial sum it received over channel-$i$ to its most recent partial sum.

At the end of $d$ concurrent exchange communication steps, each processor has its own copy of the global sum.
Figure 6: Distributed Exchange-Add algorithm on a 4-dimensional hypercube
In this algorithm, the pair-wise communications and the accumulation operations at the same step of the algorithm are performed concurrently. Furthermore, set-up times for individual communications between each pair of communicating processors are completely overlapped. The actual transmission of $w$ partial sums in two directions in an exchange operation is also completely overlapped when two physical links are present. Hence, the parallel complexity of the $EA$ algorithm is

$$T_{EA} = dS_C + dwT_C + dwT_{add}$$

(4.4)

when two physical links are present. Note that, from (4.4) and (4.3)

$$T_{diff} = T_{GSGB} - T_{EA}$$

$$= dS_C + dwT_C$$

(4.5)

can be obtained. In this expression, $dS_C$ accounts for the overlap in the set-up times for individual communications and $dwT_C$ accounts for the overlap in the transmission of the partial sums when two links are present.

It is clear from the comparison of (4.4) and (4.3) that $EA$ algorithm effectively reduces the number of communications by a factor of two. In fact, using the experimentally measured values of $S_C = 970 \mu s$ in the Intel $iPSC2$ system (Release 1.0), the term $dwT_{add}$ is negligible compared to $dS_C$ in (4.4) and $2dS_C$ in (4.3) for small values of $w$. Hence,

$$T_{EA} \sim \frac{1}{2}T_{GSGB}$$

(4.6)

can be obtained for small values of $w$. In the absence of two physical links, assuming that the set-up times can be overlapped, the complexity of the $EA$ algorithm becomes

$$T_{EA} = dS_C + 2dwT_C + dwT_{add}$$

(4.7)
which is still much better than $T_{Q}$. Note that, (4.6) will be still valid for a hypercube with high communication latency, since the term $dwT_C$ will be negligible compared to $dS_C$ in (4.7) and $2dS_C$ in (4.3) for small values of $w$.

4.3.2 Communication Protocols for Strip Partitioning Schemes

In the 1-D strip partitioning scheme, each processor except the first and the last one in the linear array, has to communicate with its right and left neighbors at each iteration in order to update its local components of the distributed $q$-vector. This communication protocol utilizes the two serial communication links to overlap the nearest neighbor communications required for the distributed sparse matrix vector product. The processors of the hypercube are ordered in graycode ordering in order to achieve nearest neighbor communications. Fig. 7 illustrates the 2 concurrent communication phases of the communication protocol for 1-D strip partitioning on a 3-dimensional hypercube. As seen from this figure, the communications between pairs of processors at the same phase of the algorithm are performed concurrently, provided that processors start each communication phase at the same time. The set-up times for individual communications between each pair of communicating processors are completely overlapped at each phase of the algorithm. The actual transmission times over the two communication links between each pair of communicating processors are also completely overlapped at each phase of the algorithm. Hence, referring to Fig. 5(a), the parallel complexity for this communication protocol is

$$T_{1D} = 2S_C + 4mT_C$$  \hspace{1cm} (4.8)

for an $m \times m$ square FE mesh with two variables associated with each FE node.

In the 2-D strip partitioning scheme, each internal processor in the processor mesh has to communicate with its right, left, up, and bottom neighbors. A two
Figure 7: Communication protocol for 1-D strip partitioning on iPSC2/d3
dimensional $P_H \times P_V$ processor array is constructed using a gray code ordering in both horizontal and vertical directions in order to achieve nearest neighbor communications, where

$$P_H = 2^{[d/2]} \quad \quad (4.9a)$$

$$P_V = 2^{[d/2]} \quad \quad (4.9b)$$

Fig. 8 illustrates the 4 concurrent nearest neighbor communication phases of this protocol on a 4-dimensional hypercube with $P_H = P_V = 4$. Note that, this protocol also utilizes the two serial communication links to overlap the nearest neighbor communications. Hence, referring to Fig. 5(b), the parallel complexity for the 2-D communication protocol is

$$T_{2D} = 4SC + (8 + \frac{4m}{P_H} + \frac{4m}{P_V})TC \quad \quad (4.10)$$

for $P_H > 2$, and $P_V > 2$.

As mentioned earlier, 1.5-D strip partitioning is a special case of 2-D partitioning for $P_V = 2$. Fig. 9 illustrates the 3 concurrent communication phases for this protocol on a 4-dimensional hypercube with $P_H = 2$ and $P_V = 8$. Hence, referring to Fig. 5(c), the parallel complexity for the 1.5-D communication protocol is

$$T_{1.5D} = 3SC + (4 + 2m + \frac{4m}{P})TC \quad \quad (4.11)$$

Note that, (4.8), (4.10), and (4.11) are valid only if processors start to communicate for each phase at the same time. This condition is satisfied under perfectly load balanced conditions.

The relative merit of one scheme over the other is a function of $P$, $SC$ and $TC$ and $m$ as follows [57]:

67
Figure 8: Communication protocol for 2-D strip partitioning on iPSC2/d4
Figure 9: Communication protocol for 1.5-D strip partitioning on iPSC2/d4
\[ T_{1D} < T_{2D} \text{ iff } m < \left( \frac{SC}{2I_{GC}} + 2 \right) \times \frac{1}{(1-1/P_H-1/P_V)} \]
\[ T_{1D} < T_{1.5D} \text{ iff } m < \left( \frac{SC}{2I_{GC}} + 2 \right) \times \frac{1}{(1-2/P)} \]
\[ T_{1.5D} < T_{2D} \text{ iff } m < \left( \frac{SC}{2I_{GC}} + 2 \right) \times \frac{1}{(1+2/P-2/P_H-2/P_V)} \] \hspace{1cm} (4.12)

For the case of a 16 processor hypercube system, \( P = 16, P_H = P_V = 2^{4/2} = 4, \)
\[ T_{1D} < T_{2D} \text{ iff } m < 2\left( \frac{SC}{2I_{GC}} + 2 \right) \]
\[ T_{1D} < T_{1.5D} \text{ iff } m < 8\left( \frac{SC}{2I_{GC}} + 2 \right) \]
\[ T_{1.5D} < T_{2D} \text{ iff } m < 8\left( \frac{SC}{2I_{GC}} + 2 \right) \] \hspace{1cm} (4.13)
is obtained. From these inequalities, it is concluded that for \( P = 16, P_H = P_V = 4 \)
the optimal approach is

\[
\begin{cases}
1-D & \text{strip partitioning, for } m < 8\left( \frac{SC}{2I_{GC}} + 2 \right) \\
1.5-D & \text{strip partitioning, for } 8\left( \frac{SC}{2I_{GC}} + 2 \right) < m < 8\left( \frac{SC}{2I_{GC}} + 2 \right) \\
2-D & \text{strip partitioning, for } m > 8\left( \frac{SC}{2I_{GC}} + 2 \right)
\end{cases}
\] \hspace{1cm} (4.14)

Using experimentally measured values of \( SC = 970 \mu s \) and \( TC = 2.88 \mu s \) per
double precision number for the iPSC2 (Release 1.0), it can be deduced from
Equation (4.14) that the 1-D approach is superior to the other two for \( m < 194 \).
This value of \( m \) is well above mesh sizes of interest in the context of a practically
realistic FE solution. While the analysis summarized in Equation (4.14) considered
a specific shape of a FE graph, it provides a good estimate of the order of magnitude
of the FE graph size that makes the 1.5-D or 2-D approaches worth using for a
parallel FE solver on the Intel iPSC2. As a consequence, only the communication
protocol required by 1-D strip partitioning was actually implemented on the Intel
iPSC2 system for the distributed sparse matrix vector product.
4.4 Implementation of the CG-SCG/s1 Algorithm on the iPSC2/d4

This section describes the implementation of the parallel SCG algorithm on a 16-node Intel 386-based iPSC2/d4 hypercube.

4.4.1 Distributed Sparse Matrix Vector Product

The communication protocol given in Section 4.3.2 is implemented to overlap the nearest neighbor communications required in the 1-D partitioning. The communication steps involved at each processor during this distributed algorithm are given below.

if my processor number is even in the graycode ordering then
  irecv \( (p_k^j \in \{\text{my right neighbor's left boundary}\}) \) from my right neighbor
  isend \( (p_k^j \in \{\text{my right boundary}\}) \) to my right neighbor
  irecv \( (p_k^j \in \{\text{my left neighbor's right boundary}\}) \) from my left neighbor
  isend \( (p_k^j \in \{\text{my left boundary}\}) \) to my left neighbor
else if my processor number is odd in the graycode ordering then
  irecv \( (p_k^j \in \{\text{my left neighbor's right boundary}\}) \) from my left neighbor
  isend \( (p_k^j \in \{\text{my left boundary}\}) \) to my left neighbor
  irecv \( (p_k^j \in \{\text{my right neighbor's left boundary}\}) \) from my right neighbor
  isend \( (p_k^j \in \{\text{my right boundary}\}) \) to my right neighbor
end if

Here, isend and irecv denote asynchronous (non-blocking) send and asynchronous receive operations, respectively. The node executive \((NX/2)\) of the iPSC2 handles short messages \((\leq 100\) bytes) and long messages \((> 100\) bytes) differently. Short incoming messages are always stored first in a buffer inside the \((NX/2)\) area regardless of a pending receive operation for that message and then copied from the \((NX/2)\) buffer to the user buffer. Long incoming messages are directly copied to

71
the user buffer if a receive operation is already pending for that message. If not, the message is kept in the $NX/2$ buffer until a receive operation is issued for that message. The 1-D strip partitioning of practical FE meshes ($m > 6$) always result in the communication of long messages. Hence, each processor of a pair coupled to exchange data issues an asynchronous send after initiating an asynchronous receive operation. This is done to ensure that a receive is already pending for the incoming long message so that it can be directly copied into the user buffer area instead of being copied to the $NX/2$ area and then transferred to the indicated user buffer due to a late issued receive.

The internal sparse matrix vector product computation is started right after the initiation of these asynchronous send/receive operations. Synchronization of the two asynchronous receive operations is performed at the completion of the internal sparse matrix vector product. The boundary sparse matrix vector product is initiated after this synchronization step. The synchronization of the asynchronous send operations are delayed until the distributed vector update operations at Step 4 of the algorithm. However, the overlap of communication and computation in this step is measured to be below 5% due to the internal architecture of the processors of the iPSC2. Significant overlap in individual nearest neighbor communications is measured in this implementation. However, the estimated bound given in Equation (4.8) is found to be overly optimistic in assuming that complete overlap of data transmission during simultaneous sends/receives is possible. In practice, contention for an internal bus during two outgoing and two incoming long messages results in a lower transmission rate.
4.4.2 Distributed Scaling

As mentioned earlier, the coefficient matrix $A$ is preconditioned before each solution step using the diagonal scaling

$$\tilde{A} = D^{-1/2}AD^{-1/2}$$  \hspace{1cm} (4.15)

to increase the convergence rate of the $CG$ algorithm. Each processor contains only the diagonal entries of the rows (of the distributed $A$-matrix) mapped to itself. Note that, only the local components of the distributed diagonal $D$-matrix are involved in the distributed pre-multiplication of the sparse $A$-matrix with $D^{-1/2}$. However, the non-local components of the distributed diagonal $D$-matrix are required for the distributed post-multiplication of the sparse $A$-matrix with $D^{-1/2}$. Hence, the same communication protocol given for the 1-D strip partitioning is implemented for each processor to communicate for its non-local components of the distributed diagonal $D$-matrix.

4.4.3 Distributed Inner Products

The $EA$ algorithm described in the Section 4.3.1 is implemented to compute the two inner products at Step 2 of the $CG$-$SCG/s1$ algorithm. The communication steps involved at each processor during this distributed algorithm are given below for $w = 1$.

Initially, each processor accumulates its partial sum corresponding to its local components of the distributed vector(s) in $myparsum$. Then,

for $i = 0, 1, \ldots, d - 1$

isend ($myparsum$) to my neighbor on channel-$i$

creceive ($hisparsum$) from my neighbor on channel-$i$

wait (isend)
\[ \text{myparsum} \leftarrow \text{myparsum} + \text{hisparsum} \]

end for

\[ \text{gsum} \leftarrow \text{myparsum} \]

Here, \text{creceive} denotes a synchronous (blocking) receive operation. The volume of communication during the \(d\) concurrent nearest neighbor communication steps of the \text{EA} operation in the \text{CG-SCG/s1} algorithm is only 16 bytes (2 DP words). As mentioned earlier, short messages are always stored first into the \(NX/2\) buffer regardless of a pending receive operation. Hence, in the implementation of the \text{EA} algorithm on the \text{iPSC2}, each processor of a pair coupled to exchange data initiate an asynchronous send operation before the synchronous receive operation in order to prevent the delay of the send operation. The updating of the partial sum is delayed until the completion of the asynchronous send operation.

4.4.4 Distributed Vector Updates

Each processor of the hypercube computes its own copies of the global scalars \(\alpha\) and \(\beta\) at Step 3 of the algorithm in terms of the two inner products computed at Step 2 of the \text{CG-SCG/s1} algorithm. Then, at Step 4 of the algorithm, each processor updates its own slices of the distributed \(x\), \(r\), and \(p\) vectors, without any inter-processor communication using these global scalars.

4.5 Experimental Results and Discussions

This section presents overall performance results for the parallel \text{CG-SCG/s1} algorithm on the Intel \text{iPSC2/d4} hypercube. The \text{CG-SCG/s1} algorithm is used for the solution phase of the \text{FE} package \text{ALPID} (Analysis of Large Plastic Incremental Deformation). \text{ALPID} requires the solution of a sequence of systems of linear subproblems, at each time step, to produce a sequence of iteration vectors. The
iteration vectors converge to the solution of the nonlinear system of equations [12]. Each of these linear subproblems involves the determination of a matrix representing the linearized version of the nonlinear system of equations and the solution of the generated matrix equation to produce the next iteration value. The details of the implementation and some of its applications are available in [58,59,60].

The tolerance value $\varepsilon = 10^{-5}$ is used for halting the $SCG$ iterations. In the first solution phase of each time step, the zero vector is used as the initial guess vector. However, in the succeeding solution steps, the solution of the previous step is used as the initial guess vector. Hence, the number of iterations required for convergence decreases in the succeeding solution steps of a time step. The experimental results given in this section use a zero initial guess vector.

**4.5.1 Comparison of the B—SCG and the CG—SCG Algorithms**

Table 1 presents the solution times of the $B-SCG$ and the $CG-SCG/s1$ algorithms with $GS-GB$ for different size $FE$ simulation problems on a 4-dimensional hypercube $iPSC2/d4$. Fig. 10 illustrates the percent performance improvement both as a function of problem size and hypercube dimension. Here, the percent performance improvement, $\eta_1$ is calculated as

$$\eta_1 = \frac{T_{BSCG(GSGB)} - T_{CGSCG(GSGB)}}{T_{CGSCG(GSGB)}} \cdot 100\%$$  \hspace{1cm} (4.16)$$

where $T_{BSCG(GSGB)}$ denotes the time taken by the $B-SCG$ algorithm with $GS-GB$, and $T_{CGSCG(GSGB)}$ denotes the time taken by the $CG-SCG/s1$ algorithm with $GS-GB$. As was mentioned in Section 3.4.2, $CG-SCG/s1$ algorithm introduces no computational overhead and reduces the number of $GS-GB$ communication steps by a factor of two without changing the volume of communication. Hence,
Table 1: Solution times (per iteration) of B-SCG and CG-SCG/al algorithms with GS-GB on iPSC2/d4 for different size FE simulation problems

<table>
<thead>
<tr>
<th>Test Prob</th>
<th>No. of Vars</th>
<th>No. of Iters</th>
<th>iPSC2/d1 sol. time per iter(ms)</th>
<th>iPSC2/d2 sol. time per iter(ms)</th>
<th>iPSC2/d3 sol. time per iter(ms)</th>
<th>iPSC2/d4 sol. time per iter(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Basic SCG</td>
<td>CG SCG</td>
<td>Basic SCG</td>
<td>CG SCG</td>
</tr>
<tr>
<td>T1</td>
<td>550</td>
<td>72</td>
<td>92.8</td>
<td>91.1</td>
<td>56.8</td>
<td>54.1</td>
</tr>
<tr>
<td>T2</td>
<td>734</td>
<td>99</td>
<td>122.8</td>
<td>120.9</td>
<td>70.9</td>
<td>68.0</td>
</tr>
<tr>
<td>T3</td>
<td>1175</td>
<td>130</td>
<td>200.5</td>
<td>197.8</td>
<td>109.5</td>
<td>106.5</td>
</tr>
<tr>
<td>T4</td>
<td>1952</td>
<td>165</td>
<td>-</td>
<td>-</td>
<td>177.1</td>
<td>173.3</td>
</tr>
<tr>
<td>T5</td>
<td>2143</td>
<td>201</td>
<td>-</td>
<td>-</td>
<td>190.9</td>
<td>187.1</td>
</tr>
<tr>
<td>T6</td>
<td>3120</td>
<td>126</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T7</td>
<td>4752</td>
<td>297</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Figure 10: Percent performance improvement of \textit{CG-SCG/s1} over \textit{B-SCG} (a) as a function of problem size (b) as a function of hypercube dimension.
the expression for $\eta_1$ under perfectly load balanced conditions is

$$\eta_1 = \frac{2dS_C}{TC_{CGSCG(GSGB)}} \cdot 100\% \tag{4.17}$$

$$\eta_1 = \frac{2dS_C}{T_{comp} + 2dS_C + 4dT_C + L_C} \cdot 100\% \tag{4.18}$$

where $T_{comp} = (z+5)N\frac{K}{T_{calc}}$ is the total computation time per SCG iteration, and $L_C = T_{1D}$ is the total local communication time required per distributed sparse matrix vector product. As seen in Fig. 10(a), $\eta_1$ decreases monotonically with increasing problem size $N$ since the ratio of $2dS_C$ to the total solution time per iteration of the $CG-SCG/s1$ algorithm decreases with increasing problem size (for fixed $d$). For example, on a 4-dimensional ($iPSC2/d4$) hypercube, $\eta$ monotonically decreases from 24% for the smallest test problem T1 to 5% for the largest test problem T7. It can also be observed from Fig. 10(b) that $\eta_1$ increases exponentially with increasing hypercube dimension. For example, for the smallest size test problem T1, $\eta$ increases exponentially from 2% on $iPSC2/d1$ to 24% on $iPSC2/d4$. This is because, $T_{comp}$ in the denominator of (4.18) decreases exponentially with the dimension, as the domain is divided among $2^d$ processors and the only term in the numerator of (4.18) increases linearly with increasing dimension. The local communication term $L_C$ in the denominator of (4.18) is independent of the hypercube dimension for 1-D strip partitioning as given in equation (4.8). Hence, the proposed $CG-SCG/s1$ algorithm is expected to result in significant performance improvement for large dimensional hypercubes.

### 4.5.2 Comparison of GS–GB and EA

Table 2 presents distributed global sum computation times for GS–GB and EA algorithms on the $iPSC2/d4$ hypercube. The measured performance for the GS–GB algorithm is found to be within 10% of the estimated performance given in
Table 2: Distributed global sum computation times for \textit{GS-GB} and \textit{EA} algorithms

<table>
<thead>
<tr>
<th>No. of DP words</th>
<th>\textit{iPSC2}/d1 global sum time (ms)</th>
<th>\textit{iPSC2}/d2 global sum time (ms)</th>
<th>\textit{iPSC2}/d3 global sum time (ms)</th>
<th>\textit{iPSC2}/d4 global sum time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GSGB</td>
<td>EA</td>
<td>GSGB</td>
<td>EA</td>
</tr>
<tr>
<td>1</td>
<td>1.25</td>
<td>0.81</td>
<td>2.45</td>
<td>1.58</td>
</tr>
<tr>
<td>2</td>
<td>1.28</td>
<td>0.84</td>
<td>2.48</td>
<td>1.63</td>
</tr>
<tr>
<td>3</td>
<td>1.31</td>
<td>0.86</td>
<td>2.52</td>
<td>1.66</td>
</tr>
<tr>
<td>4</td>
<td>1.33</td>
<td>0.89</td>
<td>2.55</td>
<td>1.69</td>
</tr>
<tr>
<td>6</td>
<td>1.38</td>
<td>0.94</td>
<td>2.62</td>
<td>1.76</td>
</tr>
<tr>
<td>8</td>
<td>1.44</td>
<td>1.00</td>
<td>2.69</td>
<td>1.84</td>
</tr>
<tr>
<td>10</td>
<td>1.49</td>
<td>1.06</td>
<td>2.77</td>
<td>1.91</td>
</tr>
<tr>
<td>12</td>
<td>1.54</td>
<td>1.11</td>
<td>2.85</td>
<td>1.98</td>
</tr>
<tr>
<td>13</td>
<td>2.45</td>
<td>1.84</td>
<td>4.65</td>
<td>3.42</td>
</tr>
<tr>
<td>15</td>
<td>2.50</td>
<td>1.89</td>
<td>4.72</td>
<td>3.49</td>
</tr>
<tr>
<td>20</td>
<td>2.63</td>
<td>2.03</td>
<td>4.91</td>
<td>3.68</td>
</tr>
<tr>
<td>30</td>
<td>2.93</td>
<td>2.31</td>
<td>5.36</td>
<td>4.06</td>
</tr>
</tbody>
</table>
equation (4.3). However, the measured performance of the EA algorithm varies between $1.3T_{EA}^{est}$ and $1.5T_{EA}^{est}$ where the estimated complexity $T_{EA}^{est}$ is given in Equation (4.4). This discrepancy can be attributed to the software overhead associated with asynchronous receive operations and internal bus conflicts for long messages. Fig. 11 illustrates the performance of the GS-GB and EA algorithms with respect to both the number of double precision (DP) words ($w$) added and the dimension of the hypercube. As mentioned earlier, the node executive $(NX/2)$ of the iPSC2 handles short messages ($\leq 100$ bytes) and long messages ($> 100$ bytes) differently. Short messages are routed directly, whereas extra handshaking is performed between two communicating processors to establish the circuit for the transmission of a long message. This extra overhead causes the set-up time to increase from $S_C = 550\mu s$ to $S_C = 970\mu s$ for long messages. This explains the jumps in the curves given in Fig. 11(a) at $w = 13$ (104 bytes). This also explains the jump in $T_{diff} = T_{GSGB} - T_{EA}$ at $w = 13$ since the expression for estimated $T_{diff}$ contains the term $dS_C$ as given in equation (4.5). Since $T_C$ is only $2.88\mu s$ per DP word, the increase in measured $T_{diff}$ with increasing $w$, which was expected due to the term $dwT_C$ in the expression for estimated $T_{diff}$ given in (4.5), is hardly seen in Fig. 11(a). Fig. 11(b) illustrates the linear increase in the measured $T_{diff}$ with increasing dimension which was expected due to the term $dS_C$ in the expression for estimated $T_{diff}$.

Table 3 presents the solution times (per iteration) of the B-SCG algorithm using the GS-GB and the CG-SCG/s1 algorithm using the EA on the iPSC2/d4 hypercube for different size FE simulation problems. Fig. 12 illustrates the percent performance improvement, $\eta_2$, obtained by the incorporation of the EA algorithm for the distributed inner product computations of the CG-SCG/s1 algorithm in place of the GS-GB algorithm. It can be observed by comparing Fig. 12 and
Figure 11: Performance of GS-GB and EA algorithms on iPSC2/d4 (a) as a function of w (b) as a function of hypercube dimension
Table 3: Solution times (per iteration) of B-SCG algorithm with GS-GB and CG-SCG/sl algorithm with EA on iPSC2/d4 for different size FE simulation problems

<table>
<thead>
<tr>
<th>Test Prob</th>
<th>No. of Vars</th>
<th>No. of Iters</th>
<th>iPSC2/d1 sol. time per iter(ms)</th>
<th>iPSC2/d2 sol. time per iter(ms)</th>
<th>iPSC2/d3 sol. time per iter(ms)</th>
<th>iPSC2/d4 sol. time per iter(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Basic SCG</td>
<td>CG SCG</td>
<td>Basic SCG</td>
<td>CG SCG</td>
</tr>
<tr>
<td>T1</td>
<td>550</td>
<td>72</td>
<td>92.8</td>
<td>90.3</td>
<td>56.8</td>
<td>52.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>734</td>
<td>99</td>
<td>122.8</td>
<td>120.1</td>
<td>70.9</td>
<td>66.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>1175</td>
<td>130</td>
<td>200.5</td>
<td>197.3</td>
<td>109.5</td>
<td>104.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>1952</td>
<td>165</td>
<td></td>
<td></td>
<td>177.1</td>
<td>171.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>2143</td>
<td>201</td>
<td></td>
<td></td>
<td>190.9</td>
<td>185.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>3120</td>
<td>126</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T7</td>
<td>4752</td>
<td>297</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

82
Figure 12: Percent performance improvement of EA over GS-GB in the CG-SCG/s1 Algorithm (a) as a function of problem size (b) as a function of hypercube dimension.
Fig. 10, that experimental curves for $\eta_2$ are very similar to the curves for $\eta_1$, except for the fact that they are scaled down by factors slightly smaller than two. This was expected because the expression for $\eta_2$, under perfectly load balanced conditions, is

$$\eta_2 = \frac{T_{CGSCG(GSGB)} - T_{CGSCG(EA)}}{T_{CGSCG(EA)}} \cdot 100\%$$

$$= \frac{T_{GSGB} - T_{EA}}{T_{CGSCG(GSGB)} - (T_{GSGB} - T_{EA})} \cdot 100\% \text{ and from (4.5)}$$

$$= \frac{dSC + 2dT_C}{T_{CGSCG(GSGB)} - dSC - 2dT_C} \cdot 100\%$$

$$\sim \frac{\eta_1}{2 - \eta_1} \text{ and from (4.17)}$$

(4.19)

Note that, the term $2dT_C$ is negligible compared to $dSC$ since $2T_C = 5.76\mu s \ll 970\mu s = S_C$ for iPSC2.

Fig. 13 illustrates the overall percent performance improvement obtained by implementing $CG-SCG/sI$ algorithm with $EA$ in place of the $B-SCG$ algorithm with $GS-GB$. It can be seen that, $CG-SCG/sI$ algorithm with $EA$ results in higher performance improvements for small grain problems and large dimensional hypercubes as expected. An improvement of 40% is obtained for the smallest size test problem $T1$ on a 4-dimensional hypercube.

### 4.5.3 Experimental Performance

Table 4 presents solution times (per iteration) of the sequential $B-SCG$ algorithm on the $\mu$VAX II and on the $iPSC2/d0$, and the parallel $CG-SCG/sI$ algorithm on the $iPSC2/d1-d4$ for different size $FE$ simulation problems. Since, the $CG-SCG/sI$ algorithm introduces almost no overhead over the $B-SCG$ algorithm, sequential complexities of these two algorithms are almost equal. Experimental speed-up
Figure 13: Percent performance improvement of $CG-SCG/s1$ (EA) over $B-SCG$ ($GS-GB$) (a) as a function of problem size (b) as a function of hypercube dimension.
and efficiency curves are illustrated in Fig. 14 and Fig. 15, respectively. Speed-up and efficiency are calculated from

\[
\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}} \quad (4.20)
\]

\[
\text{Efficiency} = \frac{\text{Speedup}}{P} \quad (4.21)
\]

It can be seen from Fig. 14(a) and Fig. 15(a) that, speed-up and efficiency increase with increasing problem size as was expected. It can also be seen from Fig. 14(b) and Fig. 15(b) that the implementation is scalable and that an almost linear speed-up is achieved for larger problems (T7, T6, and T5).

Table 5 presents the measured performance in MFLOPS of the B-SCG algorithm on the µVAX II and on the iPSC2/d0, and the CG-SCG/s1 algorithm on the iPSC2/d1-d4 for different size FE problems. The sustained performance, \( P_s \), in MFLOPS, for a particular test problem of size \( N \) is calculated from

\[
P_s(N) = \frac{2(z + 5)N}{T_{sol}} \text{MFLOPS} \quad (4.22)
\]

where \( T_{sol} \) is the measured solution time per iteration in \( \mu \text{secs} \). Fig. 16 illustrates the measured performance of the B-SCG algorithm on the µVAX II and the CG-SCG/s1 algorithm on the iPSC2/d1-d4 both as a function of problem size and as a function of hypercube dimension. It can be seen that almost 15 times speed-up over a µVAX II is obtained on a 16-node iPSC2/d4 hypercube for the larger test problems (T7, T6, and T5). It can be also seen that, a performance of 2 MFLOPS is achieved on the iPSC2/d4 for the larger test problems (T7, T6, and T5).
Table 4: Solution times (per iteration) of B-SCG on $\mu$VAX II and on iPSC2/d0, and CG-SCG/d1 on iPSC2/d1-d4 for different size FE simulation problems

<table>
<thead>
<tr>
<th>Test Prob</th>
<th>No of Vars</th>
<th>No. of Iters for Conv</th>
<th>$\mu$VAX II sol time per iter (ms)</th>
<th>d0 sol time per iter (ms)</th>
<th>d1 sol time per iter (ms)</th>
<th>d2 sol time per iter (ms)</th>
<th>d3 sol time per iter (ms)</th>
<th>d4 sol time per iter (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>550</td>
<td>72</td>
<td>179.4</td>
<td>174.6</td>
<td>90.3</td>
<td>52.4</td>
<td>29.7</td>
<td>18.9</td>
</tr>
<tr>
<td>T2</td>
<td>734</td>
<td>99</td>
<td>238.6</td>
<td>233.0</td>
<td>120.1</td>
<td>66.2</td>
<td>36.5</td>
<td>22.3</td>
</tr>
<tr>
<td>T3</td>
<td>1175</td>
<td>130</td>
<td>385.0</td>
<td>374.5</td>
<td>197.3</td>
<td>104.4</td>
<td>55.4</td>
<td>31.6</td>
</tr>
<tr>
<td>T4</td>
<td>1952</td>
<td>165</td>
<td>672.8</td>
<td>629.7</td>
<td>-</td>
<td>171.1</td>
<td>88.8</td>
<td>48.0</td>
</tr>
<tr>
<td>T5</td>
<td>2143</td>
<td>201</td>
<td>695.6</td>
<td>681.8</td>
<td>-</td>
<td>185.0</td>
<td>95.9</td>
<td>51.4</td>
</tr>
<tr>
<td>T6</td>
<td>3120</td>
<td>126</td>
<td>1021.8</td>
<td>1007.0</td>
<td>-</td>
<td>-</td>
<td>136.8</td>
<td>72.3</td>
</tr>
<tr>
<td>T7</td>
<td>4752</td>
<td>297</td>
<td>1552.0</td>
<td>1538.1</td>
<td>-</td>
<td>-</td>
<td>207.6</td>
<td>107.3</td>
</tr>
</tbody>
</table>
Figure 14: Speed-up curves (a) as a function of problem size (b) as a function of hypercube dimension
Figure 15: Efficiency curves (a) as a function of problem size (b) as a function of hypercube dimension
Table 5: MFLOPS performance of B-SCG on μVAX II and on iPSC2/d0, and CG-SCG/s1 on iPSC2/d1-d4 for different size FE simulation problems

<table>
<thead>
<tr>
<th>Test Prob</th>
<th>No of Vars</th>
<th>Mesh Size</th>
<th>μVAX II MFLOPS</th>
<th>d0 MFLOPS</th>
<th>d1 MFLOPS</th>
<th>d2 MFLOPS</th>
<th>d3 MFLOPS</th>
<th>d4 MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>550</td>
<td>15 x 20</td>
<td>0.137</td>
<td>0.141</td>
<td>0.272</td>
<td>0.469</td>
<td>0.827</td>
<td>1.300</td>
</tr>
<tr>
<td>T2</td>
<td>734</td>
<td>11 x 36</td>
<td>0.137</td>
<td>0.141</td>
<td>0.273</td>
<td>0.495</td>
<td>0.898</td>
<td>1.470</td>
</tr>
<tr>
<td>T3</td>
<td>1175</td>
<td>25 x 25</td>
<td>0.139</td>
<td>0.143</td>
<td>0.272</td>
<td>0.514</td>
<td>0.969</td>
<td>1.669</td>
</tr>
<tr>
<td>T4</td>
<td>1952</td>
<td>32 x 32</td>
<td>0.134</td>
<td>0.143</td>
<td>-</td>
<td>0.527</td>
<td>1.016</td>
<td>1.879</td>
</tr>
<tr>
<td>T5</td>
<td>2143</td>
<td>33 x 33</td>
<td>0.141</td>
<td>0.144</td>
<td>-</td>
<td>0.538</td>
<td>1.038</td>
<td>1.937</td>
</tr>
<tr>
<td>T6</td>
<td>3120</td>
<td>40 x 40</td>
<td>0.141</td>
<td>0.144</td>
<td>-</td>
<td>-</td>
<td>1.064</td>
<td>2.013</td>
</tr>
<tr>
<td>T7</td>
<td>4752</td>
<td>49 x 49</td>
<td>0.143</td>
<td>0.145</td>
<td>-</td>
<td>-</td>
<td>1.074</td>
<td>2.080</td>
</tr>
</tbody>
</table>
Figure 16: Performance of the sequential B-SCG (μVAX II, iPSC2/d0) and the parallel CG-SCG/s1 (iPSC2/d1-d4) (a) as a function of problem size (b) as a function of hypercube dimension
CHAPTER V

Vectorization and Parallelization of the CG–SCG/s1 Algorithm on a Vector Hypercube Multiprocessor

5.1 Introduction

The computations in the SCG algorithm consist mainly of matrix operations that can be vectorized. The Intel vector hypercube iPSC-VX has a Vector Processor (VP) attached to each node. In this chapter, an implementation of the CG–SCG/s1 algorithm that exploits both vectorization and parallelization is described [61,62]. The microprogramming techniques on the VP are discussed in Section 5.2.1. Section 5.2.2 describes the vectorization of the matrix operations in the CG–SCG/s1 algorithm. The node-to-node communication architecture of the Intel 286-based iPSC hypercube precludes the possibility of overlap between individual communications, as described in Section 5.3. Sections 5.3.1 and 5.3.3 describe the redesign of the communication protocols for the distributed sparse matrix vector product and the inner product computations respectively. Section 5.3.2 presents an efficient local ordering scheme that avoids the indexing overhead in the communications for the distributed sparse matrix vector product. Experimental results for vectorization and parallel implementation on a 4 node Intel iPSC–VX/d2 vector hypercube are presented and discussed in Section 5.4.
5.2 Vectorization of the CG–SCG/s1 Algorithm on the iPSC–VX

As described earlier, each processor of the hypercube has to perform the following vector operations to compute its local components of the distributed vectors per SCG iteration: one sparse matrix vector product, two inner products, and three vector updates. These operations are very suitable for vectorization. A vector processor (VP) board manufactured by Sky Computer Inc. is tightly coupled to each node processor of the iPSC–VX (Vector eXtension) via MULTIBUS II iLBX. Successful vectorization of an algorithm requires attention to architectural features of the processor on which the vectorization is being done.

Fig. 17 illustrates the basic architecture of an iPSC–VX computational node. The 80286–based node processor board serves as a general purpose microcomputer. It contains 512 Kbytes of local memory and hosts a small message–based node executive called NX. The node processor with its NX is primarily responsible for coordinating message traffic into and out of the node, for scheduling and executing user processes and for controlling its companion VP. Another feature of the iPSC–VX node architecture is the dual–ported access to the memory on the VP board, which is shared between the node CPU and the VP. All user data is placed on the VP board where it is accessible to both the 80286 and the VP. Fig. 18 illustrates the iPSC–VX VP organization. Vector supercomputers like Cray, Cyber 205, etc., have special units that perform vector operations [63,64]. In the VP, a number of scalar hardware units with very low number of pipe stages are provided to achieve functional parallelism. The independence of these functional units makes possible their parallel operation in any situation where their operands are functionally independent. Appendix A summarizes the architectural features of the iPSC–VX VP.
Figure 17: *iPSC-VX* computational node
Figure 18: iPSC-VX Vector Processor (VP) Architecture
5.2.1 Microprogramming Techniques on the iPSC–VX VP

Parallelism on the VP can be achieved in two ways: overlapped and pipelined programming [65]. In overlapped programming, local parallelism between the independent hardware units is exploited whenever the operands of these units are functionally independent. In pipelined programming, global parallelism between the independent computations on the sequence of elements of a vector is exploited. The overlapped method is usually applied to scalar computations and the initializations required during the start-up of a pipelined loop. In overlapped programming various sequences of dependent computations required are examined to find the longest or critical path and then all the shorter paths are overlapped with the critical path. The pipelined method is usually applied to vector computations. In pipelined programming, the independent scalar hardware units are programmed to construct software loops in performing vector operands. In this software loop the entire computation on one element of a vector is overlapped with those on preceeding and succeeding elements. This process is called a software pipeline[65] because of its similarity to the hardware pipeline concept used in conventional vector supercomputers [63,64].

For vectors of length $n$, the vector operations take a multiple of $n$ clockcycles and additional start-up time which can be neglected for sufficiently large $n$. A period of $n$ clockcycles is called a chime [66]. The number of chimes ($c$), corresponds to the number of microinstructions ($I$) in the pipelined loop and it determines the computation rate of the software pipeline. The lower bound on the minimum number of chimes, $I_{\text{min}}$, is determined by the most frequently used functional unit called the critical unit. Sometimes, it may not be possible to achieve this lower bound due to reasons such as: interconnect–bus conflicts, very limited number of
registers available in the arithmetic units (especially for DP operations), limitations introduced during the DP multiplication ($I \geq 3$). The number of times around the pipelined loop required to complete the entire computation on one element of a vector is the number of stages $S$ in the software pipeline which determines the start-up overhead required to fill the pipeline. The following four step procedure[65] is used to construct the tightest software pipeline loop for a vector computation.

1. Estimate the critical path length $L$ by programming the computation on one element of a vector using the overlapped method.

2. Find the critical functional unit or interconnect-bus to determine $I_{\min}$.

3. Generate the pipelined loop microcode from the overlapped microcode by trying to fold the overlapped microcode back onto itself every $I_{\min}$ microinstructions. If unavoidable conflicts arise, let $I_{\min} \leftarrow I_{\min} + 1$ and repeat Step 3, else set $S \simeq \lfloor L/I_{\min} \rfloor$, and then go to Step 4.

4. Generate $S - 1$ prelude and $S - 1$ postlude sections.

Prelude and postlude sections are required to fill and flush the software pipeline respectively. The $i$th prelude section includes the first $i$ stages and the $i$th postlude section includes of all but the first $i$ stages of the pipelined loop.

5.2.2 Vectorization of the CG–SCG/s1 Computations on the VP

As mentioned earlier, the computational load for each processor of the hypercube consists of the following matrix operations per iteration of the parallel CG–SCG/s1 algorithm.

- 1 sparse matrix vector product (Step 1)
• 2 inner products (Step 2)

• 3 vector updates (two of them are DAXPY's in BLAS notation[67]) (Step 4)

$DP$ arithmetic is used to minimize the accumulation of errors during the iterations of the $SCG$ algorithm. Hence, the implementation and the performance for $DP$ vector operations are given here.

Row–Wise Sparse Matrix Vector Product

The standard column index compressed data storage scheme which loops over the nonzero column indices of the sparse coefficient matrix is used here. The row–wise sparse matrix vector product can be expressed by the following double nested loop.

```
for i = 1, n
    start ← AROW(i)
    last ← AROW(i + 1) − 1
    sum ← 0
    for j = start, last
        sum ← sum + $A(j) \times Pk(ACOL(j))$
    end for
    $Qk(i) ← sum$
end for
```

Since the nonzero entries of the sparse coefficient matrix are stored in one dimensional array $A$ in completely compressed form, their column indices are stored in one dimensional array $ACOL$. The elements of the array $AROW$ point to the beginning of the rows in arrays $A$ and $ACOL$. This scheme can be implemented on the $VP$ by converting the column indices in array $ACOL$ into address offsets using

$$ACOL(i) = s_t \times (ACOL(i) - 1) \quad (5.1)$$
where $s_t$ is the stride for the $DP$ vector ($s_t = 2$ for contiguous $DP$ vectors). Then, the following three microinstruction sequence can be used to read $P_k(ACOL(j))$ from the $DM$.

$$R_5 = R_5 + R_6, \text{ offset} = \text{MEM}; \text{(read next offset value from DM pointed by } R_5)$$

$$\text{ENFDB}; \text{(FBRB loaded from DM (default), enable FBRB onto L-bus on next cycle)}$$

$$R_0 = R_0 + FBAC K, \text{ pj} = \text{MEM}; \text{(read } P_k(ACOL(j)) \text{ from DM pointed by } R_0)$$

Note that, the $RALU$ register $R_5$ is a pointer to the elements of the array $ACOL$, $R_6$ holds the stride for this array ($s_t = 1$ for contiguous integer vectors) and $R_0$ holds the base address for the array $P_k$. The $RALU$ field of the third microinstruction, $R_0 = R_0 + FBAC K$, routes the $L$-bus directly to one of the inputs of the integer $ALU$ inside the $RALU$. Unfortunately, this mode uses no more than 10 bits of the $L$-bus. Hence, this flexible offset scheme cannot be used for $DP$ vectors longer than 512 elements. In order to handle longer vectors, the column indices in the array $ACOL$ are converted into absolute addresses of the corresponding elements of the array $P_k$ using

$$ACOL(i) = \text{baseaddress}(P_k) + s_t \times (ACOL(i) - 1). \quad (5.2)$$

Then, the three microinstruction sequence given above should be modified as

$$R_5 = R_5 + R_6, \text{ absaddr} = \text{MEM}; \text{(read next address from DM pointed by } R_5)$$

$$\text{ENFDB}; \text{(FBRB is loaded from DM-BUS, enable FBRB onto L-BUS on next cycle)}$$

$$R_0 = FBAC K, \text{ pj} = \text{MEM}; \text{(read } P_k(ACOL(j)) \text{ from DM pointed by } R_0)$$

The microoperation $R_0 = FBAC K$ in the third microinstruction indicates that the $RALU$ register $R_0$ is loaded from the $L-BUS$. The maximum size of the $DP$ vectors that can be handled by this scheme is $32K$. The only disadvantage of this scheme is in situations where the same coefficient matrix is to be multiplied by
vectors allocated in different memory locations since the micro subroutine performing the indicated conversion has to be reexecuted for each one of these vectors. Fortunately, the second scheme does not introduce any overhead in this implementation since the same coefficient matrix $A$ is iteratively multiplied by one array $P_k$ throughout the iterations.

The four step procedure described in Section 5.2.1 is applied to pipeline the inner loop of the sparse matrix vector product as follows:

**STEP 1 : overlapped code**

1. a. process address of $ACOL(j)$ in RALU and initiate read from DM
2. a. process address of $A(j)$ in RALU and initiate read from DM
   b. $FBRB$ loaded with $ACOL(j)$ from $DM$-BUS (default)
   c. enable $FBRB$ onto $L$-BUS on next cycle
3. a. load $ACOL(j)$ from $L$-BUS into a RALU reg. and initiate read from DM
   b. register $MEM$ loaded with $A(j)$ from $DM$-BUS (default)
4. a. load $A(j)$ from register $MEM$ into multiplier register $M10$ via $A$-BUS
   b. register $MEM$ loaded with $P_k(ACOL(j))$ from $DM$-BUS (default)
5. a. load $P_k(ACOL(j))$ from $MEM$ into multiplier register $M00$
   b. initiate DP multiplication of $M00$ by $M10$ ($A(j) \times P_k(ACOL(j))$)
6-9 DP multiplication continues inside the multiplier (default)
10. a. chain product (PROD) into ALU register A00
    b. initiate DP addition of A00 by A10 which holds the partial sum
11-12 DP addition continues inside the ALU
13. a. feed new partial sum (ALUR) back to ALU register A10
    b. decrement PS counter, test, and branch to loop head

Therefore, critical path length is $L = 13$ for the overlapped code.

**STEP 2 : functional unit usage**
3 RALU operation (address calculation)
3 DM unit operation (read from DM)
1 Multiplier operation
1 ALU operation
1 PS operation
2 Arithmetic bus operation
1 Literal bus operation

RALU and DM which are used three times are the critical units, hence $I_{\text{min}} = 3$.

**STEP 3**: Success is obtained during the first folding process with $I_{\text{min}} = 3$ giving

$$S = \left\lceil \frac{L}{I_{\text{min}}} \right\rceil = 5 .$$

(5.3)

The 3 microinstruction pipelined loop written in VP assembler is given Fig. 19. Note that, fifth stage can be avoided in this loop by performing the first and the only microoperation (other than the loop test microoperation), $A10 = ALUR \rightarrow s$ (13.a in the overlapped code) on the first microinstruction of the previous (fourth) stage. However, one stage (three clockcycles) early assertion of this microoperation requires the initiation of an ALU operation to initialize the partial sum to zero on the first cycle of the third section of the prelude. This is achieved by loading one of the unused registers of the ALU with a DP zero during the start-up and then initiating an ALU bypass operation on the first microinstruction of the last prelude section. Note that, the sequencer operation which performs the loop test on 13.b of the overlapped code is shifted accordingly to the last microinstruction of the loop. Hence,

$$S = 4$$

(5.4)

is obtained. The 3 microinstruction loop with $S = 4$ is illustrated in Fig. 20.

**STEP 4**: $S - 1 = 3$ prelude and the postlude sections with three cycles each are developed (Figs. 21(a)–(b)).
<table>
<thead>
<tr>
<th>Stage-1(S1)</th>
<th>Stage-2(S2)</th>
<th>S3</th>
<th>Stage-4(S4)</th>
<th>Stage-5(S5)</th>
</tr>
</thead>
</table>
| $R_5 = R_5 + R_6, j = MEM$  
(Read $ACOL(j)$) | $M_{10} = A_j$  
(Id $M_{10}$ with $A(j)$) | | $A_{00} = PROD \rightarrow A_j P_j$  
(chain prod to $ALU$)  
$s = A_{00} + D \cdot A_{10}$  
(initiate $DP$ add) | $A_{10} = ALUR \rightarrow s$  
(feed partial sum)  
(back to $ALU$) |
| $R_3 = R_3 + r, A_j = MEM$  
(Read $A(j)$) | $M_{00} = P_j$  
(Id $M_{00}$ with $P_k(ACOL(j))$)  
$A_j P_j = M_{00} \cdot *D \cdot M_{10}$  
(initiate $DP$ mult.) | | | |
| $R_0 = BACK, P_j = MEM$  
(Read $P_k(ACOL(j))$) | | | | $JRS P_2$  
(loop test) |

$R_3$ and $R_5$ initially hold the base addresses of $A$ and $ACOL$, respectively.  
$R_4$ and $R_6$ hold the strides for the arrays, $A$ and $ACOL$, respectively.

Figure 19: The microcode ($S = 5$) in $VP$ assembler for the inner loop of sparse matrix vector product.
<table>
<thead>
<tr>
<th>$S1$</th>
<th>$S2$</th>
<th>$S3$</th>
<th>$S4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R5=R5+R6,j=\text{MEM}$</td>
<td>$M10=Aj$</td>
<td>$A00=\text{PROD} \rightarrow AjPj$</td>
<td>(chain product to ALU)</td>
</tr>
<tr>
<td>(Read $ACOL(j)$)</td>
<td>(load $M10$ with $A(j)$)</td>
<td>$A10=\text{ALUR} \rightarrow s$</td>
<td>(feed sum back to ALU)</td>
</tr>
<tr>
<td>$R3=R3+R4,Aj=\text{MEM}$</td>
<td>$M00=Pj$</td>
<td>$s=A00 \cdot +D \cdot A10$</td>
<td>(initiate DP addition)</td>
</tr>
<tr>
<td>(Read $A(j)$)</td>
<td>(load $M00$ with $Pk(ACOL(j))$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{ENFDB}$</td>
<td>$AjPj=M00 \cdot +D \cdot M10$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(enable FBRB)</td>
<td>(initiate DP mult.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R0=\text{FBACK},Pj=\text{MEM}$</td>
<td>$JRS \ P2$</td>
<td></td>
<td>(loop test)</td>
</tr>
<tr>
<td>(Read $Pk(ACOL(j))$)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 20: The microcode ($S = 4$) in VP assembler for the inner loop of sparse matrix vector product
On the last clockcycle of the postlude section, the final sum value corresponding to the result for $Qk(i)$ is latched to the output register of the ALU. Hence, it takes two cycles to store the result on the DM via FIFO thus yielding a critical path length of 11 clockcycles for the tail section. Before entering the pipelined inner loop, one of the counters in the PS should be loaded with the length of row–i which can be computed from $AROW(i + 1) - AROW(i)$. Most of the microinstructions in the prelude section use the RALU and hence the microinstructions of the start–up section, which require six RALU operations, cannot be overlapped efficiently with the prelude section. The length of the critical path in the overlapped code for the start–up section is 11 microinstructions and only the last two microinstructions can be overlapped with the first two microinstructions of the prelude section. Hence, for this scheme (Fig. 21(b)), the total number of clockcycles per iteration of the outer loop (chimes) is

$$c(z) \sim 9 + 3 \times 3 + 3 \times (z - 3) + 3 \times 3 + 2$$

startup prelude inner loop postlude tail

$$c(z) \sim 3 \times z + 20 \text{ clockcycles} \quad (5.5)$$

where $z$ is the average number of non–zero entries per row of the sparse coefficient matrix $A$.

The start–up and prelude sections for any iteration of the outer loop can be pipelined with the postlude and tail sections of the previous iteration since the operations required in the latter sections do not depend on the results obtained at the end of the former sections. Hence, the microoperations in the start–up section of the $(i + 1)$–th iteration which make heavy use of the RALU and PS are successfully overlapped with the microoperations in the postlude section of the $i$–th
iteration which only uses the arithmetical units. The number of chimes for this pipelined outer loop scheme (Fig. 21(c)) is

\[ c(z) = 3 \times (z - 3) + 18 = 3 \times z + 9 . \]  

(5.6)

Hence, for \( n = N/P \) variables mapped to a VP board, it takes \( \sim (3z + 9) \times n \) clockcycles since the initial start-up overhead for the outer loop and all the other system overhead can be neglected for sufficiently large \( n \). In general, \( n_{1/2} \sim 30 \) for the VP of the iPSC–VX, where \( n_{1/2} \) indicates the number of elements in the dense column vector required to reach the half of the peak performance of the outer loop. The constant 9 in Equation (5.6) indicates the total number of overhead clockcycles per iteration of the outer loop. There are two 64-bit and one 32-bit operand DM read microoperations in the 3-cycle inner loop. For \( n \leq 640 \), the elements of the frequently referenced column array \( Pk \) can be allocated in the SRAM. In this case, a single iteration of the inner loop takes 550\( \text{ns} \) thus resulting in a peak performance of 3.64 MFLOPS. There are one 64-bit and one 32-bit operand fetches from the DRAM during the overhead section. Hence, the expression, \( c_t(z) \), for the execution time per iteration of the outer loop as a function of \( z \) is

\[ c_t(z) = 0.55 \times z + 1.15 \mu\text{secs} . \]  

(5.7)

with

\[ z_{1/2} \sim 2.1 . \]  

(5.8)

Here, \( z_{1/2} \) denotes the average number of non–zero entries per row of \( A \)-matrix required to reach half of the peak performance (3.64 MFLOPS) of the inner loop. Unfortunately, for \( n \geq 640 \) the frequently used array \( Pk \) cannot be totally allocated in the SRAM due to the size limitations and hence the performance of the inner
Figure 21: (a) Pipelined loop microcode structure for sparse matrix vector product, (b) Pipelined inner and overlapped outer loop, (c) Pipelined inner and pipelined outer loop.
loop reduces to 2.86 MFLOPS, when the \( P_k \)-array is totally allocated in the DRAM. The expression for \( c_t(z) \) in this case becomes

\[
c_t(z) = 0.70 \times z + 1.15 \, \mu s
\]  

(5.9)

with

\[
z_{1/2} = 1.5.
\]  

(5.10)

**Comparison with Diagonal–Wise Sparse Matrix Vector Product**

Sparse matrix vector products on vector supercomputers using special vector units with long pipeline lengths are carried out in a diagonal–wise fashion for coefficient matrices arising from finite difference or finite element discretizations [68]. For example, the coefficient matrix \( A \) in Fig. 2 has 9 dense diagonal strips (\( \sigma = 9 \)). The inner loop in a diagonal–wise sparse matrix vector product is the accumulation of the product of two dense vectors (diagonal strip vectors of \( A \) and the column vector \( P_k \)) of sizes nearly equal to \( n \). The outer loop is iterated only \( \sigma \) times. Since, the inner loop is iterated almost \( n \) times, the overhead during \( \sigma \) iterations of the outer loop can be neglected for sufficiently large \( n \). Since 3 reads (one for strip vector, one for column vector and one for partial sum vector) and one write operation (for partial sum vector) are required in the inner loop, \( I_{\text{min}} = 4 \). Hence, \( I = 5 \) is found because of the limitation introduced during the DP multiplication. Thus, diagonal–wise sparse matrix vector product takes \( \sim (5 \times n) \times \sigma \) clockcycles with

\[
c(\sigma) = 5 \times \sigma
\]  

(5.11a)

\[
c_t(\sigma) = 0.8\sigma \mu s
\]  

(5.11b)
when the partial sum vector can be totally allocated in the \textit{SRAM}. Since, $\sigma = z$ for such $A$ matrices, equating the expression for $c_t(\sigma)$ and $c_t(z)$ for \textit{diagonal-wise} and \textit{row-wise} schemes, $z = 4.6$ is obtained. This indicates that the \textit{diagonal-wise} scheme yields better performance only for $z < 4.6$. The expression $c_t(\sigma)$ given for the \textit{diagonal-wise} scheme is a very optimistic estimate since finite element or difference discretizations of regions with appendages and holes yield a much larger number of strips. A considerable overhead is associated with finding an ordering for the $A$-matrix that gives a near minimal number of strips. For the sample (regular) $FE$ meshes encountered in metalwork simulation $\sigma \sim z = 18$.

The physical domains arising in metalwork simulation are in general very irregular with appendages and holes. As a consequence, the \textit{row-wise} sparse matrix vector product scheme was chosen for vectorization on \textit{iPSC-VX VP}.

\textbf{Innerproducts}

The pipelined loop for the inner product is microcoded by following the four step procedure given in Section 5.2.1. The estimated \textit{critical} path $L = 12$ is obtained from the \textit{overlapped} microcode. The \textit{critical} functional units are \textit{RALU}, \textit{DM}, and \textit{A-BUS} which are used twice and thus $I_{\text{min}} = 2$. However, this lower bound is not achieved during the first \textit{folding} process due to the limitations in \textit{DP} multiplication. Success is obtained during the second \textit{folding} process with $I_{\text{min}} \leftarrow I_{\text{min}} + 1 = 3$.

For the innerproduct, $\langle p_k, Ap_k \rangle$, each iteration of the pipelined loop takes $c_t = 450\text{ns}$ resulting in $4.44\text{MFLOPS}$, since there are two reads from the \textit{DM} and one of the vectors, $p_k$, is stored in \textit{SRAM}. Similarly, for the innerproduct $\langle Ap_k, Ap_k \rangle$, $c_t = 450\text{ns}$ resulting in $4.44\text{MFLOPS}$, since there is only one read which is from the \textit{DRAM}. 

108
Vector Updates

Two of the vector updates are of the form

\[ y = y + \alpha x \]  \hspace{1cm} (5.12)

which are called \textit{DAXPY}’s in \textit{BLAS} terminology \cite{67} and one is of the form

\[ y = x + \beta y. \]  \hspace{1cm} (5.13)

There is no difference in their performance in terms of clockcycles. The \textit{critical} path length is estimated as \( L = 12 \) from the overlapped code. The \textit{critical} functional units are the \textit{RALU}, \textit{DM}, and \textit{A-BUS} which are used three times, hence \( I_{\text{min}} = 3 \). Success is obtained in the first folding process with the lower bound. For the \textit{DAXPY} operation

\[ r_{k+1} = r_k + \alpha_k A p_k \]  \hspace{1cm} (5.14)

\( c_l = 750ns \) resulting in \( 2.67 \text{MFLOPS} \) since all the three 64-bit operands are stored in the \textit{DRAM}. For the \textit{DAXPY} operation

\[ x_{k+1} = x_k - \alpha_k p_k \]  \hspace{1cm} (5.15)

\( c_l = 600ns \) resulting in \( 3.33 \text{MFLOPS} \) since the \( p \)-vector is stored in the \textit{SRAM}. For the vector update

\[ p_{k+1} = r_{k+1} - \beta_k p_k \]  \hspace{1cm} (5.16)

\( c_l = 450ns \) resulting in \( 4.44 \text{MFLOPS} \) since the \( p \)-vector, which is stored in the \textit{SRAM} is referenced twice for read and write.
5.3 Parallel Implementation of the Vectorized CG–SCG on iPSC–VX

In the iPSC system, node-to-node interconnection is implemented with a single bidirectional communication link managed by the Intel 82586 LAN communication coprocessor. Unpredictable increases in message transfer times have been observed when two messages are sent in opposite directions on the single link. Collision of two such messages results in timeouts and retransmissions of these two messages. Similarly, multiple simultaneous communications to/from a processor can result in overwriting of messages due to contention for an internal DMA bus. This will also cause timeouts and retransmissions. The communication protocols for distributed sparse matrix vector product and inner product computations are redesigned considering all these points.

5.3.1 A Communication Protocol for 1-D Strip Partitioning

The following communication protocol for 1-D strip partitioning scheme precludes the possibility of collision of messages on serial communication links, during the nearest neighbor communication steps.

\[
\text{if my processor number is even in the gray code ordering then}
\begin{align*}
\text{csend } (p^1_k \in \{\text{my right boundary}\}) \text{ to my right neighbor} \\
\text{creceive } (p^1_k \in \{\text{my right neighbor's left boundary}\}) \text{ from my right neighbor} \\
\text{csend } (p^1_k \in \{\text{my left boundary}\}) \text{ to my left neighbor} \\
\text{creceive } (p^1_k \in \{\text{my left neighbor's right boundary}\}) \text{ from my left neighbor}
\end{align*}
\]

\[
\text{else if my processor number is odd in the gray code ordering then}
\begin{align*}
\text{creceive } (p^1_k \in \{\text{my left neighbor's right boundary}\}) \text{ from my left neighbor} \\
\text{csend } (p^1_k \in \{\text{my left boundary}\}) \text{ to my left neighbor} \\
\text{creceive } (p^1_k \in \{\text{my right neighbor's left boundary}\}) \text{ from my right neighbor} \\
\text{csend } (p^1_k \in \{\text{my right boundary}\}) \text{ to my right neighbor}
\end{align*}
\]

end if
where \texttt{csend} and \texttt{creceive} denote synchronous (blocking) send and receive operations. In this protocol, each \textit{even} numbered processor (in the graycode ordering) first sends data to its right neighbor (\textit{odd} numbered), and then it delays sending data to its left neighbor (\textit{odd} numbered) until it receives data from its right neighbor. Each odd numbered processor sends data first to its left neighbor (\textit{even} numbered) and then to its right neighbor (\textit{even} numbered) only after it receives data from those neighbor processors. The possibility of collisions on the serial communication links are thus avoided. Note that, the given protocol does not achieve the overlap of set-up and message transfer times for individual communications between a pair of communicating processors. Therefore, the set-up and message transfer times for individual communications between a pair of processors are additive.

The simple model given earlier

\[ T_{comm} = S_C + \ell T_C \]

with \( S_C = 1.2 ms \) and \( T_C = 1.27 \mu s \) is found to be quite accurate for the \textit{iPSC} system for message lengths less than 1025 bytes. In the \textit{iPSC} system, larger messages are broken into multiple packets, and extra set-up time is required for each extra packet of such a multipacket message. In fact, the \textit{1-D} strip partitioning scheme which has a larger communication volume compared to the other two will result in message lengths of \( 2m \times DP \text{ words} < 1025 \text{ bytes} \) for square \textit{FE} meshes with \( m \leq 64 \). Note that, \( m = 49 \) for the largest \textit{FE} mesh simulated in this chapter.

Hence, the parallel complexity for the \textit{1-D} communication protocol becomes

\[ T_{1D} = 4S_C + 8mT_C \]
Figure 22: Communication protocol for 1-D strip partitioning on iPSC/d3
for an $m \times m$ square $FE$ mesh for $m \leq 64$. Fig. 22 illustrates the 4 concurrent communication phases for this protocol. Note that, the time complexity given in (5.18) is twice the complexity given in (4.8) for the 1-D communication protocol proposed for the $iPSC2$ architecture. Communication protocols that prevent the possibility of such collisions can be similarly devised for 1.5-D and 2-D partitioning schemes. Similarly, $T_{1.5D}$ and $T_{2D}$ will be twice the complexity given in (4.11) and (4.10) respectively. That is,

$$T_{1.5D} = 6S_C + 2(4 + 2m + \frac{4m}{P})T_C$$  \hfill (5.19)\\
$$T_{2D} = 8S_C + 2(8 + \frac{4m}{P_H} + \frac{4m}{P_V})T_C.$$  \hfill (5.20)

Therefore, the analysis summarized in (4.14) [57] for the selection of the optimal strip partitioning scheme for the $iPSC2$ system is still valid for the $iPSC$ system for $m \leq 64$. Using the experimentally measured values of $S_C = 1.2ms$ and $T_C = 10.16\mu s$ per $DP$ word for the $iPSC$, it can be deduced from equation (4.14) that 1-D strip partitioning scheme is superior to the other two for

$$m < 70$$  \hfill (5.21)

on the $iPSC$. As a consequence, only the communication protocol required by 1-D strip partitioning was actually implemented on the Intel $iPSC$ system, for the distributed sparse matrix vector product.

This algorithm does not prevent the possibility of contention for the internal $DMA$ buses, of the processors due to the possibility of one incoming and one outgoing message for each processor. The possibility of such contentions is precluded under perfectly load balanced conditions. The percentage of retransmission due to internal bus contention has been measured to be below 1% for a wide range of sample problems.
5.3.2 An Efficient Local Ordering Scheme for 1-D Strip Partitioning

Since $NX$ requires the user data requested for communication to be in contiguous memory locations, each processor has to perform two vector *gather* operations to collect the most recently updated values corresponding to its right and left *boundary* $FE$ nodes, from its $Pk$ array in two user communication buffers. Similarly, each processor has to perform two vector *scatter* operations to insert the non-local (*redundant*) components of the distributed $p$-vector received from its two neighbors to the appropriate locations in its own $Pk$ array. Fig. 23 illustrates the vector *gather* and *scatter* operations required for communication in the 1-D strip partitioning scheme. Note that, these operations are repeated for each SCG iteration. These *scatter/gather* operations introduce a considerable amount of computational overhead for a high performance vector processor and for the 1-D strip partitioning scheme which results in larger number of *boundary* $FE$ nodes. For example, in the 1-D strip partitioning of an $m \times m$ square $FE$ mesh, each processor has to perform $8m$ *scatter/gather* operations per SCG iteration. Therefore, the computational overhead is

$$ O_{sg} = \frac{8m}{(z + 5)2m^2/P} \times \frac{T_{sg}}{T_{calc}} \cdot 100\% $$

$$ = \frac{4P}{m(z + 5)} \times \frac{T_{sg}}{T_{calc}} \cdot 100\% $$  \hspace{1cm} (5.22)

where $T_{sg}$ denotes the time taken per *scatter/gather* operation and $T_{calc}$ is the average time taken per floating point multiplication and addition in the CG–SCG/s1 algorithm. The microsubroutines for the vector *scatter/gather* operations (available in the iPSC–VX library) require a 5 cycle pipelined inner loop with 1 DP read, 1 SP read, and one 1 DP write. Hence, the time taken per single *scatter/gather* operation is, $T_{sg} \sim 0.75\mu s$, when $Pk$ is allocated in $SRAM$. In our sample prob-
Figure 23: Vector scatter/gather operations required for communication in 1-D strip partitioning scheme
Figure 24: An efficient local ordering scheme for 1-D strip partitioning
lems, $z \sim 17$ due to the inactive boundary $FE$ nodes. For $z \sim 17$, $T_{calc} \sim 0.6\mu s$, when $P_k$ is allocated in $SRAM$. Hence,

$$T_{sg} \sim 1.25T_{calc} \cdot \tag{5.23}$$

For a medium size $25 \times 25$ $FE$ mesh on $iPSC-VX$ with 4 $VP$'s, $O_{sg} \sim 4\%$. The computational overhead increases with increasing number of processors and decreasing $z$. The above simplified analysis does not include the overhead due to 4 extra microsubroutine calls from the node board per $SCG$ iteration.

Fig. 24 illustrates a local ordering scheme that avoids vector scatter/gather operations for 1-D strip partitioning. The overlap between right and left boundary components of the local $P_k$-array is due to the possibility of very narrow strips, with $FE$ nodes interacting with nodes in both right and left strips. Each processor rearranges, in parallel, the rows of its $A$-array and $ACOL$-array and modifies the entries in its $ACOL$-array according to this ordering. This parallel reordering operation is repeated only when the topology of the $FE$ mesh changes.

5.3.3 Overlapping Communication and Computation

Fig. 17 shows the steps to receive/send data into/from a node for processing by the $VP$. A message sent from an adjacent node is received over one of the serial communication ports and is automatically deposited in a system message buffer. If a request for the message is pending (or is made some time later), $NX$ will then transfer the data from the message buffer which resides on the node memory to the buffer on the $VP$ memory indicated by the requesting user process. The computational results can be sent to other nodes following a similar sequence of events. Hence, node-to-node communication operations, supported by $NX$ on the node processor, can be effectively overlapped by the floating operations performed on the $VP$ board.
It can be easily seen from Fig. 24 that the ordering given in Section 5.3.2 naturally achieves the in place separation for internal and boundary sparse matrix vector computations required to overlap communication with computation. The internal sparse matrix vector product computation on each VP is initiated before starting the four synchronous nearest neighbor communication steps required for the boundary sparse matrix vector. Thus, the internal sparse matrix vector product computations performed on the VP are effectively overlapped with the four nearest neighbor communication steps performed by NX on the node board. Each processor initiates its local boundary sparse matrix vector product computations on the VP only after its node board completes these synchronous communications.

5.3.4 Distributed Inner Product Computations

The communication pattern of the EA algorithm causes collisions of messages in opposite directions on the serial inter-processor links. The GS-GB algorithm prevents the possibility of such collisions. Hence, the advantage of the EA algorithm over the GS-GB algorithm is lost due to timeouts and retransmissions caused by message collisions. Such collisions can also be avoided in the EA algorithm by using synchronous send and synchronous receive operations. Fig. 25 illustrates the implementation of the EA algorithm on a 4-dimensional iPSC/d4 hypercube. However, in this implementation, the parallel complexity of the EA algorithm becomes equal to the parallel complexity of the GS-GB algorithm.

The GS-GB algorithm does not preclude the possibility of contention for the internal DMA buses of some processors. Contention may be caused by multiple incoming messages during the later \((d-1)\) communication phases of the GS algorithm. However, the possibility of such internal bus contention exists for each processor during each \(d\) communication step of the EA algorithm shown in Fig. 25. Both
Figure 25: Implementation of the Exchange-Add algorithm on the iPSC/860.
the GS-GB algorithm and the EA algorithm are implemented on the iPSC system. Experimental results have shown that, GS-GB algorithm, in general, yields better performance compared to the EA algorithm. As a consequence, the GS-GB algorithm is used in the distributed inner product computations.

The communication steps executed by each processor in the distributed GS-GB algorithm, with processor $P_0$ chosen as the root processor, is given below for $w = 1$. Initially, each processor accumulates its partial sum corresponding to its local components of the distributed vector(s) in $mypassum$.

\begin{verbatim}
for $i = 0, 1, \ldots, d - 1$
    if $(mynode MOD[2i+1] = 0)$ then
        creceive (hisparsum) from my neighbor on channel-$i$
        $mypassum \leftarrow mypassum + hisparsum$
    else if $(mynode MOD[2i+1] = 2^i)$ then
        csend ($mypassum$) to my neighbor on channel-$i$
    end if
end for
if $(mynode = 0)$ then
    $gsum \leftarrow mypassum$
end if
for $i = d - 1, d - 2, \ldots, 1, 0$
    if $(mynode MOD[2i+1] = 0)$ then
        csend ($gsum$) to my neighbor on channel-$i$
    else if $(mynode MOD[2i+1] = 2^i)$ then
        creceive ($gsum$) from my neighbor on channel-$i$
    end if
end for
\end{verbatim}

where \texttt{csend} and \texttt{creceive} denote synchronous send and receive operations.
5.4 Experimental Results

This section presents the performance results for the vectorization and the parallel implementation of the CG-SCG/s1 algorithm on a 4-node Intel iPSC-VX/d2 vector hypercube, for FE simulation of metalforming problems.

5.4.1 Performance of the Microcoded Sparse Matrix Vector Product

Table 7 presents the estimated and the measured performance (MFLOPS) of the microcoded sparse matrix vector product for different $z$ values. The sustained performance, $P_S$, in MFLOPS, for a particular $z$ value is calculated from

$$P_S(z) = \frac{2zn}{T_{MVP}} \text{MFLOPS}$$  \hspace{1cm} (5.24)

where $T_{MVP}$ is the measured time for the multiplication of an $n \times n$ sparse matrix $A$ having $z$ nonzero entries per row with a dense column vector $P_k$ of $n$ elements.

In these measurements $n$ is typically chosen large enough to observe the effects of $z$ on the performance. The estimated performance, $P_E$, as a function of $z$ is calculated from

$$P_E(z) = \frac{2z}{c_d(z)} \text{MFLOPS}$$

$$= \frac{2z}{0.55 \times z + 1.15} \text{MFLOPS}$$  \hspace{1cm} (5.25)

for $P_k$ totally allocated in SRAM and

$$P_E(z) = \frac{2z}{0.70 \times z + 1.15} \text{MFLOPS}$$  \hspace{1cm} (5.26)

for $P_k$ totally allocated in DRAM. The measured performance is found to be within 4% of the estimated performance.

Fig. 26 illustrates the estimated and the measured performance of the microcoded sparse matrix vector product with respect to the number of nonzero entries per
row of the sparse matrix. As seen from this figure, peak performance of the inner loop is approximately achieved for very low \( z \) values. For example, \( z_{1/2} \approx 2.26 \) and \( z_{1/2} \approx 1.84 \) are attained when \( Pk \) is allocated in \( SRAM \) and \( DRAM \), respectively. This low value for \( z_{1/2} \) is achieved by exploiting the parallelism at both levels and short pipe lengths of the functional units. As it can be seen from Fig. 26, \( \sim 3.15 MFLOPS \) and \( \sim 2.53 MFLOPS \), are attained for \( z \approx 17 \) when the \( Pk \)-array is totally allocated in \( SRAM \) and \( DRAM \), respectively. It should be noted here that, the sustained performance of a node CPU (80286/80287) of the iPSC-VX is measured to be only \( \sim 30 KFLOPS \) for the sparse matrix vector product.

5.4.2 Comparison of the B-SCG and the CG-SCG/s1 Algorithms

The vector operations involved in the B-SCG and the CG-SCG algorithms are quite similar. The B-SCG algorithm is also implemented on the iPSC-VX/d2. Table 7 presents the solution times (per iteration) of the B-SCG and the CG-SCG/s1 algorithms on one and two dimensional vector hypercubes (iPSC-VX/d1-d2) for different size FE problems. Fig. 27 illustrates the percent performance improvement (\( \eta_1 \)) as a function of problem size. Here, the percent performance improvement (\( \eta_1 \)) is calculated as

\[
\eta_1 = \frac{T_{BSCG} - T_{CGSCG}}{T_{CGSCG}} \cdot 100\%.
\]

The estimated expression for \( \eta_1 \) under perfectly load balanced conditions is

\[
\eta_1 = \frac{4dSC}{T_{CGSCG}} \cdot 100\%.
\]

Fig. 27(a) shows that \( \eta_1 \) decreases monotonically with increasing problem size. This is because, the ratio of \( 4dSC \) to the total solution time per iteration of the CG-SCG/s1 algorithm decreases with increasing problem size. As seen in Fig. 27(a), \( \eta_1 \) is greater on larger dimensional hypercubes. The high floating point performance
Table 6: Measured performance (MFLOPS) of the microcoded sparse matrix vector product for different $z$ values

<table>
<thead>
<tr>
<th>$z$</th>
<th>Pk in SRAM Perf. (MFLOPS)</th>
<th>Pk in DRAM Perf. (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.18</td>
<td>1.13</td>
</tr>
<tr>
<td>2</td>
<td>1.77</td>
<td>1.71</td>
</tr>
<tr>
<td>3</td>
<td>2.14</td>
<td>2.11</td>
</tr>
<tr>
<td>4</td>
<td>2.39</td>
<td>2.31</td>
</tr>
<tr>
<td>5</td>
<td>2.56</td>
<td>2.47</td>
</tr>
<tr>
<td>6</td>
<td>2.70</td>
<td>2.60</td>
</tr>
<tr>
<td>7</td>
<td>2.80</td>
<td>2.69</td>
</tr>
<tr>
<td>8</td>
<td>2.88</td>
<td>2.83</td>
</tr>
<tr>
<td>9</td>
<td>2.95</td>
<td>2.86</td>
</tr>
<tr>
<td>10</td>
<td>3.00</td>
<td>2.90</td>
</tr>
<tr>
<td>12</td>
<td>3.10</td>
<td>2.97</td>
</tr>
<tr>
<td>15</td>
<td>3.19</td>
<td>3.08</td>
</tr>
<tr>
<td>18</td>
<td>3.26</td>
<td>3.16</td>
</tr>
<tr>
<td>20</td>
<td>3.29</td>
<td>3.19</td>
</tr>
<tr>
<td>25</td>
<td>3.36</td>
<td>3.25</td>
</tr>
<tr>
<td>30</td>
<td>3.40</td>
<td>3.30</td>
</tr>
<tr>
<td>50</td>
<td>3.49</td>
<td>3.39</td>
</tr>
<tr>
<td>80</td>
<td>3.54</td>
<td>3.44</td>
</tr>
</tbody>
</table>
Figure 26: Measured performance (MFLOPS) of the microcoded sparse matrix vector product as a function of $z$. 
Table 7: Solution times (per iteration) of \textit{B-SCG} and \textit{CG-SCG/s1} algorithms on \textit{iPSC-VX/d1-d2} for different size \textit{FE} simulation problems

<table>
<thead>
<tr>
<th>Test Prob.</th>
<th>Mesh Size</th>
<th>Number of Var's</th>
<th>Number of Iter's for Conv.</th>
<th>\textit{iPSCVX/d1 sol. time per iter (ms)}</th>
<th>\textit{iPSCVX/d2 sol. time per iter (ms)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Basic SCG</td>
<td>CG SCG</td>
</tr>
<tr>
<td>T1</td>
<td>11 × 36</td>
<td>734</td>
<td>99</td>
<td>12.73</td>
<td>9.49</td>
</tr>
<tr>
<td>T2</td>
<td>25 × 25</td>
<td>1175</td>
<td>130</td>
<td>16.15</td>
<td>12.77</td>
</tr>
<tr>
<td>T3</td>
<td>32 × 32</td>
<td>1952</td>
<td>165</td>
<td>22.00</td>
<td>18.54</td>
</tr>
<tr>
<td>T4</td>
<td>33 × 33</td>
<td>2143</td>
<td>202</td>
<td>23.27</td>
<td>20.05</td>
</tr>
<tr>
<td>T5</td>
<td>40 × 40</td>
<td>3120</td>
<td>126</td>
<td>31.98</td>
<td>28.73</td>
</tr>
<tr>
<td>T6</td>
<td>49 × 49</td>
<td>4752</td>
<td>297</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Figure 27: Percent performance improvement of CG-SCG/s1 over B-SCG as a function of problem size.
of the *iPSC-VX VP* results in fine granularity even for large *FE* problems on small dimensional hypercubes. As a consequence, the *CG-SCG/s1* algorithm yields a substantial performance improvement of 20% to 40% on a two dimensional vector hypercube *iPSC-VX/d2*.

### 5.4.3 Comparison of Overlapped and Non-Overlapped Schemes

Table 8 presents the solution times (per iteration) of the *overlapped* and *non-overlapped* *CG-SCG/s1* algorithms on one and two dimensional vector hypercubes (*iPSC-VX/d1-d2*). Fig. 28 illustrates the percent performance improvement (\(\eta_2\)) obtained by overlapping. Note that, \(\eta_2\) on the *iPSC-VX/d1* decreases as the size of the problem increases. This is because, the computational time for the *internal* sparse matrix vector product on each VP board is larger, even for small size problems, than the local communication time required for the *boundary* sparse matrix vector product. However, the computational load of each VP is reduced by a factor of two on *iPSC-VX/d2* for the same size problems. The number of concurrent nearest neighbor communications required for the distributed *boundary* sparse matrix vector is four in *iPSC-VX/d2* compared to two in *iPSC-VX/d1*. The local communication time is greater than the *internal* sparse matrix vector product computation time for each test problem except for the largest one (*T_6*). Hence, \(\eta_2\) on the *iPSC-VX/d2*, increases as the problem size increases for the first 5 test problems (*T_1-T_5*) and then decreases for the largest problem *T_6*. As seen in Fig. 28, overlapping local communications with computation in the *CG-SCG/s1* algorithm yields a substantial performance improvement of 13% to 42% on a two dimensional vector hypercube *iPSC-VX/d2*. 

127
Table 8: Solution times (per iteration) of overlapped and non-overlapped CG-SCG/Sl algorithms on iPSC-VX/d1-d2 for different size FE problems

<table>
<thead>
<tr>
<th>Test Prob.</th>
<th>Mesh Size</th>
<th>Number of Var's</th>
<th>Number of Iter's for Conv.</th>
<th>iPSCVX/d1 sol. time per iter(ms)</th>
<th>iPSCVX/d2 sol. time per iter(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Non Overlap</td>
<td>Overlap</td>
</tr>
<tr>
<td>T1</td>
<td>11 × 36</td>
<td>734</td>
<td>99</td>
<td>13.54</td>
<td>9.49</td>
</tr>
<tr>
<td>T2</td>
<td>25 × 25</td>
<td>1175</td>
<td>130</td>
<td>17.56</td>
<td>12.77</td>
</tr>
<tr>
<td>T3</td>
<td>32 × 32</td>
<td>1952</td>
<td>165</td>
<td>23.64</td>
<td>18.54</td>
</tr>
<tr>
<td>T4</td>
<td>33 × 33</td>
<td>2143</td>
<td>202</td>
<td>25.30</td>
<td>20.05</td>
</tr>
<tr>
<td>T5</td>
<td>40 × 40</td>
<td>3120</td>
<td>126</td>
<td>34.13</td>
<td>28.73</td>
</tr>
<tr>
<td>T6</td>
<td>49 × 49</td>
<td>4752</td>
<td>297</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Figure 28: Percent performance improvement obtained by overlapping communication with computation.
5.4.4 Experimental Performance

Table 9 presents solution times (per iteration) for the sequential $B$-$SCG$ algorithm ($\mu$VAX II, $iPSC$-$VX/d0$), and the parallel $CG$-$SCG/s1$ algorithm ($iPSC$-$VX/d1$-$d2$). Experimental speed-up and efficiency curves are illustrated in Fig. 29 and Fig. 30 respectively. Speed-up and efficiency are calculated from (4.20) and (4.21). As expected, Fig. 29(a) and Fig. 30(a) show that, speed-up and efficiency increase with increasing problem size. As seen in Table 9, better performance is obtained on two $VP$s compared to a single $VP$ for all sample problems. The efficiency achieved on $iPSC$-$VX/d1$ for the three larger sample problems $T_3$, $T_4$, and $T_5$ are 83%, 84%, and 88%, respectively. However, for the two smallest size problems, $T_1$ and $T_2$, the $iPSC$-$VX/d2$ has worse performance compared to the $iPSC$-$VX/d1$. The efficiency achieved on $iPSC$-$VX/d2$ for the largest sample problem ($T_6$) is $\sim 75\%$.

Table 10 presents the measured performance ($MFLOPS$) for the sequential $B$-$SCG$ algorithm ($\mu$VAX II, $iPSC$-$VX/d0$), and for the $CG$-$SCG/s1$ algorithm ($iPSC$-$VX/d1$-$d2$). The performance in $MFLOPS$ for a particular test problem is calculated from (4.22). Fig. 31 illustrates the measured performance for the sequential $B$-$SCG$ algorithm ($\mu$VAX II, $iPSC$-$VX/d0$) as a function of the problem size and for the parallel $CG$-$SCG/s1$ algorithm ($iPSC2/d1$-$d2$) both as a function of problem size and as a function of hypercube dimension. The estimated peak performance of the microcoded $SCG$ algorithm on a single $VP$ is $\sim 3.31 MFLOPS$ if the $P_h$-array is allocated in $SRAM$ and $\sim 2.67 MFLOPS$ if it is allocated in $DRAM$. The estimated peak performance ($3.22 MFLOPS$) is almost attained ($3.19 MFLOPS$) on a single $VP$ board for the smallest size sample problem $T_1$. Most of the elements of the $p$-vector are allocated in the $SRAM$ (640 out of 734) for this problem.
Table 9: Solution times (per iteration) for the $B-SCG$ ($\mu$VAX II, $iPSC\text{-}VX/d0$), and the $CG-SCG/\alpha1$ ($iPSC2/d1\text{-}d4$) for different size $FE$ problems

<table>
<thead>
<tr>
<th>Test Prob</th>
<th>Number of Vars</th>
<th>Number of Iters for Conv</th>
<th>BSCG $\mu$VAX II sol time per iter (ms)</th>
<th>BSCG $d0$ sol time per iter (ms)</th>
<th>CGSCG $d1$ sol time per iter (ms)</th>
<th>CGSCG $d2$ sol time per iter (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>734</td>
<td>99</td>
<td>238.59</td>
<td>10.30</td>
<td>9.49</td>
<td>14.54</td>
</tr>
<tr>
<td>T2</td>
<td>1175</td>
<td>130</td>
<td>384.92</td>
<td>17.23</td>
<td>12.77</td>
<td>16.38</td>
</tr>
<tr>
<td>T3</td>
<td>1952</td>
<td>165</td>
<td>672.79</td>
<td>30.78</td>
<td>18.54</td>
<td>17.70</td>
</tr>
<tr>
<td>T4</td>
<td>2143</td>
<td>201</td>
<td>695.60</td>
<td>32.52</td>
<td>20.05</td>
<td>18.22</td>
</tr>
<tr>
<td>T5</td>
<td>3120</td>
<td>126</td>
<td>1021.83</td>
<td>-</td>
<td>28.73</td>
<td>19.84</td>
</tr>
<tr>
<td>T6</td>
<td>4752</td>
<td>297</td>
<td>1551.58</td>
<td>-</td>
<td>-</td>
<td>26.70</td>
</tr>
</tbody>
</table>
Figure 29: Speed-up curves (a) as a function of problem size, (b) as a function of hypercube dimension
Figure 30: Efficiency curves (a) as a function of problem size, (b) as a function of hypercube dimension
Table 10: MFLOPS performance for the B-SCG (μVAX II, iPSC-VX/d0), and the CG-SCG/s1 (iPSC-VX/d0-d2) for different size FE problems

<table>
<thead>
<tr>
<th>Test Prob</th>
<th>Number of Vars</th>
<th>Number of Iters for Conv</th>
<th>BSCG μVAX II Perfor. in MFLOPS</th>
<th>BSCG d0 Perfor. in MFLOPS</th>
<th>CGSCG d1 Perfor. in MFLOPS</th>
<th>CGSCG d2 Perfor. in MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>734</td>
<td>99</td>
<td>0.14</td>
<td>3.18</td>
<td>3.45</td>
<td>2.25</td>
</tr>
<tr>
<td>T2</td>
<td>1175</td>
<td>130</td>
<td>0.14</td>
<td>3.14</td>
<td>4.24</td>
<td>3.30</td>
</tr>
<tr>
<td>T3</td>
<td>1952</td>
<td>165</td>
<td>0.14</td>
<td>2.93</td>
<td>4.86</td>
<td>5.09</td>
</tr>
<tr>
<td>T4</td>
<td>2143</td>
<td>201</td>
<td>0.14</td>
<td>2.91</td>
<td>4.95</td>
<td>5.46</td>
</tr>
<tr>
<td>T5</td>
<td>3120</td>
<td>126</td>
<td>0.14</td>
<td>-</td>
<td>5.07</td>
<td>7.34</td>
</tr>
<tr>
<td>T6</td>
<td>4752</td>
<td>297</td>
<td>0.14</td>
<td>-</td>
<td>-</td>
<td>8.40</td>
</tr>
</tbody>
</table>
Figure 31: Performance of the sequential $B$-$SCG$ ($\mu$VAX II, iPSC-VX/d0) and the parallel $CG$-$SCG/s1$ (iPSC-VX/d1-d2) (a) as a function of problem size, (b) as a function of hypercube dimension.
(T_1). As seen from Fig. 31, the performance of the sequential SCG algorithm on a single VP decreases slightly with increasing problem size since the portion of the p–vector allocated in SRAM decreases. A performance of 8.40 MFLOPS is attained on the 4–node iPSC–VX/d2 vector hypercube for the largest sample problem T_6. A speed-up of 58 is obtained on the iPSC–VX/d2 for this sample problem compared to the μVAX II.
CHAPTER VI

A Concurrent Error Detecting SCG (CED SCG) Algorithm on a Hypercube Multiprocessor

6.1 Introduction

In this chapter, a cost effective concurrent error detecting SCG (CED SCG) algorithm is presented[69]. Both the SCG algorithm and the CED SCG algorithm have been implemented on a 32-node Intel 286-based iPSC/d5 hypercube. Error detection is achieved by checking algorithm-specific properties such as orthogonality of certain vectors at the end of each iteration as described in Section 6.2. The computations required for error checking are also distributed among the processors and incorporated in the algorithm with redundant computation when necessary and minimal overhead as described in Section 6.3. A low cost error detection scheme is achieved as discussed in Section 6.4. Section 6.5 presents error recovery in the case of transient errors and Section 6.6 presents a scheme for reconfiguration and recovery after a permanent failure. Implementation issues and experimental results are discussed in Section 6.7.

6.2 Error Detection Conditions

The Conjugate Gradient theorem[38] given in Section 3.3 shows the set of necessary conditions that the sequence of vectors \(\{p_i\}\) and \(\{r_i\}\) should satisfy during the iterations. These conditions are:
\[ \langle p_k, A p_i \rangle = \langle p_k, A r_i \rangle = 0 \text{ for } i = 0, 1, \ldots, k - 1 \]

\[ \langle r_k, r_i \rangle = \langle r_k, p_i \rangle = 0 \text{ for } i = 0, 1, \ldots, k - 1 \]

These inner product terms are not computed in the original SCG algorithm. All of these vectors \( p, A p \) and \( r \) are formed at each iteration but they are updated at the next iteration. Checking these conditions requires an order of \( i \) extra inner product computation steps at the \( i \)-th iteration thus introducing a total of \( O(k^2 \frac{N}{n}) \) time redundancy for \( k \) iterations. However, the computational time complexity of the concurrent SCG algorithm is in the order of \( O(k \frac{N}{n}) \) for \( k \) iterations. Hence, it is not feasible to check these conditions in their original form. Low cost check conditions for the sequence \( \{r_i\} \) can be introduced by checking the orthogonality of the new residue vector \( r_{k+1} \) only to the previous residue vector \( r_k \). Hence, the condition

\[ \langle r_{k+1}, r_k \rangle = 0 \] 

at iteration \( k \), constitutes a check on the vector update

\[ r_{k+1} = r_k - \alpha_k A p_k \] 

and also the matrix-vector product, \( A p_k \), and the inner product, \( \langle p_k, A p_k \rangle \), required for this computation. Similarly, a low cost error detection for the vector update

\[ p_{k+1} = r_{k+1} + \beta_k p_k \] 

and the inner product computation, \( \langle r_{k+1}, r_{k+1} \rangle \), required for this computation at iteration \( k \) can be achieved by the checking the \( A \)-orthogonality of the new direction vector, \( p_{k+1} \), only to the previous direction vector \( p_k \). That is,

\[ \langle p_{k+1}, A p_k \rangle = 0 . \]
It can be easily shown that, if the sequence of nonzero vectors \( \{ r_k \} \) and \( \{ p_k \} \) are generated according to
\[
\begin{align*}
    r_{k+1} &= r_k - \delta_k A p_k \quad (6.5a) \\
    p_{k+1} &= r_{k+1} + \gamma_k p_k \quad (6.5b)
\end{align*}
\]
for any sequence of scalars \( \{ \delta_k \} \) and \( \{ \gamma_k \} \) starting from any initial vector \( r_0 = p_0 \), then the conditions
\[
\begin{align*}
    < r_{k+1}, r_k > &= 0 \quad (6.6a) \\
    < p_{k+1}, A p_k > &= 0 \quad (6.6b)
\end{align*}
\]
are satisfied if and only if
\[
\begin{align*}
    \delta_k &= \frac{< r_k, r_k >}{< p_k, A p_k >} = \alpha_k \quad (6.7a) \\
    \gamma_k &= \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >} = \beta_k \quad (6.7b)
\end{align*}
\]
Hence, it can be deduced that if the sequence of matrix vector products \( A p_k \) and the sequence of vector updates
\[
\begin{align*}
    r_{k+1} &= r_k - \alpha_k A p_k \quad (6.8a) \\
    p_{k+1} &= r_{k+1} + \beta_k p_k \quad (6.8b)
\end{align*}
\]
during the \( SCG \) iterations are computed correctly, the given sequence of checks ensure the correct computation of the sequence of global scalars \( \alpha_k \) and \( \beta_k \) which involve distributed inner product computations.

A similar low cost error detection scheme should be devised for the vector update computation
\[
    x_{k+1} = x_k + \alpha_k p_k \quad (6.9)
\]
required for generating the sequence of approximate solution vectors \( \{x_k\} \). It can be easily shown that, the sequence of vectors \( \{r_k\} \) and \( \{x_k\} \) generated according to

\[
\begin{align*}
    r_{k+1} &= r_k - \delta_k A u_k \\
    x_{k+1} &= x_k + \delta_k u_k
\end{align*}
\]  

(6.10a, 6.10b)

for any sequence of scalars \( \{\delta_k\} \) and any sequence of vectors \( \{u_k\} \) starting from \( r_0 = b - A x_0 \), for any initial vector \( x_0 \), satisfy

\[
    r_{k+1} = b - A x_{k+1} \text{ for any } k = 0, 1, \ldots
\]

(6.11)

Hence, any check on the sequence of equations in (6.11) during the SCG iterations constitutes a check on the sequence of matrix-vector products \( A p_k \) and the sequence of vector updates

\[
\begin{align*}
    r_{k+1} &= r_k - \alpha_k A p_k \\
    x_{k+1} &= x_k + \alpha_k p_k
\end{align*}
\]

(6.12a, 6.12b)

performed during the SCG iterations. As it is indicated above, the relation in (6.11) holds for any sequence of global scalars \( \{\delta_k\} \) and vectors \( \{u_k\} \) which may or may not be equal to the sequence of global scalars \( \{\alpha_k\} \) and vectors \( \{p_k\} \) respectively. Here, \( \{\alpha_k\} \) and \( \{p_k\} \) represent the correct values that should be computed during the SCG iterations. However, two given conditions given in (6.6a) and (6.6b) already constitute checks on \( \delta_k = \alpha_k \) and \( \{u_k\} = \{p_k\} \).

The direct check on the equality condition (6.11) requires an extra distributed matrix-vector product \( A x_{k+1} \) which introduces high computational redundancy. Here, instead, a low cost check on this equality condition is introduced as follows.
Take the inner product of both sides of the relation (6.11) by $p_k$ to obtain

$$< r_{k+1}, p_k > = < b, p_k > - < Ax_{k+1}, p_k > .$$ (6.13)

The term $< r_{k+1}, p_k >$ should vanish by c) of the CG Theorem. Hence, (6.13) becomes

$$< Ax_{k+1}, p_k > = < b, p_k > .$$ (6.14)

Since the matrix-vector product $Ap_k$ is already computed during the SCG iterations, and noting that $< Ax_{k+1}, p_k > = < x_{k+1}, Ap_k >$, the check condition becomes

$$< x_{k+1}, Ap_k > = < p_k, b > .$$ (6.15)

thus requiring only two extra inner products per iteration. In general, if there is no better guess for the initial unknown vector $x_0$, the common approach is to use $x = 0$ to start the SCG iterations. Considering this fact, the following corollary of the CG Theorem is proven below in order to introduce another low cost check.

**Corollary 6.1** If the initial guess vector $x_0 = 0$ in the CG algorithm, then

$$< p_k, b > = < r_k, r_k >$$ (6.16)

**Proof:** Take the inner product of both sides of the equation

$$p_k = r_k + \beta_{k-1}p_{k-1}$$ (6.17)

used in Step 6 of the SCG algorithm by $r_k$ to obtain

$$< p_k, r_k > = < r_k, r_k > + \beta_{k-1} < r_k, p_{k-1} >$$

$$= < r_k, r_k >$$ (6.18)

since $< r_k, p_{k-1} > = 0$ by c) of the CG Theorem. Now, applying the Equation in Step 3 of the CG algorithm recursively

$$r_k = r_0 - \sum_{i=0}^{k-1} \alpha_i A p_i$$

141
\[ k - 1 = b - a_i \]  

is obtained since \( r_0 = b - Ax_0 = b \). Inserting this expression for \( r_k \) into (6.18)

\[ < r_k, r_k > = < p_k, b > - < p_k, \sum_{i=0}^{k-1} a_i Ap_i > 
= < p_k, b > - \sum_{i=0}^{k-1} a_i < p_k, Ap_i > 
= < p_k, b > \]

(6.20)

is obtained since \( < p_k, Ap_i > = 0 \) for \( i = 0, 1, \ldots, k-1 \) by b) of the CG Theorem.

Q.E.D.

The inner product term \( < r_k, r_k > \) is computed at each iteration of the original CG algorithm and \( < p_k, b > \) is to be computed to achieve the proposed check \( < x_{k+1}, Ap_k > = < p_k, b > \). Hence, the check \( < p_k, b > = < r_k, r_k > \) can be achieved simply by a scalar comparison without introducing any extra inner product computations. Note that, the condition \( < p_k, Ap_i > = 0 \) for \( i = 0, 1, \ldots, k-1 \) is used in the proof of the above corollary. Hence, this check constitutes an overall check on the A-orthogonality of the set of vectors \( \{ p_i \}^{k}_{i=0} \). If there is an initial guess \( x_0 \neq 0 \) better than the zero vector, then this powerful check can be replaced by

\[ < p_k, r_0 > = < r_k, r_k > \]

(6.21)

which requires the computation of an extra inner product term. The inner products \( < r_{k+1}, r_{k+1} > \) and \( < p_k, Ap_k > \), generated during the iterations of the basic SCG algorithm, should satisfy

\[ < r_{k+1}, r_{k+1} > \geq 0 \]

(6.22)
since it is an Euclidean norm and

\[ < p_k, A p_k > > 0 \]  \hspace{1cm} (6.23)

since \( A \)–matrix is positive definite and \( p_k \) cannot vanish during the \( CG \) iterations (if \( < r_k, r_k > \neq 0 \)). These conditions can be easily checked by only performing two extra scalar comparisons per iteration. Due to the accumulation of numerical errors a small tolerance should be allowed in checking the equalities during implementation. Only four extra inner product computations

1. \( < r_{k+1}, r_k > \)
2. \( < p_{k+1}, A p_k > \)
3. \( < x_{k+1}, A p_k > \)
4. \( < p_k, b > \)

are required per \( SCG \) iteration to perform these checks.

6.3 Distribution of Computation Steps for Error Detection

As was mentioned earlier, in order to compute the distributed inner products, the partial sums computed by each processor must be added and the final sum should be distributed to all processors. In the distributed \( GS-GB \) algorithm, any error due to a single processor or link failure during the distributed \( GS \) phase may cause the root processor to accumulate a wrong value for the distributed inner product. In this case, all the processors of the hypercube may receive the same wrong copy of the inner product. In the distributed \( EA \) algorithm, \( P \) copies of the inner product are accumulated in \( P \) processors by running \( P \) concurrent \( GS \) algorithms with \( P \) different root processors. Hence, any error due to a single processor or
link failure during the distributed EA operations will cause different inner product values to be accumulated at some processors. Therefore, the EA algorithm is more suitable for error detection by comparison. As a consequence, the EA algorithm is incorporated into the distributed inner product computations required in the CED SCG algorithm. The EA algorithm that avoids message collisions on serial communication links was shown in Fig. 25 for a 4-dimensional iPSC/d4 hypercube.

Now, the issue is to find the best points of the original parallel SCG algorithm to compute these extra inner products in order to achieve a low cost error detection scheme. As was mentioned earlier, the set up time \( S_G \) is the real dominating factor in the overall communication costs. It can be noted that, only one DP word is transferred between nearest neighbor processors during 2d concurrent communication steps of the EA algorithm. Therefore, it is desirable to compute these extra inner products in the same EA communication step together with the inner products required in the original SCG algorithm. The redundancy in communication is kept low by this way. For example, the extra inner products \(< r_{k+1}, r_k >\) and \(< x_{k+1}, Ap_k >\) can be computed together with the inner product \(< r_{k+1}, r_{k+1} >\) required in the original SCG algorithm during the same EA communication step. Similarly, the extra inner products \(< p_{k+1}, Ap_k >\) and \(< p_{k+1}, b >\) can be computed together with the inner product \(< p_{k+1}, Ap_{k+1} >\).

A low cost check procedure for inner product computations under a single faulty processor or single faulty communication link assumption, can be formulated as follows: after the EA steps, processors concurrently exchange the three inner products on a predetermined dimension-\(i\) (channel-\(i\)) and then check the inner product values for the given conditions and compare with their own copies. Thus, two \((d-1)\)-dimensional subcubes monitor each other’s results. The distributed EA starts with an exchange step over dimension-\(i\). The errors during the summation and
communication steps of the EA algorithm will be detected by comparison. The result of this concurrent comparison check will be used to determine efficient recovery points in the CED SCG algorithm for retries in the case of transient errors. This issue will be discussed in the next section. The CED SCG algorithm under a single fault assumption is given below. The initial guess vector is assumed to be chosen as the zero vector. It is also assumed that the EA algorithm is started with an exchange on dimension-i. Only Step 2 and Step 5 of the CED SCG algorithm are given since the other steps are similar to the original concurrent SCG algorithm.

2.a. each processor $P_\ell$ concurrently computes

\begin{enumerate}
  \item $R_\ell = \sum_{i \in M_\ell} p^i_k q^i_k$
  \item $S_\ell = \sum_{i \in M_\ell} p^i_k q^i_{k-1}$
  \item $T_\ell = \sum_{i \in M_\ell} p^i_k b^i$
\end{enumerate}

b. each processor computes its own copy of the following inner products

\begin{enumerate}
  \item $\langle p_k, A p_k \rangle = \sum_{\ell=0}^{P-1} R_\ell$
  \item $\langle p_k, A p_{k-1} \rangle = \sum_{\ell=0}^{P-1} S_\ell$
  \item $\langle p_k, b \rangle = \sum_{\ell=0}^{P-1} T_\ell$
\end{enumerate}
in one EA communication step.

c. each processor exchanges its own copies of three inner products with its neighbor on dimension-i.

d. each processor compares these inner products it received with its own copies.
e. each processor checks these inner products it received for:

\begin{enumerate}
  \item $\langle p_k, A p_k \rangle > 0$
  \item $\langle p_k, A p_{k-1} \rangle = 0$
  \item $\langle p_k, b \rangle = \langle r_k, r_k \rangle$
\end{enumerate}
5.a. each processor $P_\ell$ concurrently computes

\begin{enumerate}
  \item $R_\ell = \sum_{i \in M_\ell} r_{k+1}^i + r_{k+1}^i$
  \item $S_\ell = \sum_{i \in M_\ell} r_{k+1}^i r_k^i$
  \item $T_\ell = \sum_{i \in M_\ell} x_{k+1}^i q_k^i$
\end{enumerate}

b. each processor computes its own copy of the following inner products

\begin{enumerate}
  \item $\langle r_{k+1}, r_{k+1} \rangle = \sum_{\ell=0}^{P-1} R_\ell$
  \item $\langle r_{k+1}, r_k \rangle = \sum_{\ell=0}^{P-1} S_\ell$
  \item $\langle x_{k+1}, A_p_k \rangle = \sum_{\ell=0}^{P-1} T_\ell$
\end{enumerate}

in one EA communication step.

c. each processor exchanges its own copies of three inner products with its neighbor on dimension-i.

d. each processor compares these inner products it received with its own copies.
e. each processor checks these inner products it received for:

\begin{enumerate}
  \item $\langle r_{k+1}, r_{k+1} \rangle \geq 0$
  \item $\langle r_{k+1}, r_k \rangle = 0$
  \item $\langle x_{k+1}, A_p_k \rangle = \langle p_k, b \rangle$
\end{enumerate}

Errors during the check computations are either detected during the comparison step at 2.d and 5.d by subcubes or at check step 2.e or 5.e, in the worst case, by $P-1$ processors. Note that, the comparison checks at Step 2.d and Step 5.d cannot detect any error(s) that may occur during the computations required for the calculation of the initial partial sums at Step 2.a and Step 5.a, if no errors occur during the steps 2.b-d and 5.b-d respectively. These errors are detected by the checks at Step 2.e and 5.e. Also note that, errors that are detected during the comparison Steps at 2.d and 5.d can be also detected during the checks at Steps 2.e and 5.e. These comparison checks are introduced to achieve fast recovery points and also to increase the fault coverage.
6.4 Time Redundancy for the CED SCG Algorithm

The expression for the time redundancy of the proposed CED SCG algorithm per SCG iteration is

\[ T_{\text{red}} = 4 \frac{N}{P} T_{\text{calc}} + 4S_C + 8dT_C + 16T_C \]  

(6.24)

where \( N \) is the number of equations, \( P \) is the number of processors in a \( d \) dimensional hypercube (\( P = 2^d \)), \( T_{\text{calc}} \) is the time taken per single DP addition and multiplication, \( S_C \) is the set-up time, \( T_C \) is the data transmission time per single DP word over the serial link. The term \( 4 \frac{N}{P} T_{\text{calc}} \) in (6.24) is due to the four extra inner product computations performed for error checking. The terms \( 4S_C \) and \( 16T_C \) are the total set-up and transmission times respectively, for the additional exchange performed after the EA steps. The term \( 8dT_C \) denotes the increase in the volume of communication during the EA communication steps due to the extra distributed inner product computations.

Hence, the time redundancy \( R \) can be given as

\[ R = \frac{4 \frac{N}{P} T_{\text{calc}} + [4S_C + 8(d + 2)T_C]}{(z + 5)4 \frac{N}{P} T_{\text{calc}} + [4d(S_C + T_C) + L_C]} \cdot 100\% \]  

(6.25)

where \( z \) is the number of nonzero entries per row of the \( A \)-matrix, \( L_C \) is the concurrent local communication time required during a distributed sparse matrix vector product. The simplified analysis given above holds under perfect load balanced conditions.

The percent time redundancy in parallel computations is

\[ R_{\text{comp}} = \frac{4}{z + 5} \cdot 100\%. \]  

(6.26)

For example, in a quadrilateral 2D FE discretization as shown in Figure 2, \( z \approx 18 \) thus resulting in a computational time redundancy of 17.4%. Note that, the percent
computational time redundancy decreases as the number of nonzero entries per row of the coefficient matrix increases. For example, in 3D FE discretization with brick elements, $z \sim 81$. Hence, the computational time redundancy reduces to 4.7%.

The percent time redundancy in communication can be expressed as:

$$R_{\text{comm}} = \frac{4S_C + 8(d+2)T_C}{4dS_C + 4dT_C + 4S_C + 8mT_C} \cdot 100\% \quad (6.27)$$

For 1-D strip partitioning, $L_C = T_{1D}$. Using the expression given in (5.18) for $T_{1D}$, (6.27) becomes

$$R_{\text{comm}} = \frac{4S_C + 8(d+2)T_C}{4dS_C + 4dT_C + 4S_C + 8mT_C} \cdot 100\% \quad (6.28)$$

for a square $m \times m$ FE mesh. In the above equation, the term $(4d+8m)T_C$ in the denominator will be greater than $8(d+2)T_C$ in the numerator for $m > d/2 + 2$ ($m > 4.5$ for $d = 5$). For practical FE meshes implemented in this work $m > 10$. Also note that, $T_C \sim 0.08S_C$ for the iPSC system. As a consequence, the terms associated with the volume of communication can be neglected in (6.28). Hence, (6.28) becomes

$$R_{\text{comm}} \sim \frac{4S_C}{4(d+1)S_C} \cdot 100\% \sim \frac{1}{d+1} \cdot 100\% \quad (6.29)$$

For example, $R_{\text{comm}} \sim 16.67\%$ for a 5-dimensional hypercube. Note that, the percentage of time redundancy in communication decreases with increasing hypercube dimension. Analyzing (6.26) and (6.29), it can be deduced that the overall percent time redundancy depends on $z$ (average number of nonzero entries per row of the $A$-matrix), $d$ (hypercube dimension) and the granularity of the problem. The percent time redundancy is expected to be below 20% on the iPSC/d5 for 2D FE
simulation problems implemented in this work. Hence, it can be concluded that
the proposed CED SCG algorithm provides a considerably low cost error detection
scheme.

6.5 Error Recovery

Error recovery in the proposed CED SCG algorithm is achieved by using retries.
If any one of the check conditions fails, processors retry the computations starting
from a recovery point to get rid of the transient errors. In the following paragraphs,
the recovery points for different check failures are examined. If at least one of the
checks fails after the retry, processors decide that a permanent fault exists and
then they run a distributed fault diagnosis algorithm to locate the faulty processor.
The error recovery in this case, is achieved by carrying out a graceful degradation
procedure which is discussed in the next section.

Two check points exist at each iteration of the CED SCG algorithm. The inner
product values computed during Step 2.a-b and 5.a-b are first checked by com­
parison at 2.d and 5.d, respectively. In order to achieve fast recovery in the case
of transient errors, it is assumed that at most one transient error can occur be­
tween two consecutive check points. Hence, if any one of the comparison checks
at Step 2.d or Step 4.d fails, the failure can be assumed to be due to a transient
error during Steps 2.b-d or Steps 5.b-d, respectively. Therefore, the retry points
for the failure of comparison checks at 2.d and 5.d are 2.b and 5.b, respectively.

A failure in one of the checks \(< p_k, A p_{k-1} > = 0 \) and \(< p_k, b > = < r_k, r_k > \)
indicate error(s) in the computation of the scalar \( \beta_{k-1} \) and/or the vector update

\[ p_k = r_k + \beta_{k-1} p_{k-1} \]
which is performed during the previous iteration. Note that, the vectors \( r_k \) and \( p_{k-1} \) are already checked during the previous iterations. However, in the worst case, a memory error that can occur during fetching the elements of the vector \( p_{k-1} \) for the above vector update, may cause the failure(s) in the above check(s). Hence, the computations starting from the calculation of \( p_{k-1} \) has to be repeated. Therefore, the recovery point for a failure in any one of these two checks at Step 2.e of the \( k \)-th iteration is Step 6 of the \((k-2)\)-th iteration of the CED SCG algorithm.

If however, both of these checks pass, then a failure in the check \( < p_k, Ap_k > > 0 \), in the worst case, indicates an error in the matrix-vector product since the initial sums of the inner products \( < p_k, Ap_{k-1} > \) and \( < p_k, b > \) are computed after the computation of the initial sum for \( < p_k, Ap_k > \) at Step 2.a. Therefore, if the condition \( < p_k, Ap_k > > 0 \) is the only check in failure at Step 2.e, then the recovery point is Step 1 of the current iteration.

A failure in the check \( < r_{k+1}, r_k > = 0 \) indicates error(s) in the computation of the scalar \( \alpha_k \) which includes the matrix-vector product \( Ap_k \) and/or in the vector update

\[
r_{k+1} = r_k - \alpha_k Ap_k .
\]

Similarly, a failure in the check \( < x_{k+1}, Ap_k > = < p_k, b > \) indicates an error either in

\[
r_{k+1} = r_k - \alpha_k Ap_k
\]

or in

\[
x_{k+1} = x_k + \alpha_k p_k
\]

or both. Hence, if one of these checks fails, computations starting from the calculation of \( r_k \) has to be repeated. Therefore, the recovery point for a failure in any
one of these two checks at Step 5.e is Step 3 of the previous iteration. Similarly, if
the condition \( r_{k+1}^T r_{k+1} > 0 \) is the only check in failure at Step 5.e, then the
recovery point is Step 5.a of the current iteration.

6.6 Graceful Degradation

If at least one of the checks again fails after the retry, the SCG iterations are
stopped and a distributed fault diagnosis\[70] algorithm is run to locate the faulty
processor or the faulty link. After diagnosis, the processors of a \((d - 1)\) dimen-
sional hypercube which does not contain the faulty component can continue the
CED CG algorithm. However, in order to achieve this graceful degradation proce-
dure, duplicate copies of the distributed intermediate results of the CG algorithm
should be kept at different processors of the hypercube. A minimal set of duplicate
information necessary to achieve this goal is given in the following paragraph.

In the basic CG algorithm, if the sequence of scalars \( \{\alpha_i\}_{i=0}^k \) and the sequence
of vectors \( \{p_i\}_{i=0}^k \) are known then the vectors \( x_{k+1} \) and \( r_{k+1} \) can be generated
according to

\[
\begin{align*}
x_{k+1} &= x_0 + \sum_{i=0}^{k} \alpha_i p_i & \quad \text{(6.30a)} \\
r_{k+1} &= b - A x_{k+1} & \quad \text{(6.30b)}
\end{align*}
\]

Then the SCG iterations can be continued starting from Step 5 of the SCG al-
gorithm. The vector \( r_k \) which is required in Step 5, can be generated from
\( r_k = r_{k+1} + \alpha_k A p_k \). Hence, the sequence \( \{\alpha_i, p_i\} \) constitutes a minimal set of
information necessary to continue the SCG iterations. Note that, during the CED
SCG iterations, each processor already computes its own copies of the sequence
of scalars \( \{\alpha_i\} \). Also note that, during distributed sparse matrix vector product
(Step 1), each processor \( P_\ell \) already sends the boundary components of its portion
of the distributed $p_k$-vector to a number of neighbor processors $P_m$ for $m \in C_\ell$. Here, $C_\ell$ denotes the set of processors that processor $P_\ell$ has to communicate with. Hence, each processor $P_\ell$ can also distribute the *internal* components of its portion of the distributed $p_k$-vector evenly among its neighbor processors $P_m$ for $m \in C_\ell$ during these concurrent local communication steps. Therefore, the time redundancy in this concurrent duplication step is only due to the increase in the volume of the communication. Note that, the percent increase in the volume of communication is equal to the ratio of the *internal FE* nodes to the *boundary FE* nodes mapped to a processor. As was mentioned earlier, 1-D strip partitioning scheme is found to be superior to the other two schemes on the iPSC system. This ratio $(\text{internal}/\text{boundary})$ is lower in 1-D partitioning scheme compared to the other two schemes. For 1-D strip partitioning, the volume of communication increases from $8m$ to $4m^2/P$ for an $m \times m$ square FE mesh. The percent time redundancy in the local communication for 1-D strip partitioning is

$$R_{LC} = \frac{4m^2/P}{4S_C + 8mT_C} \cdot 100\%.$$  

Since $T_C \sim 0.08S_C$ for iPSC system, $R_{LC}$ is moderately low for 1-D strip partitioning scheme. For a medium size $24 \times 24$ square FE mesh, $R_{LC} \sim 10.8\%$.

The portions of the vectors $x_0$ and $b$ and the rows of the coefficient matrix $A$ mapped onto each processor $P_\ell$ ($\{x_0^i\}_{i \in M_\ell}$, $\{b^i\}_{i \in M_\ell}$ and $\{a^{i,j}\}_{i \in M_\ell,j \in R_i}$, respectively) should be also duplicated once accordingly during the initialization phase of the CED SCG algorithm. Note that, the initial mapping and duplication information should be also kept globally.

The proposed scheme for graceful degradation requires an order of $2k\frac{N}{P}$ extra memory locations for each processor at the $k$-th iteration, since each processor has to preserve its own portion and a subset of its neighbors' portions of the sequence.
of direction vectors generated up to that iteration. This memory problem can be resolved by allowing each processor $P_i$ to distribute its own portion of the unknown vector $x_k$ (i.e. $\{x_k^j\}_{j \in P_i}$) to neighbor processors $P_m$ for $m \in C_i$ during the same local communications at Step 1. However, in the original SCG algorithm, processors do not communicate for the non-local components of the distributed $x_k$-vector. Hence, the time redundancy in communication due to this duplication step can be high if it is repeated at each iteration. The solution is to repeat this duplication for the distributed $x$-vector at certain iteration intervals. The equations given for recovery in Equations (6.30a) and (6.30b) can be modified in the case of duplication of the distributed vector $x$ after each $s$ SCG iterations, as follows:

$$x_{i\sigma+k} = x_{i\sigma} + \sum_{j=i\sigma}^{i\sigma+k-1} \alpha_j p_j \quad (6.32a)$$

$$r_{i\sigma+k} = b - Ax_{i\sigma+k} \quad (6.32b)$$

for $i = 0, 1, \ldots$ and $k = 1, \ldots, s - 1$. The selection of the parameter $s$ is a trade off between time redundancy in communication and memory redundancy.

Now, the issue is to find the most recent fault-free direction vector $p_i$ when the check conditions again fail after the retry. If any one of the checks $<p_k, Ap_{k-1}> = 0$ and $<p_k, b> = <r_k, r_k>$ causing the retry at Step 2.e fails again after the retry, the most recent fault-free direction vector is $p_{k-2}$. If the condition $<p_k, Ap_k> > 0$ which was the only check on failure causing the retry is again the only check on failure at Step 2.e after the retry, the most recent fault-free direction vector is $p_k$. Similarly, if the condition $<r_{k+1}, r_{k+1}> \geq 0$ which was the only check on failure causing the retry is again the only check on failure at Step 5.e after the retry, the most recent fault-free direction vector is $p_k$. If any one of the checks $<r_{k+1}, r_k> = 0$ and $<x_{k+1}, Ap_k> = <p_k, b>$ causing the retry
at Step 5.e fails again after the retry, the most recent fault-free direction vector is $p_{k-1}$. If the comparison checks causing the retries at Step 2.d fail again after the retry, the results of the checks at Step 2.e will be examined. If any one of the checks $<p_k, A p_{k-1}> = 0$ and $< p_k, b > = < r_k, r_k >$ at Step 2.e fails, most recent fault-free direction vector is $p_{k-2}$ otherwise it will be $p_{k-1}$. If the comparison checks causing the retry at Step 5.d fail again after the retry at Step 5.d fail again, the most recent fault-free direction vector will be determined by the results of the following checks at the Step 5.e.

Assume that the faulty processor is found to be the processor $P_f$. Also, assume that the most recent fault-free direction vector is decided to be $p_g$ for some integer $g$. Then each processor $P_m$ for $m \in C_f$ having some duplicate information about the portion of the direction vector mapped onto the faulty processor $P_f$ will concurrently compute

$$
x_{g+1}^i = x_{ts}^i + \sum_{k=ts}^{g} \alpha_k p_k^i \quad (6.33a)
$$

$$
r_{g+1}^i = b^i - \sum_{j \in R_i} a_{i,j} x_{g+1}^j \quad (6.33b)
$$

$$
r_g^i = r_{g+1}^i + \alpha_i \sum_{j \in R_i} a_{i,j} p_j^i \quad (6.33c)
$$

where $i \in M_{fm}$ and $t$ is the largest integer satisfying $s \times t \leq g$. Here, the set $M_{fm}$ denotes the set of duplicate information that processor $P_f$ sends to the processor $P_m$ during the CED SCG iterations. The sequence of vector additions in Equation $(6.33a)$ can be done concurrently by the processors $P_m$ for $m \in C_f$ without involving any communication. Each processor $P_m$ should perform local communication with processors $P_{\ell}$ for $\ell \in C_f$ such that $\ell \neq m$, in order to compute the summation terms in Equations $(6.33b)$ and $(6.33c)$. However, any processor $P_m$ for $m \in C_f$ do not need to communicate with the faulty processor since the
processor $P_m$ can find the fault-free copy of the duplicate information it requires from $P_f$ in one of the processors $P_\ell$ for $\ell \in C_f$ such that $\ell \neq m$.

After the completion of this step, fault-free copies of $A, b, p_g, A p_g, r_g, r_{g+1}$ and $x_{g+1}$ reside distributively in the fault-free processors of the hypercube. Note that, this set represents a complete set of information for the CED SCG iterations to continue. Now, a fault free $(d - 1)$-dimensional subcube should be chosen and a repartitioning of the problem domain onto the chosen fault-free subcube should be performed. Algorithms for finding the maximum dimensional fault-free subcubes in the presence of multiple faulty processors and faulty links are described in Chapter VII.

6.7 Implementation and Experimental Results

The CED SCG algorithm described has been implemented on a 32-node Intel 286-based iPSC/d5 hypercube as part of a FE simulation program for metalforming where linear system of equations are repeatedly generated and solved. The test problem was a 2-dimensional spike forging problem with a 11 by 36 FE mesh and 734 variables (active degrees of freedom). The SCG algorithm required 124 iterations to converge ($tolerance = 10^{-7}$). Both the basic SCG algorithm and the CED CG algorithm were run on the iPSC/d5 hypercube. The parallel execution times for the basic SCG algorithm and the CED SCG algorithm were measured to be 12.15$\text{secs}$ and 14.81$\text{secs}$. Hence, the measured time redundancy is $\sim 22\%$ which is within $5\%$ of the estimated value.

To demonstrate the effectiveness of the error checks, errors were injected by software to various matrix and vector elements at the end of each of the SCG steps during certain iterations. For example, errors during the vector update

$$r_{k+1} = r_k - \alpha_k q_k$$
are simulated by first letting the processor $P_f$ (selected as the faulty processor) calculate the correct values from

$$r_{k+1}^i = r_k^i - \alpha_k q_k^i$$

for all $i \in M_f$ and then changing a randomly selected subset of these values. Hence, a broad class of errors in the scalar-vector product and/or the vector update and/or the memory read/write operations are covered. Similarly generated errors for the matrix-vector product $Ap_k$, cover the errors during the local communication steps as well as those during computations. Errors are introduced in $A$ by changing a randomly chosen subset of the entries of the coefficient matrix mapped to processor $P_f$. Errors during the $EA$ communication step are generated by changing the correct value of the partial inner product after each intermediate communication step. For each case discussed above, the following different types of errors are injected:

1. sign-reversal ($SR$) error
2. zero ($ZE$) error
3. single stuck-at-fault ($SA$) in the binary representation of the $DP$ number
4. random error ($RE$)

Table 11 shows the check outcomes of the $CED SCG$ algorithm for various types of errors injected to processor $P_8$ at different points of the $20^{th}$ iteration. First column indicates the step number of the $CED CG$ algorithm during which the error is introduced. Second column shows the erroneous variable and third column indicates the type of error injected. Columns 4 and 5 show the correct value and the erroneous value of the variable. Column 6 shows the iteration at which the
error is detected. Columns 7-14 show the outcomes of the checks of the \textit{CED CG} algorithm. In these columns, '0' indicates that the check fails to detect the error, '1' indicates that the check detects the error and 'x' indicates that the \textit{CED CG} algorithm has detected the error in a previous step and has already initiated the retry procedure. As discussed earlier, checks were introduced to cover errors affecting different calculations steps in the algorithm and the experimental results show that they are effective. It can be seen that all of the errors are detected by at least one check, within one iteration after they are introduced.

All the errors in Table 1, except the errors injected in Steps 2.b and 5.b, are detected by all the processors of the hypercube. As was discussed earlier, errors injected in Steps 2.b and 5.b are detected by the processors belonging to a subcube during the comparison checks at Steps 2.d and 5.d respectively. Results for single errors are presented in Table 11. Faults which produce errors in all or most of the vector elements or coefficient matrix entries mapped to a processor were also simulated. Such multiple errors found to be even more easily detected than single errors.

The equality checks given in Steps 2.e and 5.e may not be satisfied exactly during the correct operation of the \textit{CED SCG} algorithm due to the accumulation of numerical errors. This situation may introduce spurious false alarms during the iterations. In order to overcome this difficulty, these checks are modified as

\begin{align*}
\left( \frac{\langle r_k+1, r_k \rangle}{\langle b, b \rangle} \right)^{1/2} & \leq \epsilon_1 \quad (6.34a) \\
\left( \frac{\langle p_{k+1}, Ap_k \rangle}{\langle b, Ab \rangle} \right)^{1/2} & \leq \epsilon_1 \quad (6.34b) \\
\left( \frac{\langle x_{k+1}, Ap_k \rangle - \langle p_k, b \rangle}{\langle b, b \rangle} \right)^{1/2} & \leq \epsilon_1 \quad (6.34c)
\end{align*}

157
Table 11: Experimental results showing error detection by the CED SCG algorithm

<table>
<thead>
<tr>
<th>CG Step No.</th>
<th>erroneous variable at iter 20</th>
<th>error type</th>
<th>correct value</th>
<th>erroneous value</th>
<th>det. at iter.</th>
<th>CHECK OUTCOMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$r_{31}^{400}$</td>
<td>SR</td>
<td>$0.58 \times 10^{-2}$</td>
<td>$-0.58 \times 10^{-2}$</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZE</td>
<td>$0.58 \times 10^{-2}$</td>
<td>0.0</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>$0.58 \times 10^{-2}$</td>
<td>$0.77 \times 10^{-2}$</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RE</td>
<td>$0.58 \times 10^{-2}$</td>
<td>$-0.61 \times 10^{-1}$</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td>4</td>
<td>$x_{31}^{400}$</td>
<td>SR</td>
<td>$-0.1485$</td>
<td>$0.1485$</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZE</td>
<td>$-0.1485$</td>
<td>0.0</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>$-0.1485$</td>
<td>$-0.2110$</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RE</td>
<td>$-0.1485$</td>
<td>$0.42 \times 10^{11}$</td>
<td>20</td>
<td>x x x x 0 0 1 0</td>
</tr>
<tr>
<td>5</td>
<td>$p_{31}^{400}$</td>
<td>SR</td>
<td>$-0.17 \times 10^{-1}$</td>
<td>$0.17 \times 10^{-1}$</td>
<td>21</td>
<td>0 0 1 1 x x x x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZE</td>
<td>$-0.17 \times 10^{-1}$</td>
<td>0.0</td>
<td>21</td>
<td>0 0 1 1 x x x x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>$-0.17 \times 10^{-1}$</td>
<td>$-0.23 \times 10^{-1}$</td>
<td>21</td>
<td>0 0 1 1 x x x x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RE</td>
<td>$-0.17 \times 10^{-1}$</td>
<td>$0.89 \times 10^{13}$</td>
<td>21</td>
<td>0 0 1 1 x x x x</td>
</tr>
<tr>
<td>2.a</td>
<td>$p_{20}^{500} \times q_{20}^{400}$</td>
<td>SR</td>
<td>$0.45 \times 10^{-2}$</td>
<td>$-0.45 \times 10^{-2}$</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZE</td>
<td>$0.45 \times 10^{-2}$</td>
<td>0.0</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>$0.45 \times 10^{-1}$</td>
<td>$0.64 \times 10^{-2}$</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RE</td>
<td>$0.45 \times 10^{-2}$</td>
<td>$-0.36 \times 10^{18}$</td>
<td>20</td>
<td>0 0 1 0 x x x x</td>
</tr>
<tr>
<td>5.a</td>
<td>$r_{21}^{500} \times r_{21}^{400}$</td>
<td>SR</td>
<td>$0.14 \times 10^{-1}$</td>
<td>$-0.14 \times 10^{-1}$</td>
<td>21</td>
<td>0 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZE</td>
<td>$0.14 \times 10^{-1}$</td>
<td>0.0</td>
<td>21</td>
<td>0 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>$0.14 \times 10^{-1}$</td>
<td>$0.10 \times 10^{-1}$</td>
<td>21</td>
<td>0 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RE</td>
<td>$0.14 \times 10^{-1}$</td>
<td>$-0.24 \times 10^{20}$</td>
<td>20</td>
<td>x x x x 0 1 0 0</td>
</tr>
<tr>
<td>6</td>
<td>$q_{31}^{500}$</td>
<td>SR</td>
<td>$-0.63 \times 10^{-2}$</td>
<td>$0.63 \times 10^{-2}$</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZE</td>
<td>$-0.63 \times 10^{-2}$</td>
<td>0.0</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>$-0.63 \times 10^{-2}$</td>
<td>$-0.44 \times 10^{-2}$</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RE</td>
<td>$-0.63 \times 10^{-2}$</td>
<td>$0.85 \times 10^{25}$</td>
<td>20</td>
<td>0 1 0 0 x x x x</td>
</tr>
<tr>
<td>1</td>
<td>$a_{500,500}$</td>
<td>SR</td>
<td>$-0.4152$</td>
<td>$0.4152$</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZE</td>
<td>$-0.4152$</td>
<td>0.0</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SA</td>
<td>$-0.4152$</td>
<td>$-0.2902$</td>
<td>20</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RE</td>
<td>$-0.4152$</td>
<td>$-0.96 \times 10^{28}$</td>
<td>20</td>
<td>0 1 0 0 x x x x</td>
</tr>
<tr>
<td>2.b</td>
<td>$\Sigma_{i=1}^{R_1} R_i$</td>
<td>SA</td>
<td>0.1458</td>
<td>0.2083</td>
<td>20</td>
<td>1 x x x x x x x x</td>
</tr>
<tr>
<td></td>
<td>$\Sigma_{i=2}^{R_1} R_i$</td>
<td>SA</td>
<td>0.1620</td>
<td>0.2245</td>
<td>20</td>
<td>1 x x x x x x x x</td>
</tr>
<tr>
<td></td>
<td>$\Sigma_{i=3}^{R_1} R_i$</td>
<td>SA</td>
<td>0.1876</td>
<td>0.2581</td>
<td>20</td>
<td>1 x x x x x x x x</td>
</tr>
<tr>
<td></td>
<td>$\Sigma_{i=4}^{R_1} R_i$</td>
<td>SA</td>
<td>0.2340</td>
<td>0.17158</td>
<td>20</td>
<td>1 x x x x x x x x</td>
</tr>
<tr>
<td>5.b</td>
<td>$\Sigma_{i=1}^{R_1} R_i$</td>
<td>SA</td>
<td>0.1735</td>
<td>0.2360</td>
<td>20</td>
<td>x x x x x x x x</td>
</tr>
<tr>
<td></td>
<td>$\Sigma_{i=2}^{R_1} R_i$</td>
<td>SA</td>
<td>0.1798</td>
<td>0.2423</td>
<td>20</td>
<td>x x x x x x x x</td>
</tr>
<tr>
<td></td>
<td>$\Sigma_{i=3}^{R_1} R_i$</td>
<td>SA</td>
<td>0.1840</td>
<td>0.2465</td>
<td>20</td>
<td>x x x x x x x x</td>
</tr>
<tr>
<td></td>
<td>$\Sigma_{i=4}^{R_1} R_i$</td>
<td>SA</td>
<td>0.2185</td>
<td>0.1560</td>
<td>20</td>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>

158
The right hand side vector $\mathbf{b}$ can be considered as the initial residue and the direction vector for a zero initial guess vector $\mathbf{x}_0 = 0$. Hence, the inner product values are normalized before checking for error detection. The tolerance parameter $\epsilon$ is a small positive number that should be chosen carefully. If the parameter $\epsilon$ is chosen extremely small, false alarms may still be encountered during the iterations thus causing retries hence increasing the completion time of the algorithm. A large value for $\epsilon$ may cause undetected faults. The right choice for the parameter depends on the application program and the precision of the floating point arithmetic used. Note that, two different tolerance values $\epsilon_1$ and $\epsilon_2$ are given in the above inequalities. After performing tests on a wide range of FE sample problems, it was found that the equality condition $\langle \mathbf{p}_k, \mathbf{b} \rangle = \langle \mathbf{r}_k, \mathbf{r}_k \rangle$ becomes more susceptible to the accumulation of numerical errors, compared to the other three equality conditions as the SCG iterations proceed. This is because, the condition in (6.34d) checks the A-orthogonality of the most recent direction vector to all the previous direction vectors generated up to that iteration. For the FE sample problems $\epsilon_1 = 10^{-6}$ and $\epsilon_2 = 0.5 \times 10^{-3}$ are found to give satisfactory results.

During the implementation of the CED SCG algorithm, a local error flag is associated with each processor. A processor sets its own error flag if any one of its own checks fail. These error flags are communicated (twice per iteration) during the communication steps of the EA together with the intermediate partial sum results. A processor sets its own error flag to the faulty state if either its error flag is currently set or the error flag it received from its neighbor is set. Hence, any error decision taken by one or more processors at Steps 2.d-e or Steps 5.d-e of an iteration are propagated to the remaining processors of the hypercube in
Step 5.b of the current iteration or the Step 2.b of the next iteration, respectively. Thus, Steps 2.b and 5.b are in fact the decision points for continuing normal SCG iterations or initiating a software rollback. Furthermore, these error flags are encoded to give information about the retry point decision of each processor when an error is detected. Hence, each processor changes its own error flag to a worse retry point if needed, by comparing its own error flag to the error flag it receives from its neighbor. It is assumed here that no errors occur during the communication of these error flags so that all processors of the hypercube can reach a correct unified decision on either proceeding with the normal SCG iterations or initiating a software rollback to the same point of the algorithm. It should be noted here that, this assumption is made for ease of programming on the iPSC as neither the hardware nor the system software could be modified to incorporate fault-tolerance features.
CHAPTER VII

Reconfiguration Algorithms for a Hypercube Multiprocessor

7.1 Introduction

The set of faulty processors and links in a $d$-dimensional hypercube can be identified by the fault-free processors, by a distributed fault diagnosis algorithm [70,71,72] when the number of faulty processors and links $r \leq d$[70]. Once the faulty elements are identified, graceful degradation can be achieved by reconfiguring the multiprocessor and the distributed algorithm running on the multiprocessor[69]. Most parallel algorithms can be formulated with the dimension $d$ of the hypercube being the parameter of the algorithm[73]. Hence, the reconfiguration problem in a hypercube multiprocessor reduces to finding the maximum dimensional fault-free subcube(s). A simple distributed procedure to find the maximum dimension $m_d$ of a fault-free subcube is given in[73]. However, as indicated in[73], this procedure does not always find $m_d$ and furthermore, it does not construct the set of fault-free $m_d$-subcubes. In this chapter, algorithms which always find $m_d$ and also the complete set of fault-free $m_d$-subcubes are presented. An algorithm using the Inclusion-Exclusion principle of counting is devised for finding the maximum dimension $m_d = d - k$ of the fault-free subcubes as described in Section 7.2. Section 7.3 presents two schemes to construct the $m_d$ dimensional fault-free subcubes once $m_d$ is determined.
7.2 Finding the Maximum Dimension of the Fault–Free Subcubes

Each processor in a $d$-dimensional hypercube can be represented by a $d$-tuple $(b_{d-1} \cdots b_i \cdots b_0)$, where $b_i \in \{0,1\}$. A subcube in a hypercube can be represented by a $d$-tuple $\{0,1,x\}^d$. Coordinate values "0" and "1" can be referred to as bound coordinates and "x" as free. A $(d-k)$-dimensional subcube ($(d-k)$-subcube) in a $d$-hypercube is a $d$-tuple with $k$ bound coordinates and $d-k$ free coordinates. Hence, there are $2^k \binom{d}{k}$ different $(d-k)$-subcubes in a $d$-hypercube, where $\binom{d}{k}$ denotes the $k$-combination of a $d$-set. It can be shown, by making use of the Binomial theorem, that there are a total number of $\sum_{i=1}^{d} 2^i \binom{d}{i} = 3^d - 1$ different subcubes in a $d$-hypercube.

All $(d-1)$-dimensional fault–free subcubes in a $d$-hypercube can be found by searching for a coordinate position that has the same value in all $d$-tuples representing the faulty processors. Such coordinates will be referred to as fixed coordinates. Once such a fixed coordinate position is found, the $d$-tuple obtained by complementing the value of the bound variable at that coordinate position and assigning "x"s to the remaining $d-1$ coordinate positions defines a fault–free $(d-1)$-subcube. For example, for all of the three faulty processors $F_1$, $F_2$, and $F_3$ in Fig. 32, coordinate positions $b_0$ and $b_3$ are both 0, and therefore (xxx1) and (1xxx) are the only 3-subcubes that do not contain $F_1$, $F_2$, and $F_3$. The search for fixed coordinate position can be simply achieved by defining a commutative and associative intersection operation $I_d = \{I\}^d$ on the faulty processors, where $I$ denotes the intersection of individual coordinates as follows;

$$
\begin{align*}
0 I 0 &= 0 \\
1 I 1 &= 1 \\
0 I 1 &= q
\end{align*}
\quad (7.1a) \quad (7.1b) \quad (7.1c)
$$
The bound coordinates ("0"s and "1"s) in the intersection of all faulty processors indicate the fixed coordinates among them. Hence, the number of fixed coordinates, \( n_f \), gives the number of \((d-1)\)-dimensional fault-free subcubes. Finding \((d-k)\)-dimensional fault-free subcubes for \( k > 1 \) is more complex and will be described in Section 7.3.2. First, an algorithm for finding the maximum dimension, \( m_d \), of the fault-free subcubes will be given in this section. This algorithm is based on using the Inclusion–Exclusion\[74\] principle to count the number of subcubes of a given dimension \((d-k)\), that can be formed in the presence of faulty processors and links. An algorithm to construct \( m_d \) dimensional fault-free subcubes, once \( m_d \) is determined, will be described in the next section. The case of faulty processors is discussed next.

7.2.1 Faulty Processors

Lemma 7.1: The number of \((d-k)\)-subcubes destroyed by a faulty processor is

\[
D_{d-k} = \binom{d}{k}
\]  

(7.2)

Proof: Each faulty processor \((b_{d-1} \cdots b_i \cdots b_0)\) in a \( d \) dimensional hypercube destroys or disables the formation of \( d \) fault-free \((d-1)\)-subcubes which can be denoted by \((x \cdots xb_i'x \cdots x)\) for \( i = 0,1,\ldots,d - 1 \), since the faulty processor is a common node for all of these \((d-1)\)-subcubes. In general, a faulty processor will be a common node of those \((d-k)\)-subcubes that can be obtained by fixing any \( k \) positional coordinates of the \( d \)-tuplet representing the faulty processor and then assigning free coordinates ("x"s) to the remaining \((d-k)\) coordinate positions. There are \( \binom{d}{k} \) different ways of choosing \( k \) out of \( d \) coordinate positions. \( Q.E.D. \)
Figure 32: 4-dimensional hypercube with faulty elements

- : faulty processor
- - - : faulty link
For example, the faulty processor $F_2$ in Fig. 32, belongs to the 2-subcubes $(xx00)$, $(x1x0)$, $(0xx0)$, $(x10x)$, $(0x0x)$, $(01xx)$, and therefore destroys these subcubes. Note that, the sets of $(d-k)$-subcubes destroyed by a number of faulty processors may not be disjoint. Let $S^d_{F_i}$ denote the set of $(d-k)$-subcubes that $F_i$ belongs to. Subcubes jointly destroyed by $\ell$ faulty processors $F_{i_1}, F_{i_2}, \ldots, F_{i_\ell}$ or subcubes in $S^d_{F_{i_1}} \cap S^d_{F_{i_2}} \cap \ldots S^d_{F_{i_\ell}}$ can be counted without explicit enumeration of subcubes in each set by using the following lemma.

**Lemma 7.2:** The number of $(d-k)$-dimensional subcubes jointly destroyed by $\ell$ faulty processors $F_{i_1}, F_{i_2}, \ldots, F_{i_\ell}$ is

\[ K_{d-k}(F_{i_1}, F_{i_2}, \ldots, F_{i_\ell}) = \binom{n_f}{k} \]  

where $n_f$ is the number of fixed coordinates in $I_{i_1i_2\ldots i_\ell} = F_{i_1}I_dF_{i_2}I_d\ldots I_dF_{i_\ell}$.

**Proof:** The $\binom{d}{k}$ subcubes of dimension $(d-k)$ in $S^d_{F_i}$ can be constructed by fixing $k$ coordinates in $F_i$ and assigning “x”s to the remaining $(d-k)$ coordinates. The $\binom{n_f}{k}$ subcubes constructed by taking $k$-combinations of the $n_f$ fixed coordinates in $F_{i_1}I_dF_{i_2}I_d\ldots I_dF_{i_\ell}$ and assigning “x”s to the remaining $(d-k)$ coordinates will be common to $S^d_{F_{i_1}} \cap S^d_{F_{i_2}} \cap \ldots S^d_{F_{i_\ell}}$. Q.E.D.

For example, considering the faulty processors $F_1 = (0000)$ and $F_2 = (0100)$ in Fig. 32, $I_{12} = F_1I_dF_2 = (0000)$. The 2-subcubes $(0x0x)$, $(xx00)$ and $(0xx0)$ are destroyed by both $F_1$ and $F_2$.

In counting the number of distinct $(d-k)$-subcubes destroyed by faulty processors $F_1, F_2, \ldots, F_r$, the common $(d-k)$-subcubes destroyed should not be included more than once in the count and should be excluded by using the Inclusion-Exclusion[74] principle of counting given below.

**Theorem 7.1[74]: Principle of Inclusion-Exclusion.** If $N$ is the number of elements in a set $S$, the number of elements of $S$ not having any of the properties
$p_1, p_2, \ldots, p_r$ is given as:

$$N(p_1 p_2 \cdots p_r) = N - \sum_{i=1}^{r} N(p_i) + \sum_{i<j} N(p_ip_j) - \sum_{i<j<k} N(p_ip_jp_k) + \cdots + (-1)^r N(p_1p_2\cdots p_r). \quad (7.4)$$

Here, $N(p_i)$ denotes the number of objects having property $p_i$, $N(p_ip_j)$ denotes the number of objects having both properties $p_i$ and $p_j$, and so on. Objects having the same property $p_i$ are elements of the set $S_i$. The second summation on the right hand side of the equation is over all pairs of sets $S_iS_j$ ($i < j$) and therefore enumerates the number of elements in pairwise intersections of sets $S_i$ and $S_j$. The third summation is over triples $S_iS_jS_k$ ($i < j < k$). Finally, the last term enumerates the number of elements in the intersection of all sets $S_1 \cdots S_r$. This principle can be applied to counting the number of fault-free or available subcubes of a given dimension $(d-k)$, $G_{d-k}$, as described by the following theorem.

**Theorem 7.2:** The number of fault-free (available) $(d-k)$-subcubes, $G_{d-k}$, in the presence of $r$ faulty processors $F_1, F_2, \ldots, F_r$ is

$$G_{d-k} = 2^k \binom{d}{k} - r \binom{d}{k} + \sum_{i<j} K_{d-k}(F_iF_j) - \sum_{i<j<k} K_{d-k}(F_iF_jF_k) + \cdots + (-1)^r K_{d-k}(F_1F_2\cdots F_r) \quad (7.5)$$

**Proof:** To calculate the number of fault-free subcubes, we need to find the number of elements, in $S = \bigcup_{i=1}^{r} S_{F_i}^{d-k}$, where $\bigcup$ denotes the set union, and subtract from $2^k \binom{d}{k}$, the total number of $(d-k)$-dimensional subcubes in a $d$-hypercube. Thus, equation (7.4) can be applied as follows: $N$ is the total number of $(d-k)$-dimensional subcubes in a $d$-hypercube, and property $p_i$ corresponds to being destroyed by processor $F_i$. $\sum_{i=1}^{r} N(p_i) = r \binom{d}{k}$ since each $F_i$ destroys $\binom{d}{k}$ subcubes by Lemma 7.1. $N(p_ip_j)$ is the number of subcubes destroyed by both $F_i$ and $F_j$ or the number of elements in $S_{F_i}^{d-k} \cap S_{F_j}^{d-k}$ and $\sum_{i \neq j} N(p_ip_j)$.
is the sum of the number of subcubes jointly destroyed by all pairs of faulty processors. Note that $N(p_i p_j) = |S_{F_i}^{d-k} \cap S_{F_j}^{d-k}| = K_{d-k}(F_i F_j)$ by Lemma 7.2. 

$N(p_{i_1}, p_{i_2}, \ldots, p_{i_t}) = K_{d-k}(F_{i_1}, F_{i_2}, \ldots, F_{i_t})$ is the number of $(d-k)$-subcubes jointly destroyed by $F_{i_1}, F_{i_2}, \ldots, F_{i_t}$. Thus, $G_{d-k}$ is the number of subcubes not destroyed by any one of $F_1, F_2, \ldots, F_r$ or not having any one of the properties $p_1, p_2, \ldots, p_r$. Q.E.D.

**Example 7.1**: Let the faulty elements in a 4-hypercube be

$F_1 = (0000), F_2 = (0100), F_3 = (0110), F_4 = (1001)$

Thus $d = 4, r = 4$. For $k = 2 (d - k = 2)$;

$I_{12} = (0q00) n_f = 3, I_{13} = (0qq0) n_f = 2, I_{23} = (01q0) n_f = 3,$

$I_{14} = (q00q) n_f = 2, I_{24} = (qg0q) n_f = 1, I_{34} = (qqqg) n_f = 0,$

$I_{123} = (0qq0) n_f = 2, I_{124} = (qq0q) n_f = 1,$

$I_{134} = I_{234} = I_{1234} = (qqqq) n_f = 0,$

$$\sum_{i \neq j} K_{d-k}(F_i F_j) = \binom{3}{2} + \binom{2}{2} + \binom{3}{2} + \binom{2}{2} = 8$$

$$\sum_{i, j, k} K_{d-k}(F_i F_j F_k) = \binom{2}{2} = 1$$

$G_2 = 2^2 \left( \binom{3}{2} \right) - 4 \left( \binom{4}{2} \right) + 8 - 1 = 7$. Since $G_2 \neq 0$, the maximum dimension $m_4 = 2$.

The Venn diagram showing the elements of the sets and their intersections is illustrated in Figure 33.

Note that, if $n_f(I_{i_1 i_2 \ldots i_t}) < 2$, then there is no need to construct higher order intersections of the form $(I_{i_1 \ldots i_t}) I_d F_{i_{t+1}}$ for any $F_{i_{t+1}}$, since $n_f < 2$ for these intersections. Also note that, if $n_f(I_{i_1 i_2 \ldots i_t}) \geq 2$, but $n_f(I_{i_j i_{t+1}}) < 2$ for any $i_j \in \{i_1, i_2, \ldots, i_t\}$, then there is no need to construct the higher order intersection $(I_{i_1 \ldots i_t}) I_d F_{i_{t+1}}$, since $n_f(I_{i_1 \ldots i_t i_{t+1}}) < 2$. These intersections are constructed once,
all 2-d subcubes in a 4-cube

Figure 33: Venn diagram showing the subcubes in $S_{F_1}^2, S_{F_2}^2, S_{F_3}^2, S_{F_4}^2$ and their intersections
and only those intersections for which \( n_f \geq k \) are considered for the computation of \( G_{d-k} \), since these are the only subsets of faulty processors that can destroy common \((d-k)\)-subcubes. Hence, the steps of the algorithm can be given as follows:

1. construct the intersection \( I_{12...r} \) and compute \( n_f \) for this intersection. If \( n_f \geq 1 \) then exit with \( m_d = d - 1 \) and \( G_{d-1} = n_f \).

2. construct all pairwise intersections \( I_{i_1i_2} \) and compute \( n_f \) for these intersections.

3. construct intersections \( (I_{i_1...i_{\ell-1}})I_d F_\ell, 2 < \ell < r, \) for which \( n_f(I_{i_1...i_{\ell-1}}) \geq 2 \) and \( n_f(I_{ij}) \geq 2 \) for all \( ij \in \{i_1, i_2, \ldots, i_{\ell-1}\} \), and compute \( n_f \) for these intersections.

4. for \( k = 2, 3, \ldots \) do
   
   (a) compute \( K_{d-k}(F_{i_1}, F_{i_2}, \ldots, F_{i_\ell}) \) from eq(2), for \( 1 < \ell < r \).
   
   (b) compute \( G_{d-k} \) from Equation (7.5).
   
   (c) exit the loop if \( G_{d-k} \neq 0 \) with \( m_d = d - k \) and \( G_{m_d} = G_{d-k} \).

The initial pairwise intersections required at Step 2 can be constructed by using the logic Exclusive-OR (EXOR) operation. That is,

\[
I_{i_1i_2} = F_{i_1} I_d F_{i_2} = F_{i_1} \oplus F_{i_2} . \tag{7.6}
\]

The "1"s in the resulting \( d\)-tuples of the logic operations indicate the conflicting coordinates "q"s between the pairs of processors. The higher order intersections required at Step 3 can be constructed using the logic OR operation as follows:

\[
I_{i_1i_2...i_{\ell}i_{\ell+1}} = (I_{i_1...i_\ell}) I_d F_{i_{\ell+1}} = (I_{i_1...i_\ell}) \lor (I_{ij}) \tag{7.7}
\]
for any $i_j \in \{i_1, \ldots, i_\ell \}$. Equation (7.7) holds due to (7.1d) and the associativity of the intersection operation. The “0”s in the resulting $d$-tuplets of the logic operations represent the fixed coordinates among the faulty processors. For example, in Fig. 32, $I_{23} = F_2 \oplus F_3 = (0100) \oplus (0110) = (0010)$, $n_f = 3$, and $I_{123} = I_{12} \lor I_{13} = I_{12} \lor I_{23} = (0110)$, $n_f = 2$.

The complexity of the algorithm is analyzed as follows. Each one of the intersections computed at Steps 2 and 3 is constructed in constant time as described above. Note that at most $(2^r - r - 1)$ intersections will be constructed. The time complexity of computing $n_f$ for each intersection in Steps 1, 2 and 3 is $d$. However, each $n_f$ can be computed in constant time by a table look-up of the number of “0”s in the first $P = 2^d$ positive integers. Similarly, the factorial operations required to compute $\binom{n_f}{k}$ in Step 4.a can be performed in constant time by a table look-up of the factorials of the first $d$ positive integers. Thus an upper bound for the time complexity of the above computations for one value of $k$ is $O(2^r)$. Since Step 4 is repeated $(k-1)$ times, the complexity of the given algorithm is

$$T = O(k2^r)$$

which occurs only when all of the intersections have to be constructed and $n_f \geq k$ for each intersection. It has been shown in [73] that an upper bound for $k$ in terms of the number of faulty processors is the largest integer value of $k$ satisfying

$$2^{k-1} \leq r$$

or $k \leq \log_2 r + 1$. Thus the complexity in terms of $r$, the number of faulty elements is

$$T = O(2^r \log_2 r).$$
With the assumption that \( r \leq d[70] \), the complexity can be expressed as

\[
T = O(2^d \log_2 d) \quad (7.11a)
\]

\[
T = O(P \log_2(\log_2 P)) \quad (7.11b)
\]

in terms of \( d \), the dimension of the hypercube and in terms of the number of processors in the hypercube, respectively.

### 7.2.2 Faulty Links

A faulty link is uniquely specified by a \( d \)-tuplet \( \{0, 1, z\}^d \), which contains exactly one "z" at a particular coordinate position. Two unique processors connected via this faulty link can be obtained by assigning "0" and "1" to the single "z" coordinate.

**Lemma 7.3**: The number of \( (d-k)\)-snbc's destroyed by a faulty link is

\[
D_{d-k} = \binom{d-1}{k} \quad (7.12)
\]

**Proof**: It can be easily seen that, a subcube containing two adjacent (directly connected) processors must contain the link between these processors. Hence, a faulty link belongs to those subcubes containing both of the two processors connected via this link. Thus, the \( (d-k)\)-subcubes destroyed by a faulty link can be obtained by assigning one of the "x"s to the unique coordinate position having "z" and then assigning the remaining \( (d-k-1) \) "x"s to combinations of the remaining \( d-1 \) coordinate positions of the \( d\)-tuplet. Since there are \( \binom{d-1}{d-1-k} = \binom{d-1}{k} \) different ways of choosing \( (d-1-k) \) out of \( d-1 \) coordinate positions, the number of \( (d-k)\)-subcubes destroyed by a faulty link is \( D_{d-k} = \binom{d-1}{k} \). \( Q.E.D. \)

For example, the faulty link \( F_5 = (011z) \) in Figure 32 destroys the 2-subcubes, \((x11x), (0x1x), \) and \((01xx)\). Hence, the term \( r \binom{d}{k} \) in Equation (7.5) should be
replaced by \( s \binom{d}{k} + t \binom{d-1}{k} \), where \( s \) and \( t \) denote the number of faulty processors and the faulty links, respectively and \( r = s + t \).

The number of common \((d-k)\)-subcubes destroyed by a number of faulty links and faulty processors can be computed using Equation (7.3) by expanding the definition of the intersection operation \( I \) used to find the fixed coordinates. Since the subcubes destroyed by a faulty link have an "x" in the "z" coordinate position, the intersection operation \( I \) for the "z" coordinates is defined as follows:

\[
1 \ I z = 0 \ I z = z \ I z = z \ I q = q . 
\]  

\[ (7.13) \]

For example, considering the faulty processor \( F_2 = (0100) \) and the faulty link \( F_5 = (01lz) \) in Fig. 32, \( I_{25} = (01qq) \). The 2-subcube \((01xx)\) is destroyed by both \( F_2 \) and \( F_5 \).

Logic operations can be used to construct the intersections containing faulty links as before. The \( z \) coordinate positions of the \( d \)-tuplets representing the faulty links are ignored in performing the logic \textit{EXOR} operation for the initial pairwise intersections required at Step 2 and the \( z \) coordinate positions are then set to "1" by masking using the logic \textit{OR} operation. This assures that they will be treated as "q"s. The higher order intersections required at Step 3 can be constructed by \textit{OR}’ing these initial pairwise intersections as described for the faulty processor case. Hence, the worst case time complexity of the algorithm remains the same with \( r = s + t \).

It should be noted here that, the "z" coordinates of the faulty links should be ignored during constructing the intersection \( I_{12...r} \) at Step 1 of the algorithm. The reason for this requirement will be discussed in Section 7.3.2. The set of subcubes destroyed by a faulty link is a proper subset of the set of subcubes destroyed by any one of the two processors connected via this link. Hence, a faulty link can be
removed from the list of faulty elements if one of the processors connected by this link is also faulty. This can be done by checking for \( n_f = d - 1 \) for intersections between a faulty link and a faulty processor which indicates that the faulty link is connected to the faulty processor.

**Example 7.2:** Including the faulty links, the faulty elements in the 4-hypercube shown in Fig. 32 are:

\[
F_1 = (0000), F_2 = (0100), F_3 = (0110), F_4 = (1111) F_5 = (011z)
\]

**k = 1:** Since \( I_{12345} = (qqqq) \), ignoring z-coordinates, \( m_4 \neq 3 \).

**k = 2:** Note that, \( I_{35} = (011q) \) with \( n_f = d - 1 = 3 \), indicating that faulty link \( F_5 \) is connected to the faulty processor \( F_3 \). Hence, remove \( F_5 \) from the above list and construct all possible intersections of the first 4 faulty elements \((s = 3, t = 1)\).

\[
\begin{align*}
I_{12} &= (qq00) n_f = 3, & I_{13} &= (qq0q) n_f = 2, & I_{23} &= (01q0) n_f = 3, \\
I_{14} &= (qqqq) n_f = 0, & I_{24} &= (q1qq) n_f = 1, & I_{34} &= (q1qq) n_f = 1, \\
I_{123} &= (0qq0) n_f = 2.
\end{align*}
\]

\[
G_2 = 2^2 \left( \binom{4}{2} - 3 \binom{3}{3} + \left[ 2 \binom{3}{2} + \binom{2}{2} \right] - \binom{2}{2} \right) = 9
\]

Hence, \( m_4 = 2 \).

Note that, the third order intersections, \( I_{124}, I_{134}, \) and \( I_{234} \) are not constructed since they contain \( I_{14}, I_{24}, \) and \( I_{34} \) for which \( n_f < 2 \).

### 7.2.3 Faulty Subcubes

If some of the faulty processors are neighbors then the complexity of given algorithm can be decreased by combining the 0-subcubes representing the faulty processors into higher dimensional faulty subcubes using the well known Quine-McCluskey[75] tabular minimization method. Note that, all the processors belonging to a faulty subcube are faulty. The Quine-McCluskey method is basically a very cleverly organized method for making an exhaustive search for all combinations of 0-subcubes.
(minterms) of a logic function into larger subcubes and then selecting the minimal combination of subcubes required to realize the logic function. Hence, a single output switching function of \( d \) binary variables whose minterms are the 0-subcubes representing faulty processors can be defined. Then, the Quine-McCluskey algorithm can be used to find the minimal realization of the defined switching function. After selecting the minimal number of faulty subcubes covering the faulty processors, the given algorithm can be applied to this minimal set of \( p \) faulty subcubes instead of the complete set of \( r \) faulty processors where \( p \leq r \) is guaranteed by the Quine-McCluskey algorithm.

Lemma 7.4: The number of \((d-k)\)-subcubes destroyed by a \(d_f\)-dimensional faulty subcube \( C \) is

\[
D_{d-k}(C) = \sum_{i=\text{Max}(0,k-d_f)}^{\text{Min}(k,d-d_f)} 2^{k-i} \binom{d-d_f}{i} \binom{d_f}{k-i}
\]

(7.14)

Proof: A faulty \(d_f\)-subcube \( C \) will contain \(d_f\) free coordinates ("x"s) and \(d-d_f\) bound coordinates. The \((d-k)\)-subcubes that have at least one common node with a faulty \(d_f\)-subcube can be obtained as follows. First, fix any \(i\) coordinates of the \(d-d_f\) bound coordinate positions in \( C \). Then, fix any \(k-i\) coordinates of the \(d_f\) free coordinate positions in \( C \) by assigning either "0"s or "1"s to these coordinate positions. Finally, assign "x"s to the remaining \(d-k\) coordinate positions. There are \(\binom{d-d_f}{i}\) different ways of choosing \(i\) out of \(d-d_f\) coordinate positions. There are \(2^{k-i}\binom{d_f}{k-i}\) different ways of assigning bound variables to \(k-i\) out of \(d_f\) free coordinate positions.

Q.E.D.

For example, the faulty 2-subcube (01xx) in a 4-hypercube has common nodes with the 3-subcubes (x1xx), (0xxx), (xxx0), (xx1x), (x0x0), (xx1x), and therefore destroys these subcubes. The intersection between the faulty subcubes can be
constructed by expanding the intersection operation defined earlier as follows:

\[ x \cap 0 = 0 \]  
\[ x \cap 1 = 1 \]  
\[ q \cap x = q \]  
\[ x \cap x = x \]  

(7.15a)  
(7.15b)  
(7.15c)  
(7.15d)

The intersections of the subcubes will be \( d \)-\textit{tuplets} of the form \([0, 1, x, q]^d\). Each bound coordinate ("0" or "1") in the intersection of a subset of faulty subcubes indicates a fixed coordinate among them. Each free coordinate ("x") in the intersection indicates two fixed coordinates among them. These fixed coordinates can be obtained by assigning either "1" or "0" to that coordinate position. Let \( S_{C_i}^{d-k} \) denote the set of \((d-k)\)-subcubes that have at least one common node with the faulty subcube \( C_i \). Subcubes jointly destroyed by \( \ell \) faulty subcubes \( C_{i_1}, C_{i_2}, \ldots, C_{i_\ell} \) or subcubes in \( S_{C_{i_1}}^{d-k} \cap S_{C_{i_2}}^{d-k} \cap \cdots \cap S_{C_{i_\ell}}^{d-k} \) can be counted without explicit enumeration of subcubes in each set by using the following lemma.

**Lemma 7.5:** The number of \((d-k)\)-subcubes jointly destroyed by \( \ell \) faulty subcubes \( C_{i_1}, C_{i_2}, \ldots, C_{i_\ell} \) is

\[ K_{d-k}(C_{i_1}, C_{i_2}, \ldots, C_{i_\ell}) = \sum_{i=\text{Max}(0,k-n_x)}^{\text{Min}(k,n_b)} 2^{k-i} \binom{n_b}{i} \binom{n_x}{k-i} \]  

(7.16)

where \( n_b \) is the number of bound coordinates and \( n_x \) is the number of free coordinates in \( I_{i_1}I_{i_2}\cdots I_{i_\ell} = C_{i_1}I_{d}C_{i_2}I_{d} \cdots I_{d}C_{i_\ell} \).

**Proof:** The \((d-k)\)-subcubes common to \( S_{C_{i_1}}^{d-k}, S_{C_{i_2}}^{d-k}, \ldots, S_{C_{i_\ell}}^{d-k} \) can be obtained as follows. First, fix any \( i \) coordinates of the \( n_b \) bound coordinate positions in \( C_{i_1}I_{d}C_{i_2}I_{d} \cdots I_{d}C_{i_\ell} \). Then, fix any \( k-i \) coordinates of the \( n_x \) free coordinates positions in \( C_{i_1}I_{d}C_{i_2}I_{d} \cdots I_{d}C_{i_\ell} \) by assigning either "0"s or "1"s to these coordinate positions. Finally, assign "x"s to the remaining \( d-k \) coordinate positions. There
are \( \binom{n_b}{i} \) different ways of choosing \( i \) out of \( n_b \) bound coordinate positions. There are \( 2^{k-i} \binom{n_x}{k-i} \) different ways of assigning bound variables to \( k-i \) out of \( n_x \) free coordinate positions. \( Q.E.D. \)

For example, considering the faulty subcubes \( C_1 = (001x) \) and \( C_2 = (10xx) \), \( I_{12} = C_1 I_d C_2 = (q01x) \). The 2-subcubes \((x01x), (x0x0), (x0x1), (xx10)\) and \((xx11)\) are destroyed by both \( C_1 \) and \( C_2 \).

Thus, the same Equation (7.5) used in Step 3.b of the algorithm can be used by replacing \( r \left( \binom{d}{k} \right) \) with \( \sum_{i=1}^{P} D_{d-k}(C_i) \), where the expression for each \( D_{d-k}(C_i) \) is given in (7.14), and replacing (7.3) with (7.16). In Step 2 of the algorithm, only those intersections \((I_{ij}...I_{i\ell-1})I_d F_{\ell} \) for which \( n_b(I_{i1}...i_{\ell-1}) + n_x(I_{i1}...i_{\ell-1}) \geq 2 \) and \( n_b(I_{ij}i_{\ell}) + n_x(I_{ij}i_{\ell}) \geq 2 \) for all \( i_j \in \{i_1,i_2,...,i_{\ell-1}\} \) are constructed. Similarly, only those intersections with \( n_b + n_x \geq k \) are considered in the computation of \( G_{d-k} \).

It can be easily seen that, faulty links cannot combine with other faulty links to form a faulty subcube. Similarly, faulty links cannot combine with faulty processors to form a faulty subcube unless the processors connected via these faulty links are also faulty. These exceptional situations can be easily detected as described earlier and those faulty links can be removed from the list of faulty elements. Then, the Quine-McCluskey algorithm can be applied exclusively on the set of faulty processors. The intersections of the faulty links with the faulty subcubes can be constructed by defining the commutative intersection operation between “x” and “z” coordinates as

\[
\text{z I x} = \text{z I q} = q .
\] (7.17)

In this manner, the expression given for \( K_{d-k} \) in Equation (7.16) can be also used for intersection of a faulty link with a faulty subcube.
Example 7.3: Let the faulty processors and links in a 4-hypercube be

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$F_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$F_3$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$F_4$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$F_5$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$F_6$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$F_7$</td>
<td>1</td>
<td>1</td>
<td>z</td>
<td>1</td>
</tr>
<tr>
<td>$F_8$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>z</td>
</tr>
</tbody>
</table>

$k=1$: Since $I_{12} \cdots 8 = (q \cdots q)$, ignoring z-coordinates, $m_4 \neq 3$.

Apply the Quine-McCluskey algorithm on the set of faulty processors ($F_1 \cdots F_6$) to find the minimal set of faulty subcubes $C_1 = (0xx0)$ and $C_2 = (10x1)$. Note that, $I_{58} = (100q)$ with $n_b = d - 1 = 3$, indicating that faulty-link $F_8$ is connected to the faulty processor $F_5$. Hence, remove $F_8$ from the above list. Then the reduced set of faulty elements:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>$C_2$</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>$L_3$</td>
<td>1</td>
<td>1</td>
<td>z</td>
<td>1</td>
</tr>
</tbody>
</table>

where the faulty link $L_3 = F_7$ in the original list.

$k=2$: Construct all possible intersections.

$I_{12} = (q0xq), n_b = 1, n_x = 1$

$I_{13} = (q1qq), n_b = 1, n_x = 0$

$I_{23} = (1qq1), n_b = 2, n_x = 0$
\[ D_2(C_1) = \sum_{i=0}^{2} 2^{2-i} \binom{2}{i} \binom{2}{2-i} = 13 \]
\[ D_2(C_2) = \sum_{i=1}^{2} 2^{2-i} \binom{3}{i} \binom{1}{2-i} = 9 \]
\[ D_2(L_3) = \binom{3}{2} = 3 \]
\[ K_2(C_1C_2) = \sum_{i=1}^{2} 2^{2-i} \binom{1}{i} \binom{1}{2-i} = 2 \]
\[ K_2(C_2L_3) = \sum_{i=2}^{2} 2^{2-i} \binom{2}{i} \binom{0}{2-i} = 1 \]

Hence,
\[ G_2 = 2^k \binom{d}{k} - [D(C_1) + D(C_2) + D(L_3)] + [K(C_1C_2) + K(C_2L_3)] \]
\[ G_2 = 2^2 \binom{4}{2} - 22 = 2 \neq 0 \text{ hence, } m_d = 2 \]

The individual intersection operations required at Step 2 can be performed in \( d \)-time instead of constant time, since logical EXOR and OR operations cannot be used in this case. Hence, if the complexity of the Quine-McCluskey algorithm used to find the minimal set of \( p \) faulty-subcubes is neglected for small \( r \leq d \), the upper bound for the complexity of the algorithm becomes

\[ T = O(dk2^p) \quad (7.18) \]

where \( p \leq r \) is the minimal number of faulty–subcubes covering \( r \) faulty processors. Note that, the algorithm proposed here becomes exponential in terms of the total number of processors \( (P = 2^d) \) as \( r \) approaches \( P \). An alternative algorithm for very large number of faulty processors \( (r \gg d) \) can be formulated by defining a switching function having all \( P - r \) of the fault-free processors as its minterms. Then, by applying only the first step of the Quine-McCluskey algorithm, the set of all prime implicants of this function is constructed. The dimension of the maximum dimensional prime implicant(s) gives \( m_d \) and the set of maximum dimensional prime implicants gives the set of maximum dimensional fault–free subcubes. However, this approach cannot handle faulty links. Furthermore, for large dimensional hypercubes with small number of faults \( (r \leq d) \), the number of minterms in the
switching function defined above will be close to \( P \), hence the complexity of this approach will be much worse when compared to the algorithms proposed here.

7.3 Finding the Maximum Dimensional Fault-Free Subcubes

The problem now is to find the \( G_{m_d} \), \( m_d \)-dimensional fault-free subcubes, once \( m_d = d - k \) and \( G_{m_d} \) are found using the algorithm given in Section 7.2. First, a brute force approach will be given. Then, a much more efficient procedure, which checks the missing combinations for the bound coordinates at all possible coordinate position combinations, will be presented.

7.3.1 The Brute Force Approach

The brute force approach is to form the \( d \)-tuplet representations of all \( 2^k \binom{d}{k} \) \((m_d = d - k)\)-subcubes and check each candidate subcube to see whether it contains any one of the faulty processors or links. Detection of a fault-free subcube can be achieved by defining a new commutative intersection operation, \( I \), between individual coordinates of the \( d \)-tuplets representing the candidate subcubes and the faulty processors and links as follows:

\[
\begin{align*}
0 \times 0 & = x \times 0 = 0 \quad (7.19a) \\
1 \times 1 & = x \times 1 = 1 \quad (7.19b) \\
0 \times 1 & = 0 \times z = 1 \times z = q \quad (7.19c) \\
x \times z & = z \quad (7.19d)
\end{align*}
\]

Hence, the \( d \)-tuplets representing the intersections will be of the form \( \{0, 1, q, z\}^d \). If any intersection between a subcube and a faulty processor (link) is of the form \( \{0, 1\}^d(\{0, 1, z\}^d) \) then that intersection will be equal to the faulty processor (link) and it will indicate that the subcube contains that faulty processor (link). For
example, considering a candidate 2-subcube (01xx) and a faulty processor (0100) and a faulty link (01z1), \( (01xx)_{d}(0100) = (0100) \), and \( (01xx)_{d}(01z0) = (01z0) \). The candidate 2-subcube (01xx) contains both the faulty processor and the faulty link. On the other hand, if the faulty processor and link are (1011) and (0z11), respectively, then \( (01xx)_{d}(1011) = (0q11) \), and \( (01xx)_{d}(0z11) = (0q11) \). In this case, the candidate 2-subcube (01xx) does not contain any faulty element. Hence, a subcube can be declared as fault-free iff all of its intersections with the faulty processors and links contain at least one conflicting-coordinate ("q").

If there are faulty subcubes, then the intersection operation defined above should be expanded as follows:

\[
\begin{align*}
0 \times x &= 0 \\
1 \times x &= 1 \\
x \times x &= x
\end{align*}
\]

Hence, the \( d \)-tuplets representing these intersections will be of the form \( \{0, 1, q, x\}^d \). Note that, any intersection of the form \( \{0, 1, x\}^d \) will represent the common lower dimensional subcube that belongs to both subcubes. For example, considering a candidate 3-subcube (x0xx), and a faulty 2-subcube (0x0x), \( (0xx)_{d}(0x0x) = (000x) \). The 1-subcube (000x) belongs to both the candidate and the faulty subcube.

Each intersection operation and searching for "q"-coordinates can be performed in \( d \)-time. In the worst case, \( r \) intersection operations have to be performed for each candidate \( m_d \)-subcube, where \( r \) is the total number of faults. Hence, the worst case time complexity of the algorithm is

\[
T = O\left(r d^{k+1}\binom{d}{k}\right)
\]
where $m_d = d - k$ and it requires $d2^k \binom{d}{k}$ extra memory locations to hold the candidate $m_d$-subcubes.

### 7.3.2 Missing Combinations Approach

The search procedure described for $k = 1$ at the beginning of Section 7.2, corresponds to a search for a missing value for a single bound variable at each one of the $d$ coordinate positions. For general $k$, there exist no $(d-k)$-dimensional fault-free subcubes, if all the $2^k$ combinations of $\{0, 1\}^k$ are exhaustively covered at each $(\frac{d}{k})$ different $k$-coordinate position combinations by the set of faulty processors. If any one of the $2^k$ combinations of $\{0, 1\}^k$, at any one of the $k$-coordinate position combinations, is found to be missing in the list of faulty processors, then the $d$-tuplet constructed by assigning that missing $k$-tuplet combination to those particular $k$ coordinate positions and also assigning the remaining $d-k$ coordinates as free ("x"), defines a $(d-k)$-dimensional fault-free subcube. The faulty link case requires some more attention. A faulty link does not destroy the subcubes that contain either "0" or "1" on the "z" coordinate position, since these subcubes contain only one of the two processors connected by that link. In the search for missing $k$-tuplets at a particular $k$-coordinate position combination, the faulty links which contain "z" in one of these $k$-coordinate positions should be ignored. This is because, a fault-free subcube, constructed by assigning the missing combination found by searching the list of faulty processors and faulty links which do not have "z" in one of these $k$-coordinate positions, will have bound coordinates at the "z" coordinate positions and therefore cannot contain those faulty links.

The search for missing combination(s) for each faulty element can be avoided by encoding each $k$-tuplet at a $k$-coordinate position combination of faulty elements as a $2^k$-tuplet and then simply using the logic $OR$ operation on these encoded $2^k$-
tuplets. Encoding is achieved by extracting the \( k \)-bit binary number, \( \{0,1\}^k \), at a particular \( k \)-coordinate position combination and then computing \( 2^{\{0,1\}^k} \). Note that, if the integer value of the \( 2^k \)-tuplet reaches the value \( 2^{2^k} - 1 \) during OR'ing, the search at that \( k \)-coordinate position combination can be aborted, since this indicates that all the \( 2^k \) combinations have already been encountered. Decoding is performed by checking the positions of the "0"s in the resulting \( 2^k \)-tuplet, which represent the missing combinations.

**Example 7.4:** Let the faulty elements in a 6-hypercube with \( m_6 = 4 \) and \( G_{m_6} = 2 \) be

\[
\begin{array}{cccccc}
   b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
F_1 : & 0 & 1 & 1 & 0 & 0 & 0 \\
F_2 : & 0 & 0 & 0 & 1 & 0 & 1 \\
F_3 : & 0 & 0 & 1 & z & 1 & 0 \\
F_4 : & 1 & 0 & 0 & 0 & 1 & 0 \\
F_5 : & 1 & 1 & 0 & 1 & 1 & 1 \\
F_6 : & 1 & 0 & 1 & 1 & 0 & 1 \\
\end{array}
\]

**Encodings for the 2-coordinate position combination "b_1b_0" are:**
\( E_1 = (0001), E_2 = (0010), E_3 = (0100), E_4 = (0100), E_5 = (1000). \)

Note that, \( E_1 \lor E_2 \lor E_3 \lor E_4 \lor E_5 = (1111) = 2^4 - 1 = 15. \) Hence, there is no missing combination at "b_1b_0", and the search at this position combination is aborted after OR'ing the first five \( E_i \)'s.

**Encodings for the 2-coordinate position combination "b_2b_0" are:**
\( E_1 = (0001), E_2 = (1000), E_3 = ignored, E_4 = (0001), E_5 = (1000), E_6 = (1000). \)
\( E_1 \lor E_2 \lor \cdots \lor E_6 = (1001) \neq 15, \) hence, two missing combinations can be decoded as, "01" and "10". Therefore, two fault-free subcubes are (xxx0x1) and (xxx1x0).

Computational complexity of the encoding operation is calculated as follows. For each of the \( r \) faulty elements, \( \binom{d}{k} \) coordinate position combinations and therefore
a total of \( r \binom{d}{k} \) combinations need to be encoded. The complexity of encoding one of these \( k \)-combinations is \( O(k) \), since each combination has to be packed into a \( k \)-tuplet before encoding. Thus the complexity of encoding all combinations is \( O(rk \binom{d}{k}) \). An upper bound on the complexity of the logic OR operation on these combinations is \( O(r \binom{d}{k}) \) since each logic operation can be done in constant time.

The decoding operation to find the missing combinations or the positions of the "0”s in the \( 2^k \)-bit encoded words can be done in \( 2^k \)-time and is repeated once for each of the \( \binom{d}{k} \) coordinate position combinations, if an early abort does not occur. Hence, the upper bound for the complexity of the algorithm is

\[
T = O \left( (rk + 2^k) \binom{d}{k} \right) \tag{7.22}
\]

Since \( 2^k \leq 2r \) from eq. (7.9), the complexity is

\[
T = O \left( rk \binom{d}{k} \right) \tag{7.23}
\]

An upper bound in terms of the number of processors \( P \) can be obtained as follows.

The function \( f(i) = \binom{d}{i} \) is monotonically increasing for values of \( i \) in the interval \([0, \lfloor d/2 \rfloor] \). From eq. (7.9) and since \( r \leq d \) one can conclude that \( k \leq 1 + \lfloor \log_2 d \rfloor \).

On the other hand, \( 1 + \lfloor \log_2 d \rfloor \leq \lfloor d/2 \rfloor \) for \( d > 4 \). Thus,

\[
k \leq 1 + \lfloor \log_2 d \rfloor \leq \lfloor d/2 \rfloor \tag{7.24}
\]

for \( d > 4 \) and therefore \( \binom{d}{k} \leq \binom{d}{\lfloor d/2 \rfloor} \). Substituting \( d = \log_2 P \) and replacing \( r \) with \( d \) since \( r \leq d \) and from (7.24), the complexity \( T \) is expressed as

\[
T = O \left( \left( \frac{\log_2 P}{[\log_2(\log_2 P)] + 1} \right)(\log_2 n)(\log_2(\log_2 P)) \right) \tag{7.25}
\]

for \( d > 4 \). For \( d \leq 4 \), \( \binom{d}{k} \leq \binom{d}{\lfloor d/2 \rfloor} \) and the complexity is

\[
T = O \left( \left( \frac{\log_2 P}{[1/2 \log_2 P]} \right)(\log_2 P)(\log_2(\log_2 P)) \right). \tag{7.26}
\]

183
Note that, the maximum $k$ that can be handled by this scheme is restricted by the logarithm of the number of bits in a single word of the computer used. For a 32-bit word, only $k \leq 5$ can be handled. For large $k > 5$, multiple words can be used for the encoding and decoding described above. However, in this case, the complexity for each logical OR operation will be $O(2^{k-5})$. An alternative data structure for large $k$ is to reserve a one dimensional integer array of size $2^k$ which is initialized to all "0's" at the beginning of the search for each new coordinate position combination. Then, the integer value of each encoded $2^k$-tuple for a particular $k$-coordinate position combination can be used to index and set the appropriate entry of the array to a "1". The decoding is done at the end by checking the positions of the "0"s in this array. The indexing of the array and incrementing the counter can be done in unit time after each encoding. Thus, the same complexity given above holds for any $k$. The only overhead for this scheme is the extra memory requirement of $2^k$ words which is much smaller when compared to the brute force approach.

This approach can also be used to find $m_d$ by exhaustively searching for missing combinations at each $i$-coordinate position combination starting from $i = 2$ and at each step incrementing $i$ by 1 until at least one missing combination is encountered at $i = k$ giving $m_d = d - k$. The time complexity in this case is

$$T = \sum_{i=2}^{k} (ri + 2^i) \binom{d}{i}. \quad (7.27)$$

In fact, for large number of faults, this approach is more efficient than the procedure given in Section 7.2. However, for large dimensional hypercubes with small number of faults, the procedure given in Section 7.2 becomes more efficient to find $m_d$. 

184
CHAPTER VIII

Conclusion

In this dissertation, several techniques for increasing the performance of parallel algorithms on distributed memory message-passing multiprocessor systems were investigated. The techniques derived were effectively implemented for the parallelization of the Scaled Conjugate Gradient (SCG) algorithm on a hypercube connected message-passing multiprocessor. Significant performance improvement was achieved by using these techniques. The implementation achieved almost linear speed-up and showed the scalability of the hypercube topology for Finite Element class of problems. The SCG algorithm was also shown to be suitable for vectorization and near supercomputer performance was achieved on a vector hypercube multiprocessor by exploiting both parallelization and vectorization.

Fault tolerance issues for the parallel SCG algorithm and for the hypercube topology were also addressed. A low cost concurrent error detecting SCG algorithm was developed by exploiting the algorithmic properties and the topological features of the parallel architecture. This algorithm was implemented on a hypercube multiprocessor and it was shown that cost effective algorithm-based fault tolerance could be achieved on commercially available multiprocessor systems. In addition, efficient reconfiguration algorithms for fault tolerance were developed for hypercube multiprocessors.

In Chapter III, coarse grain Scaled Conjugate Gradient (SCG) algorithms with re-
duced inner product dependencies were presented. In these algorithms, the basic $SCG$ algorithm was restructured in such a way that the number of inner products accumulated in a single global sum communication step was increased. Hence, the granularity of the parallel $SCG$ algorithm was increased by decreasing the frequency of the global communication steps and increasing the volume of data involved at each global communication step. Serious limitations were encountered in the implementation of the general $CG-SCG$ algorithms due to the accumulation of numerical errors. The coarse grain partial $SCG$ algorithm proposed to avoid the numerical instability problem was found to be feasible only for applications that require the exclusive use of the partial $SCG$ algorithm. $S$-step $CG-SCG$ algorithms, in general, introduce $(s-1)$ extra matrix vector products per $s$ $SCG$ iterations. Two special cases for $s=1$ ($CG-SCG/s1$) and for $s=2$ ($CG-SCG/s2$) that avoid the matrix vector product overhead were presented. These algorithms are expected to yield higher performance for cases in which the partitioning of the problem domain results in fine grain computations (small problems or large problems on large hypercubes) and for large dimensional hypercubes as the communication overhead is a function of the dimension of the hypercube, independent of the problem size. Although the coarse grain $SCG$ algorithms proposed here were tailored for the hypercube architecture they can easily be modified for implementation on any distributed memory message-passing multiprocessor with a high communication latency.

In Chapter IV, the implementation of a coarse grain $SCG$ ($CG-SCG/s1$) algorithm on a 16-node Intel 386-based $iPSC^2/d4$ hypercube and experimental results were discussed. The algorithm, as was expected, resulted in a higher performance improvement for fine grain problems and for large dimensional hypercubes. The incorporation of a distributed Exchange-Add algorithm that overlaps individual
inter-processor communications in place of the $GS-GB$ algorithm resulted in further performance improvement for fine grain problems and for large dimensional hypercubes. The communication protocol implemented for the local communications during the distributed sparse matrix vector product achieved substantial communication overlap by utilizing the two physical links connecting neighbor processors. The implementation of the algorithm was shown to be scalable and an almost linear speed-up was achieved for large problems. An efficiency of 90% and a speed-up of 15 over a $\mu VAX$ II was achieved on a 16 node $iPSC2/d4$ hypercube for larger problems. The parallel $CG-SCG$ algorithm was incorporated in a Finite Element modeling system (ALPID) for metal deformation, based on a viscoplastic formulation.

In Chapter V, the vectorization and the parallel implementation of $CG-SCG/s1$ algorithm on a 4 node Intel $iPSC-VX/d2$ vector hypercube and experimental results were discussed. The high floating point performance of the vector processors attached to the nodes of the $iPSC-VX$ resulted in fine granularity even for large $FE$ problems on small dimensional hypercubes. The algorithm, as was expected, resulted in substantial performance improvement for fine grain problems on small dimensional hypercubes. The sparse matrix vector computation was efficiently vectorized by exploiting the independent functional units with very low number of pipeline stages. The peak performance of the innermost loop was almost obtained for very low number of non-zero entries per row of the coefficient matrix. The ability of the $iPSC-VX$ system to overlap communication with computation was successfully exploited during the distributed sparse matrix vector computations. An efficiency of 75% and a speed-up of 58 over a $\mu VAX$ II was obtained on a 4 node $iPSC-VX/d2$ vector hypercube for large $FE$ problems.

In Chapter VI, a concurrent error detecting ($CED$) parallel $SCG$ algorithm was
presented. Error detection was achieved by checking the algorithm-specific properties such as orthogonality of certain vectors at the end of each iteration. The computations required for error checking were also distributed among the processors and incorporated in the algorithm with redundant computation when necessary and minimal overhead. A scheme for reconfiguration and recovery after a failure was also presented. The algorithm was implemented on a 32 node Intel hypercube iPSC/D5 and successfully detected the errors injected by software. The resulting time redundancy was estimated and measured to be around 20% for 2-dimensional FE problems. The time redundancy is expected to be around 5% for 3-dimensional FE problems. Although the CED SCG algorithm presented here was tailored for a hypercube architecture, it can be modified for implementation on any multiprocessor.

In Chapter VII, algorithms for reconfiguration of a hypercube multiprocessor in the case of faulty processors and faulty links were presented. The algorithm for finding the dimension and the number of maximum dimensional fault-free subcubes is devised by using the inclusion and exclusion principle of counting. This algorithm was found to be efficient for large dimensional hypercubes with small number of faulty elements. An algorithm for finding maximum dimensional fault-free subcubes was also presented.
APPENDIX A

Architectural Features of the iPSC–VX Vector Processor

In this Appendix, the architectural features of the iPSC–VX vector processor VP are summarized.

A.1 Multiple Functional Units

Arithmetic Functional Units: The multiplier and ALU (ADSP 3210/3220) chip set from Analog Devices is used to perform arithmetic and logic computations. The data path of the multiplier is a two-stage pipe: the first stage performs the mantissa multiply using a 32 × 32 parallel multiplier array and the second performs format adjustment, rounding and exception detection. The ALU also designed as a two stage pipe: the first stage performs alignment and the mantissa addition and the second does format adjustment and rounding. The pipe delay in both units is equal to the VP clockcycle time which is 100ns. The data I/O for both units is performed via 32-bit wide input and output ports. 32-bit operands can be loaded or unloaded every 50ns and hence 64-bit operands can be loaded or unloaded at every VP clockcycle (100ns). The inputs and outputs of both units are fully registered.

The multiplier performs integer (32-bit), single precision (32-bit) and double precision (64-bit) floating point multiplication. The multiplier contains a register file of four registers (M00, M01, M10, M11) which supply the operands for the mul-
tiplication. It is possible to load a whole register bank (M00–M01, or M10–M11) with a 64-bit value. The result of a single precision (SP) floating point multiplication is available on the output port (PROD) of the multiplier three cycles after the initiation of multiplication. The multiplier can sustain the full 10MFLOPS throughput rate for SP scalar vector multiplication. It can only sustain 5MFLOPS performance for the product of two (SP) vectors because of its single input port. Because of the size of its 32 × 32 multiplier array, it performs double precision (DP) floating point multiplication iteratively in four cycles. The result of a DP floating point multiplication is available on the output port of the multiplier 5 cycles after the initiation of the multiplication and a new multiplication can be initiated only three cycles after the initiation of the previous one. Hence, the performance for DP scalar vector multiplication is reduced to 3.33MFLOPS.

The ALU performs integer and SP/DP floating point addition/subtraction and 32-bit logic operations. The ALU contains a register file of eight registers which supply the operands for the ALU operation. Each register file contains four registers designated as A01–A03 and A10–A13. The ALU has two input ports. For a DP operand load, an even indexed (Ax0 or Ax2) register is specified and its register pair is loaded (Ax0/Ax1, or Ax2/Ax3) in a single clock cycle. An ALU operation for both 32 and 64 bit operands can be initiated on any clock cycle. The three port structure of the ALU accommodates the full 10MFLOPS throughput rate for both SP and DP operations. If an ALU operation is initiated on cycle $k$, the result is available on its output port (ALU R) on cycle $k+3$.

Memory Functional Unit: Data Memory (DM) contains a SRAM and a DRAM. SRAM runs ($t_{access} = 100/100ns$ for SP/DP data) at twice the DRAM speed ($t_{access} = 200/250ns$), but the amount of SRAM is limited (16Kbytes and 1Mbytes, respectively). The VP's clockcycle is lengthened from 100ns to 200/250ns for the
DRAM cycles. Several of the VP system data structures reside in SRAM and 5Kbytes of the SRAM can be used by the user to store frequently used vector and scalar data. DRAM is used as the main source for vector and scalar data.

Address Calculation Unit: The Registered ALU (RALU) contains a 32-bit wide dual ported register file and a simple integer ALU. 32 registers are available for use as pointers in reads and writes to DM or as general purpose integer registers. The output of the RALU is directly connected to the Data Memory Address Register (DMAR).

Execution Control Unit: The operation of the VP is controlled by the execution of 64-bit microinstructions from the Program Memory (PM). The execution sequence of the VP microcode is controlled by the ADSP-1401 Program Sequencer (PS). Non-sequential addresses are generated by including conditional and unconditional jump codes in the sequencer's microinstruction field. The PS has four decrementing 16-bit counters to track loops and events and has 64 words of 16-bit RAM that serves as the stack for storing subroutine linkages, jump addresses, counters, and status register.

A.2 Multiple Interconnect Buses

The multiple functional units available on the VP are interconnected to three different buses: Arithmetic bus (A-bus), Data Memory bus (DM-bus), and microcode Literal bus (L-bus). The input and the output ports of the multiplier and the ALU are connected to the A-bus. Furthermore, the output of the multiplier is chained to the ALU. In situations where the results of the multiplier unit is the operand to the ALU unit, chaining permits the result of the first functional unit to be transmitted directly to the second.

The I/O port of the DM and the RALU output latch are connected to the DM-
bus. The A-bus and the DM-bus are interconnected through two sets of staging registers. All data returned from the arithmetic unit to the memory unit must pass through the output FIFO. The output FIFO can store two words and it provides staging of arithmetic results until the DM-bus is available. Similarly, staging of the DM-bus to the A-bus is realized by the Memory Register (MEM). If a RALU operation is performed on cycle \( k \) to initiate a read from a DM location, then the result of this operation is latched to the MAR at the end of this cycle and the addressed operand is latched to the register MEM at the end of cycle \( k+1 \), and it is available (under microcode control) on the A-bus on cycle \( k+2 \).

The literal fields of the microinstruction register and the input ports of the RALU and the PS are connected to the L-bus. One way staging is possible from the DM-bus to the L-bus via two feedback registers, FBRA and FBRB. Using this feedback mechanism the RALU or the PS can be loaded from the FIFO or from any specified location in the DM. External data can be also loaded into the RALU and the PS directly from the literal field of the microinstruction via the L-bus. Note that, the arbitration of these interconnect buses are controlled by the microcode.

A.3 Multioperation Microinstructions

Each of the functional units needs its own microinstruction on every cycle to achieve maximum functional parallelism. The 64-bit microinstruction word contains a separate microinstruction field for each of the functional units and also for the control of the interconnect-bus staging registers.
REFERENCES


193


198


199