New Parallel Algorithms for Direct Solution of Sparse Linear Systems

A Thesis

Submitted by

G. Kartik

for the award of the degree

of

MASTER OF SCIENCE

(by Research)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

July 1996

THESIS CERTIFICATE

This is to certify that the thesis titled **New Parallel Algorithms for Direct Solution of Sparse Linear Systems**, submitted by **G. Kartik**, to the Indian Institute of Technology, Madras, for the award of the degree of Master of Science by Research, is a bonafide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Madras 600 036. [C. Siva Ram Murthy]

Date:

Acknowledgements

First and foremost, I would like to express gratitude to my guide Dr. C. Siva Ram Murthy, whose constant guidance has been the principal moving force behind my thesis work. It is the outcome of his gentle encouragement, invaluable feedbacks, and the countless hours he spent going through the drafts, that this thesis has materialized in the present form. His perseverance and his ability to always keep the larger picture in view without compromising on the finer details are qualities worth being emulated.

I would like to thank the Head of the Department, the department office and the department library for providing all the help when required. I would like to express special thanks to Dr. P. Sreenivasa Kumar who spent time with me discussing important aspects of my work. I would also like to thank Dr. V. V. Rao for providing me with useful tips and suggestions during GTC meetings.

This work was supported by *Indian National Science Academy* and the *Department of Science and Technology*.

It is hard to work alone without company. The company of my PDC lab mates went a long way in filling that void. Balu, who spent so many hours discussing my work and suggesting new ideas, in spite of his busy schedule, was a constant source of encouragement to me. It was indeed a pleasure to work with Bhuvana, Manimaran, Santosh, Sudhakar and Tom. The help given by Godbole and Murthy in running my simulation programs have proved invaluable.

Then, of course, it is imperative that I mention all those MS-ites, (both *chai* and non-*chai* types), from Tapti, Brahms, and Ganga whose friendship I will remember forever. There was never a dull moment during my stay in Cauvery, thanks to the excellent company of my numerous friends, who are too many to mention individually.

It is difficult to express in words, all the support and encouragement I received from my family members. I derive strength from their love and affection.

Abstract

The problem of solving large *sparse systems of linear equations* of the form ($Ax = b$) - i.e. systems of linear equations in which majority of coefficients ($A[i, j]$) are zero - arise in various applications such as finite element analysis, computational fluid dynamics, and power systems analysis. The techniques for solving sparse linear systems involve more complex data structures and algorithms than their dense counterparts. We have developed new parallel algorithms for solution of three classes of sparse linear systems - (i) block tridiagonal linear systems, (ii) sparse symmetric linear systems, and (iii) general sparse linear systems. For the solution of block tridiagonal system of linear equations, we propose a new mapping of the Cyclic Elimination (CE) algorithm onto hypercube multiprocessors. Unlike the previous mapping schemes, in our mapping of the CE algorithm, all communications are restricted to physically adjacent processors, using the concept of *data replication*. For the solution of sparse symmetric linear systems, we propose a new *bidirectional algorithm*, based on Cholesky factorization. Unlike the regular algorithm based on Cholesky factorization, in our algorithm, the numerical factorization phase is carried out in such a manner that the entire back substitution component of the substitution phase is replaced by a single step division. On similar lines, for the solution of general sparse system of linear equations, we propose a new bidirectional algorithm, based on LU factorization. As with the sparse symmetric case, the substitution phase of our algorithm does not have a back substitution component. However, due to absence of symmetry, important differences arise in the ordering technique, the symbolic factorization phase, and message passing during numerical factorization phase. Extensive simulations, comparing the two bidirectional algorithms with their corresponding existing algorithms indicate that, when solving for multiple b-vectors, the speedups obtained from these two bidirectional algorithms steadily overtake those obtained from the corresponding regular algorithms, as the number of b-vectors for which the system is solved increases.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Multiprocessing Systems and Parallel Algorithms

Various scientific computing problems, such as computational fluid dynamics and numerical weather prediction, are highly computationally intensive. The high computational power required for fast solution of such problems is beyond the reach of present day conventional uniprocessors. Furthermore, the performance of uniprocessors tends to display an early saturation in relation to their costs. This implies that even modest gains in performance of a uniprocessor comes at an exorbitant increase in its cost. Thus ordinary microprocessors, which cost many orders of magnitude lower than the fastest serial computers, have only marginally lower performance. By connecting many such microprocessors together to form a *multiprocessor*, we can obtain raw computing power comparable to that of the fastest serial computers available, that too at a considerably lower price.

However, this raw power of multiprocessors needs to be translated to high computational rates that are realizable for actual applications. For this purpose, we need to design efficient *parallel algorithms* that can exploit the maximum possible parallelism available in the problem and deliver the high performance required. Unlike a sequential algorithm, which simply executes a sequence of instructions on a single processor, a parallel algorithm proceeds by dividing a problem into multiple sub-problems. Each of these sub-problems can in turn be solved on different processors in an asynchronous fashion. In addition, a parallel algorithm handles the various interactions that occur between these sub-problems in the form of exchange of messages. In the next section, we look at some of the fundamental issues that crop up in the design of a parallel algorithm.

1.2 Key Issues in Design of Parallel Algorithms

The following two principal issues arise in the design of parallel algorithms.

- *Problem partitioning and mapping* : refers to dividing a problem into a number of co-operating sub-problems (tasks) which can be executed concurrently and assigning these tasks to various processors.
- *Communication* : refers to interaction between various tasks of a parallel algorithm by exchange of messages containing data or control information across the inter-processor links.

A parallel algorithm may execute different number of tasks simultaneously at different instants of time. The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*. The degree of concurrency depends principally upon how amenable a given problem is to parallelization.

The measure of the amount of computation involved in each task of a parallel algorithm is called *task granularity*. Task granularity can be classified as *fine*, *medium*, or *coarse* depending upon the processing levels involved.

Speedup is a simple metric to measure the performance of a parallel algorithm. It refers to the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by a parallel algorithm for solving the same problem on p identical processors. For an ideal multiprocessor system, the speedup is equal to p. In practice, however, depending upon the inter-task dependencies and communication overheads, the speedup is less than p.

1.3 Statement of the Problem

In this thesis, we address the problem of solving the sparse system of linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N \end{aligned}$$

where majority of the coefficients a_{ij} are zero. In other words, we have to solve the linear system $Ax = b$, where A is a *sparse* coefficient matrix (i.e., majority of its elements are zero) of dimension $N \times N$, x is an $N \times 1$ unknown solution vector, and b is an $N \times 1$ known right hand side vector.

In this work we have considered the solution of the following three classes of sparse linear systems.

- *Block tridiagonal linear systems* : in which the coefficient matrix A has nonzeros along the three diagonals as shown below.

$$A = \begin{pmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \ddots & \ddots & \ddots & \\ & & \times & \times & \times \\ & & & \times & \times \end{pmatrix}$$

  Each $\times$ is an $n \times n$ matrix block.

- *Sparse symmetric linear systems* : in which the relation $A[i, j] = A[j, i]$ holds for each element of the coefficient matrix A.

- *General sparse linear systems* : in which the coefficient matrix does not have any specific pattern in the location of nonzeros.

The techniques for obtaining solution for sparse linear systems can be divided into two broad categories - *iterative* and *direct*. Iterative methods, such as Jacobi, Gauss-Seidel, and conjugate gradient methods, converge towards an approximate final solution by means of a sequence of iterations. The number of iterations required to solve a system of linear equations with a desired precision is not known beforehand. Iterative methods do not guarantee convergence towards a final solution, but when they do yield a solution, they are usually less computationally expensive than the direct methods.

Direct methods, such as Guassian elimination, LU factorization, and Cholesky factorization based methods, yield an exact final solution by executing a predetermined number of arithmetic operations. Although these methods are more computationally

intensive than iterative methods, they are important for solving sparse linear systems due to their accuracy, robustness, and generality. In this work we consider the direct methods for solution of sparse linear systems.

1.4 Brief Survey of Relevant Work

The problem of solving a system of linear equations ($Ax = b$) is central to many problems in engineering and scientific computing. Large sparse systems of linear equations arise in various applications such as finite element analysis, computational fluid dynamics, and power systems analysis. Developing fast parallel algorithms for solving sparse linear systems has been the focus of research in recent years not only because they are encountered frequently in scientific computing problems, but also because they usually form the most computationally intensive part of these problems. Furthermore, the techniques for solving sparse linear systems involve more complex data structures and algorithms than their dense counterparts. There is an enormous amount of literature available in this field. The current state of art in developing parallel algorithms for sparse linear systems can be found in [19, 13, 20, 30].

Although there is substantial parallelism inherent in sparse linear systems, efforts made till date to develop efficient parallel algorithms for solving these have achieved only limited success. This is because most of the attempts are based on trying to parallelize good sequential algorithms. However, the goal of a good sequential algorithm i.e., minimizing the total operation count, directly conflicts with the goal of a good parallel algorithm, which is maximizing the number of concurrent sub-problems. Hence, parallelizing the good sequential formulations may not yield good parallel counterparts.

Existing works on parallel algorithms for solving tridiagonal and block tridiagonal systems can be found in [3, 31, 50, 51, 52].

Existing works on solving sparse symmetric and general sparse linear systems can be classified according to the phases of solution that each work addresses. Parallelization of the numerical factorization phase has received much attention [2, 4, 14, 15, 11, 20, 44, 30] due to its being a computationally intensive phase. A class of algorithms called *multifrontal algorithms* has also gained popularity recently [9, 40].

Ashcraft et. al. [5] compare the fan-out, fan-in and multifrontal approaches to sparse numerical factorization.

The substitution phase, which involves solution of triangular systems, has limited inherent parallelism. Therefore efforts towards parallelizing this phase have received much less attention. Solving sparse triangular systems in parallel is discussed in [14, 22, 29].

Literature on the various techniques for the ordering phase can be found in [12, 26, 38, 33, 32]. Work on developing parallel ordering algorithms is fairly rudimentary till date [8, 41, 47]. Work on parallel algorithms for the symbolic factorization phase can be found in [2, 18, 28].

1.5 Contribution of the Thesis

We have proposed new parallel algorithms for the following three problems in our work:

- In the first problem, we have proposed a new mapping of the Cyclic Elimination (CE) algorithm [25] for the solution of block tridiagonal system of linear equations onto hypercube multiprocessors. Unlike the previous mapping schemes, in our mapping of the CE algorithm, all communications are restricted to physically adjacent processors, using the concept of *data replication*.
- In the second problem, we have proposed a new parallel *bidirectional algorithm*, based on *Cholesky factorization*, for the solution of sparse symmetric system of linear equations. Traditionally, the process of obtaining a direct solution of a sparse symmetric linear system, $Ax = b$, where A is a sparse symmetric matrix, involves the four distinct phases - (i) *Ordering*, (ii)*Symbolic factorization* (iii)*Numerical factorization*, and (iv) *Substitution*. For solution of multiple b-vectors, the first three phases are carried out only once to obtain the Cholesky factor L. The substitution phase is then repeated for each b-vector in order to obtain a different solution vector x in each case. Thus, in problems which involve solution of multiple b-vectors, the time taken by repeated execution of substitution phase dominates the overall solution time.

In the bidirectional algorithm based on Cholesky factorization, that we have proposed, the numerical factorization phase is carried out in such a manner that the entire back substitution component of the substitution phase is replaced by a single step division. The application of the novel concept of bidirectional elimination to dense linear systems can be found in [42, 43].

- In the third problem, we have proposed a new parallel *bidirectional algorithm*, based on *LU factorization*, for the solution of general sparse system of linear equations. The traditional method for parallel solution of this class of problem consists of the four phases mentioned above. As with sparse symmetric systems, the numerical factorization phase is carried out in such a manner that the entire back substitution component of the substitution phase is replaced by a single step division. However, due to absence of symmetry, important differences arise in the ordering technique, the symbolic factorization phase, and message passing during numerical factorization phase. The bidirectional substitution phase for solving general sparse systems is the same as that for sparse symmetric systems.

The effectiveness of all our algorithms have been demonstrated by comparing them with their corresponding existing parallel algorithms using extensive simulation studies.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows. In chapter 2, we present an improved mapping of the cyclic elimination algorithm onto hypercube multiprocessors. We also present analytical and experimental performance studies for the new mapping scheme. In chapter 3, we describe new parallel algorithms based on Cholesky factorization for solving sparse symmetric linear systems. We consider the case where the system needs to be solved for multiple b-vectors and compare the new scheme with the existing method for solving sparse symmetric linear systems. In chapter 4, we present new parallel algorithms, based on LU factorization, for solving general sparse linear systems with multiple b-vectors and present comparison with the existing methods based on LU factorization. Chapter 5 concludes the work with a summary of the thesis and pointers to some directions in which the work presented here can be extended.

Chapter 2

Solving Block Tridiagonal Linear Systems on Hypercube Multiprocessors

2.1 Introduction

The numerical solution of block tridiagonal linear system of equations is one of the important classes of problems which occurs in many areas of numerical analysis such as solving partial differential equations using finite difference schemes. The most efficient method for solving block tridiagonal linear systems on a uniprocessor is the Block Gaussian Elimination (BGE) [19]. However, the BGE algorithm is not suitable for multiprocessor environment because of lack of adequate parallelism. On the other hand algorithms such as block Cyclic Reduction (CR) [24], Buneman's algorithm [7], block Cyclic Elimination (CE) [25, 19] and recursive doubling [31] exploit the inherent parallelism present in the problem. For efficient implementation of these algorithms on multiprocessors, the principal challenge lies in reducing the overhead involved in communication between processors. This aim can be achieved by using efficient mapping schemes and overlapping the communication and computation steps.

A mapping of any algorithm onto a hypercube is said to be *desirable* if all communications are restricted to physically adjacent processors. However, the following (statement) result due to Lakshmivarahan and Dhall [31] relates to non-existence of a desirable mapping of the CR and CE algorithms onto base-2 (binary) hypercube.

"In any mapping of the CR or CE algorithm onto a p-node base-2 hypercube, it is necessary that at least $\frac{\log p}{2} - 1$ steps involve communication between processors that are at a distance two or more apart." (For proof refer to [31], pp 364-365.) Further, it has been shown by Johnsson [27] that upon using the binary reflected Gray code mapping [48], the distance between any two communicating processors is no more than two.

However, we show, in this chapter, that it is possible to obtain a desirable mapping of CE algorithm onto hypercube multiprocessors using the concept of *data replication*.

Complete details about mapping of CR or CE algorithm onto a hypercube multiprocessor can be found in [31]. Here we give a brief overview of the major differences between the CR and CE algorithms. The CR algorithm consists of two phases - *reduction* and *substitution*. The CE algorithm consists of only one phase, namely, *reduction*. The degree of parallelism in the reduction phase of CR algorithm halves with every consecutive stage. On the other hand, the degree of parallelism in the reduction phase of CE algorithm remains constant through all stages. Thus, theoretically, CE algorithm ought to be preferred over CR algorithm. However, the communication overhead incurred in the existing mapping of CE algorithm onto hypercubes is much higher than that of CR algorithm. In particular, the communication graph of the CR algorithm is a sub-graph of the communication graph of CE algorithm. The communication overhead incurred by the existing mapping of CE algorithm becomes costly, especially since successive stages of the reduction phase call for data communication between processors which are not neighbours. A large number of such multiple hop data communications lead to link contentions and, consequently, lower performance.

In order to gainfully exploit the higher degree of parallelism of the CE algorithm we propose an improved mapping of the CE algorithm onto a hypercube multiprocessor with which the data communications are restricted to occur between neighbouring processors only. This is achieved by efficient duplication of data at every stage of the algorithm. Thus the problem due to link contentions are overcome and better performance achieved. Two significant features of our algorithm are that, the computational load is balanced among all the processors at all stages of the algorithm and secondly, much of the communication gets overlapped with computation giving an overall better performance.

The rest of the chapter is organised as follows. In section 2.2, we make a problem statement and introduce some notations which will be used in the subsequent sections. In section 2.3, we discuss the sequential BGE algorithm on a uniprocessor, and the parallel CR and CE algorithms. In section 2.4, using an example, we first look at the existing schemes for mapping CR and CE algorithms onto hypercube multiprocessors

and then present our improved mapping scheme for the CE algorithm followed by its analytical performance study. In section 2.5, we present numerical results for the speedups obtained from our new mapping scheme and the existing mapping of CR algorithm, and compare the two schemes. Section 2.6 concludes the work with some pointers for future research.

2.2 Problem Statement and Notations

The block tridiagonal matrix A is defined as

$$A = \begin{pmatrix} d_1 & f_1 & & & \\ e_2 & d_2 & f_2 & & \\ & \ddots & \ddots & \ddots & \\ & & e_{N-1} & d_{N-1} & f_{N-1} \\ & & & e_N & d_N \end{pmatrix}$$

where the components e_i, d_i and f_i are $n \times n$ matrices (or blocks) with $e_1 = f_N = 0$. There are N such blocks along principal diagonal of A where N is a power of 2. So the overall dimension of A is $(Nn) \times (Nn)$. We are to solve the system $AX = b$, where the vector $X = (x_1, x_2, \ldots, x_N)^t$, the vector $b = (b_1, b_2, ..., b_N)^t$, the components x_i and b_i are n-vectors and

$$e_j x_{j-1} + d_j x_j + f_j x_{j+1} = b_j \quad , \quad j = 1, \ldots, N.$$

The CR algorithm for solving the system $Ax = b$ consists of the reduction phase followed by the back substitution phase. Each of these two phases, in turn, is divided into $\log N$ stages. The CE algorithm consists of reduction phase alone which is divided into $\log N$ stages. In both CR and CE algorithms, at the beginning of stage $l = 1$ of the reduction phase, we define the 5-tuple $row_i^{(0)}$ as

$$row_i^{(0)} = (e_i^{(0)}, d_i^{(0)}, (d_i^{(0)})^{-1}, f_i^{(0)}, b_i^{(0)}) = (e_i, d_i, (d_i)^{-1}, f_i, b_i).$$

At each stage $l \in \{1, \ldots, \log N\}$ of reduction phase, we define the tuple $row_i^{(l)}$ as

$$row_i^{(l)} = \begin{cases} (e_i^{(l)}, d_i^{(l)}, (d_i^{(l)})^{-1}, f_i^{(l)}, b_i^{(l)}) & , \quad \forall i \in \{1, \ldots, N\} \\ (0, I, I, 0, 0) & , \quad \forall i \leq 0 \text{ or } i > N. \end{cases}$$

Here $e_i^{(l)}$ is the value of e_i at the end of stage l, $f_i^{(l)}$ is the value of f_i at the end of stage l and so on. The matrix I is the $n \times n$ identity matrix. Note that $(d_i^{(l)})^{-1}$ is included as a member of the tuple $row_i^{(l)}$. This is done because, the inverse, once computed at a source processor, can be transferred along with the tuple $row_i^{(l)}$ to other processors which need it, thus avoiding its re-computation at the destination processors. Figure 2.1 gives an example of the above notations for an 8×8 block tridiagonal system.

$$\begin{pmatrix} d_1 & f_1 & & & \\ e_2 & d_2 & f_2 & & \\ & \ddots & \ddots & \ddots & \\ & & e_7 & d_7 & f_7 \\ & & & e_8 & d_8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_7 \\ b_8 \end{pmatrix}$$

$$A \qquad x \qquad = \qquad b$$

*Stage*1	*Stage*2	*Stage*3
$row_1^{(0)} = (e_1, d_1, (d_1)^{-1}, f_1, b_1)$	$row_1^{(1)} = (e_1^{(1)}, d_1^{(1)}, (d_1^{(1)})^{-1}, f_1^{(1)}, b_1^{(1)})$	$row_1^{(2)} = (e_1^{(2)}, d_1^{(2)}, (d_1^{(2)})^{-1}, f_1^{(2)}, b_1^{(2)})$
$row_2^{(0)} = (e_2, d_2, (d_2)^{-1}, f_2, b_2)$	$row_2^{(1)} = (e_2^{(1)}, d_2^{(1)}, (d_2^{(1)})^{-1}, f_2^{(1)}, b_2^{(1)})$	$row_2^{(2)} = (e_2^{(2)}, d_2^{(2)}, (d_2^{(2)})^{-1}, f_2^{(2)}, b_2^{(2)})$
$row_3^{(0)} = (e_3, d_3, (d_3)^{-1}, f_3, b_3)$	$row_3^{(1)} = (e_3^{(1)}, d_3^{(1)}, (d_3^{(1)})^{-1}, f_3^{(1)}, b_3^{(1)})$	$row_3^{(2)} = (e_3^{(2)}, d_3^{(2)}, (d_3^{(2)})^{-1}, f_3^{(2)}, b_3^{(2)})$

Figure 2.1: An 8×8 block tridiagonal system and listing of $row_i^{(l)}$ at various stages

2.3 Solving Block Tridiagonal Linear Systems

In this section, we first briefly present the theoretical concepts behind the sequential BGE and then the parallel versions of CR and CE algorithms.

2.3.1 Sequential Block Gaussian Elimination (BGE)

There are two phases in this algorithm - forward elimination and back substitution. Computation within each phase is completely sequential in nature.

Algorithm 1

(*Forward elimination phase*)

for $i = 2$ **to** N **do**

Calculate $(d_{i-1})^{-1}$

$a_i = e_i(d_{i-1})^{-1}$

$e_i = 0$

$d_i = d_i - a_i f_{i-1}$

$f_i = fi$

$b_i = b_i - a_i b_{i-1}$

endfor

(*Back substitution phase*)

Calculate $(d_N)^{-1}$

$x_N = (d_N)^{-1} b_N$

for $i = (N-1)$ **downto** 1 **do**

$x_i = (d_i)^{-1}(b_i - f_i x_{i+1})$

endfor.

The time taken for calculating the inverse of an $n \times n$ matrix block, using the *exchange method*, is $T_{inv} = 3n^3 - 4n^2 + 2n$ computational time units. Multiplying two $n \times n$ matrices takes $T_{mult} = 2n^3 - n^2$ time units, whereas multiplying an $n \times n$ matrix with an n-vector takes $T'_{mult} = 2n^2 - n$ time units. The sequential *BGE algorithm* executes N matrix inversions, $2(N-1)$ matrix-matrix multiplications, $3N-2$ matrix-vector multiplications, $N-1$ matrix subtractions, and $2(N-1)$ vector subtractions. Summing up all the components, this step takes

$$T_{BGE} = N(3n^3-4n^2+2n)+2(N-1)(2n^3-n^2)+(3N-2)(2^2-n)+(N-1)n^2+2(N-1)n$$

$$= (N-1)(7n^3+n^2+n) + (3n^3+2n^2+n) \text{ time units.}$$

2.3.2 The Basic Elimination Step

Both CE and CR algorithms, have a basic elimination step in common. We name this step *Compute* $row_i^{(l)}$, where $i \in \{1, \ldots, N\}$ is the index of a row of blocks and $l \in \{1, \ldots, \log N\}$ is the stage being considered. Let $h = 2^{(l-1)}$. The *Compute* $row_i^{(l)}$ step

eliminates the dependence of equation i on the variables x_{i+h} and x_{i-h} by subtracting appropriate multiples of equations $i+h$ and $i-h$ from equation i. The *Compute* $row_i^{(l)}$ step consists of the following computation steps.

$$
\begin{aligned}
&u_i^{(l)} = -e_i^{(l)}(d_{i-h}^{(l-1)})^{-1} \\
&v_i^{(l)} = -f_i^{(l)}(d_{i+h}^{(l-1)})^{-1} \\
&e_i^{(l)} = u_i^{(l)} e_{i-h}^{(l-1)} \\
&d_i^{(l)} = d_i^{(l-1)} + u_i^{(l)} f_{i-h}^{(l-1)} + v_i^{(l)} e_{i+h}^{(l-1)} \\
&\text{Calculate } (d_i^{(l)})^{-1} \\
&f_i^{(l)} = v_i^{(l)} f_{i+h}^{(l-1)} \\
&b_i^{(l)} = b_i^{(l-1)} + u_i^{(l)} b_{i-h}^{(l-1)} + v_i^{(l)} b_{i+h}^{(l-1)}
\end{aligned}
$$

The *Compute* $row_i^{(l)}$ step involves six matrix-matrix multiplications, two matrix-vector multiplications, one matrix inversion, two matrix additions, and two vector additions. Summing up the components, this step takes $e = (15n^3 - 4n^2 + 2n)$ time units.

2.3.3 The Block Cyclic Reduction Algorithm (CR)

The CR algorithm consists of two phases, namely reduction (or elimination) phase and back substitution phase. These two phases are essentially sequential although the computations within each phase can be carried out in parallel. Therefore, the total parallel time is the sum of the individual parallel times. Figure 2.2 shows the pattern of elimination and back substitution steps for the case of $N = 8$ block equations.

Algorithm 2

(*Reduction phase*)

1. **for** $l = 1$ **to** $\log N$ **do**

   $h = 2^{(l-1)}$

   for $i \in \{2^l, 2 \times 2^l, 3 \times 2^l, \ldots, \log N\}$**do in parallel**

   Compute $row_i^{(l)}$

   endfor

   endfor

(*Back substitution phase*)

2. $x_N = (d_N^{(\log N)})^{-1} b_N^{\log N}$

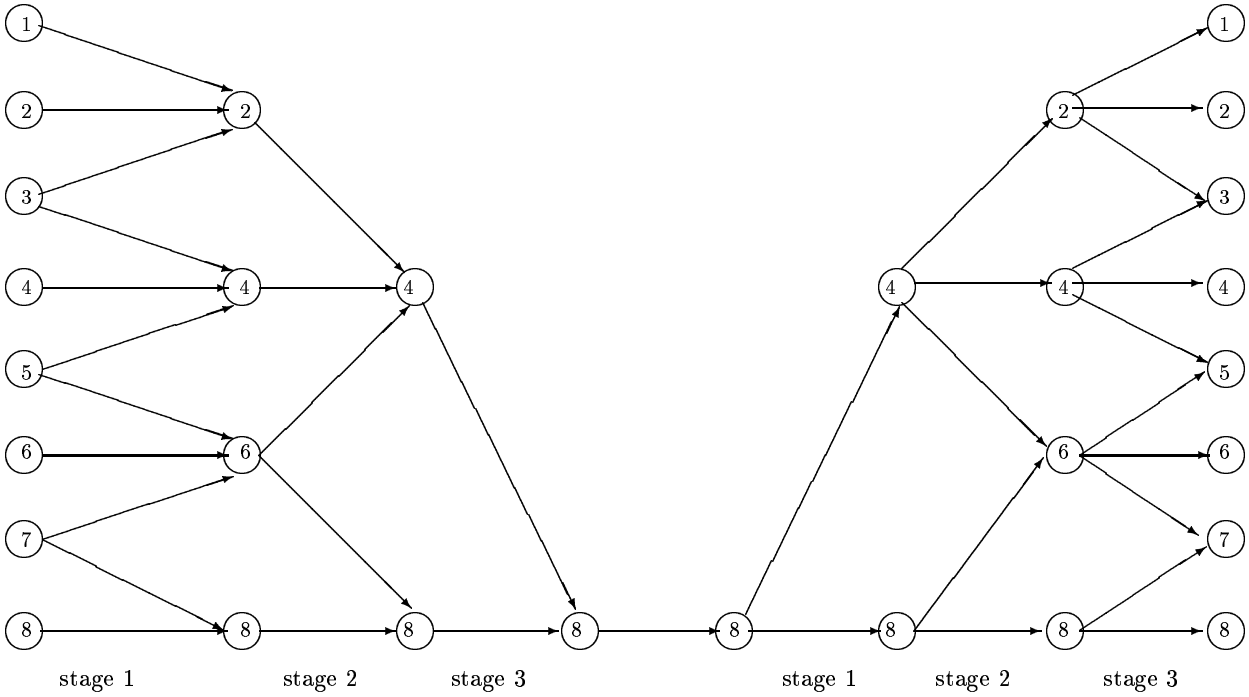


Figure 2.2: Elimination and back substitution pattern in CR algorithm for N=8

3. **for** $l = \log N$ **downto** 1 **do**

   $h = 2^{(l-1)}$

   for $i \in \{2^{(l-1)}, 3 \times 2^{(l-1)}, 5 \times 2^{(l-1)}, \ldots, N - 2^{(l-1)}\}$**do in parallel**

   $x_i = (d_i^{(l-1)})^{-1}(b_i^{(l-1)} - e_i^{(l-1)} x_{i-h} - f_i^{(l-1)} x_{i+h})$

   endfor

endfor.

2.3.4 The Block Cyclic Elimination Algorithm (CE)

The CE algorithm consists of only the elimination phase followed by a single step division. Here the elimination phase recursively converts the given system of equations into two independent systems of equations each of which can be solved in parallel using the CE algorithm. Figure 2.3 shows the pattern of *Compute* $row_i^{(l)}$ steps for the case of $N = 8$ block equations.

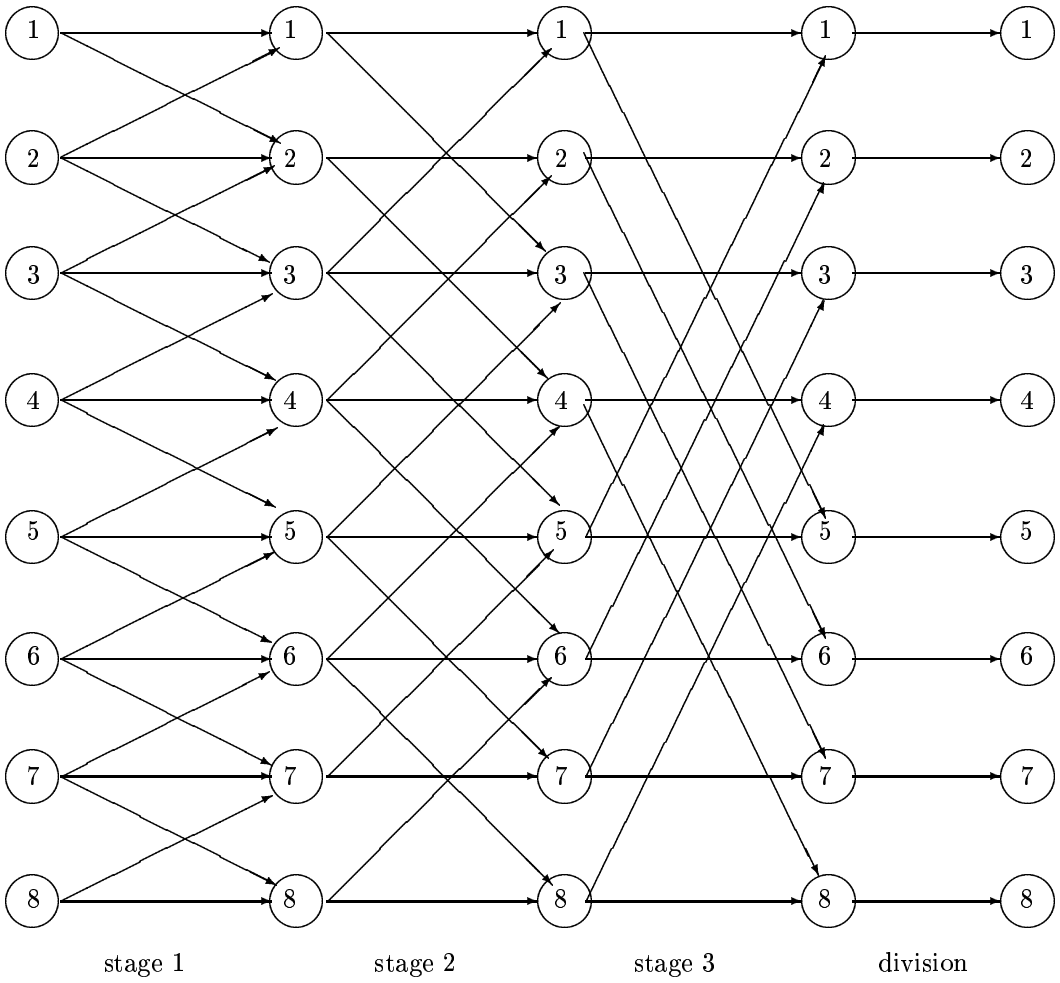


Figure 2.3: Elimination pattern in CE algorithm for N=8

Algorithm 3

1. **for** $l = 1$ **to** $\log N$ **do**

    $h = 2^{(l-1)}$

    for $i \in \{1, 2, \ldots, N\}$**do in parallel**

        Compute $row_i^{(l)}$

    endfor

  endfor

2. **for** $i \in \{1, 2, \ldots, N\}$**do in parallel**

    $x_i = (d_i^{(\log N)})^{-1} b_i^{(\log N)}$

  endfor.

2.4 Solving Block Tridiagonal Linear Systems on Hypercubes

The hypercube, one of the most popular architecture for multiprocessor systems, is a generalization of a cube to d dimensions such that each of the 2^d processors has d neighbours. In this section, we present an improved mapping of the CE algorithm on a hypercube multiprocessor which achieves neighbouring processor communication by efficient use of the concept of data duplication. We begin by comparing the three mapping schemes, namely, the existing mapping of the CR algorithm, the existing mapping of the CE algorithm, and our improved mapping of the CE algorithm with the help of a simple example. We then proceed to formally present our algorithm and explain the various steps.

2.4.1 Comparison of Three Schemes

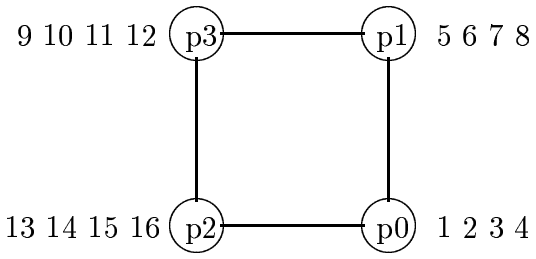
Let us consider the simple problem of solving a block tridiagonal system with $N = 16$ block equations and block size 1×1 (i.e., $n = 1$) on a two-dimensional hypercube (i.e., there are four processors in the hypercube). We trace the step by step execution of each of the schemes below and calculate the time taken in each case. For the sake of simplicity, we consider only the non-overlapped execution of computation and communication steps.

We define the following notations to make our comparison clearer.

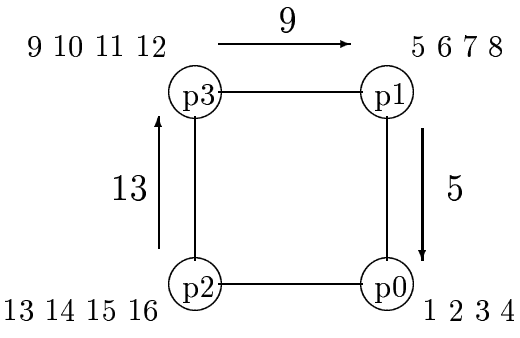
- p_k symbolically represents the kth processor of a hypercube.
- p represents the number of processors in a hypercube. Thus the dimension of the hypercube is $\log p$.
- e represents the number of operations involved in executing the *Compute* $row_i^{(l)}$ with no communication overheads. As shown in section 3.2, this works out to be $e = 15n^3 - 4n^2 + 2n$ computational time units.
- s represents the number of operations involved in executing one back substitution step, which involves three matrix-vector multiplications and two vector subtractions. This works out to be $s = 6n^2 - n$ computational time units.
- *Communication to Computation ratio*, C/E, represents the the ratio of time taken to communicate one floating point value between two neighbouring processors to the time taken to execute one floating point operation.
- T_b represents the time taken to communicate the contents of an $n \times n$ matrix block between two neighbouring processors. This works out to be $n^2(C/E)$ computational time units.
- T_e represents the time taken to communicate the contents of a 5-tuple $row_i^{(l)}$ between two neighbouring processors. This works out to be $5T_b = 5n^2(C/E)$ computational time units.
- *kth dimension* of a hypercube is represented by a set of links each of which connects some processor p_j to its neighbour $p_{j'}$, such that j' is obtained by inverting the kth bit in the binary representation of j.

2.4.1.1 Existing Mapping of the CR Algorithm

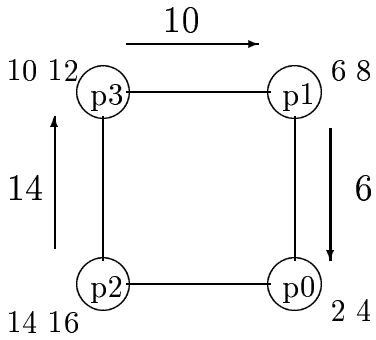
Figure 2.4 shows the step by step execution of the CR algorithm for solving the tridiagonal system of 16 equations using a hypercube of four processors. The equations are initially mapped onto processors in a block wrap manner (see figure 2.4(a)). The reduction phase of the mapped algorithm consists of 4 (i.e., $\log 16$) stages. The first stage consists of a one hop communication of tuples $row_5^{(0)}$ (from processor p_1 to p_0), $row_9^{(0)}$ (from p_3 to p_1), $row_{13}^{(0)}$ (from p_2 to p_3) followed by the computation steps



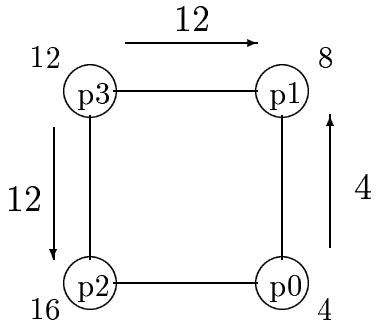
(a) initial data distribution



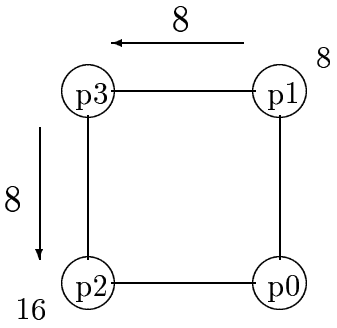
(b) stage 1



(c) stage 2



(d) stage 3



(e) stage 4

Figure 2.4: Progression of the CR algorithm with the existing mapping for N=16 and p=4

Table 2.1: Counts of tasks executed by the CR algorithm

Reduction phase

stage	*task count*	
	T_e	e
1	1	2
2	1	1
3	1	1
4	2	1

Substitution phase

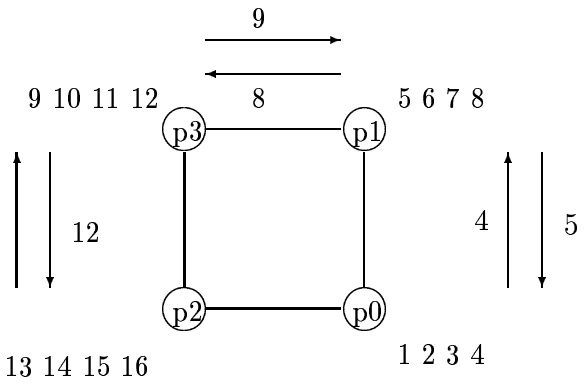
stage	*task count*	
	T_b	s
1	2	1
2	1	1
3	1	1
4	1	2

Compute $row_2^{(1)}$ and *Compute* $row_4^{(1)}$ at p_0, *Compute* $row_6^{(1)}$ and *Compute* $row_8^{(1)}$ at p_1, *Compute* $row_{10}^{(1)}$ and *Compute* $row_{12}^{(1)}$ at p_3 and *Compute* $row_{14}^{(1)}$ and *Compute* $row_{16}^{(1)}$ at p_2. This completes the first stage of reduction phase. Similarly, second and third stages involve one hop communication of $row_i^{(l)}$ tuples and one step each of the form *Compute* $row_i^{(l)}$. Stage 4 consists of a two hop communication of $row_8^{(3)}$ from p_1 to p_2 followed by the step *Compute* $row_{16}^{(4)}$. The substitution phase of the algorithm follows a completely reverse pattern of communication and can be described by reversing the order of the stages and the direction of the arrows in the reduction phase. The data items communicated are the floating point values of the variables x_i (instead of $row_i^{(l)}$ as in reduction phase).

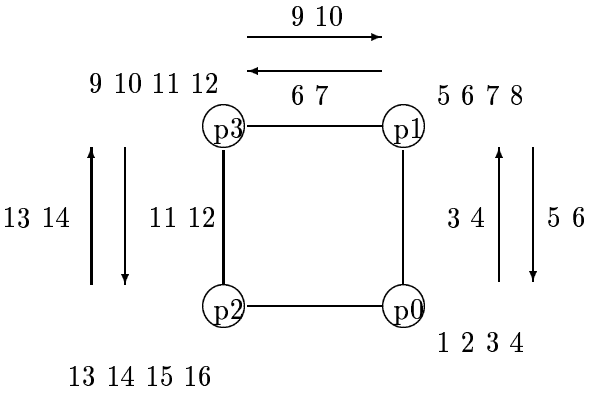
The counts of various tasks executed at each stage of the algorithm are summarised in table 2.1. We see from table 2.1 that it takes $5T_e + 5e$ computational time units for the reduction phase, followed by a division step, followed by $5T_b + 5s$ units for the substitution phase. Thus the total execution time is $T_{CR} = 5(T_e + T_b) + 5(e + s) + 1$ units. Typically the communication to computation ratio (C/E) is of the order of 100. Thus with $N = 16$, $n = 1$ and $p = 4$ we have $T_e = 500$, $T_b = 100$, $e = 13$ and $s = 5$. Thus $T_{CR} = 3091$ computational time units from the above expression.

2.4.1.2 Existing Mapping of the CE Algorithm

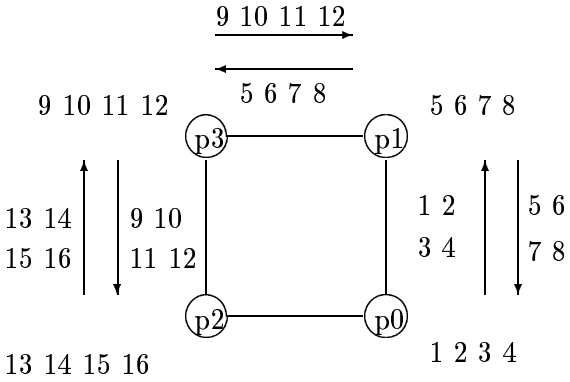
Figure 2.5 shows the step by step execution of the CE algorithm for solving the tridiagonal system of 16 equations using a hypercube with four processors. The equations are initially mapped onto processors in a block wrap manner (see figure 2.5(a)). The



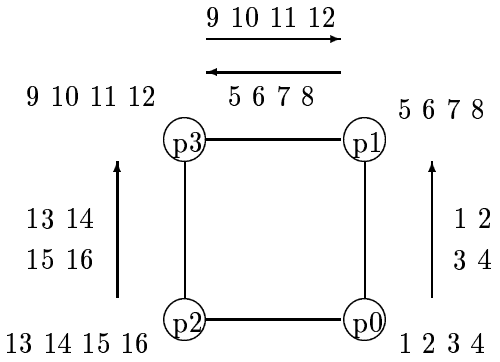
(a) stage 1



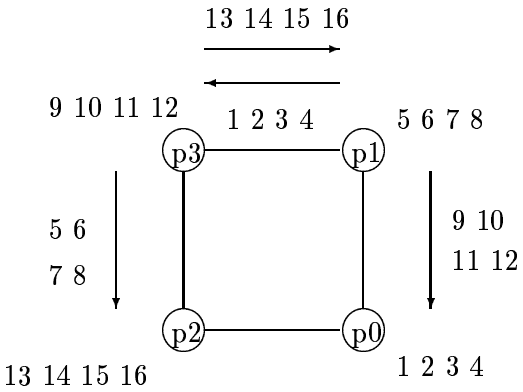
(b) stage 2



(c) stage 3



(d) stage 4 - first hop



(e) stage 4 -second hop and elimination

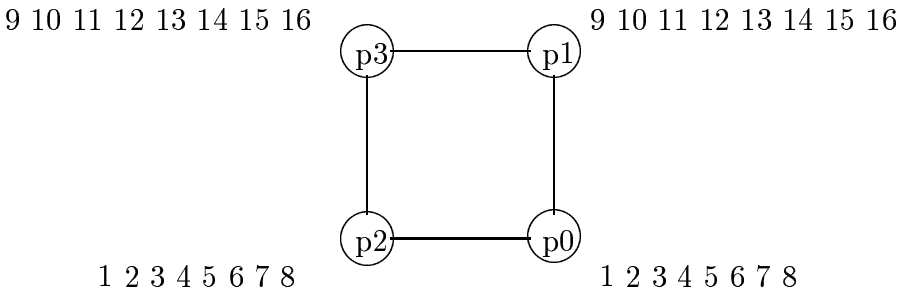
Figure 2.5: Progression of the CE algorithm with existing mapping for N=16 and p=4

Table 2.2: Counts of tasks executed by the CE algorithm with the existing mapping

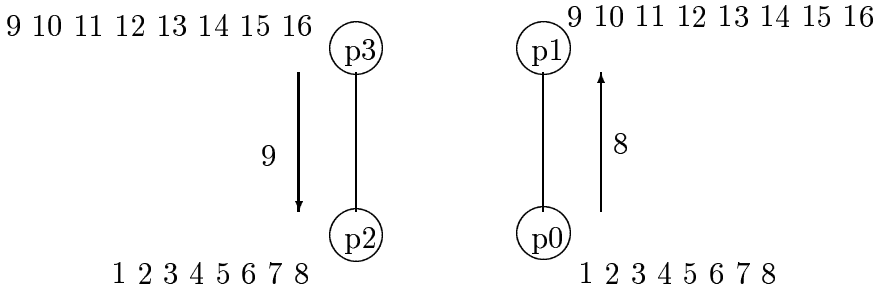
stage	*task count*	
	T_e	e
1	1	4
2	2	4
3	4	4
4	$8(2hops)$	4

algorithm consists of only reduction phase which has 4 (i.e., $\log 16$) stages. In the first stage, $row_5^{(0)}$ tuple is communicated from p_1 to p_0 preceding the step *Compute* $row_4^{(1)}$. Simultaneously, $row_4^{(0)}$ tuple is communicated from p_0 to p_1 preceding the step *Compute* $row_5^{(1)}$ and so on. Thus stage 1 consists of one-hop communication of $row_i^{(l)}$ tuples followed by four *Compute* $row_i^{(l)}$ steps per processor. At the end of stage 1, there are two independent sets of equations, namely, $\{1, 3, 5, 7, 9, 11, 13, 15\}$ and $\{2, 4, 6, 8, 10, 12, 14, 16\}$. Similarly, stage 2 consists of two one-hop communication of $row_i^{(l)}$ tuples followed by four *Compute* $row_i^{(l)}$ steps per processor. At the end of stage 2 there are four independent sets of equations, namely $\{1, 5, 9, 13\}$, $\{3, 7, 11, 15\}$, $\{2, 6, 10, 14\}$, and $\{4, 8, 12, 16\}$. Stage 3 consists of four one hop communications of $row_i^{(l)}$ tuples followed by four *Compute* $row_i^{(l)}$ steps.

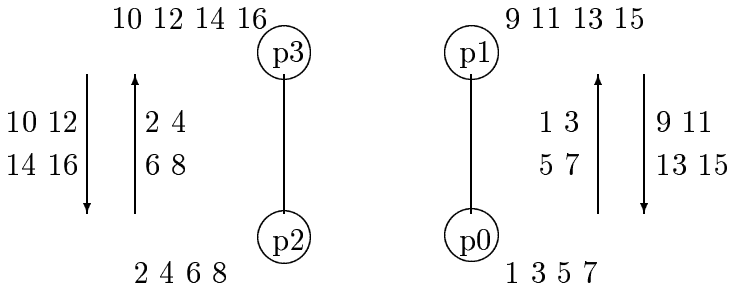
The counts of various tasks executed at each stage of the algorithm are summarised in table 2.2. We see from table 2.2 that communication overhead doubles with each stage as the number of independent sets of equations doubles at each stage. Further, the last stage consists of four consecutive two-hop communication of $row_i^{(l)}$ tuples. Stage 4 is followed by four divisions per processor. Thus the total execution time taken in the present case is $T_{CE} = 15T_e + 16e + 4$ computational time units. Substituting the values for T_e and e, we get $T_{CE} = 7712$ time units. Thus, in the present case, the existing mapping of CE algorithm performs poorly in comparison to the mapping of CR algorithm onto hypercubes.



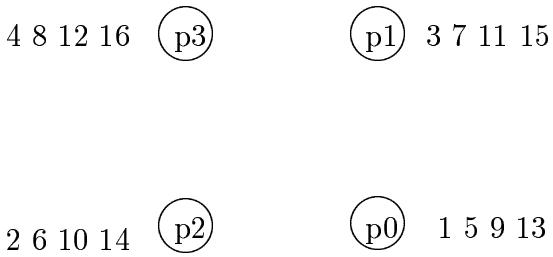
(a) initial data distribution



(b) elimination phase at stage 1



(c) copying phase at stage 1



(d) after elimination at stage 2

Figure 2.6: Progression of the CE algorithm with improved mapping for N=16 and p=4

2.4.1.3 The Improved Mapping of CE Algorithm

Figure 2.6 shows the step by step execution of our improved mapping of CE algorithm for solving the tridiagonal system of 16 equations using a hypercube with four processors. In this improved mapping scheme, all the communication steps occur between neighbouring processors only. The initial distribution of data is as follows. We divide the processors of the hypercube into two sets - $\{p_0, p_1\}$ and $\{p_2, p_3\}$ - the former being the set of processors in the lower half of the hypercube along 2nd dimension and the latter being the set of processors in the upper half of the hypercube along the 1st dimension. The 16 equations are then mapped onto each of the two sets of processors in a block wrap manner. Thus we get the initial data distribution as shown in figure 2.6(a). There are $\log p$ stages of the improved mapping. Each of the first $\log p - 1$ stages (only the first stage in the present case) consists of two phases - *elimination* and *replication (copying)*. The elimination phase corresponds to the reduction stage of the CE algorithm in which *Compute* $row_i^{(l)}$ steps are executed. Thus in stage 1 of the algorithm (figure 2.6(b)), the processors in the set $\{p_0, p_1\}$ execute *Compute* $row_i^{(l)}$ steps for odd-indexed equations and the processor set $\{p_2, p_3\}$ executes *Compute* $row_i^{(l)}$ for even-indexed equations. This involves a one-hop communication of $row_i^{(l)}$ tuples followed by four *Compute* $row_i^{(l)}$ steps per processor. At the end of elimination phase of stage 1, the processor set $\{p_0, p_1\}$ holds the independent set of equations $\{1, 3, 5, 7, 9, 11, 13, 15\}$ and the processor set $\{p_2, p_3\}$ holds the independent set of equations $\{2, 4, 6, 8, 10, 12, 14, 16\}$. The next phase of stage 1 is the *copying phase* in which each processor copies the $row_i^{(1)}$ tuples of its set of equations to the neighbouring processor along the 1st dimension of the hypercube. Thus p_0 copies the $row_i^{(1)}$ tuples of equations $\{1, 3, 5, 7\}$ to p_1 and p_1 copies those of equations $\{9, 11, 13, 15\}$ to p_0. Similar copying occurs between processors p_2 and p_3. Stage 2 of the algorithm consists of only the elimination phase. Thus p_0 executes *Compute* $row_i^{(2)}$ steps for $i = 1, 5, 9, 13$, p_1 executes *Compute* $row_i^{(2)}$ steps for $i = 3, 7, 11, 15$, p_2 executes *Compute* $row_i^{(2)}$ steps for $i = 2, 6, 10, 14$, and p_3 executes *Compute* $row_i^{(2)}$ steps for $i = 4, 8, 12, 16$. Thus at the end of $\log p$ stages (i.e., elimination phase of stage 2 in the present case) each processor contains an independent set of equations which can be solved using BGE algorithm without communicating with any other processor.

Table 2.3: Counts of tasks executed by the CE algorithm with improved mapping

stage		*task count*	
		T_e	e
1	*elimination*	1	4
	copying	4	0
2	*elimination*	0	4

The counts of various tasks executed at each stage of the algorithm are summarised in table 2.3. The BGE algorithm for solving 4 equations per processor takes $T_{BGE} = 33$ computational time units (see section 3.1). Thus the total time taken in the present case is $T_{new} = 5T_e + 8e + T_{BGE}$ units. Substituting the values for T_e and e, we get $T_{new} = 2637$ time units.

Thus we see that in the case of $N = 16$, $n = 1$ and $p = 4$, our improved mapping of CE algorithm performs better than the existing mappings of both CR and the CE algorithms. Further, the existing mapping of CR algorithm performs better than the existing mapping of CE algorithm due to lower communication overhead. We now present some definitions and then formally present our improved mapping of the CE algorithm. We then evaluate its performance by comparing with the existing mapping of the CR algorithm only, since this mapping fares better than the existing mapping of CE algorithm, as shown in the above example.

2.4.2 Definitions

- *Binary reflected gray codes* [48] are a class of codes useful in embedding a ring structure onto a binary hypercube. Let $G(n)$ denote the set of all n-digit code words of the base-2 (binary) reflected gray code i.e.,

$$G(n) = \{G_0(n), G_1(n), \ldots, G_{2^n-1}(n)\}$$

where, $G_i(n)$ ith code word of binary reflected gray code, $i \in \{0, \ldots, 2^n - 1\}$. Let

$$i = i_n i_{n-1} \cdots i_2 i_1 i_0$$

in binary with $i_n = 0$ and

$$G_i(n) = g_n g_{n-1} \cdots g_2 g_1$$

in binary. If $\oplus$ denotes the exclusive-OR addition of binary bits, then the encoding function $E_n :< \mathcal{N} >\rightarrow G(n)$ is given by

$$E_n(i) = G_i(n) = g_n g_{n-1} \cdots g_1$$

where

$$g_j = i_j \oplus i_{j-1}$$

for all $j = 1, 2, \ldots, N$, and the decoding function $D_n : G(n) \rightarrow < \mathcal{N} >$ is given by

$$D_n(g) = i$$

where

$$i_j = g_{j+1} \oplus g_{j+2} \oplus \cdots \oplus g_n.$$

- $p_j : send(row_i^{(l)}, p_{j'})$ indicates that processor p_j sends contents of $row_i^{(l)}$ to processor $p_{j'}$.
- $p_j : receive(row_i^{(l)}, p_{j'})$ indicates that processor p_j receives contents of $row_i^{(l)}$ from processor $p_{j'}$.
- $neighbour(j, k)$ indicates the neighbour of processor p_j along the kth dimension of hypercube. If $j' = neighbour(j, k)$ then j' is obtained by complementing the kth bit in the binary representation of j.
- Let d be the dimension of the hypercube and $l \in \{1, \ldots, d\}$ be the dimension across which the hypercube is to be divided into two halves. We define two sets $P_{upper}^{(l)}$ and $P_{lower}^{(l)}$ as

$$P_{upper}^{(l)} = \{j \mid j > neighbour(j, l)\}$$

$$P_{lower}^{(l)} = \{j \mid j < neighbour(j, l)\}$$

where $j \in \{0, \ldots, p-1\}$. Further,

$$P_{upper}^{(0)} = \{p/2, p/2 + 1, \ldots, p-1\}$$

$$P_{lower}^{(0)} = \{0, 1, \ldots, p/2 - 1\}.$$

In the next two sub-sections the following assumptions hold.

- Each processor contains sufficient local memory and no global memory exists.
- $N/p \geq 1$, where N is the number of rows of blocks in the block tridiagonal linear system.
- All links between the processors of the hypercube are capable of full-duplex communication.
- For each communication step between a pair of neighbouring processors, the startup time is assumed to be negligible.
- Each processor can overlap its computation with the data communication from/to its neighbours.
- Inversion of matrix blocks is done using the *exchange method*.
- The matrix blocks $d_i^{(l)}$, $i = 1, \ldots, N$, are non-singular at all stages $l = 1, \ldots, \log N$.

2.4.3 Our Improved Mapping of CE onto Hypercubes

Initially, all $row_i^{(0)}$, $i = 1, \ldots, N$ in the block tridiagonal linear system are partitioned into $p/2$ sets $S_1^{(0)}, S_2^{(0)}, \ldots, S_{p/2}^{(0)}$ of $2N/p$ rows each such that

$$S_i^{(0)} = \{row_{2(i-1)\frac{N}{p}+1}^{(0)}, row_{2(i-1)\frac{N}{p}+2}^{(0)}, \ldots, row_{2i\frac{N}{p}}^{(0)}\}$$

$i = 1, \ldots, p/2$.
One copy of each set $S_i^{(0)}$ is stored in a pair of processors p_j and $p_{j'}$, $j \in \{0, \ldots, p/2-1\}$ and $j' \in \{p/2, \ldots, p-1\}$such that

$$j = E_{\log p - 1}(i - 1)$$

i.e., $j = (i-1)$th code word of the binary reflected gray code with $\log p - 1$ bits and

$$j' = neighbour(j, \log p)$$

At any stage l of the algorithm, we maintain sets $C_j^{(l)}$ at every processor p_j such that

$$C_j^{(l)} = \{row_i^{(l-1)} \mid row_i^{(l)} \text{is computed at processor } p_j\}.$$

For all $j \in P_{lower}^{(0)}$, let $k = D_{\log p-1}(j) + 1$. Thus the members of the set $S_k^{(0)}$ are stored at processor p_j. Initially, let

$$C_j^{(1)} = \{row_i^{(0)} \mid row_i^{(0)} \in S_k^{(0)} \text{ and } i \in \{1, 3, \ldots, N-1\}\}$$

i.e., *Compute* $row_i^{(l)}$ step is executed at p_j for all odd indexed equations which are members of the set $S_k^{(0)}$. Similarly, for all $j' \in P_{upper}^{(0)}$, let $k = D_{\log p-1}(neighbour(j', \log p)) + 1$. Then

$$C_{j'}^{(1)} = \{row_i^{(0)} \mid row_i^{(0)} \in S_k^{(0)} \text{ and } i \in \{2, 4, \ldots, N\}\}$$

i.e., *Compute* $row_i^{(l)}$ step is executed at p_j for all even indexed equations which are members of the set $S_k^{(0)}$. We now formally present our CE algorithm for hypercubes.

Algorithm 4

(*Cyclic elimination on hypercube*)

1. **for** $j \in \{0, 1, \ldots, p/2 - 1\}$ **do in parallel**
2. $\quad p_j,\ p_{j+p/2} : k = D_{\log p-1}(j) + 1$
3. $\quad\quad\quad\quad\quad\quad h = 2^{l-1}$
4. **endfor**
5. **for** $l = 1$ **to** $\log p - 1$ **do**
6. $\quad$ (*Elimination phase*)
7. $\quad$ **for** all $j \in \{0, \ldots, p-1\}$ **do in parallel**
8. $\quad\quad p_j$: **for** all i such that ($row_i^{(l)} \in C_j^{(l)}$) **do**
9. $\quad\quad\quad\quad$ *Compute* $row_i^{(l)}$
10. $\quad\quad\quad$ **endfor**
11. $\quad$ **endfor**
12. $\quad$ **if** ($l < \log p - 1$) **then**
13. $\quad\quad$ (*Copying phase*)
14. $\quad\quad$ **for** $j \in \{0, \ldots, p-1\}$ **do in parallel**
15. $\quad\quad\quad\quad p_j : S_k^{(l)} = C_j^{(l)}$

16. **for** all i such that ($row_i^{(l-1)} \in C_j^{(l)}$)
17. $send(row_i^{(l)}, p_{neighbour(j,l)})$
18. $receive(row_{i'}^{(l)}, p_{neighbour(j,l)})$
19. $S_k^{(l)} = S_k^{(l)} \cup \{row_i^{(l)}\}$
20. **endfor**
21. **endfor**
22. (*Updating $C_j^{(l+1)}$*)
23. **for** $j \in P_{lower}^{(l)}$ and $j' \in P_{upper}^{(l)}$ **do in parallel**
24. $p_j : min = minimum\{i \mid row_i^{(l)} \in S_k^{(l)}\}$
25. $C_j^{(l+1)} = \phi$
26. **for** $i = min$ **to** $min + (\frac{N}{p} - 1)h$ **step** $2h$ **do**
27. $C_j^{(l+1)} = C_j^{l+1} \cup \{row_i^{(l)}\}$
28. **endfor**
29. $p_{j'} : min = minimum\{i \mid row_i^{(l)} \in S_K^{(l)}\}$
30. $C_{j'}^{(l+1)} = \phi$
31. **for** $i = min + h$ **to** $min + Nh/p$ **step** $2h$ **do**
32. $C_{j'}^{(l+1)} = C_{j'}^{l+1} \cup \{row_i^{(l)}\}$
33. **endfor**
34. **endfor**
35. **endif**
36. **endfor**

37. (*Obtaining x_i*)
38. **for** $j \in \{0, \ldots, p-1\}$ **do in parallel**
39. p_j :**if** ($\frac{N}{p} > 1$) **then**
40. Solve the independent system of $\frac{N}{p}$ block equations in $C_j^{(\log p-1)}$ using BGE algorithm to obtain $\{x_i \mid row_i^{(\log p-1)} \in C_j^{(\log p-1)}\}$
41. **else** (*$N = p$*)
42. **for** all i such that ($row_i^{(\log p-1)} \in C_j^{(\log p-1)}$) **do**
43. $x_i = (d_i^{(\log p-1)})^{-1} b_i^{(\log p-1)}$
44. **endfor**

45. **endif**

46. **end**

We see that the communication of data occurs in the lines 7-11 (elimination phase) and lines 14-21 (copying phase). Lines 7-11 for computing $row_i^{(l)}$ at processor p_j require data of $row_{i-h}^{(l-1)}, row_i^{(l-1)}$, and $row_{i+h}^{(l-1)}$. Of the three, $row_i^{(l-1)}$ is available on p_j. If $row_{i-h}^{(l-1)}$ and $row_{i+h}^{(l-1)}$ are not available on p_j, then they have to be brought in from its neighbouring processors. In lines 14-21 of the copying phase, (see figures 2.7(c),(e)), at every stage l, exactly $\frac{N}{p}$ rows of blocks are copied in each direction between every pair of neighbouring processors along dimension $l-1$ of the hypercube. Again the communication is between neighbouring processors only. Hence the number of hops in any communication step is no more than one at any stage of the algorithm.

Note that after $\log p - 1$ stages, the above algorithm switches over to BGE algorithm on uniprocessor. This is because after $\log p - 1$ stages each processor contains an independent set of equations which can be solved without communicating with any other processor. Since on a uniprocessor, the BGE algorithm is the most efficient one, switching over to BGE enhances the performance.

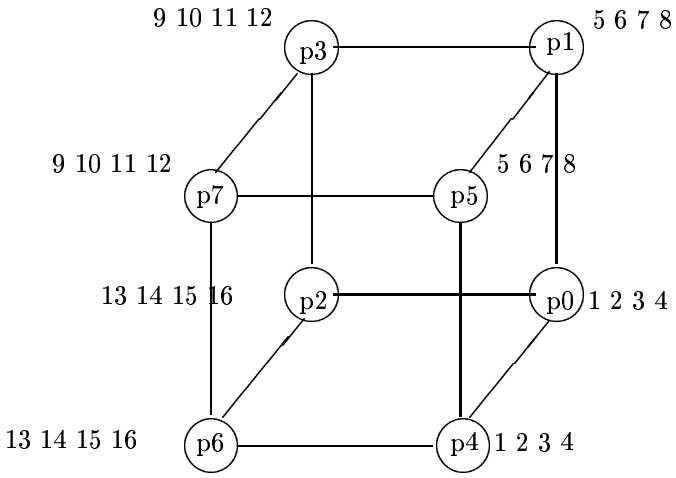
2.4.4 Analytical Performance Studies

We now derive expressions for the execution time of our algorithm and also the CR algorithm.

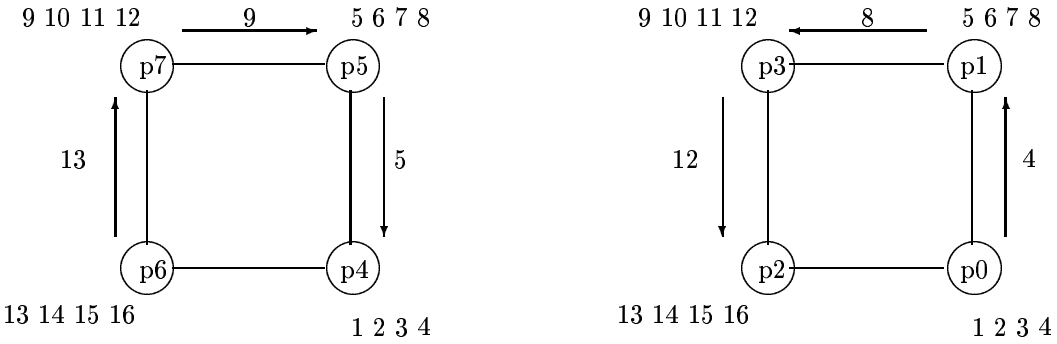
2.4.4.1 Our Improved CE Algorithm

The lines 1-4 take $T_1 = 3$ time units to execute in parallel on p processors. In lines 5-36, the copying phase of every iteration l overlaps with the computation phase of $(l+1)$th iteration. Thus this step (lines 5-36) takes

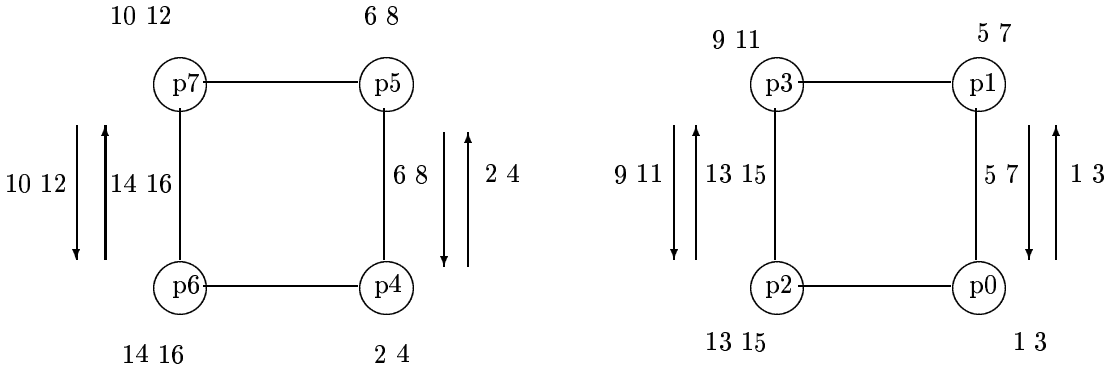
$$T_2 = max\{(\tfrac{N}{p} - 1)e, T_e\} + e + \\ (\log p - 2)(max\{(\tfrac{N}{p} - 1)e, \tfrac{N}{p}(T_e + 1)\} + T_e + e) + \\ (T_e + (\tfrac{N}{p} - 1)max(e, T_e) + e) \text{ units.}$$



(a) initial distribution of data

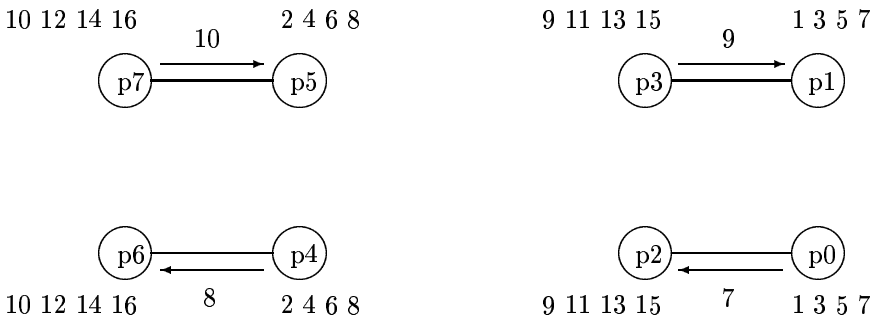


(b) elimination phase at l = 1

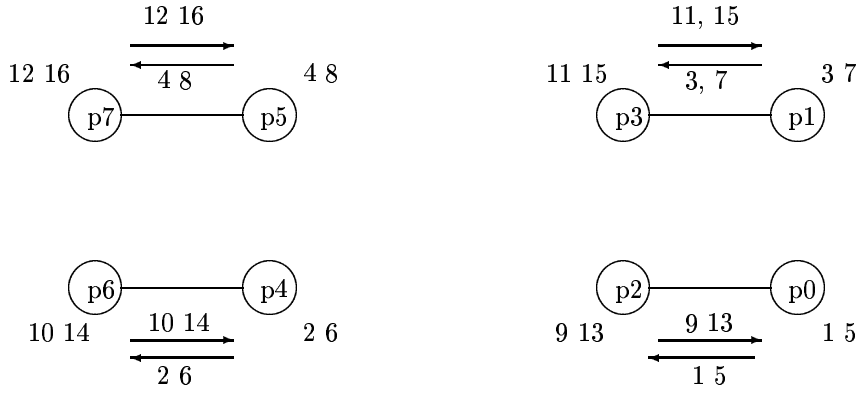


(c) copying phase at l = 1

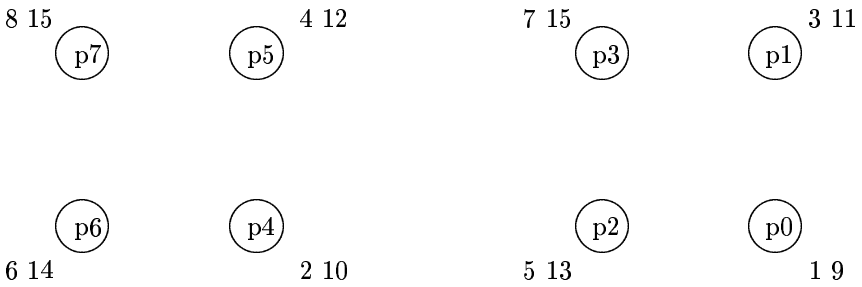
Figure 2.7: (a) Progression of our algorithm on hypercube for N=16 and p=8



(d) elimination phase at l = 2



(e) copying phase at l = 2



(f) after elimination phase at l = 3

Figure 2.7: (b) Progression of our algorithm on hypercube for N=16 and p=8

For lines 38-46, $T_3 = (\frac{N}{p} - 1)(e + s) + (2n^3 - n^2)$. Thus the total time taken, $T_{total} = T_1 + T_2 + T_3$.

Let us look at the communication complexity of our algorithm without considering any overlap between the communication and computation steps. The contribution from elimination phase (lines 7-11) alone is $(\log p - 1)T_e$ and that from copying phase (lines 14-21) alone is $\frac{N}{p}(\log p - 2)T_e$. Thus the total communication complexity of our algorithm is a sum of these two, given by

$$\left((\frac{N}{p} + 1)\log p - 2\frac{N}{p} - 1\right) T_e \text{ units}$$

where $T_e = 5n^2(C/E)$, $n \times n$ being the size of each block.

2.4.4.2 CR Algorithm

In reduction phase, the first $\log(N/p)$ stages involve one hop communication of rows of blocks and $N/(p2^l)$ computations of $row_i^{(l)}$ (figure 2.4(b) and (c)). Here the communication of a row of blocks and $(\log(N/p) - 1)$ computations of $row_i^{(l)}$ are overlapped. The $\log(N/p) + 1$th stage involves one hop communication of a row of blocks and one $row_i^{(l)}$ computation step in a non-overlapped manner (figure 2.4(d)). The remaining $(\log p - 1)$ stages involve two hop communication of a row of blocks and one $row_i^{(l)}$ computation step in a non-overlapped manner (figure 2.4(e)). Thus the total time for the reduction phase works out to be

$$\begin{aligned} T_{reduction} = {} & (e + 4)\log(\tfrac{N}{p}) + \\ & \textstyle\sum_{l=1}^{\log(\frac{N}{p})} (max\{(N/(p2^l) - 1)(e + 1), T_e\} + \\ & (T_e + e + 4) + (\log p - 1)(2T_e + e + 4) \text{ units} \end{aligned}$$

Similar communication pattern exists for back substitution but in reversed manner. Thus the time taken for back substitution phase works out to be

$$T_{back\ substitution} = (s+3)\log(\tfrac{N}{p}) + \sum_{l=1}^{\log(\frac{N}{p})} max\{(N/(p2^l)-1)s, T_b\} + (T_b+s+3) + (\log p - 1)(2T_b+s+3) \text{ units.}$$

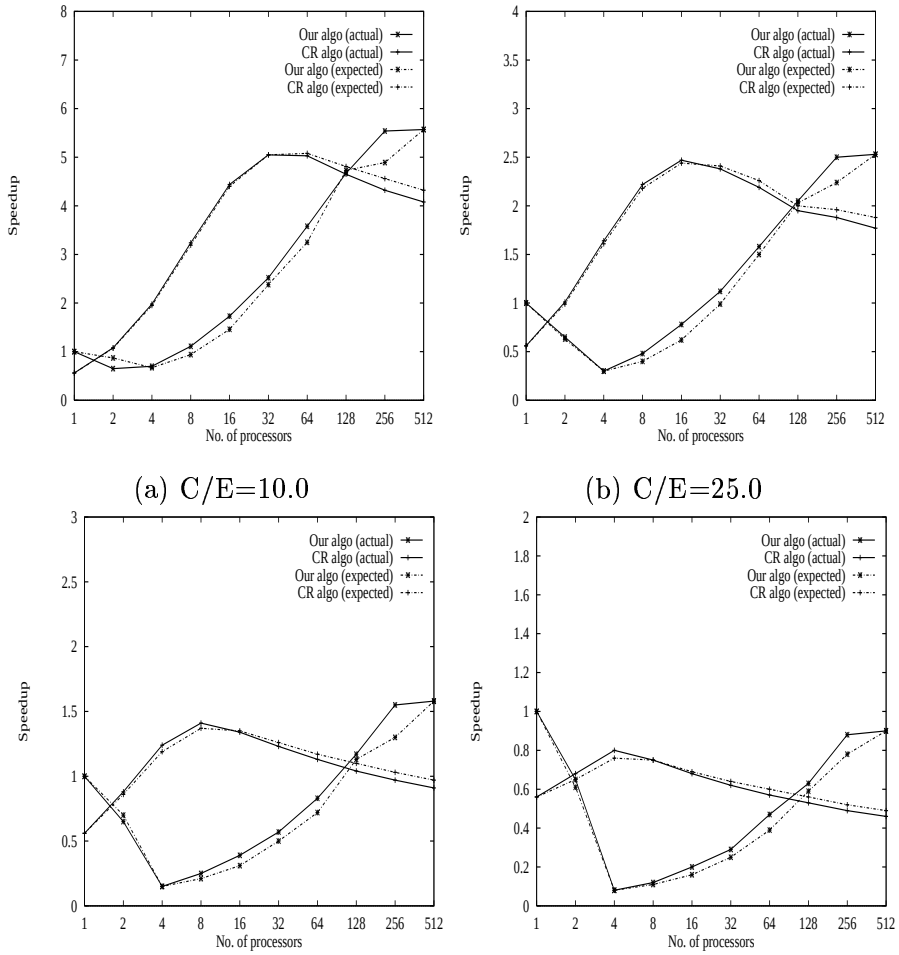
Taking a block multiplication step between these two phases into account, the total time $T_{CR} = T_{reduction} + T_{mult} + T_{backsubstitution}$. Let us look at the communication complexity of CR algorithm without considering any overlap of the communication and computation steps. The contribution from reduction phase alone is $(\log N + \log p - 1)T_e$ and contribution from back substitution phase alone is $(\log N + \log p - 1)T_b$. Thus the total communication complexity of CR algorithm, as a sum of these two, is given by

$$(\log N + \log p - 1)(T_e + T_b) \text{ units}$$

where $T_e = 5n^2(C/E)$ and $T_b = n^2(C/E)$, $n \times n$ being the size of each block.

2.5 Experimental Results

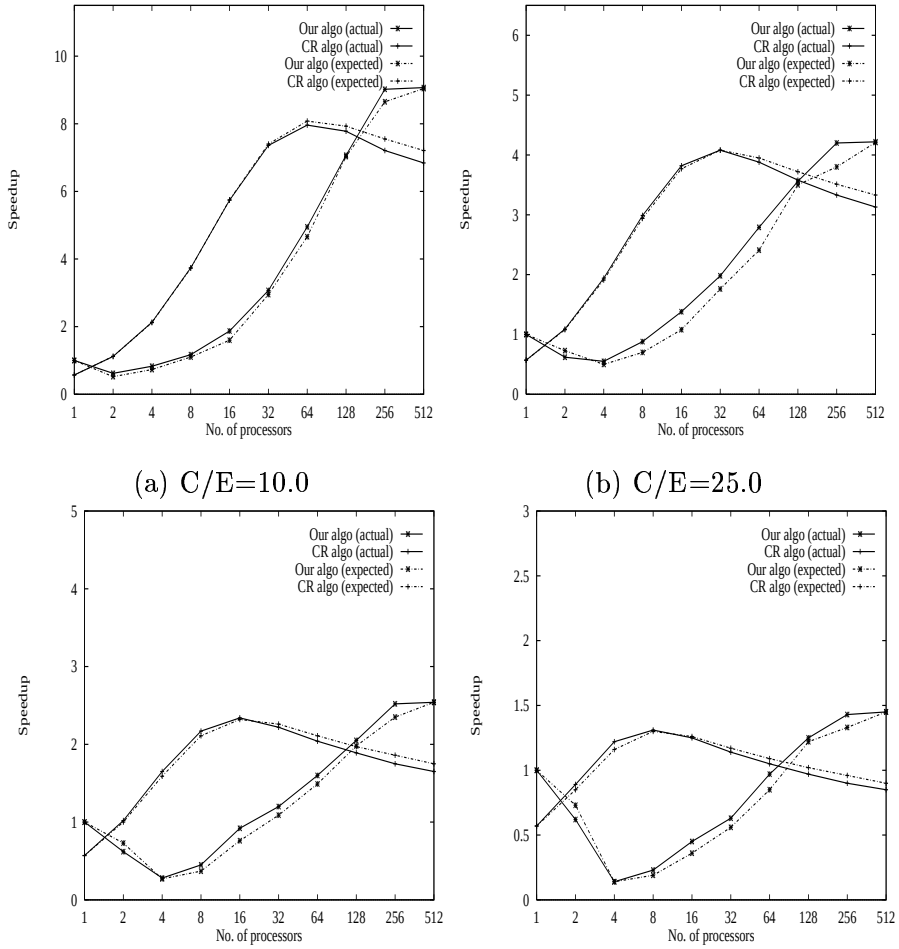
To evaluate the accuracy of the above analytical expressions, we implemented a hypercube simulator in C language and compared the *speedups* obtained from our new mapping of CE algorithm with those obtained from the existing mapping of CR algorithm. We used SPARC Classic machines to carry out our simulations. The parameters that were varied were the number of rows of blocks N (512 and 1024), the block size n (1,2, and 4), the ratio of communication step to computation step C/E (10, 25, 50, and 100), and the number of processors p (1 to 1024). The figures 2.8-2.13 show the comparison of measured speedups of the two algorithms for various values of the above parameters. We observe the following facts.



(a) C/E=10.0 (b) C/E=25.0

(c) C/E=50.0 (d) C/E=100.0

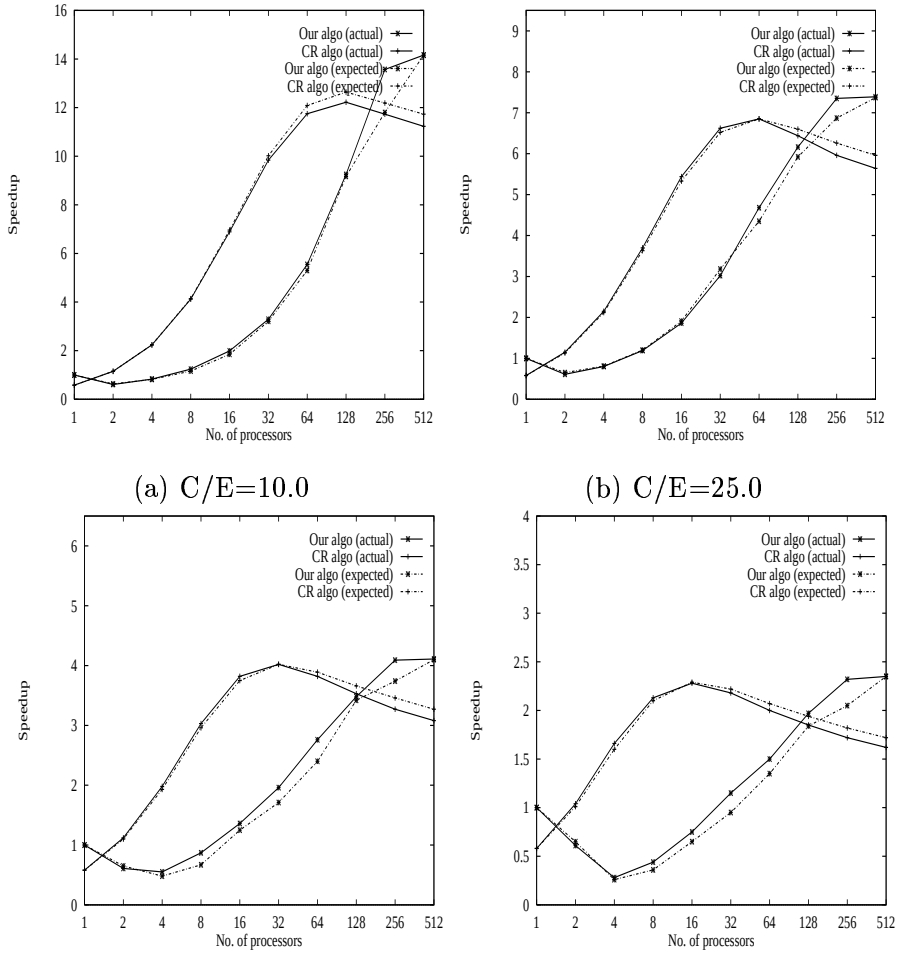
Figure 2.8: Speedups obtained for our algorithm versus CR algorithm for N=512 and n=1



(a) C/E=10.0 (b) C/E=25.0

(c) C/E=50.0 (d) C/E=100.0

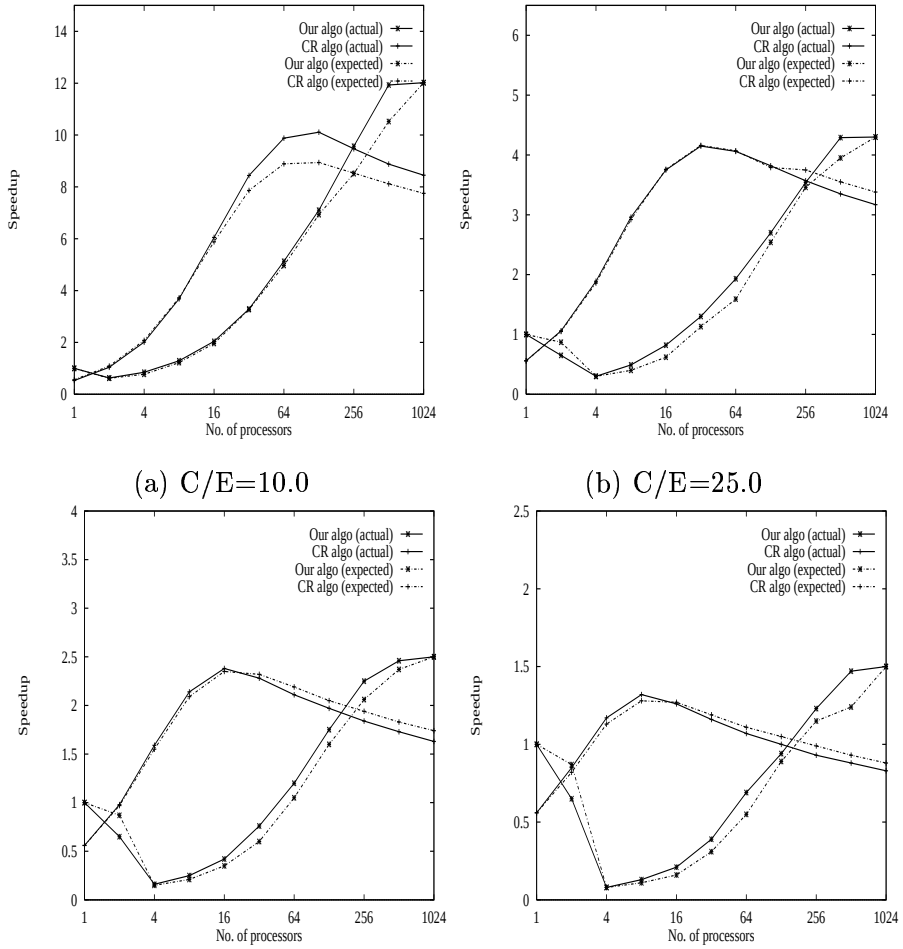
Figure 2.9: Speedups obtained for our algorithm versus CR algorithm for N=512 and n=2



(a) C/E=10.0 (b) C/E=25.0

(c) C/E=50.0 (d) C/E=100.0

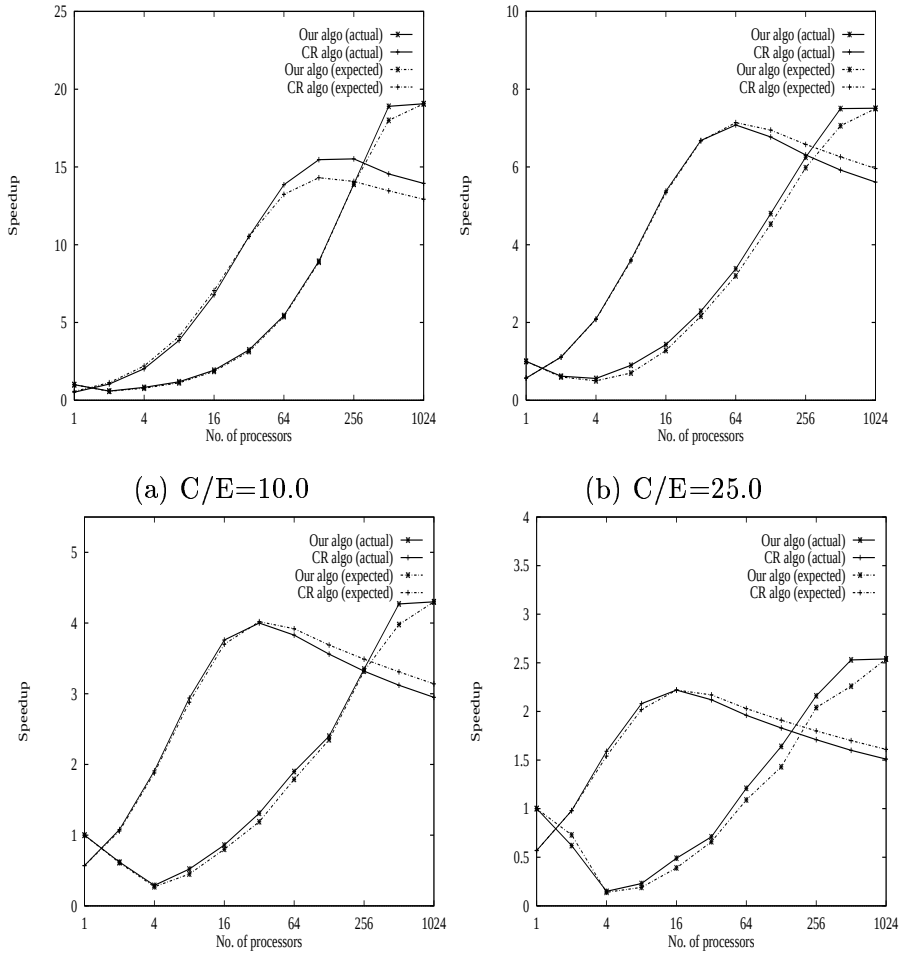
Figure 2.10: Speedups obtained for our algorithm versus CR algorithm for N=512 and n=4



(a) C/E=10.0 (b) C/E=25.0

(c) C/E=50.0 (d) C/E=100.0

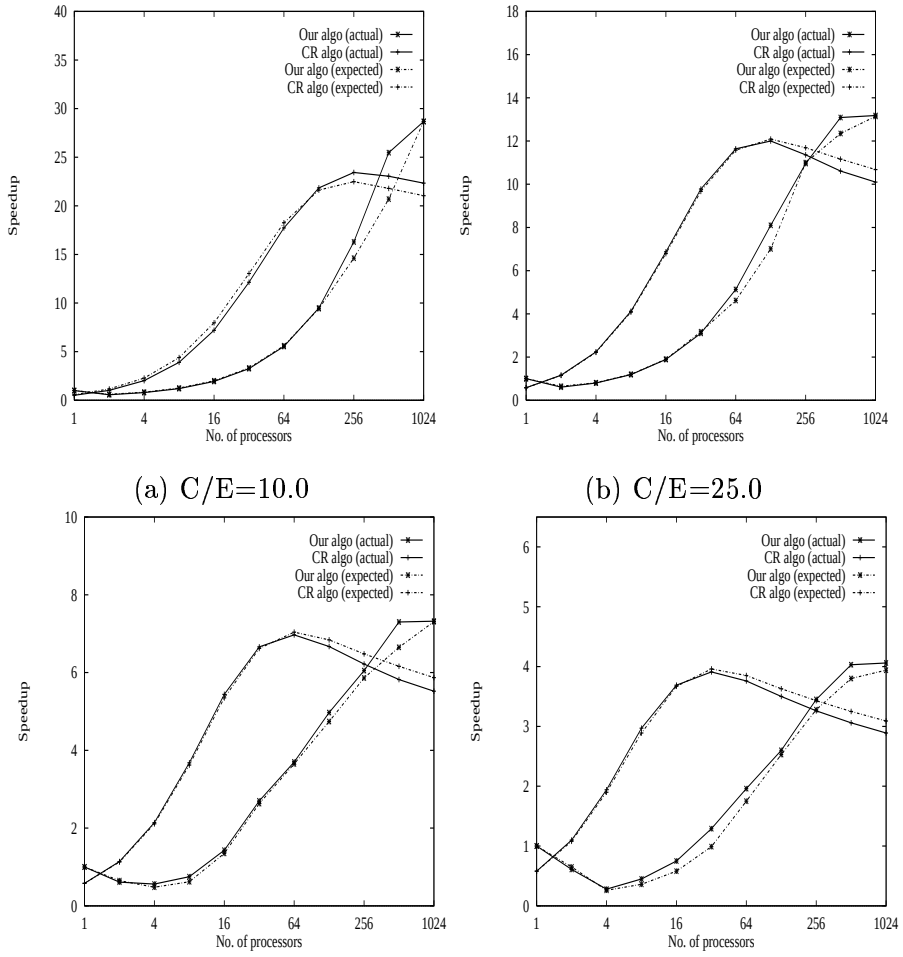
Figure 2.11: Speedups obtained for our algorithm versus CR algorithm for N=1024 and n=1



(a) C/E=10.0 (b) C/E=25.0

(c) C/E=50.0 (d) C/E=100.0

Figure 2.12: Speedups obtained for our algorithm versus CR algorithm for N=1024 and n=2



(a) C/E=10.0 (b) C/E=25.0

(c) C/E=50.0 (d) C/E=100.0

Figure 2.13: Speedups obtained for our algorithm versus CR algorithm for N=1024 and n=4

- For $N = p$ our improved mapping scheme for CE algorithm always gives higher speedup than the CR algorithm.
- Increasing the block size n increases the magnitude of speedups obtained by the two schemes (see figures 2.11(a),2.12(a), and 2.13(a)). Increasing the number of rows of blocks, N, shows up a similar trend (see figures 2.8 and 2.11, 2.9 and 2.12 and, 2.10 and 2.13). On the other hand, as the C/E ratio increases, the magnitude of speedup reduces in both the algorithms (see figures 2.8(a), (b), and (c)).

- The speedup of CR algorithm tends to saturate and even fall as the number of processors increases. Such a saturation effect is absent from our algorithm in which the speedup progressively increases with the number of processors and reaches its maximum value at $N = p$.
- The results obtained from the simulation studies compared well with the theoretical predictions obtained from the analytical method. The small differences between the speedups obtained from both the methods arise due to the following reason. The analytical method tries to estimate, as closely as possible, the amount of overlap between the computation and communication steps. However, the exact amount of overlap depends on various factors such as the C/E ratio, precedence constraints between various computation and communication tasks, and routing scheme used in the multiprocessor system. The effect of all these factors on the speedup of the algorithms cannot be encapsulated neatly into a single analytical expression.

2.6 Conclusions

We have proposed a new mapping of the CE algorithm onto hypercube multiprocessors for solving block tridiagonal linear systems. This mapping maintains the same degree of parallelism throughout and uses the concept of data replication to achieve only neighbouring processor communication at all stages of the processing. We have demonstrated the effectiveness of our mapping by comparing it with the existing mapping of CR algorithm onto hypercubes using both analytical and simulation methods. Further work is possible in the direction of controlling the amount of parallelism in our implementation of the CE algorithm [42]. In its present form, our algorithm switches to the sequential BGE algorithm only after $\log p - 1$ stages when each processor has an independent set of equations which can be solved without communicating with any neighbour. However, switching over to BGE algorithm at an earlier stage (say k) may lead to further improvements in the performance of our algorithm. Determining the optimal value of k is an open problem.

Chapter 3

A New Algorithm for Direct Solution of Sparse Symmetric Linear Systems

3.1 Introduction

In this chapter, we consider the problem of solving sparse symmetric system of linear equations of the form $Ax = b$, where A is a sparse symmetric matrix of dimension $N \times N$, and x and b are N-vectors. Such equations arise in various applications such as finite element problems, power systems analysis, and circuit simulations for VLSI CAD. Traditionally, the process for obtaining the direct solution for a sparse symmetric system of linear equations, $Ax = b$, involves the following four distinct phases.

- *Ordering* : Apply an appropriate symmetric permutation matrix P such that the new system is of the form $(PAP^T)(Px) = (Pb)$.
- *Symbolic factorization* : Set up the appropriate data structures for the numerical factorization phase.
- *Numerical factorization* : Determine the Cholesky factor L such that $A = LL^T$.
- *Substitution* : Determine the solution vector x by first solving the forward triangular system $Ly = b$ and then solving the backward triangular system $L^T x = y$.

For solution of multiple b-vectors, the first three phases are carried out only once to obtain the Cholesky factor L. The substitution phase is then repeated for each b-vector in order to obtain a different solution vector x in each case. Thus, in problems which involve solution of multiple b-vectors, the time taken by repeated execution of substitution phase dominates the overall solution time. Any parallel formulation, which can reduce the time taken by the substitution phase, will contribute significantly to enhanced performance of the entire process.

Although traditional approaches to parallel solution of sparse symmetric system of linear equations have yielded efficient parallel algorithms for the numerical factorization phase [4, 15, 20, 30], not much progress has been made in the case of substitution phase due to the limited amount of parallelism inherent in this phase. Moreover, the forward and backward substitution components of the substitution phase require different parallel algorithms due to the manner in which data is distributed over various processors. Existing work on parallel formulations for this phase can be found in [14, 22, 29].

In this chapter we present a new *bidirectional algorithm*, based on Cholesky factorization, for the solution of sparse symmetric system of linear equations. In our algorithm, the numerical factorization phase is carried out in such a manner that the entire back substitution component of the substitution phase is replaced by a single step division. The application of the novel concept of bidirectional elimination to dense linear systems can be found in [42, 43].

The rest of the chapter is organized as follows. In section 3.2, we present the bidirectional sparse Cholesky factorization algorithm for sparse symmetric matrices. In section 3.3, we present the bidirectional algorithm for the substitution phase which does not have a back substitution component. In section 3.4 we develop a bidirectional heuristic algorithm for ordering on the lines of the popular *nested dissection* ordering algorithm [13, 10] for sparse symmetric matrices. In section 3.5, we describe a symbolic factorization algorithm which sets up data structures required by the bidirectional Cholesky factorization phase. In section 3.6, we evaluate the performance of the bidirectional algorithm on hypercube multiprocessors and present comparison of our algorithm with the existing scheme based on sparse Cholesky factorization. In section 3.7, we conclude the work with some observations about possible future improvements to the bidirectional scheme.

3.2 The Bidirectional Sparse Cholesky Factorization (BSCF) Algorithm

Unlike the regular Cholesky factorization algorithm which factorizes A to obtain the lower triangular matrix L, such that $A = LL^T$, the BSCF algorithm factorizes A into a series of *trapezoidal* matrices of multipliers. This series of trapezoidal matrices remove

the need for the back substitution component in the substitution phase.

In this section, we first present an overall view of the concept of bidirectional Cholesky factorization. We then proceed to describe the manner in which the sparsity of the coefficient matrix can be exploited to obtain higher degree of parallelism. Following this we present the details of implementing BSCF algorithm on multiprocessor systems.

3.2.1 Bidirectional Cholesky Factorization - The Concept

In regular Cholesky algorithm, the lower triangular matrix L is obtained by choosing columns 1 through N of matrix A as pivots so that $A = LL^T$. We name this process as *factorization in forward direction*. On the other hand, we can also choose columns N through 1 of matrix A as pivots and factorize A in a reverse fashion to obtain an upper triangular matrix U such that $A = U^T U$. We name this process as *factorization in backward direction*. The bidirectional Cholesky factorization of the coefficient matrix A proceeds as follows.

- *Step 1*: We form two matrices, namely A_0 and A_1, identical to the coefficient matrix A. We factorize A_0 in the forward direction, but only through the first $\lceil \frac{N}{2} \rceil$ pivot columns, to obtain a lower trapezoidal matrix L_0, as shown in figure 3.1, in which only the sub-diagonal entries in columns 1 to $\lceil \frac{N}{2} \rceil$ are present. Concurrently, we factorize A_1 in backward direction, through pivot columns N to $\lceil \frac{N}{2} + 1 \rceil$, to obtain an upper trapezoidal matrix L_1, as shown in figure 3.1, in which only the super-diagonal elements in columns N to $\lceil \frac{N}{2} + 1 \rceil$ are present.
- *Step 2*: We duplicate the reduced matrix A_0 to form A_{00} and A_{01}, and also duplicate the reduced matrix A_1 to form A_{10} and A_{11}. The matrices A_{00} and A_{10} are factorized halfway through in the forward direction to produce lower trapezoidal matrices L_{00} and L_{10} respectively. Similarly, the matrices A_{01} and A_{11} are factorized halfway through in the backward direction to produce upper trapezoidal matrices L_{01} and L_{11} respectively. Note that here we factorize the four matrices A_{00}, A_{01}, A_{10}, and A_{11} in parallel.
- *Step 3*: We continue this process of halving the size of the sub-matrices through simultaneous Cholesky factorization in both forward and backward directions

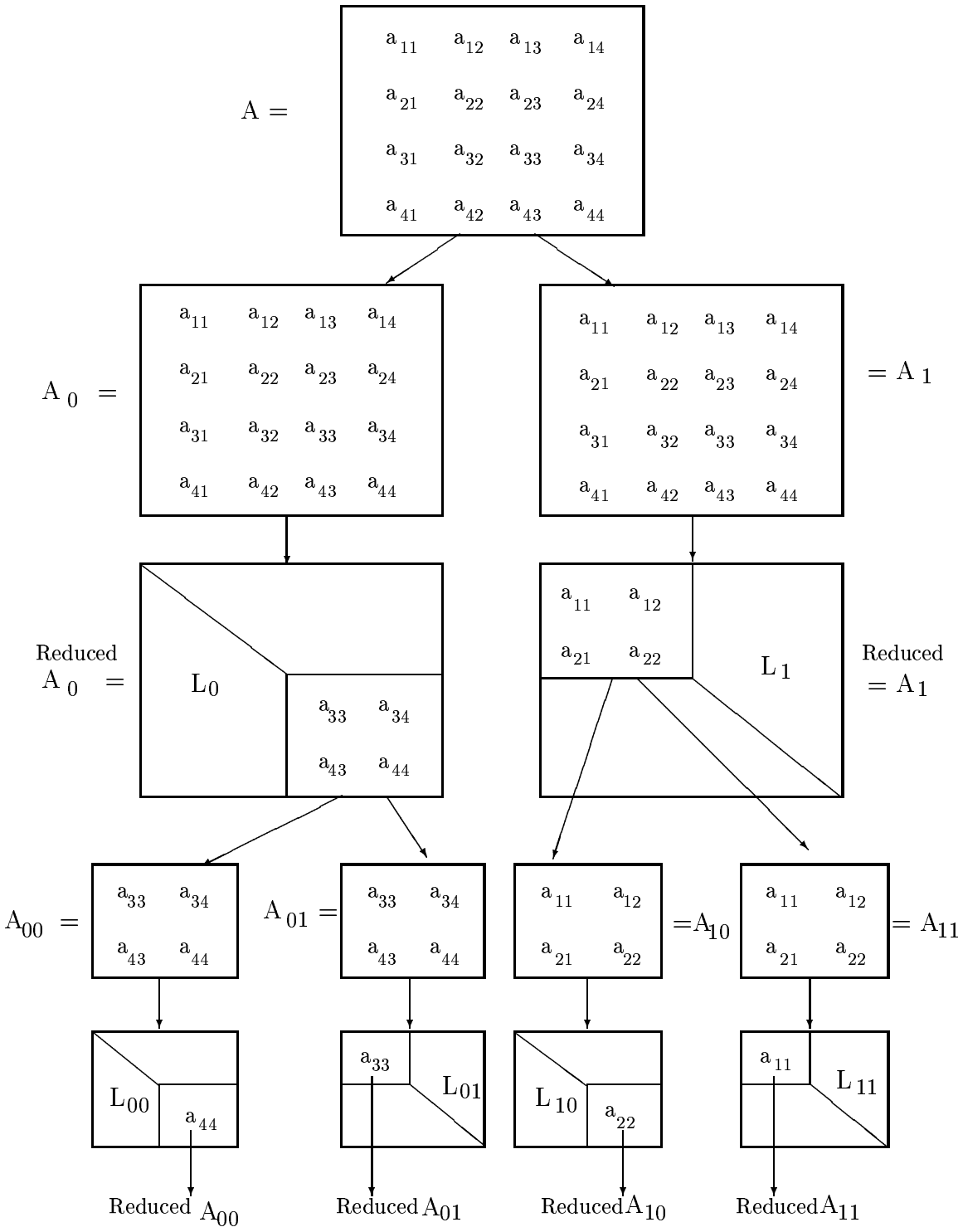


Figure 3.1: The progression of BSCF algorithm for $N = 4$

and thus doubling the number of sub-matrices for $\log N$ times. Finally we end up with N sub-matrices of order 1×1.

The bidirectional Cholesky factorization algorithm described above produces a tree of trapezoids of multipliers (i.e., L matrices). In the substitution phase, which is described in section 3.3, the b-vector, corresponding to which a solution vector x has to be found, is moved down this tree of trapezoids. At the end of this process each leaf produces an equation with just one variable which is then solved by a single step division to produce the solution vector x.

3.2.2 Exploiting the Sparsity of the Coefficient Matrix A

In regular sparse Cholesky factorization of a coefficient matrix A, column i directly modifies column j if $j > i$ and $A[i, j] \neq 0$. Column i indirectly modifies column j if column i directly modifies another column k which in turn modifies column j directly or indirectly. Columns i and j are *mutually independent* if column i does not modify column j directly or indirectly. The mutually independent columns of the sparse matrix can be used as pivots in parallel.

This concept of mutually independent columns can be easily extended to the BSCF algorithm. At any stage $s \in \{1 \cdots \log N\}$, columns i and j ($j > i$) are *forward independent* if pivot column i does not modify column j directly or indirectly during factorization in forward direction. The forward independent columns, i and j, can be simultaneously used as pivots in forward direction. The columns i and j are *backward independent* if pivot column j does not modify column i directly or indirectly during factorization in backward direction. The backward independent columns, i and j, can be simultaneously used as pivots in backward direction.

In regular sparse Cholesky factorization, the concept of mutually independent columns can be abstracted with the help of *elimination trees*. An elimination tree contains a node corresponding to each column of the coefficient matrix. The parent of a node i is defined as

$$parent(i) = min \{j \mid j > i \text{ and } L[j, i] \neq 0\} .$$

The elimination tree defines a partially ordered precedence relation which determines

when a certain column can be used as pivot.

Similarly, in BSCF algorithm, we can abstract the concepts of forward independence and backward independence by means of *forward elimination tree* and *backward elimination tree* respectively. At some stage $s \in \{1 \cdots \log N\}$, let A_{x0} be a sub-matrix being factorized in forward direction and A_{x1} be a sub-matrix being factorized in the backward direction (x being a possibly empty string of 0's and 1's). The *forward parent* of node i, is defined as

$$fparent(i, A_{x0}) = min\{j \mid j > i \text{ and } L_{x0}[j, i] \neq 0\}.$$

Similarly, the *backward parent* of node i, is defined as

$$bparent(i, A_{x1}) = max\{j \mid j < i \text{ and } L_{x1}[j, i] \neq 0\}.$$

For achieving high degree of parallelism during factorization phase, both the forward and the backward elimination trees should be as short and wide as possible. This is the function of the ordering phase (described in section 3.4).

In the next subsection, we examine the parallel implementation of BSCF algorithm on multiprocessors.

3.2.3 Implementing the BSCF Algorithm on Multiprocessors

For our present study, we consider the *medium grain model* of parallelism in which tasks perform floating point operations over nonzero elements of entire columns of coefficient matrix. The following elementary tasks are considered for the BSCF algorithm.

- *fdivide(i,s)* divides by $\sqrt{A_{x0}[i, i]}$ every nonzero element of the sub-diagonal part of the ith column of sub-matrix A_{x0}.
- *bdivide(i,s)* divides by $\sqrt{A_{x1}[i, i]}$ every nonzero element of the super-diagonal part of the ith column of sub-matrix A_{x1}.
- *fmodify(i,vector,s)* subtracts the contents of *vector* from the ith column of a sub-matrix A_{x0}, at stage $s \in \{1 \cdots \log N\}$. *vector* is an appropriate multiple of some column j of A_{x0}, which modifies column i directly in forward direction at stage s.

- *bmodify(i,vector,s)* subtracts the contents of *vector* from the ith column of a sub-matrix A_{x1}, at stage $s \in \{1 \cdots \log N\}$. *vector* is an appropriate multiple of some column j of A_{x1}, which modifies column i directly in backward direction at stage s.

To keep track of the columns that each pivot should modify at each of the $\log N$ stages, we maintain the following data structures.

- $F_i^{(s)}$ denotes the set of all columns with indices smaller than i that modify the ith column in the forward direction at stage s.
- $B_i^{(s)}$ denotes the set of all columns with indices greater than i that modify the ith column in the backward direction at stage s.

These data structures are generated during the symbolic factorization phase. This phase is described in section 3.5. In the remaining part of this section, we describe the implementation of BSCF algorithm on a message passing multiprocessor - initially for the case where each processor is responsible for only one column of the coefficient matrix and then for the case where the number of processors p is less than the order N of the coefficient matrix.

Case $p = N$: In algorithm 1 below, N processors are being used to factorize an $N \times N$ sparse symmetric matrix A. For each processor P_i, the index of the column stored in it is *mycol*. At any stage $s \in \{1 \cdots N\}$, there are two copies of column *mycol* stored in processor P_i. The first copy is a part of the forward sub-matrix A_{x0} and is represented by $A_{x0}[*, mycol]$. The second copy is a part of the backward sub-matrix A_{x1} and is represented by $A_{x1}[*, mycol]$. Thus each processor is responsible for carrying out $fmodify(mycol, vector, s)$, $bmodify(mycol, vector, s)$, $fdivide(mycol, s)$, and $bdivide(mycol, s)$ operations at every stage, s, of the BSCF algorithm.

Algorithm 1 (*The parallel BSCF algorithm for case $p = N$*)
begin
for $s := 1$ **to** $\log N$ **do**
Let A_{x0} be the forward sub-matrix and A_{x1} be the
backward sub-matrix to which column *mycol* belongs
at stage s.

parbegin

Forward_factorize($mycol$,s);

Backward_factorize($mycol$,s);

parend

end

procedure Forward_factorize(col,s)

begin

for all $i \in F_{col}^{(s)}$ **do**

receive message of the form (col,$vector$,s) from

processor storing the column i;

$fmodify(col, vector, s)$;

if col belongs to the first half of sub-matrix A_{x0} **then**

$fdivide(col, s)$;

for all j such that $col \in F_j^{(s)}$ **do**

send the message (j, $A_{x0}[j, col] \times A_{x0}[*, col]$, s)

to processor storing column j;

else if $s < \log N$ **then**

(*copy column col of A_{x0} to column col of A_{x00}*)

$A_{x00}[*, col] := A_{x0}[*, col]$;

(*copy column col of A_{x0} to row col of A_{x01} since only

sub-diagonal part of the columns of the symmetric matrix A_{x0}

are stored*)

for all j such that $A_{x0}[j, col] \neq 0$ **do**

$A_{x01}[col, j] := A_{x0}[j, col]$;

end

procedure Backward_factorize(col,s)

begin

for all $i \in B_{col}^{(s)}$ **do**

receive message of the form (*col*,*vector*,*s*) from
processor storing the column i;
$bmodify(col, vector, s)$;

if *col* belongs to the second half of sub-matrix A_{x1} **then**
$bdivide(col, s)$;
for all j such that $col \in B_j^{(s)}$ **do**
send the message (j, $A_{x1}[j, col] \times A_{x1}[*, col]$, s)
to processor storing column j;
else if $s < \log N$ **then**
(*copy column *col* of A_{x1} to row *col* of A_{x10} since only
super-diagonal part of the columns of the symmetric matrix A_{x1}
are stored*)
for all j such that $A_{x1}[j, col] \neq 0$ **do**
$A_{x10}[col, j] := A_{x1}[j, col]$;
(*copy column *col* of A_{x1} to column *col* of A_{x11}*)
$A_{x11}[*, col] := A_{x1}[*, col]$;
end

The progression of the above algorithm for the case of $p = N = 4$ is shown in figure 3.2. In this figure we note that the size of the subset of processors with which any processor P_i communicates, reduces by half with every stage. In stage $s = 1$, all processors P_1 through P_4 communicate with each other. In stage $s = 2$, P_1 and P_2 communicate only with each other, and P_3 and P_4 communicate only with each other. Thus communication gets localized with every stage. Such a pattern of communication also holds for the case of $p < N$.

In practice, algorithm 1 would be extremely inefficient due to the excessive number of messages being passed. Also, the number of processors is usually much less than N, the order of the coefficient matrix. We now discuss the modification of algorithm 1 to the case where $p < N$.

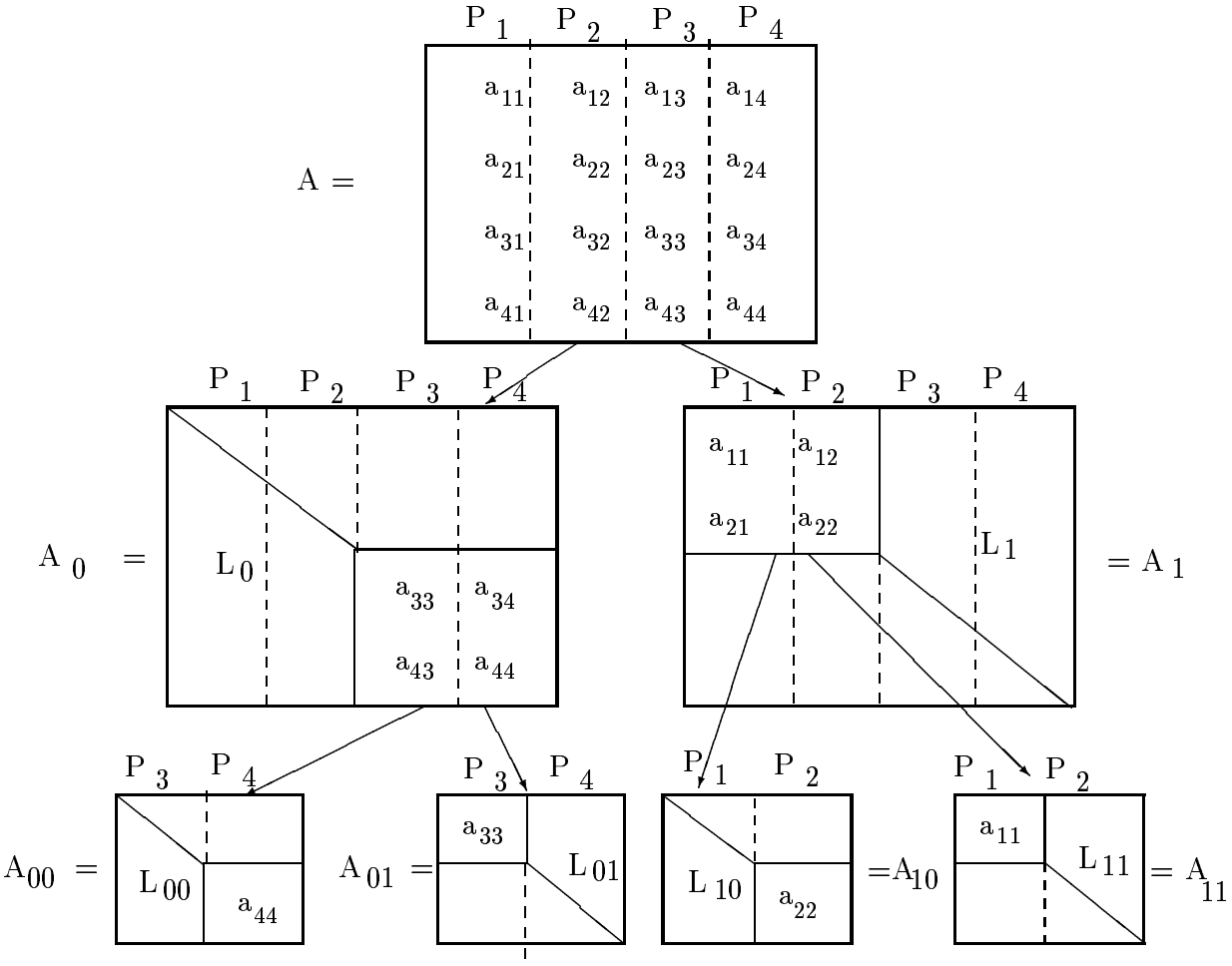


Figure 3.2: The progression of BSCF algorithm for $p = N = 4$ (one column is mapped onto each processor).

Case $p < N$: In Cholesky factorization, if column i modifies column j, then the factor, by which the modifying column i is multiplied, is an element $A[j,i]$ of the modifying column i itself. This happens due to the symmetric nature of the coefficient matrix being operated upon. Thus, as seen in algorithm 1, the multiple of the modifying column is calculated at the processor storing column i itself and the resulting vector is sent over to the processor storing column j which needs to be modified.

When $p < N$, there might be more than one column at a processor P_k, which modifies column j (i.e., more than one column stored at processor P_k might belong to the sets $F_j^{(s)}$ or $B_j^{(s)}$). In place of sending a separate vector as message corresponding to every column at P_k that modifies column j, we can add all these outgoing vectors together and send them as one vector to the processor storing column j. In this manner, the number of outgoing messages can be significantly reduced. Note that the above observation applies for modifications in both forward and backward factorizations.

In algorithm 2 below, we incorporate the above idea in the BSCF algorithm and present the *fan-in* BSCF algorithm. The set $List_{myid}$ is the set of columns stored in processor P_{myid}. Each processor maintains the sparse vectors $fUpdate_j$ and $bUpdate_j$ for $1 \leq j \leq N$. If column i is to modify column j in forward direction at stage s then, after performing $fdivide(i,s)$ operation, the processor P_{myid}, which stores the column i, adds an appropriate multiple of column i to the vector $fUpdate_j$. When such an addition has been performed for all the columns in processor P_{myid} that modify column j in forward direction at stage s, a message containing the $fUpdate$ vector is sent to the processor storing the column j. Similar mechanism operates for factorization in backward direction.

Algorithm 2 (*The parallel fan-in BSCF algorithm for case $p < N$*)

begin

  for $s := 1$ **to** $\log N$ **do**

    parbegin

      Forward_factorize($List_{myid}$,s);

      Backward_factorize($List_{myid}$,s);

    parend

end

procedure Forward_factorize($List$,s)

begin

for $i := 0$ **to** $N-1$ **do** $fUpdate_i$:=**0**;

while $List \neq \phi$ **do**

if $\exists i \in List$ such that $fdivide(j,s)$ has been performed for all $j \in F_i^{(s)}$ **then**

Let column i belong to the forward sub-matrix A_{x0} at stage s;

while messages of the form (i,$fvector$,s) have not been received from

all processors that store columns belonging to $F_i^{(s)}$ **do**

receive messages of the form (i,$fvector$,s);

$fmodify(i, fvector, s)$;

if column i belongs to the first half of sub-matrix A_{x0} **then**

$fdivide(i,s)$;

for all j such that $i \in F_j^{(s)}$ **do**

$fUpdate_j := fUpdate_j + A_{x0}[j,i] \times A_{x0}[*,i]$;

if $fdivide(k,s)$ has been done for all $k \in F_k^{(s)} \cap List$ **then**

send a message of the form (j,$fUpdate_j$,s)

to processor storing column j;

else if $s < \log N$ **then**

(*copy column i of A_{x0} to column i of A_{x00}*)

$A_{x00}[*,i] := A_{x0}[*,i]$;

(*copy column i of A_{x0} to row i of A_{x01} since only sub-diagonal

part of the columns of the symmetric

matrix A_{x0} are stored*)

for all j such that $A_{x0}[j,i] \neq 0$ **do**

$A_{x01}[i,j] := A_{x0}[j,i]$;

$List := List - i$;

end

procedure Backward_factorize($List$,s)

begin

for $i := 0$ **to** $N-1$ **do** $bUpdate_i$:=**0**;

while $List \neq \phi$ **do**

if $\exists i \in List$ such that $bdivide(j,s)$ has been performed for all $j \in B_i^{(s)}$ **then**

Let column i belong to the backward sub-matrix A_{x1} at stage s;

while messages of the form (i,$bvector$,s) have not been received from

all processors that store columns belonging to $B_i^{(s)}$ **do**

receive messages of the form (i,$bvector$,s);

$bmodify(i, bvector, s)$;

if column i belongs to the second half of sub-matrix A_{x1} **then**

$bdivide(i,s)$;

for all j such that $i \in B_j^{(s)}$ **do**

$bUpdate_j := bUpdate_j + A_{x1}[j,i] \times A_{x1}[*,i]$;

if $bdivide(k,s)$ has been done for all $k \in B_k^{(s)} \cap List$ **then**

send a message of the form (j,$bUpdate_j$,s)

to processor storing column j;

else if $s < \log N$ **then**

(*copy column i of A_{x1} to row i of A_{x10} since only

super-diagonal part of the columns of the

symmetric matrix A_{x1} are stored*)

for all j such that $A_{x1}[j,i] \neq 0$ **do**

$A_{x10}[i,j] := A_{x1}[j,i]$;

(*copy column i of A_{x1} to column i of A_{x11}*)

$A_{x11}[*,i] := A_{x1}[*,i]$;

$List := List - i$;

end

An important observation is in order in algorithm 2. Let the number of processors $p = 2^d$ (as in hypercube multiprocessors) and $N = 2^n$ ($n, d \in \mathcal{N}$, the set of natu-

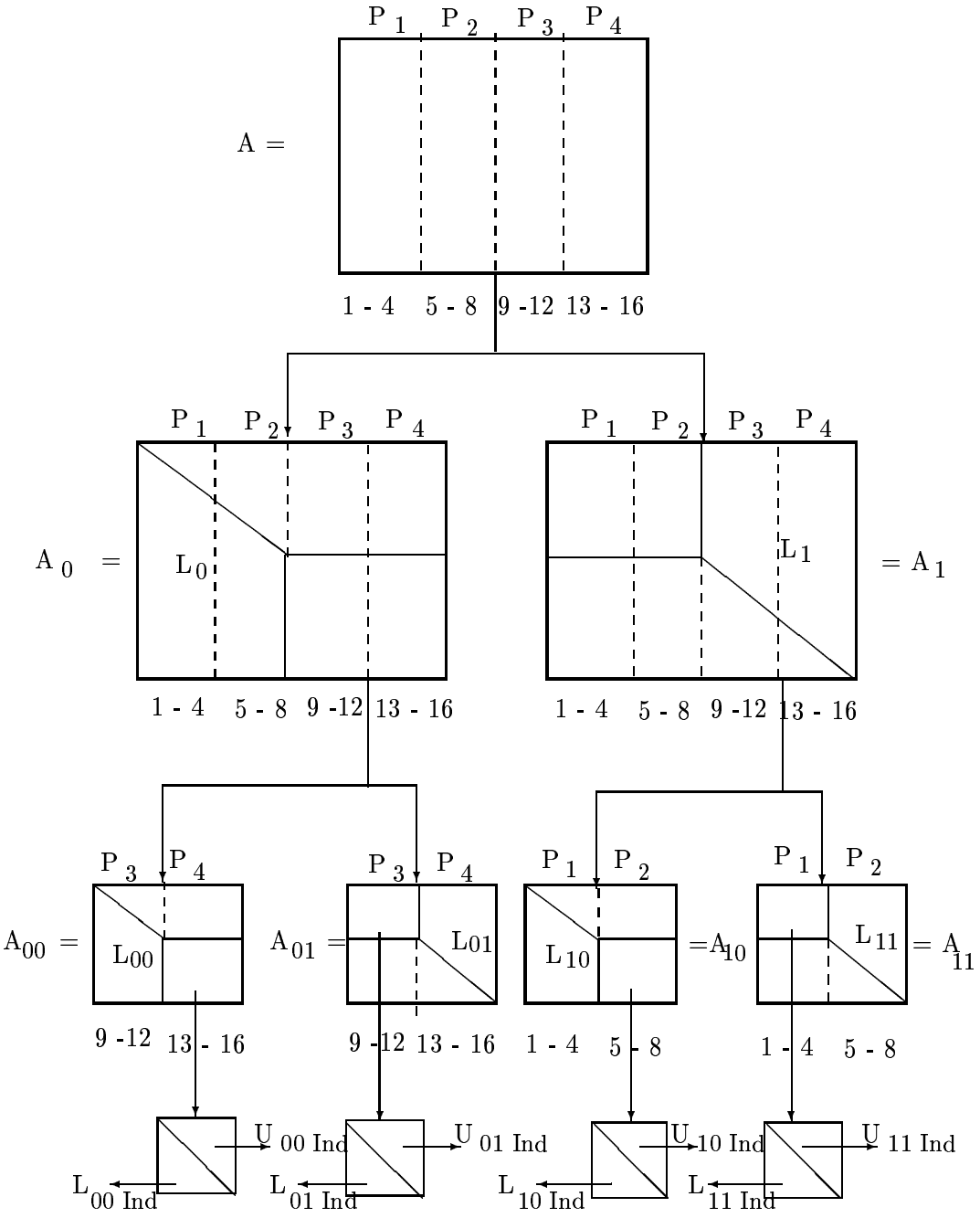


Figure 3.3: Progression of the BSCF algorithm for $p = 4$ and $N = 16$ (four columns are stored in each processor).

ral numbers). Assume that we map the equations on the processors in a block wrap manner (as shown in figure 3.3). Thus each processor holds $\frac{N}{p} = 2^{n-d}$ consecutive equations. At the end of $d = \log p$ stages of the fan-in BSCF algorithm, each processor contains an independent system of $\frac{N}{p}$ equations. This independent system can be factorized within a single processor without any communication with any other processor. Since, on a single processor, regular sequential sparse Cholesky factorization performs more efficiently than the fan-in BSCF algorithm, we can switch over to this regular sequential version after $\log p$ stages and factorize the coefficient matrix (say A_{ind}) of this independent system into the form $A_{ind} = L_{ind}L_{ind}^T$. This results in enhancing the performance of the fan-in BSCF algorithm. The manner in which this factorization proceeds is shown in figure 3.3.

3.3 The Substitution Phase

In this section we present the *bidirectional substitution* (BS) algorithm. Unlike the regular algorithm, which consists of two triangular solution components (i.e., the forward substitution followed by the backward substiution), the BS algorithm consists of only one forward solution component, which is followed by a single step division to yield the solution vector x. Following the pattern of the previous section, we first present an overall view of the concepts behind the BS algorithm. We then proceed to describe the manner in which the sparsity of the series of trapezoidal factor matrices can be exploited to obtain a higher degree of parallelism.

3.3.1 Bidirectional Substitution Algorithm - The Concept

The scheme we propose below is somewhat on similar lines to the parallel column triangular solver (PCTS) proposed by Li and Coleman in [34]. To find the solution vector x, for a given b-vector, we begin with two copies of b-vectors b_0 and b_1.

- *Step 1* : The vector b_0 is modified by successive columns of trapezoids of multipliers L_0 (i.e., from column 1 to column $\lceil \frac{N}{2} \rceil$). In other words, after modification by column $i-1$, the processor containing column i computes x_i as $x_i = b_0[i]/L_0[i,i]$

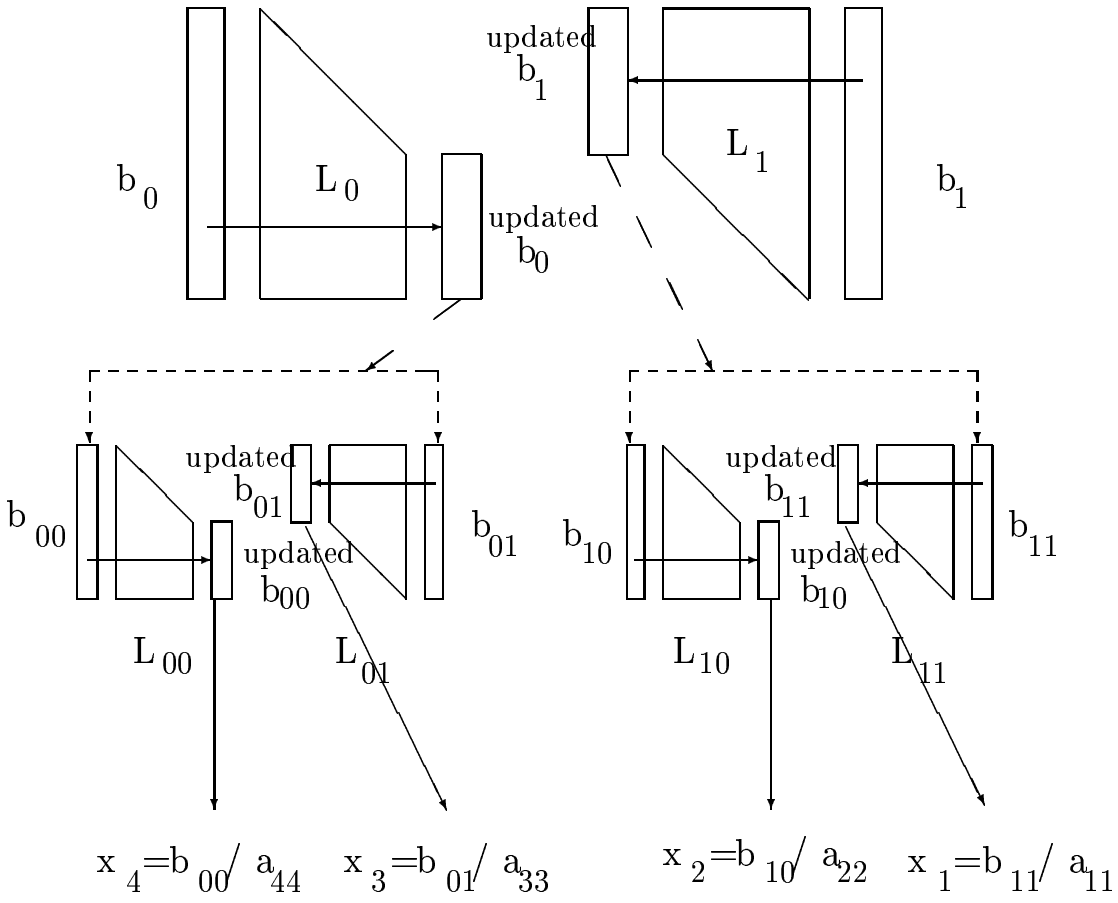


Figure 3.4: The progression of substitution phase for $N = 4$

and modifies the remaining elements of b_0-vector as $b_0[j] = b_0[j] - L_0[j,i] * x_i$ for all j such that $L_0[j,i] \neq 0$. At the end of updation by L_0, the size of vector b_0 is reduced to half its original size (see figure 3.4). Simultaneously, the vector b_1 is updated by successive columns of the trapezoidal matrix of multipliers L_1 in backward direction (i.e., from column N to column $\lceil \frac{N}{2} \rceil + 1$). In other words, after modification by column $i+1$, the processor containing column i computes x_i as $x_i = b_1[i]/L_1[i,i]$ and modifies the remaining b_1-vector as $b_1[j] = b_1[j] - L_1[j,i] * x_i$ for all j such that $L_1[j,i] \neq 0$. At the end of updation by L_1, the size of vector b_1 is reduced to half its original size (see figure 3.4).

- *Step 2* : The reduced b_0 is copied to form vectors b_{00} and b_{01} whereas the reduced b_1 is copied to form vectors b_{10} and b_{11}. The new vectors b_{00} and b_{10} are modified by L_{00} and L_{10} respectively in forward direction whereas the vectors b_{01} and b_{11} are modified by L_{01} and L_{11} respectively in backward direction. Thus the size of these new b-vectors gets reduced by another factor of half (see figure 3.4).
- *Step 3* : This process of reducing the size of b-vectors and doubling their numbers continues for $\log N$ stages by which time there will be N b-vectors of only one element each. These N b-vectors, when divided by N elements obtained at the end of factorization phase, will give N x-vector elements.

3.3.2 Increasing Parallelism by Exploiting Sparsity

In the above scheme we observe that the process of modifying a b-vector through successive columns of a trapezoid is inherently sequential and is communication intensive in case the successive columns happen to reside on separate processors. George et.al. have proposed in [14], parallel schemes for solving sparse triangular systems resulting from regular Cholesky factorization. Their scheme is an adaptaion of the corresponding dense algorithm proposed by Romine and Ortega in [49] and it uses the following inner product form to carry out forward factorization.

$$x_i = \left(b_i - \sum_{\{j | L[i,j] \neq 0\}} (L[i,j] * x_j) \right) / L[i,i] \quad i = 1, 2, \cdots, N$$

Since the columns and the corresponding solution components are distributed among the processors, the inner product computation is partitioned accordingly.

The above concept of distributed computation of inner product can be applied to the BS algorithm. Consider the case where the vector b_{x0} is to be updated by the trapezoid L_{x0} in the forward direction. Instead of moving the vector b_{x0} from left to right across the trapezoid L_{x0}, each element $b_{x0}[i]$ is updated as follows. Each processor computes the products of the elements of the row i of the trapezoid that it contains with the corresponding elements of the solution vector x and sends their sum i.e., the partial inner product, to the processor containing column i. Upon receiving the contributions to the inner product from each processor, the processor storing the column i subtracts them from b_{x0}. If column i belongs to the first half of the matrix A_{x0} then, after subtracting the complete inner product of row i in L_{x0} from $b_{x0}[i]$, the processor storing the column i computes $x_i = b_{x0}[i]/L_{x0}[i,i]$. This x_i is then used for calculating the partial inner products of rows $j > i$. On the other hand if the column i belongs to the second half then after subtracting the complete inner product of row i in L_{x0} from $b_{x0}[i]$, two copies of the element $b_{xo}[i]$, namely $b_{x00}[i]$ and $b_{x01}[i]$, are made for modification at the next stage of the BS algorithm. Similar mechanism operates while updating a vector b_{x1} with a trapezoid L_{x1} in backward direction. The complete details of the BS algorithm are given below.

Algorithm 3 (* The bidirectional substitution algorithm *)

begin

for $s := 1$ **to** $\log N$ **do**

parbegin

Forward_modify($List_{myid}$,s);

Backward_modify($List_{myid}$,s);

parend

end

procedure Forward_modify($List$,s)

begin

Let b_{x0} be the forward copy of the b-vector to be modified

by trapezoid L_{x0} at stage s.

for $i := 1$ **to** N **do** $t_i := 0$;

for all $i \in List$ **do**

for all j such that processor P_j has nonzeros belonging to row i of L_{x0} **do**

receive message (i,t) having partial inner product t from processor P_j;

$b_{x0}[i] := b_{x0}[i] - t$;

if column i belongs to the first half of L_{x0} **then**

$x_i := b_{x0}[i]/L_{x0}[i,i]$;

for all j such that $L_{x0}[j,i] \neq 0$ **do**

$t_j := t_j + x_i * L_{x0}[j,i]$;

if x_k has been calculated for all k such that $L_{x0}[j,k] \neq 0$ and

$k \in List$ **then**

send message (j,t_j) to processor storing column j;

else if $s < \log N$ **then**

$b_{x00}[i] := b_{x0}[i]$;

$b_{x01}[i] := b_{x0}[i]$;

else (* $s = \log N$ *) $x_i := b_{x0}[i]/L_{x0}[i]$;

end

procedure Backward_modify($List$,s)

begin

Let b_{x1} be the backward copy of the b-vector to be modified

by trapezoid L_{x1} at stage s.

for $i := 1$ **to** N **do** $t_i := 0$;

for all $i \in List$ **do**

for all j such that processor P_j has nonzeros belonging to row i of L_{x1} **do**

receive message (i,t) having partial inner product t from processor P_j;

$b_{x1}[i] := b_{x1}[i] - t$;

if column i belongs to the second half of L_{x1} **then**

$x_i := b_{x1}[i]/L_{x1}[i,i]$;

for all j such that $L_{x1}[j,i] \neq 0$ **do**

$t_j := t_j + x_i * L_{x1}[j,i]$;

if x_k has been calculated for all k such that $L_{x1}[j,k] \neq 0$ and

$k \in List$ **then**

send message (j,t_j) to processor storing column j;

else if $s < \log N$ **then**

$b_{x10}[i] := b_{x1}[i]$;

$b_{x11}[i] := b_{x1}[i]$;

else (* $s = \log N$ *) $x_i := b_{x1}[i]/L_{x1}[i]$;

end

As in the case of the BSCF algorithm, a special situation arises when $p = 2^d$ and $N = 2^n$ $(n, d \in \mathcal{N})$. After $d = \log p$ stages, the BSCF algorithm switches over to the regular sparse Cholesky factorization and produces triangular factor matrix of the form L_{ind} in the last stage such that $A_{ind} = L_{ind}L_{ind}^T$. Thus in the substitution phase, let b_{ind} be one of the p reduced vectors after $\log p$ stages of BS algorithm. We now switch over to the sequential substitution algorithm for solving the two triangular systems, $L_{ind}y = b_{ind}$ and $L_{ind}^T x = y$. In this manner, we avoid executing excessive number of floating point operations when all the remaining computations are resricted to occur within individual processors.

In the next two sections, we describe the ordering and the symbolic factorization algorithms that precede the BSCF algorithm.

3.4 Ordering the Sparse Symmetric Matrix for Bidirectional Factorization

A good initial ordering of a sparse matrix A is crucial to the efficient solution of the sparse symmetric system $Ax = b$. The basic aim of the ordering phase is to reorder the columns of the coefficient matrix in such a manner that during the factorization phase, the amount of fill-in is minimized and the degree of parallelism is maximized. In a parallel environment, the former aim is not as important as the latter aim since large amounts of memory are available very cheaply.

Sparse symmetric matrices chiefly arise from $k \times k$ regular grids that are encountered in finite element problems. The principal ordering heuristic used for reordering the matrices obtained from the regular grid problems is the popular *nested dissection* ordering method [13, 10]. The nested dissection ordering yields short and wide elimination trees that are well suited for parallel factorization algorithms. For regular

Cholesky factorization, this ordering technique satisfies the criteria of both low fill-in and short and wide elimination trees. However, the nested dissection ordering in its existing form is not suited for the BSCF algorithm due to reasons given below. Recall that in section 3.2.2 we defined the concepts of forward elimination tree and backward elimination tree for the BSCF algorithm. The degree of parallelism while factorizing in forward direction depends on the shape of the forward elimination tree and that for factorizing in backward direction depends on the shape of the backward elimination tree. An ideal ordering for the BSCF algorithm is one in which both the elimination trees are as short and wide as possible. The forward elimination tree obtained from nested dissection algorithm is short and wide and hence desirable for parallel factorization. On the other hand the backward elimination tree obtained from nested dissection algorithm is lean and tall and hence undesirable for parallel factorization.

In the remaining part of this section, with the help of an example of a 7×7 grid, we show why the regular nested dissection algorithm is not suited for BSCF algorithm and then we describe how it can be modified to yield orderings suitable for the BSCF algorithm.

The nested dissection algorithm begins by recursively dividing a $k \times k$ grid into two disjoint parts using a set of nodes as *separator* nodes and applying the nested dissection algorithm again to the two separated halves. Figure 3.5 shows the manner in which the separators (S1 to S15) divide a 7×7 grid. The recursive division of the grid yields a tree structure of separators and nodes as shown in figure 3.6. We call this tree a *nested dissection tree*. The internal nodes of the tree are *separator blocks* and the leaves of the tree are blocks of node(s) at lowermost level which cannot be further sub-divided using nested dissection. The dimension of such blocks can be 1×1, 1×2, 2×1 or 2×2. Such indivisible blocks are called *leaf blocks*.

In regular nested dissection ordering, all the grid points at the leaf blocks(say at level 0) are numbered in ascending order. Then the separator grid points at level 1 are numbered, then level 2 and so on until the grid points at the root separator blocks get numbered. The ordering resulting from this scheme is shown in figure 3.7 and the forward and backward elimination trees resulting from this ordering are shown in figure 3.8. As seen from figure 3.8, although the forward tree is short and wide,

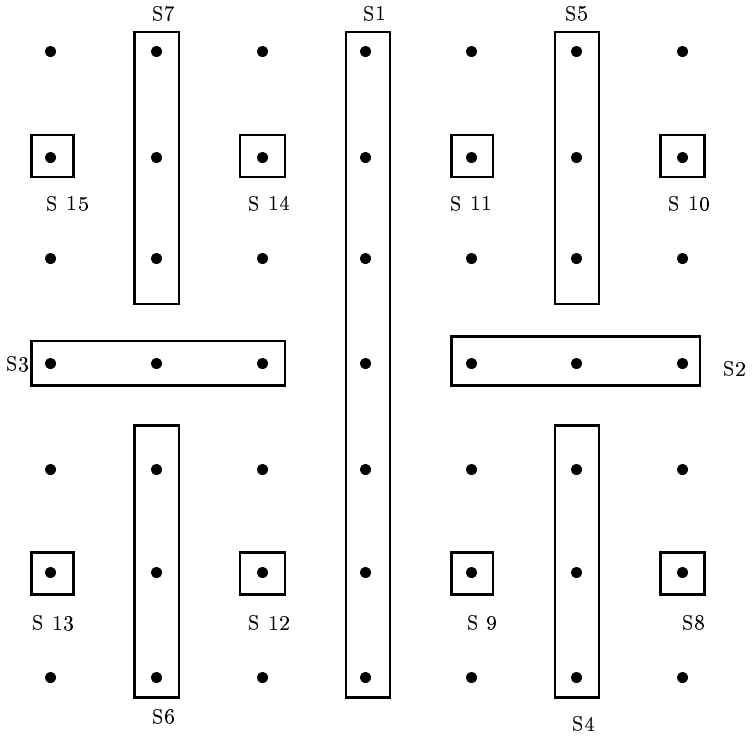


Figure 3.5: Dissection of a 7×7 grid by separators during nested dissection

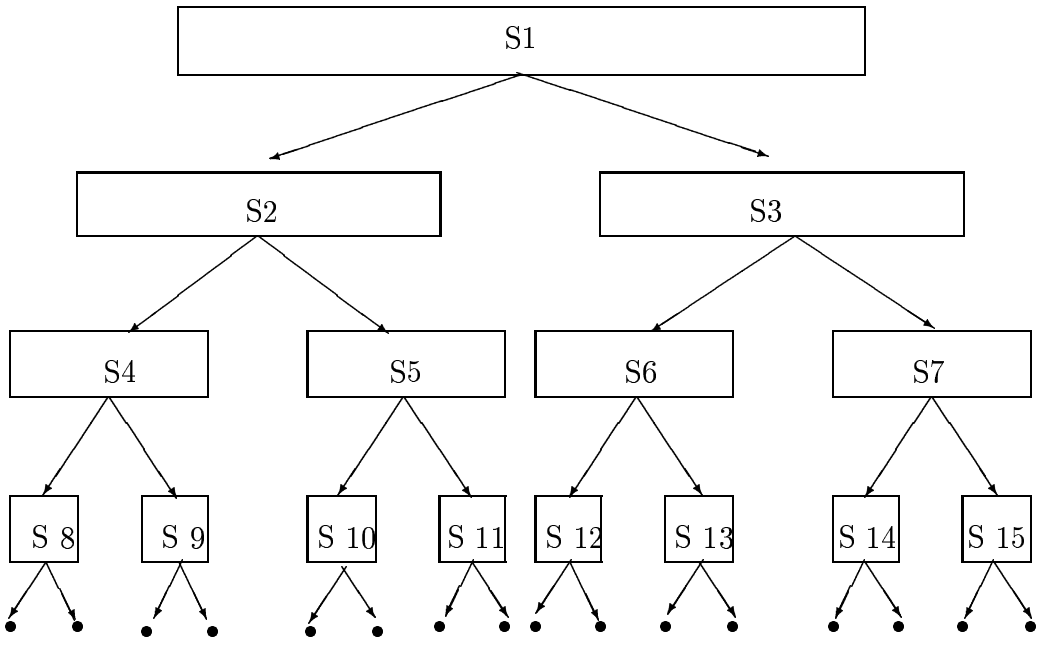


Figure 3.6: The nested dissection tree for a 7×7 grid

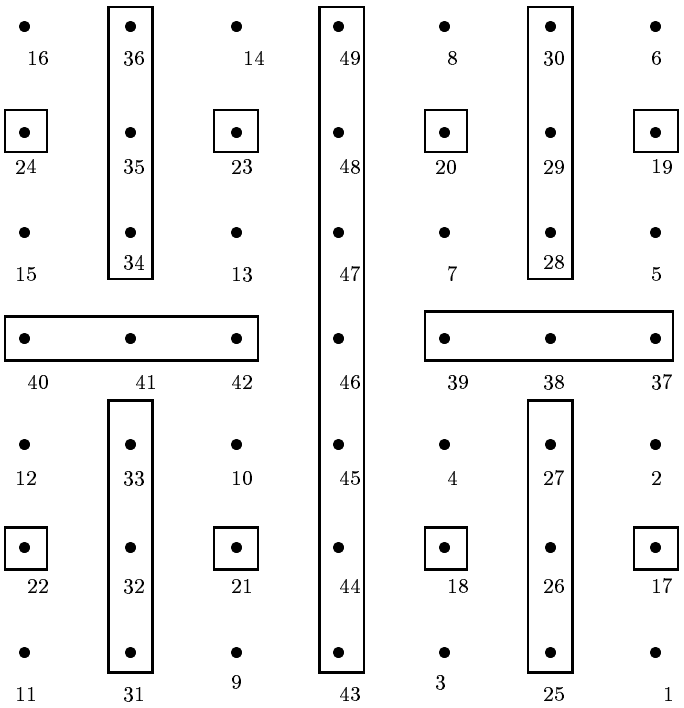


Figure 3.7: Ordering of a 7×7 grid using regular nested dissection ordering

the backward tree is lean and tall. Hence this ordering is not conducive for good performance of the BSCF algorithm.

We now look at a modification of the regular nested dissection algorithm which produces orderings that provide reasonably good parallelism properties in both forward and backward directions. We call this heuristic as the *bidirectional nested dissection ordering* which proceeds as follows.

- *Step 1* : Carry out the dissection part of the nested dissection algorithm as described above. This gives a nested dissection tree as shown in figure 3.6.
- *Step 2* : At each level of the nested dissection tree, approximately half of the tree nodes are labeled white and the other half are labeled black as shown in figure 3.9.
- *Step 3* : While numbering the grid points, we proceed as follows.

    1. Keep two counts - *whiteCount* initialized to 1 and *blackCount* initialized to $k \times k$.

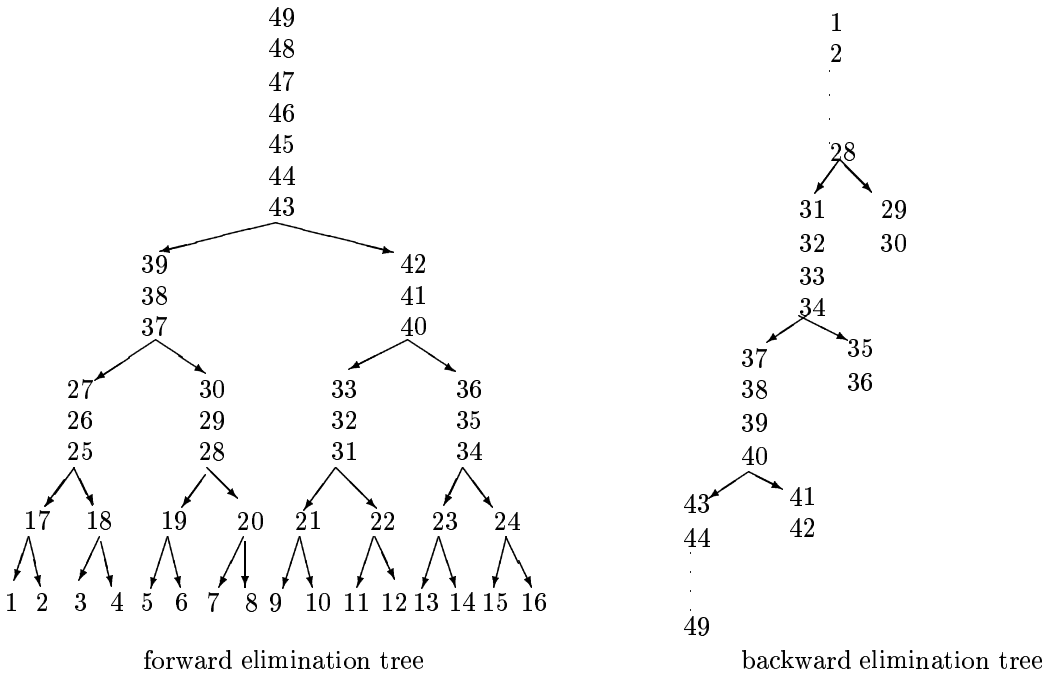


Figure 3.8: The forward and backward elimination trees for a 7×7 grid obtained using regular nested dissection ordering

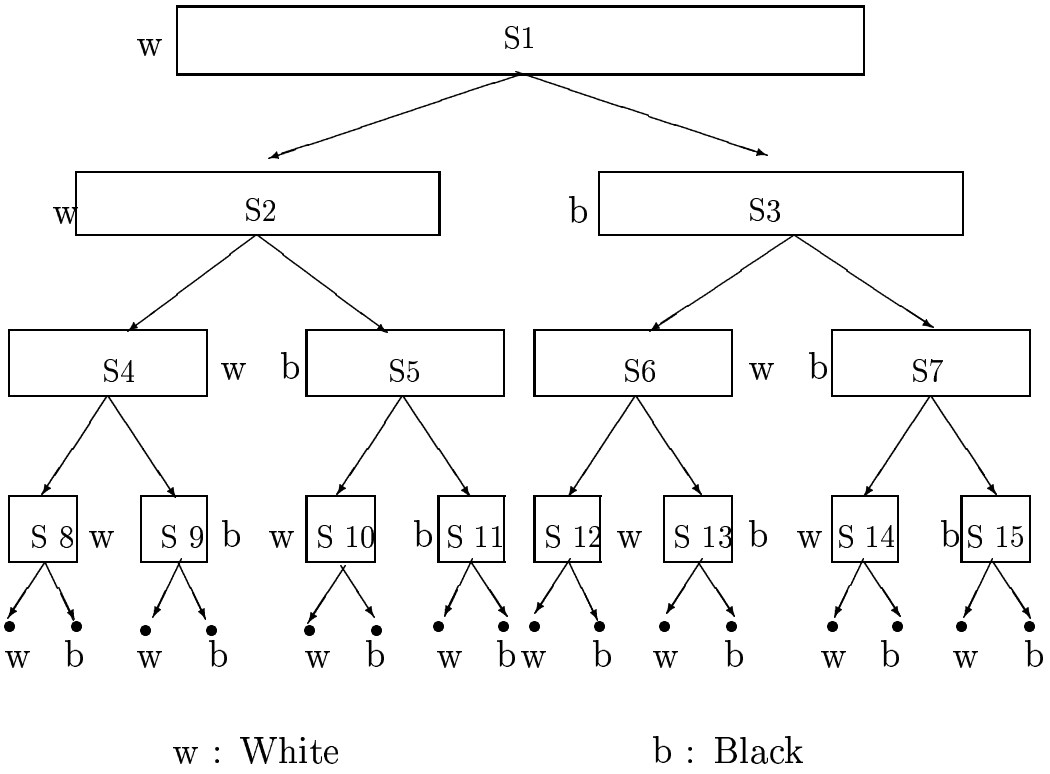


Figure 3.9: The colouring of tree nodes in bidirectional nested dissection ordering

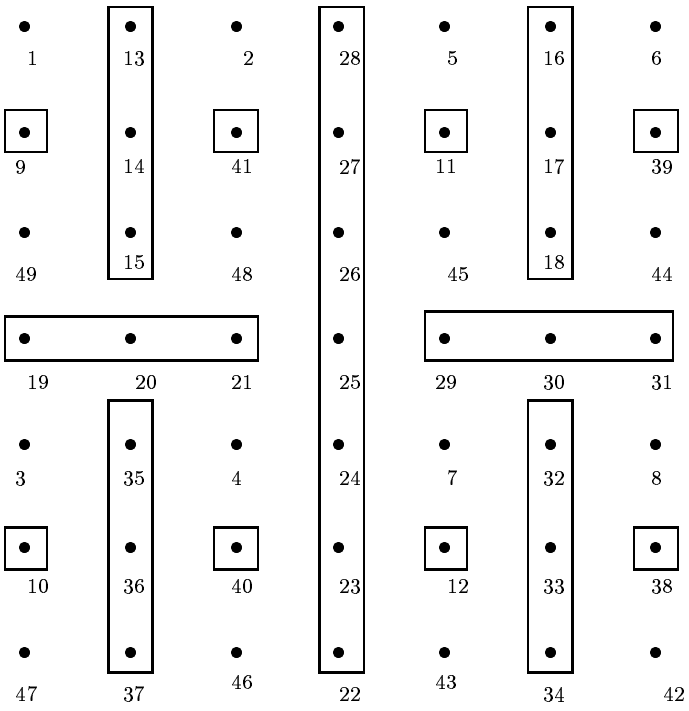


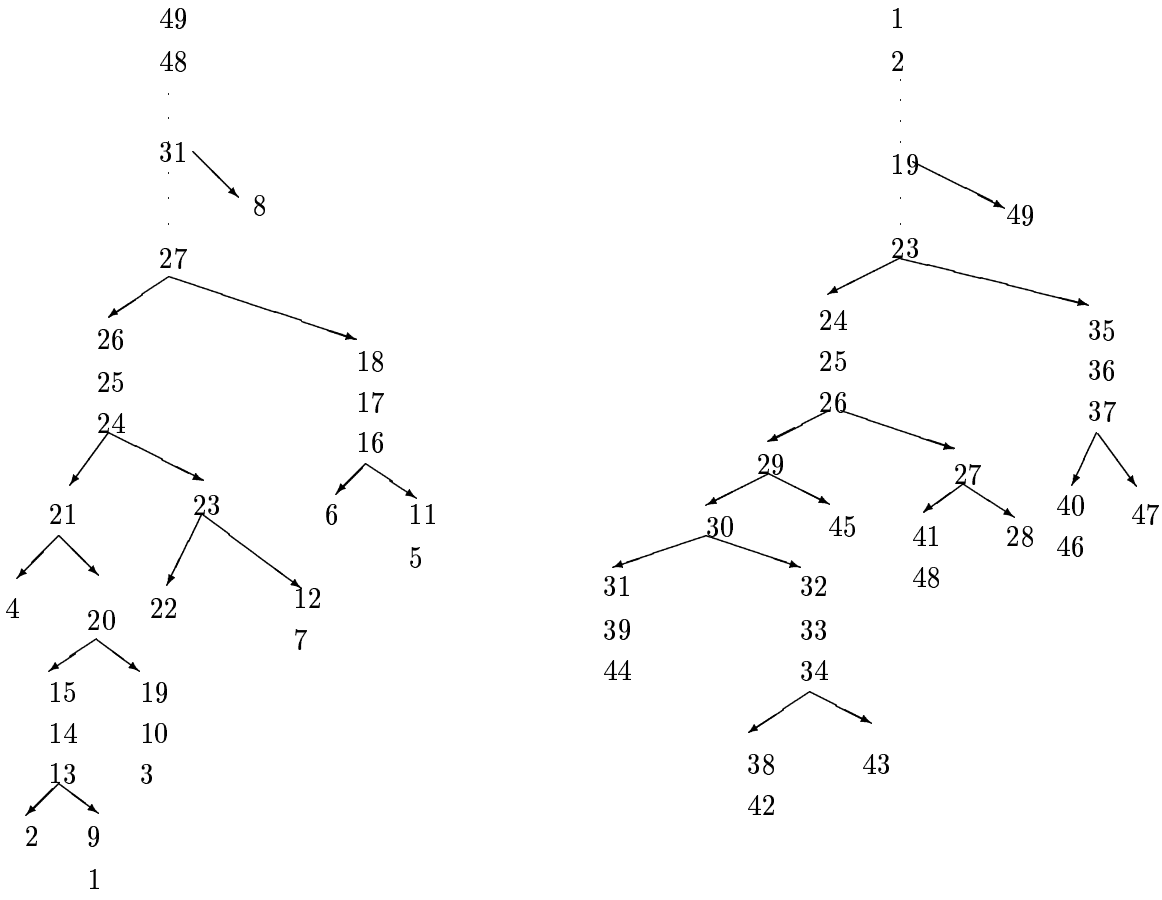
Figure 3.10: Ordering of a 7×7 grid using bidirectional nested dissection ordering

2. Take a grid point at level 0. If the leaf node to which it belongs is white then number the grid point as *whiteCount* and increment *whiteCount*. Otherwise the leaf node is black. Hence number the grid point as *blackCount* and decrement *blackcount*.

3. The above step is applied to all grid points of each node at level 0 followed by each node at level 1 and so on upto the root.

The ordering obtained from this scheme is shown in figure 3.10 and the corresponding forward and backward elimination trees are shown in figure 3.11. As seen in this figure, although the forward elimination tree is not as short and wide as in the case of regular nested dissection ordering, the backward tree is definitely more conducive for good performance of parallel factorization than in the previous case. Essentially we have succeeded in balancing the degree of parallelism in both forward and backward directions so that lack of parallelism in any one direction does not act as a bottleneck to the entire BSCF algorithm.

In the next section we look at the bidirectional symbolic factorization algorithm



forward elimination tree

backward elimination tree

Figure 3.11: The forward and backward elimination trees for a 7×7 grid obtained using bidirectional nested dissection ordering

which allocates memory and sets up the appropriate data structures prior to the BSCF algorithm.

3.5 The Bidirectional Symbolic Factorization Algorithm

The principal aim of the symbolic factorization phase is to determine apriori, the data structure of the factor matrices that result from the numerical factorization phase. As seen in section 3.2, the BSCF algorithm creates a series of trapezoidal factor matrices of multipliers. Hence, the bidirectional symbolic factorization algorithm, which precedes the BSCF phase, does the following.

- It determines the structure of each trapezoidal factor matrix at each of the $\log N$ stages and
- It initializes the data structures for the sets $F_i^{(s)}$ and $B_i^{(s)}$ which are required during the BSCF algorithm.

We define $Colstruct(A_{x0}, i)$ to denote the set of row indices of nonzeros in the sub-diagonal part of column i in the forward matrix A_{x0}.

$$Colstruct(A_{x0}, i) = \{j \mid j > i \text{ and } A_{x0}[j, i] \neq 0\}.$$

In a similar fashion, we define $Colstruct'(A_{x1}, i)$ to denote the set of row indices of nonzeros in the super-diagonal part of column i of the backward matrix A_{x1}.

$$Colstruct'(A_{x1}, i) = \{j \mid j < i \text{ and } A_{x1}[j, i] \neq 0\}.$$

We now describe the bidirectional symbolic factorization algorithm.

Algorithm 4 (*The bidirectional symbolic factorization algorithm*)
begin
  for $s := 1$ **to** $\log N$ **do**
    for $col := 1$ **to** N **do**
      $F_{col}^{(s)} := \phi; B_{col}^{(s)} := \phi;$
  for $s := 1$ **to** $\log N$ **do**
    for $col := 1$ **to** N **do**

Forward_SF(col,s);

for $col := N$ **downto** 1 **do**

Backward_SF(col,s);

end

procedure Forward_SF(col,s)

begin

Let A_{x0} be the forward sub-matrix that contains column col at stage s;

if col belongs to the first half of A_{x0} **then**

Calculate $fparent(col, A_{x0})$ using definition given in section 3.2.2;

if $fparent(col, A_{x0})$ belongs to the first half of A_{x0} **then**

$Colstruct(A_{x0}, fparent(col, A_{x0}))$:=

$Colstruct(A_{x0}, fparent(col, A_{x0}) \cup Colstruct(A_{x0}, col)$;

for all j such that j belongs to second half of A_{x0} and $A_{x0}[col, j] \neq 0$ **do**

$Colstruct(A_{x0}, j) := Colstruct(A_{x0}, j) \cup Colstruct(A_{x0}, col)$;

for all j such that $j \in Colstruct(A_{x0}, col)$ **do**

$F_j^{(s)} := F_j^{(s)} \cup \{col\}$;

else

$Colstruct(A_{x00}, col) := Colstruct(A_{x0}, col)$;

for all $j \in Colstruct(A_{x0}, col)$ **do**

$Colstruct'(A_{x01}, j) := Colstruct'(A_{x01}, j) \cup \{col\}$;

end

procedure Backward_SF(col,s)

begin

Let A_{x1} be the backward sub-matrix that contains column col at stage s;

if col belongs to the second half of A_{x1} **then**

Calculate $bparent(col, A_{x1})$ using definition given in section 3.2.2;

if $bparent(col, A_{x1})$ belongs to the second half of A_{x1} **then**

$Colstruct'(A_{x1}, fparent(col, A_{x1}))$:=

$Colstruct'(A_{x1}, fparent(col, A_{x1}) \cup Colstruct'(A_{x1}, col)$;

for all j such that j belongs to first half of A_{x1} and $A_{x1}[col, j] \neq 0$ **do**

$Colstruct'(A_{x1}, j) := Colstruct'(A_{x1}, j) \cup Colstruct'(A_{x1}, col)$;

for all j such that $j \in Colstruct'(A_{x1}, col)$ **do**

$B_j^{(s)} := B_j^{(s)} \cup \{col\}$;

else

for all $j \in Colstruct'(A_{x1}, col)$ **do**

$Colstruct(A_{x10}, j) := Colstruct(A_{x10}, j) \cup \{col\}$;

$Colstruct'(A_{x11}, col) := Colstruct'(A_{x1}, col)$;

end

The bidirectional symbolic factorization algorithm described above has time complexity proportional to the number of nonzero elements stored in trapezoids at each stage. Since the symbolic factorization algorithm is executed only once while solving for multiple b-vectors and also since this phase takes significantly lower time than the numerical factorization phase, parallelizing this phase does not yield significant improvements in the overall performance.

For the case of regular symbolic factorization, parallel algorithms have been described in [16, 28]. While the former scheme by George et.al. requires the information about the elimination tree structure apriori, the latter scheme by P. S. Kumar et.al. does not require this information and uses the concept of *false elimination trees (fet)* to compute the symbolic factorization. More specifically, the computation begins with the leaves of the false elimination tree which pass their column structure information to their true parents. Each internal node then combines the column structures of all its children with its own column structure, computes the true parent and sends its column structure information to its true parent. This process continues till all the information propagates to the root node.

We have developed a parallel bidirectional symbolic factorization algorithm based on a similar concept of *forward* and *backward* false elimination trees.

- $ffparent(i, s)$ denotes the false forward parent of a column i in the sub-matrix

A_{x0} being factorized in the forward direction at stage s.

$$ffparent(i,s) = min\{j \mid j \in \text{ first half of } A_{x0} \text{ and } j \in Colstruct(A_{x0}, i)\}.$$

- $fbparent(i,s)$ denotes the false backward parent of a column i in the sub-matrix A_{x1} being factorized in the backward direction at stage s.

$$fbparent(i,s) = max\{j \mid j \in \text{ second half of } A_{x1} \text{ and } j \in Colstruct'(A_{x1}, i)\}.$$

The details of this algorithm are described below.

Algorithm 5 (*The parallel bidirectional symbolic factorization*)

begin

for $s := 1$ to $\log N$ **do**

parbegin

Forward_SF($List_{myid}$,s);

Backward_SF($List_{myid}$,s);

parend

end.

procedure Forward_SF($List$,s)

begin

for each $i \in List$ **do**

Let A_{x0} be the forward sub-matrix to which column i belongs at stage s;

$dummy\_parent :=$ last node of sub-matrix A_{x0};

Determine the false forward parent $ffparent(i,s)$;

send $ffparent(i,s)$ to processor containing $dummy\_parent$;

if $i = dummy\_parent$ **then**

receive $ffparent(j,s)$ from each column j;

broadcast forward fet T_{ff} constructed from received

$ffparent$ information;

receive forward fet T_{ff} broadcast from $dummy\_parent$;

Let the children of column i in T_{ff} be $CHLD(i)$;

(*initialise the expected and accumulated weights for node i*)

$exp\_wt(i) :=\mid CHLD(i) \mid$; $acc\_wt(i) := 0$;

$first(i)$:=*true*;

if column i is a true leaf of T_{ff} **and** column i is in

first half of sub-matrix A_{x0} **then**

send $Colstruct(A_{x0}, i)$ to $ffparent(i, s)$ with weight 1;

send $Colstruct(A_{x0}, i)$ with weight 0 to all nodes j in second half of

A_{x0} such that $j \in Colstruct(A_{x0}, i)$;

repeat

receive a message S intended for column i;

Let the message be from processor storing column j with weight w;

if column i is in first half of sub-matrix A_{x0} **then**

case type of S

attach or ordinary:

$Colstruct(A_{x0}, i) := Colstruct(A_{x0}, i) \cup Colstruct(A_{x0}, j)$;

$acc\_wt := acc\_wt + w$;

if $j \in CHLD(i)$ **then** delete j from $CHLD(i)$;

if ($\mid CHLD(i) = 0 \mid$) **and** $(acc\_wt(i) \geq exp\_wt(i))$ **then**

$ffparent(i, s) := k \; where \; k = min(Colstruct(A_{x0}, i))$;

if $ffparent(i)$ has changed **then**

send a detach message to old parent;

if $first(i)$ **then**

$wt := acc\_wt(i) - exp\_wt(i) + 1$;

$exp\_wt(i) := 0$;

$first(i)$:=**false**;

else

$wt := w$;

send $Colstruct(A_{x0}, i)$ to $ffparent(i)$ with weight wt;

send $Colstruct(A_{x0}, i)$ to all nodes j in second half of A_{x0}

such that $j \in Colstruct(A_{x0}, i)$ with weight 0;

detach :

delete j from $CHLD_(i)$;

else

case type of S

attach or ordinary:

if $j \in Colstruct(A_{x0}, i)$ **then**

$Colstruct(A_{x0}, i) := Colstruct(A_{x0}, i) \cup Colstruct(A_{x0}, j)$;

detach:

if $i = dummy\_parent$ **then**

delete j from $CHLD(i)$;

if ($\mid CHLD(i) = 0 \mid$) **then**

broadcast forward phase over message;

until S is forward phase over message;

for each $i \in List$ **do**

if column i is in second half of sub-matrix **then**

$Colstruct(A_{x00}, i) := Colstruct(A_{x0}, i)$;

for all j such that $A_{x0}[j, i] \neq 0$ **do**

$Colstruct'(A_{x01}, j) := Colstruct(A_{x01}, j) \cup i$;

end

procedure Backward_SF($List$,s)

begin

for each $i \in List$ **do**

Let A_{x1} be the backward sub-matrix to which column i belongs at stage s;

$dummy\_parent$:= last node of sub-matrix A_{x1};

Determine the false backward parent $fbparent(i, s)$;

send $fbparent(i, s)$ to processor containing $dummy\_parent$;

if $i = dummy\_parent$ **then**

receive $fbparent(j, s)$ from each column j;

broadcast backward fet T_{fb} constructed from received

$fbparent$ information;

receive backward fet T_{fb} broadcast from $dummy\_parent$;

Let the children of column i in T_{fb} be $CHLD(i)$;

$exp\_wt(i) :=\mid CHLD(i) \mid$; $acc\_wt(i) := 0$;

$first(i)$:=*true*;

if column i is a true leaf of T_{fb} **and** column i is in second half

of sub-matrix A_{x1} **then**

send $Colstruct'(A_{x1}, i)$ to $fbparent(i, s)$ with weight 1;

send $Colstruct'(A_{x1}, i)$ with weight 0 to all nodes j in first half

of sub-matrix A_{x1} such that $j \in Colstruct'(A_{x1}, i)$;

repeat

receive a message S intended for column i;

Let the message be from processor storing column j with weight w;

if column i is in second half of sub-matrix A_{x1} **then**

case type of S

attach or ordinary:

$Colstruct'(A_{x1}, i) := Colstruct'(A_{x1}, i) \cup Colstruct'(A_{x1}, j)$;

$acc\_wt := acc\_wt + w$;

if $j \in CHLD(i)$ **then** delete j from $CHLD(i)$;

if ($\mid CHLD(i) = 0 \mid$) **and** ($acc\_wt(i) \geq exp\_wt(i)$) **then**

$fbparent(i, s) := k$ *where* $k = max(Colstruct'(A_{x1}, i))$;

if $fbparent(i)$ has changed **then**

send a detach message to old parent;

if $first(i)$ **then**

$wt := acc\_wt(i) - exp\_wt(i) + 1$;

$exp\_wt(i) := 0$;

$first(i)$:=**false**;

else

$wt := w$;

send $Colstruct'(A_{x1}, i)$ to $fbparent(i)$ with weight wt;

send $Colstruct'(A_{x1}, i)$ to all nodes j in first half of sub-matrix

such that $j \in Colstruct'(A_{x1}, i)$ with weight 0;

detach :

delete j from $CHLD_(i)$;

else

case type of S

attach or ordinary:

if $j \in Colstruct'(A_{x1}, i)$ **then**

$Colstruct'(A_{x1}, i) := Colstruct'(A_{x1}, i) \cup Colstruct'(A_{x1}, j)$;

detach:

if $i = dummy\_parent$ **then**

delete j from $CHLD(i)$;

if ($\mid CHLD(i) = 0 \mid$) **then**

broadcast backward phase over message;

until S is backward phase over message;

for each $i \in List$ **do**

if column i is in first half of sub-matrix **then**

for all j such that $A_{x1}[j, i] \neq 0$ **do**

$Colstruct(A_{x10}, j) := Colstruct'(A_{x10}, j) \cup i$;

$Colstruct'(A_{x11}, i) := Colstruct'(A_{x1}, i)$;

end

3.6 Experimental Results and Performance Analysis

To evaluate the performance of the entire bidirectional scheme presented in this work, we implemented a hypercube simulator in C language and compared the *speedups* obtained from the bidirectional scheme with those obtained from the regular scheme. We used SPARC Classic machine to carry out our simulations.

In the bidirectional scheme, we implemented each of the four phases as follows.

- *Ordering* : The bidirectional nested dissection ordering described in section 3.4.
- *Symbolic factorization* : The sequential bidirectional symbolic factorization algorithm described in section 3.5.
- *Numerical factorization* : The parallel BSCF algorithm described in section 3.2.
- *Substitution* :The parallel BS algorithm described in section 3.3.

In the regular scheme, we implemented each of the four phases as follows.

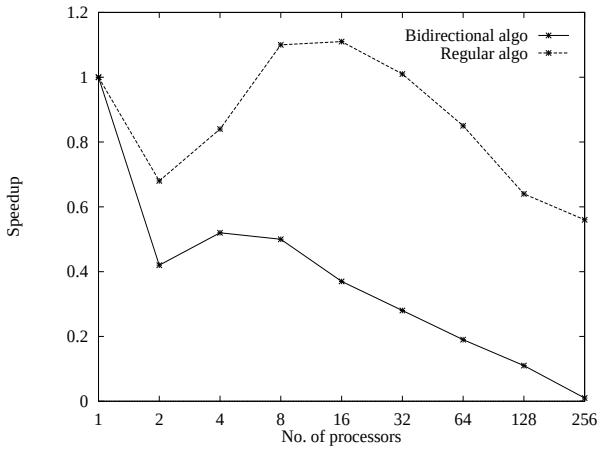
- *Ordering* : The regular nested dissection algorithm for ordering a $k \times k$ grid.
- *Symbolic factorization* : The sequential symbolic factorization algorithm presented in [16].
- *Numerical factorization* : The parallel fan-in algorithm given in [4].
- *Substitution* :The elimination tree based forward and back substitution algorithms given in [29].

Mapping of columns onto processors is an important issue. For the bidirectional scheme, we have used the *block wrap around mapping* using gray code whereas for the regular algorithm we have used the *subtree-to-processor* mapping [17] based on elimination tree.

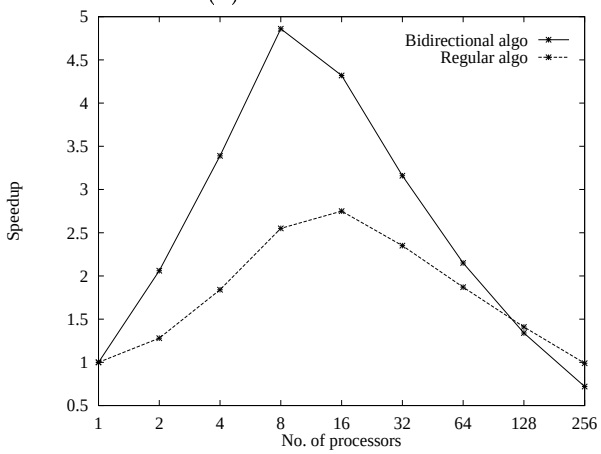
The parameters that were varied were the grid size k(16 and 32), the number of processors p(1 to 1024), the number of b-vectors for which solution vector x was obtained, and the C/E ratio i.e., the ratio of time for communicating a floating point data between two neighbouring processors to the time for a floating point operation(50 and 100). Figures 3.12, 3.13, 3.14, and 3.15 show the comparison of the measured speedups of the two schemes for various values of the above parameters.

As mentioned earlier in section 3.1, the first three phases, namely ordering, symbolic factorization, and numerical factorization, are executed only once and the substitution phase is repeatedly executed for each one of the different b-vectors.The output of the factorization phase of the bidirectional algorithm is a series of trapezoidal factor matrices whereas the output of the regular factorization algorithm is the pair of lower and upper triangular factor matrices. As a result, the inputs to the substitution phase of bidirectional and regular algorithms also differ. For separate comparison of the two phases of bidirectional and regular algorithms, we have considered a pseudo-speedup ratio for the bidirectional algorithm. This is a ratio of the time taken by the best sequential regular algorithm for the factorization (substitution) phase to the time taken by the parallel bidirectional algorithm for the factorization (substitution) phase.

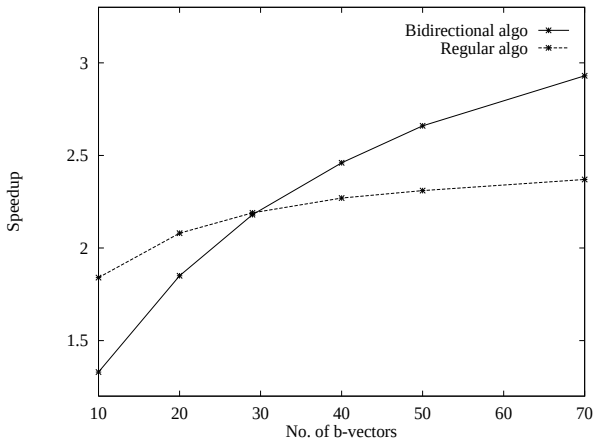
Therefore figures 3.12(a), 3.13(a), 3.14(a), and 3.15(a) compare the pseudo-speedup of the bidirectional algorithm with the speedup of the regular algorithm for the first



(a) factorization

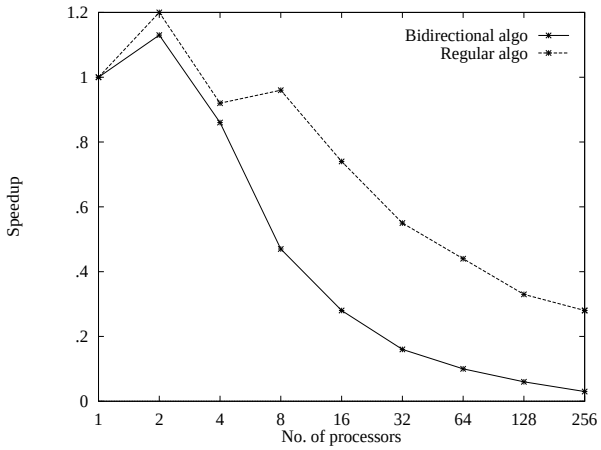


(b) substitution

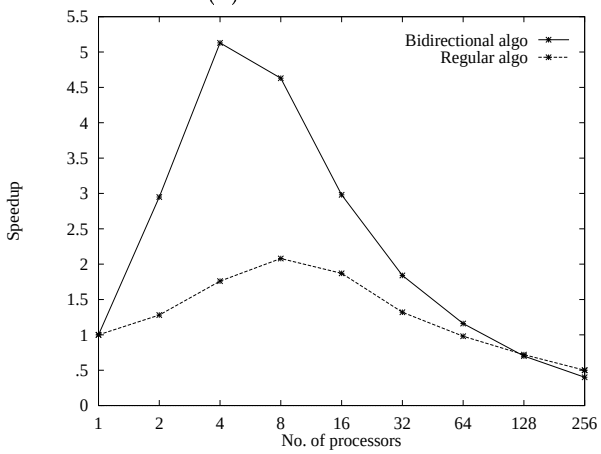


(c) solving multiple b-vectors with 8 processors

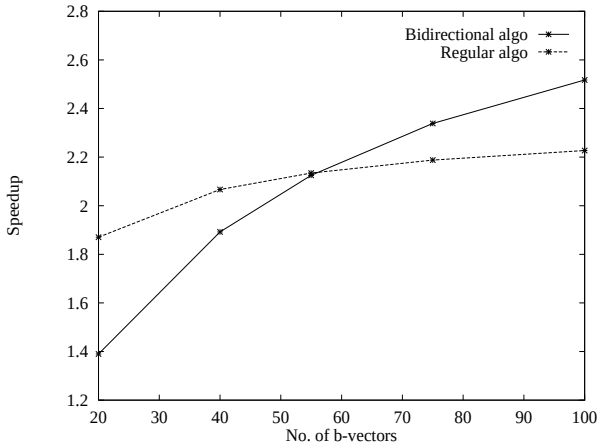
Figure 3.12: Speedups obtained for bidirectional algorithm versus regular algorithm for a 16×16 grid (i.e., $N = 256$) with $C/E = 50$



(a) factorization

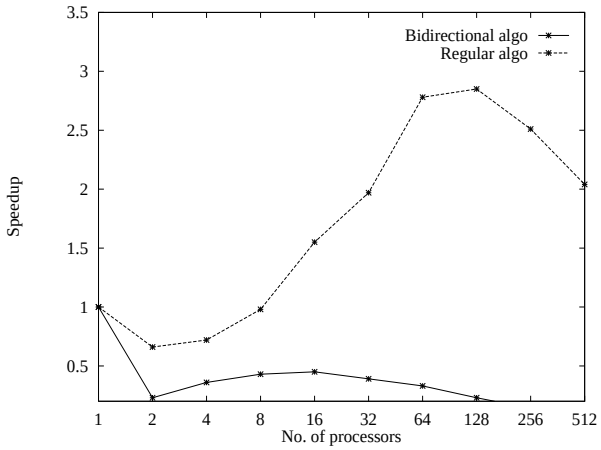


(b) substitution

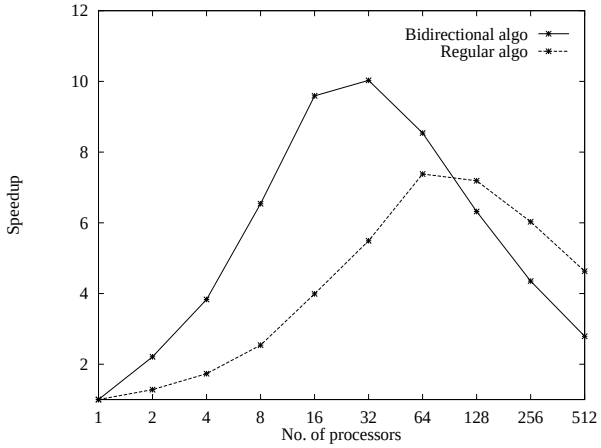


(c) solving multiple b-vectors with 8 processors

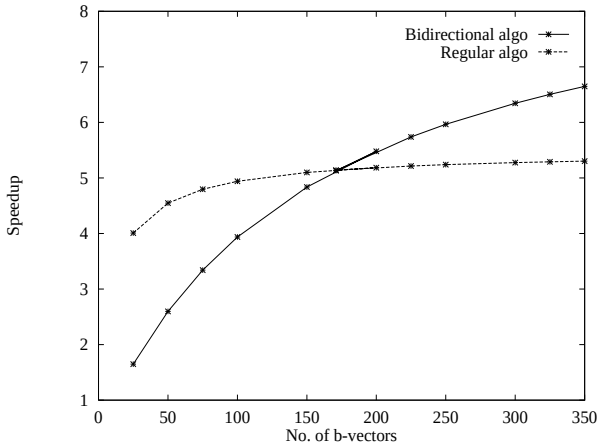
Figure 3.13: Speedups obtained for bidirectional algorithm versus regular algorithm for a 16×16 grid (i.e., $N = 256$) with $C/E = 100$



(a) factorization

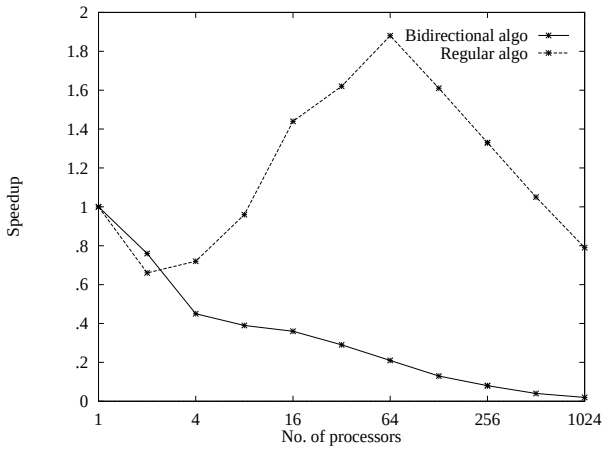


(b) substitution

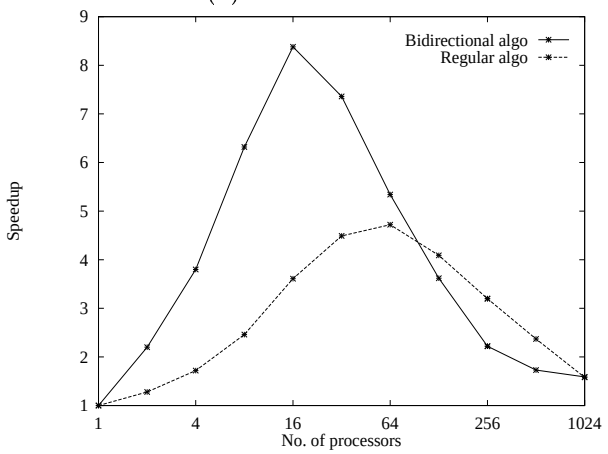


(c) solving multiple b-vectors with 8 processors

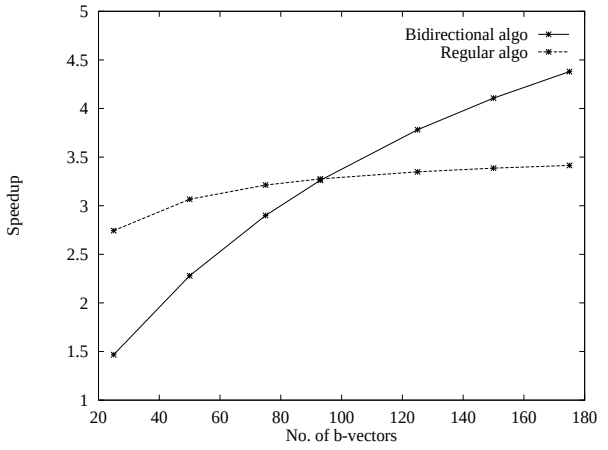
Figure 3.14: Speedups obtained for bidirectional algorithm versus regular algorithm for a 32×32 grid (i.e., $N = 1024$) with $C/E = 50$



(a) factorization



(b) substitution



(c) solving multiple b-vectors with 16 processors

Figure 3.15: Speedups obtained for bidirectional algorithm versus regular algorithm for a 32×32 grid (i.e., $N = 1024$) with $C/E = 100$

three phases put together. The figures 3.12(b), 3.13(b), 3.14(b), and 3.15(b) compare the pseudo-speedup of the bidirectional algorithm with the speedup of the regular algorithm for the substitution phase alone. Figures 3.12(c), 3.13(c), 3.14(c), and 3.15(c) plot the actual speedups of bidirectional and regular algorithms for all the four phases put together versus the number of b-vectors for which substitution phase is repeatedly executed. In figure 3.12(c), this comparison has been shown for the case when $p = 8$ and $k = 16$ (or $N = 256$) since, for $k = 16$, bidirectional factorization phase gives maximum speedup at $p = 8$. Similarly, in figure 3.13(c) $p = 8$ and $k = 16$, in figure 3.14(c) $p = 32$ and $k = 32$, and in figure 3.15(c) $p = 16$ and $k = 32$ (or $N = 1024$). These figures clearly indicate that with increasing number of b-vectors, the speedup obtained from our bidirectional scheme becomes higher than that obtained from the regular scheme. On increasing the problem size from $k = 16$ to 32, we observe that the magnitude of speedup obtained also increases. Increasing the C/E ratio causes a decrease in the magnitude of speedup obtained.

3.7 Conclusions

In this chapter, we have proposed a new bidirectional algorithm for direct solution of sparse symmetric system of linear equations. This scheme generates a series of trapezoidal factor matrices during the factorization phase due to which the substitution phase has only one forward substitution component and, unlike the regular substitution algorithms, it does not possess a back substitution component. Thus the bidirectional algorithm is well suited for situations where the system of equations has to be solved for multiple b-vectors. We have demonstrated the effectiveness of the bidirectional algorithm by comparing it with the regular methods for solving sparse symmetric systems. Further work is possible in the direction of increasing the amount of parallelism in the factorization and substitution phases of the bidirectional algorithm. In this work, we have considered a situation where computations on a particular column, say i, for both forward and backward factorizations are handled by the same processor. However, the computations for forward and backward factorizations are independent of each other (i.e., concurrent) at every stage s. Same is the case with the computations on a column i in substitution phase. This concurrency has not been

sufficiently exploited in the present work. In place of using p processors, we can use $2p$ processors, such that two processors are responsible for computations on each column - one handling computations related to forward factorization and the other related to backward factorization. Developing such a scheme is an open problem.

Chapter 4

A New Algorithm for Direct Solution of General Sparse Linear Systems

4.1 Introduction

In this chapter, we consider the problem of solving general sparse system of linear equations of the form $Ax = b$, where the coefficient matrix A has a general structure (i.e., A can be either symmetric or non-symmetric in nature), and is of dimension $N \times N$, and x and b are N-vectors. Such equations arise in various applications such as structural engineering, chemical engineering, fluid flow problems and nuclear physics. As with the sparse symmetric coefficient matrix case, the traditional process for obtaining direct solution of a general sparse system of linear equations, $Ax = b$, involves the following four distinct phases.

- *Ordering* : Apply an appropriate symmetric permutation matrix P such that the new system is of the form $(PAP^T)(Px) = (Pb)$.
- *Symbolic factorization* : Set up the appropriate data structures for the numerical factorization phase.
- *Numerical factorization* : Factorize the coefficient matrix A to the form $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix.
- *Substitution* : Determine the solution vector x by first solving the forward triangular system $Ly = b$ and then solving the backward triangular system $Ux = y$.

For solution of multiple b-vectors, the first three phases are carried out only once following which the substitution phase is repeated for each b-vector in order to obtain a different solution vector x in each case. Thus, in problems which involve solution of

multiple b-vectors, the time taken by repeated execution of substitution phase dominates the overall solution time. Although efficient parallel algorithms exist for the numerical factorization phase [5, 2, 44, 14, 11, 20, 30], not much progress has been made in the case of substitution phase [14, 22, 29] due to the limited amount of parallelism inherent in this phase.

In this chapter we present a new *bidirectional algorithm*, based on LU factorization, for the solution of general sparse system of linear equations. As in the sparse symmetric case, the numerical factorization phase is carried out in such a manner that the entire back substitution component of the substitution phase is replaced by a single step division. However, due to absence of symmetry, important differences arise in the ordering technique, the symbolic factorization phase, and message passing during numerical factorization phase. The bidirectional substitution phase for solving general sparse systems is the same as that for sparse symmetric systems (see section 3.3).

It is known that for sparse non-symmetric problems, pivoting is necessary to ensure numerical stability during numerical factorization phase. In this work, however, we consider the case where bidirectional factorization is done without pivoting so as to maintain clarity and concentrate more on other basic issues such as exploiting parallelism and reducing communication overheads. Existing work on bidirectional factorization algorithm based on LU factorization with partial pivoting for dense linear systems can be found in [42].

The rest of the chapter is organized as follows. In section 4.2, we present the bidirectional sparse factorization algorithm based on LU factorization for general sparse matrices. In section 4.3, we develop a bidirectional heuristic algorithm which produces a reordered coefficient matrix suitable for numerical factorization phase. In section 4.4, we look at a symbolic factorization algorithm which sets up data structures required by the numerical factorization phase. In section 4.5, we evaluate the performance of the bidirectional algorithm on hypercube multiprocessors and present comparison of our algorithm with the existing scheme based on sparse LU factorization. In section 4.6, we conclude the work with some observations about possible future improvements to the bidirectional scheme.

4.2 The Bidirectional Sparse Factorization (BSF) Algorithm

Unlike the regular LU factorization algorithm which factorizes A to the form $A = LU$, the BSF algorithm factorizes A into a series of *trapezoidal* matrices of multipliers. This series of trapezoidal matrices remove the need for the back substitution component in the substitution phase.

In this section, we first present an overall view of the concept of bidirectional factorization. We then proceed to describe the manner in which the sparsity of the coefficient matrix can be exploited to obtain higher degree of parallelism. Following this we present the details of implementing BSF algorithm on multiprocessor systems.

4.2.1 Bidirectional Factorization - The Concept

The basic concept behind the bidirectional factorization algorithm is the same as that presented in section 3.2.1. For $\log N$ stages, we repeatedly halve the size of sub-matrices through simultaneous factorizations in both forward and backward directions (generating lower and upper trapezoidal factor matrices in the process) and double the number of sub-matrices through copying at each stage. Finally, we end up with N sub-matrices of order 1×1 (see figure 3.1). Each pivot column operation during the forward and backward factorization is the same as in LU factorization. The substitution phase (described earlier in section 3.3) consists of moving the b-vector down the tree of trapezoids to produce N equations with one variable each, which are then solved by a single step division to produce the solution vector x (see figure 3.4).

4.2.2 Exploiting the Sparsity of the Coefficient Matrix A

In this section we look at the notion of *elimination tree* and consider as to how this notion abstracts the level of concurrency available during factorization process.

In regular sparse LU factorization, let F be the filled matrix obtained after factorizing the coefficient matrix A. An elimination tree contains a node corresponding to each column of the coefficient matrix. The parent of a node i is defined as

$$parent(i) = min\{j \mid j > i \text{ and } F[i,j] \neq 0\}.$$

The elimination tree defines a partially ordered precedence relation which determines when a certain column can be used as pivot.

Similarly, in BSF algorithm, we can define the notions of *forward elimination tree* and *backward elimination tree*. At some stage $s \in \{1 \cdots \log N\}$, let A_{x0} be a sub-matrix being factorized in the forward direction and A_{x1} be a sub-matrix being factorized in the backward direction (x being a possibly empty string of 0's and 1's). Let F_{x0} and F_{x1} be the respective filled sub-matrices generated at the end this factorization step. The *forward parent* of node i, is defined as

$$fparent(i, A_{x0}) = min\{j \mid j > i \text{ and } F_{x0}[i,j] \neq 0\}.$$

Similarly, the *backward parent* of node i, is defined as

$$bparent(i, A_{x1}) = max\{j \mid j < i \text{ and } F_{x1}[i,j] \neq 0\}.$$

For achieving a high degree of parallelism during factorization phase, both the forward and the backward elimination trees should be as short and wide as possible. This is the function of the ordering phase (described in section 4.3).

In the next subsection, we examine the parallel implementation of BSF algorithm on multiprocessors.

4.2.3 Implementing the BSF Algorithm on Multiprocessors

For our present study, we consider the *medium grain model* of parallelism in which tasks perform floating point operations over nonzero elements of entire columns of coefficient matrix. The following elementary tasks are considered for the BSF algorithm.

- *fdivide(i,s)* divides by $A_{x0}[i,i]$, every nonzero element of the sub-diagonal part of the ith column of sub-matrix A_{x0}.
- *bdivide(i,s)* divides by $A_{x1}[i,i]$, every nonzero element of the super-diagonal part of the ith column of sub-matrix A_{x1}.
- *fmodify(i,vector$_j$,s)* subtracts an appropriate multiple of *vector$_j$* from the ith column of a sub-matrix A_{x0}, at stage $s \in \{1 \cdots \log N\}$. *vector$_j$* contains the

contents of some column j of A_{x0}, which modifies column i directly in forward direction at stage s.

- *bmodify(i,vector$_j$,s)* subtracts an appropriate multiple of *vector$_j$* from the ith column of a sub-matrix A_{x1}, at stage $s \in \{1 \cdots \log N\}$. *vector$_j$* contains the contents of some column j of A_{x1}, which modifies column i directly in backward direction at stage s.

To keep track of the columns that each pivot should modify at each of the $\log N$ stages, we maintain the following data structures.

- $F_i^{(s)}$ denotes the set of all columns with indices smaller than i that modify the ith column in the forward direction at stage s.
- $B_i^{(s)}$ denotes the set of all columns with indices greater than i that modify the ith column in the backward direction at stage s.

These data structures are generated during the symbolic factorization phase. This phase is described in section 4.4. In the remaining part of this section, we describe the implementation of BSF algorithm on a message passing multiprocessor for the case where the number of processors p is less than or equal to the order N of the coefficient matrix.

In parallel fan-in BSCF algorithm (described in section 3.2), the symmetric nature of coefficient matrix is exploited to reduce the communication overheads. Multiples of various columns located in the same processor, which modify a particular column j located in some other processor, are added into a single message vector which is then sent over to the destination processor. In parallel BSF algorithm, on the other hand, the absence of symmetry in the coefficient matrix does not permit such an optimization. Thus for every column i, which modifies column j in the forward (backward) direction (i.e., i belongs to the set $F_j^{(s)}$ ($B_j^{(s)}$)), a separate message vector containing column i is sent to the processor storing column j.

In algorithm 1 below, we incorporate the above idea in the BSF algorithm and present the *fan-out* BSF algorithm. The set $List_{myid}$ is the set of columns stored in processor P_{myid}. If column i is to modify column j in forward direction at stage s then,

after performing $fdivide(i,s)$ operation, the processor which stores the column i, sends a message containing the contents of column i to the processor storing the column j. Similar mechanism operates for factorization in backward direction.

Algorithm 1 (*The parallel fan-out BSF algorithm for case $p \leq N$*)

begin

for $s := 1$ **to** $\log N$ **do**

parbegin

Forward_factorize($List_{myid}$,s);

Backward_factorize($List_{myid}$,s);

parend

end

procedure Forward_factorize($List$,s)

begin

while $List \neq \phi$ **do**

if $\exists i \in List$ such that $fvector_j$ has been received for all $j \in F_i^{(s)}$ **then**

Let column i belong to the forward sub-matrix A_{x0} at stage s;

for $k := 0$ to $i-1$ **do**

if $k \in F_i^{(s)}$ **then** $fmodify(i, fvector_j, s)$;

if column i belongs to the first half of sub-matrix A_{x0} **then**

$fdivide(i,s)$;

for all j such that $i \in F_j^{(s)}$ **do**

$fvector_i := A_{x0}[*,i]$;

send a message of the form (j,$fvector_i$,s)

to processor storing column j;

else if $s < \log N$ **then**

(*copy column i of A_{x0} to column i of A_{x00} and A_{x01}*)

$A_{x00}[*,i] := A_{x0}[*,i]$;

$A_{x01}[*,i] := A_{x0}[*,i]$;

$List := List - i$;

if there is an incoming message **then** receive and store the message;

end

procedure Backward_factorize($List$,s)
begin
  while $List \neq \phi$ **do**
    if $\exists i \in List$ such that $bvector_j$ has been received for all $j \in B_i^{(s)}$ **then**
      Let column i belong to the backward sub-matrix A_{x1} at stage s;
      for $k := N-1$ **downto** $i+1$ **do**
        if $k \in B_i^{(s)}$ **then** $bmodify(i, bvector_j, s)$;
      if column i belongs to the second half of sub-matrix A_{x1} **then**
        $bdivide(i, s)$;
        for all j such that $i \in B_j^{(s)}$ **do**
          $bvector_i := A_{x1}[*, i]$;
          send a message of the form (j,$bvector_i$,s)
            to processor storing column j;
      else if $s < \log N$ **then**
        (*copy column i of A_{x1} to column i of A_{x10} and A_{x11}*)
        $A_{x10}[*, i] := A_{x1}[*, i]$;
        $A_{x11}[*, i] := A_{x1}[*, i]$;
      $List := List - i$;
      if there is an incoming message **then** receive and store the message;
end

As noted in section 3.2.3, a special situation arises when the number of processors $p = 2^d$ (as in hypercube multiprocessors) and $N = 2^n$ ($n, d \in \mathcal{N}$). Assume that we map the equations on the processors in a block wrap manner (as shown in figure 3.3). Thus each processor holds $\frac{N}{p} = 2^{n-d}$ consecutive equations. At the end of $d = \log p$ stages of the fan-out BSF algorithm, each processor contains an independent system of $\frac{N}{p}$ equations. This independent system can be factorized within a single processor without any communication with any other processor. Since, on a single processor, regular sequential sparse LU factorization performs more efficiently than the fan-out

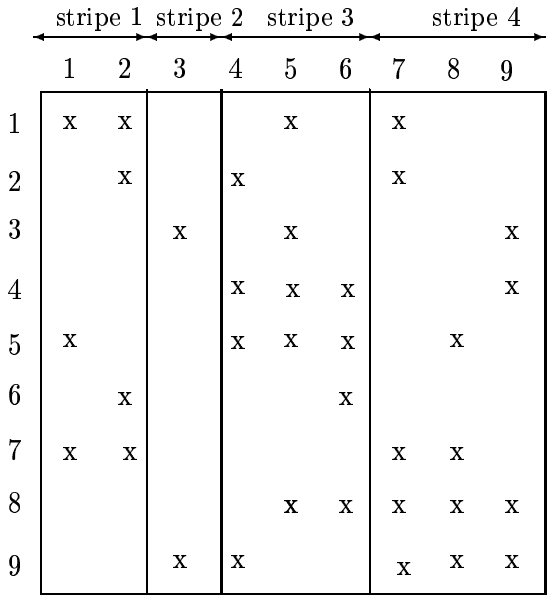
BSF algorithm, we can switch over to this regular sequential version after $\log p$ stages and factorize the coefficient matrix (say A_{ind}) of this independent system into the form $A_{ind} = L_{ind}U_{ind}$. This results in enhancing the performance of the fan-out BSF algorithm.

4.3 Ordering the General Sparse Matrix for Bidirectional Factorization

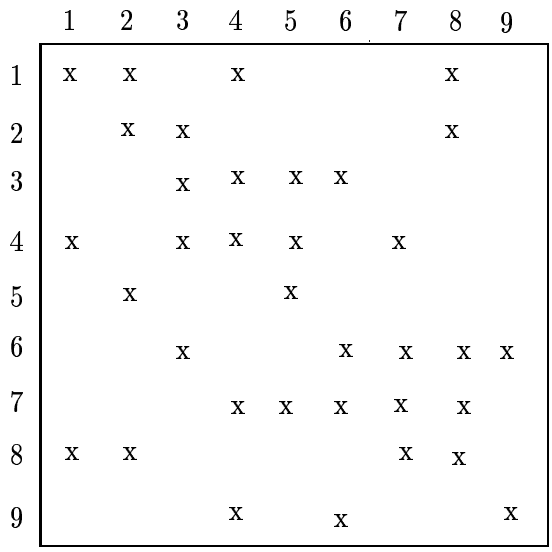
As noted earlier, the basic aim of the ordering phase is to reorder the columns of the coefficient matrix in such a manner that during the factorization phase, the amount of fill-in is minimized and the degree of parallelism is maximized. The principal ordering technique used for reordering the general sparse matrices for regular LU factorization algorithms involves two stages. In the first stage, a fill reducing ordering, such as minimum degree ordering [12], is applied to the coefficient matrix A. This is followed by application of Liu's scheme of elimination tree rotation [38, 39] which causes a reduction in the height of the elimination tree without affecting the amount of fill-in in the upper triangular factor U. The resulting elimination tree is more appropriate for parallel LU factorization.

The ordering resulting from the above scheme is, however, not suited for the BSF algorithm due to reasons given below. Recall that in section 4.2.2 we defined the concepts of forward elimination tree and backward elimination tree for the BSF algorithm. The degree of parallelism while factorizing in forward direction depends on the shape of the forward elimination tree and that for factorizing in backward direction depends on the shape of the backward elimination tree. An ideal ordering for the BSF algorithm is one in which both the elimination trees are as short and wide as possible. The forward elimination tree obtained from the above scheme is short and wide and hence desirable for parallel factorization. On the other hand the backward elimination tree obtained from the above scheme is lean and tall and hence undesirable for parallel factorization.

In the remaining part of this section we describe how the above scheme can be extended to yield ordering suitable for the BSF algorithm. We call the new heuristic as the *alternate stripe reordering method* and it proceeds as follows. First we apply a fill reducing ordering, such as the minimum degree ordering, followed by Liu's height reducing elimination tree rotation scheme to obtain a reordered matrix whose forward



(a) 9 x 9 striped sparse matrix



(b) 9 x 9 alternate stripe reordered matrix

Figure 4.1: Ordering of a 9×9 matrix using alternate stripe reordering.

elimination tree has low height. Let the reordered matrix be A'. The following steps of alternate stripe reordering method are applied to the matrix A'.

- *Step 1* : Stripe the matrix A' into groups of columns as shown in figure 4.1. The grouping of columns into stripes is done according to the following criteria. Column i and column $i+1$ belong to the same stripe if $A'[i, i+1] \neq 0$. Otherwise, column i and column $i+1$ belong to consecutive stripes.
- *Step 2* : Initialize $upCount$ to 1 and $downCount$ to N. Maintain an array $newOrder$ of size N to store the new ordering.
- *Step 3* :

  For each successive column i of stripe 1 **do**

  - $newOrder[i] := upCount$;
  - $upCount = upCount + 1$;

  For each successive column i' of stripe 2 **do**

- $newOrder[i'] := downCount$;
- $downCount = downCount - 1$;

- *Step 3* : The above numbering method is repeated for each successive pair of stripes i.e., columns belonging to odd stripes are numbered by incrementing $upCount$ and columns belonging to even stripes are numbered by decrementing $downCount$.
- *Step 4* : The row i and column i of matrix A' are numbered as row $newOrder[i]$ and column $newOrder[i]$ in the final reordered matrix.

A little thought reveals that the alternate stripe reordering method is a generalization of the bidirectional nested dissection method described in section 3.4. The latter method can be alternatively viewed as consisting of two stages - (i) applying the regular nested dissection method to the $k \times k$ grid followed by (ii) applying alternate stripe reordering to the matrix obtained from the first stage. It will be shown through experimental results at the end of this chapter that the new reordering scheme does indeed yield reorderings better suited to parallel bidirectional factorization than the scheme based on fill-reduction and elimination tree rotations alone.

In the next section we look at the bidirectional symbolic factorization algorithm which allocates memory and sets up the appropriate data structures prior to the BSF algorithm.

4.4 The Bidirectional Symbolic Factorization Algorithm

The bidirectional symbolic factorization algorithm, which precedes the BSF phase, does the following.

- It determines apriori, the structure of each one of the filled sub-matrices, F_x, at each of the $\log N$ stages and
- It initializes the data structures for the sets $F_i^{(s)}$ and $B_i^{(s)}$ which are required during the BSF algorithm.

We define $Colstruct(A_{x0}, i)$ to denote the set of row indices of nonzeros in the column i of forward matrix A_{x0}.

$$Colstruct(A_{x0}, i) = \{j \mid A_{x0}[j, i] \neq 0\}.$$

In a similar fashion, we define $Colstruct'(A_{x1}, i)$ to denote the set of row indices of nonzeros in the column i of the backward matrix A_{x1}.

$$Colstruct'(A_{x1}, i) = \{j \mid A_{x1}[j, i] \neq 0\}.$$

We now describe the bidirectional symbolic factorization algorithm.

Algorithm 2 (*The bidirectional symbolic factorization algorithm*)

begin

for $s := 1$ **to** $\log N$ **do**

for $col := 1$ **to** N **do**

$F_{col}^{(s)} := \phi; B_{col}^{(s)} := \phi;$

for $s := 1$ **to** $\log N$ **do**

for $col := 1$ **to** N **do**

Forward_SF(col,s);

for $col := N$ **downto** 1 **do**

Backward_SF(col,s);

end

procedure Forward_SF(col,s)

begin

Let A_{x0} be the forward sub-matrix that contains column col at stage s;

if col belongs to the first half of A_{x0} **then**

Calculate $fparent(col, A_{x0})$ using definition given in section 4.2.2;

if $fparent(col, A_{x0})$ belongs to the first half of A_{x0} **then**

$Colstruct(A_{x0}, fparent(col, A_{x0}))$:=

$Colstruct(A_{x0}, fparent(col, A_{x0}) \cup Colstruct(A_{x0}, col);$

for all j such that j belongs to second half of A_{x0} and $A_{x0}[col, j] \neq 0$ **do**

$Colstruct(A_{x0}, j) := Colstruct(A_{x0}, j) \cup Colstruct(A_{x0}, col);$

for all j such that $j \in Colstruct(A_{x0}, col)$ and $j < col$ **do**

$F_{col}^{(s)} := F_{col}^{(s)} \cup \{i\}$;

else

$Colstruct(A_{x00}, col) := Colstruct(A_{x0}, col)$;

$Colstruct'(A_{x01}, col) := Colstruct(A_{x0}, col)$;

end

procedure Backward_SF(col,s)

begin

Let A_{x1} be the backward sub-matrix that contains column col at stage s;

if col belongs to the second half of A_{x1} **then**

Calculate $bparent(col, A_{x1})$ using definition given in section 4.2.2;

if $bparent(col, A_{x1})$ belongs to the second half of A_{x1} **then**

$Colstruct'(A_{x1}, fparent(col, A_{x1}))$:=

$Colstruct'(A_{x1}, fparent(col, A_{x1}) \cup Colstruct'(A_{x1}, col)$;

for all j such that j belongs to first half of A_{x1} and $A_{x1}[col, j] \neq 0$ **do**

$Colstruct'(A_{x1}, j) := Colstruct'(A_{x1}, j) \cup Colstruct'(A_{x1}, col)$;

for all j such that $j \in Colstruct'(A_{x1}, col)$ and $j > col$ **do**

$B_j^{(s)} := B_j^{(s)} \cup \{col\}$;

else

$Colstruct(A_{x10}, col) := Colstruct'(A_{x1}, col)$;

$Colstruct'(A_{x11}, col) := Colstruct'(A_{x1}, col)$;

end

The bidirectional symbolic factorization algorithm described above has time complexity proportional to the number of nonzero elements stored in trapezoids at each stage.

4.5 Experimental Results and Performance Analysis

To evaluate the performance of the entire bidirectional scheme presented in this work, we implemented a hypercube simulator in C language and compared the *speedups*

obtained from the bidirectional scheme with those obtained from the regular scheme. We used the SPARC Classic machine to carry out our simulations.

In the bidirectional scheme, we implemented each of the four phases as follows.

- *Ordering* : The alternate stripe reordering method described in section 4.3.
- *Symbolic factorization* : The sequential bidirectional symbolic factorization algorithm described in section 4.4.
- *Numerical factorization* : The parallel fan-out BSF algorithm described in section 4.2.
- *Substitution* :The parallel BS algorithm described in section 3.3.

In the regular scheme, we implemented each of the four phases as follows.

- *Ordering* : The fill reducing minimum degree ordering [12] followed by Liu's elimination tree rotation scheme [38].
- *Symbolic factorization* : The sequential symbolic factorization algorithm presented in [16].
- *Numerical factorization* : The parallel fan-out algorithm given in [4, 30].
- *Substitution* :The elimination tree based forward and back substitution algorithms given in [29].

Mapping of columns onto processors is an important issue. For the bidirectional scheme, we have used the *block wrap around mapping* using gray code whereas for the regular algorithm we have used the *subtree-to-processor* mapping [17] based on elimination tree.

For the purpose of simulation we used three test matrices, described in table 4.1, from the Harwell-Boeing Collection. Due to memory constraints, the maximum dimension of the test matrix considered was 343×343. The parameters that were varied were the number of processors p (1 to 128), the number of b-vectors for which solution vector x was obtained, and the C/E ratio i.e., the ratio of time for communicating a floating point data between two neighbouring processors to the time for a floating

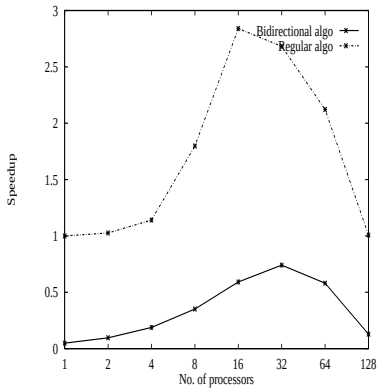
Table 4.1: Matrices from Harwell-Boeing collection

Number of equations	Number of nonzeros in A	Description
199	701	WILL199 : pattern of stress analysis matrix.
216	876	GRE216A : unsymmetric matrix from Grenoble.
343	1435	GRE343 : unsymmetric matrix from Grenoble.

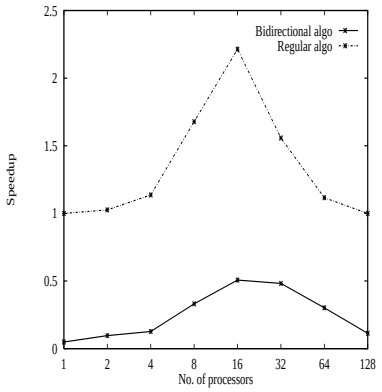
point operation (50 and 100). Figures 4.2, 4.3, and 4.4 show the comparison of the measured speedups of the two schemes for various values of the above parameters.

As mentioned earlier in section 4.1, the first three phases, namely ordering, symbolic factorization, and numerical factorization, are executed only once and the substitution phase is repeatedly executed for each one of the different b-vectors. The output of the factorization phase of the bidirectional algorithm is a series of trapezoidal factor matrices whereas the output of the regular factorization algorithm is the pair of lower and upper triangular factor matrices. As a result, the inputs to the substitution phase of bidirectional and regular algorithms also differ. For separate comparison of the two phases of bidirectional and regular algorithms, we have considered a pseudo-speedup ratio for the bidirectional algorithm. This is a ratio of the time taken by the best sequential regular algorithm for the factorization (substitution) phase to the time taken by the parallel bidirectional algorithm for the factorization (substitution) phase.

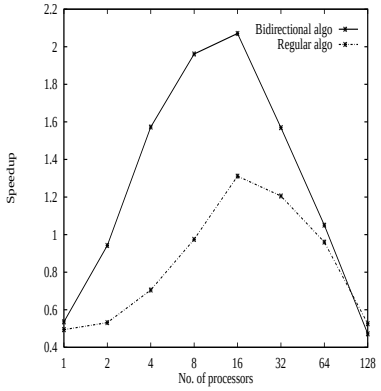
Therefore figures 4.2(a), 4.2(d), 4.3(a), 4.3(d), 4.4(a), and 4.4(d) compare the pseudo-speedup of the bidirectional algorithm with the speedup of the regular algorithm for the first three phases put together. The figures 4.2(b), 4.2(e), 4.3(b), 4.3(e), 4.4(b), and 4.4(e) compare the pseudo-speedup of the bidirectional algorithm with the speedup of the regular algorithm for the substitution phase alone. The figures 4.2(c), 4.2(f), 4.3(c), 4.3(f), 4.4(c), and 4.4(f) plot the actual speedups of bidirectional and regular algorithms for all the four phases put together versus the number of b-vectors for which substitution phase is repeatedly executed. In figure 4.2(c), this comparison has been shown for the case when $p = 16$, $N = 199$, and $C/E = 50$ since, for this combination of parameters, bidirectional factorization phase gives maximum speedup at $p = 16$. Same logic holds for figures 4.2(f), 4.3(c), 4.3(f), 4.4(c), and 4.4(f). These



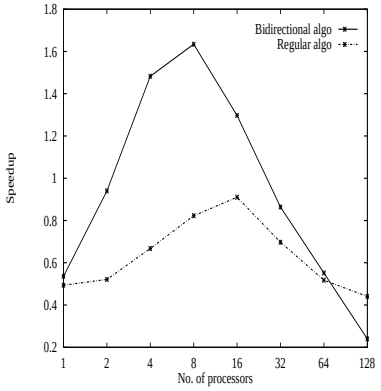
(a) factorization, C/E=50



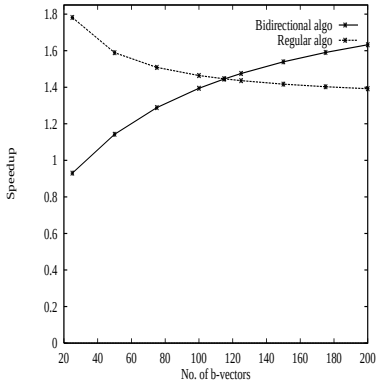
(d) factorization, C/E=100



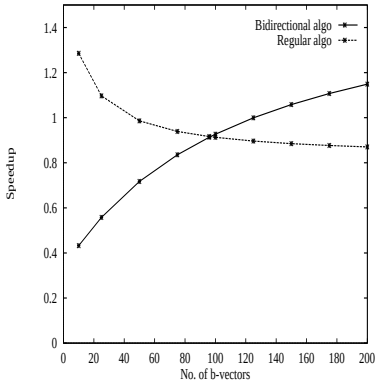
(b) substitution, C/E=50



(e) substitution, C/E=100

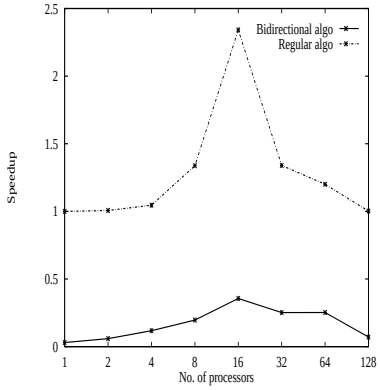


(c) solving multiple b-vectors with 16 processors, C/E=50

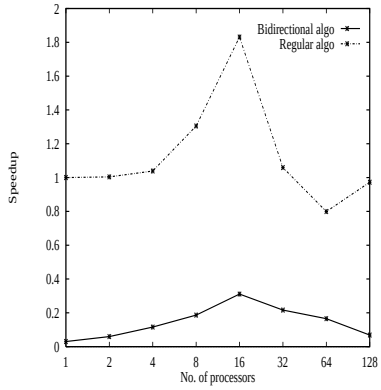


(f) solving multiple b-vectors with 8 processors, C/E=100

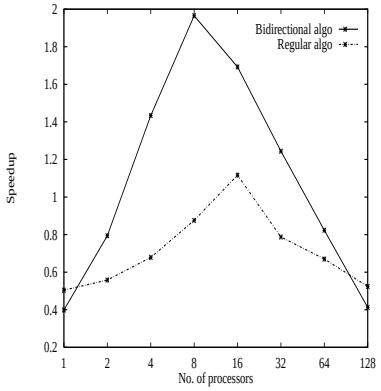
Figure 4.2: Speedups obtained for bidirectional algorithm versus regular algorithm for WILL199.



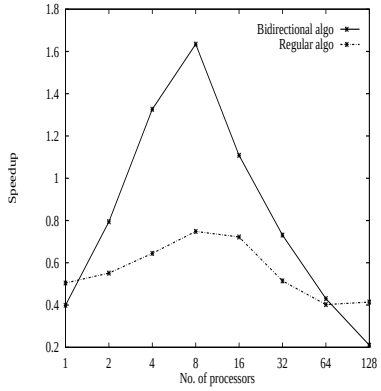
(a) factorization, C/E=50



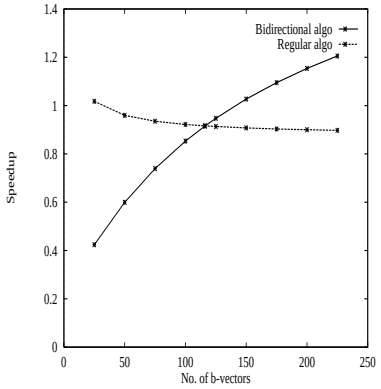
(d) factorization, C/E=100



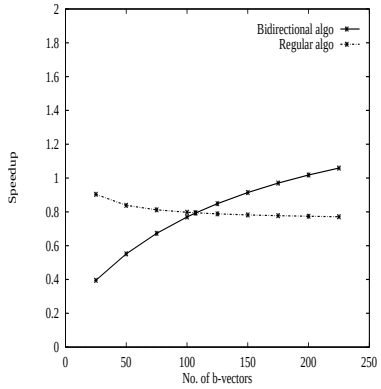
(b) substitution, C/E=50



(e) substitution, C/E=100

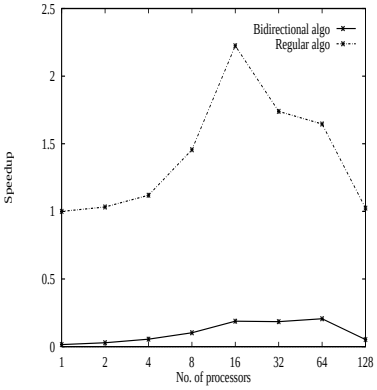


(c) solving multiple b-vectors with 8 processors, C/E=50

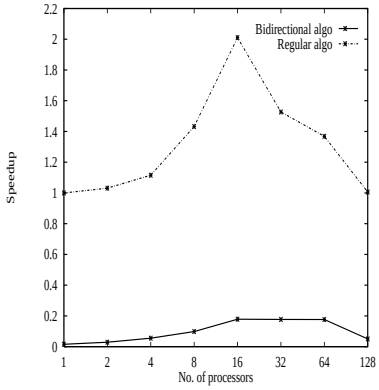


(f) solving multiple b-vectors with 8 processors, C/E=100

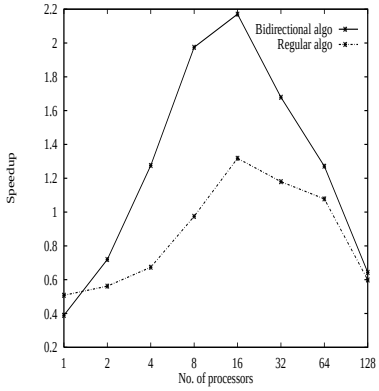
Figure 4.3: Speedups obtained for bidirectional algorithm versus regular algorithm for GRE216A.



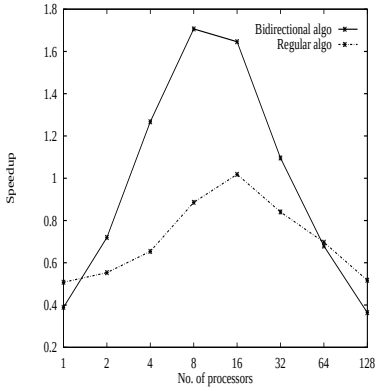
(a) factorization, C/E=50



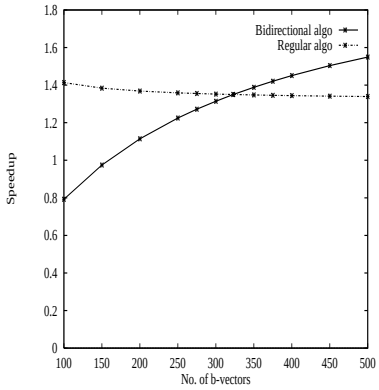
(d) factorization, C/E=100



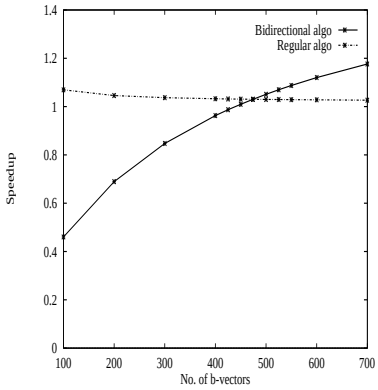
(b) substitution, C/E=50



(e) substitution, C/E=100

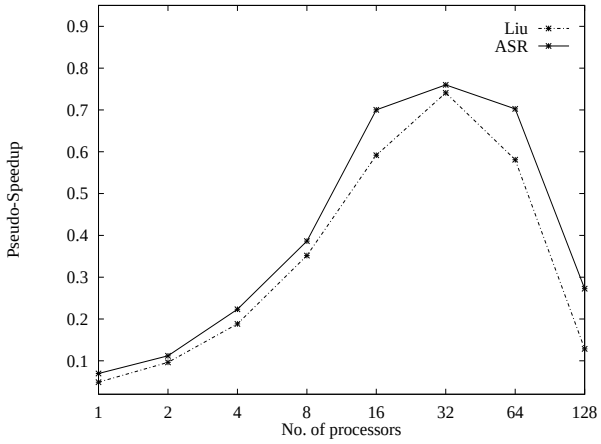


(c) solving multiple b-vectors with 16 processors, C/E=50

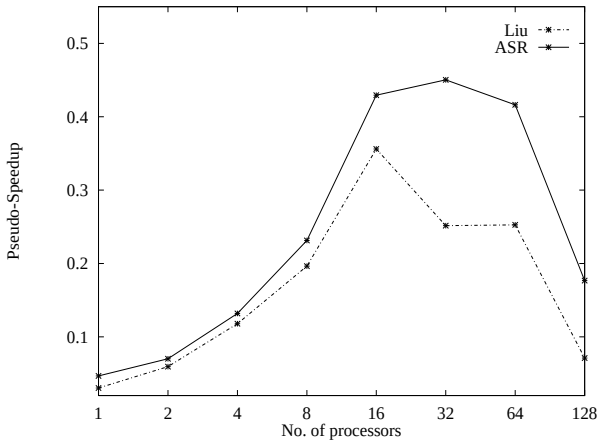


(f) solving multiple b-vectors with 8 processors, C/E=100

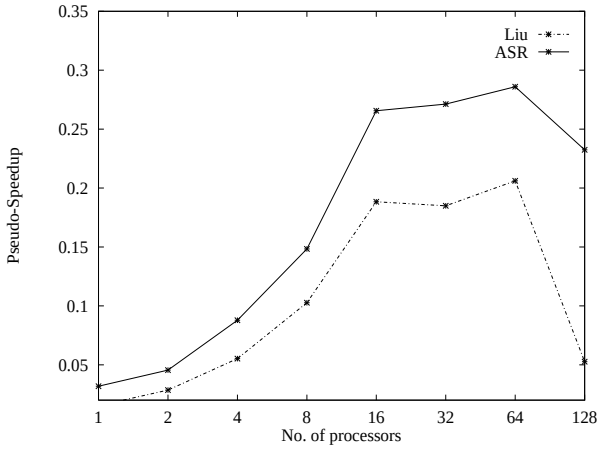
Figure 4.4: Speedups obtained for bidirectional algorithm versus regular algorithm for GRE343.



(a) WILL199 matrix



(b) GRE216A matrix



(c) GRE343 matrix

Figure 4.5: Pseudo-speedups obtained for bidirectional factorization with matrices reordered by ASR method versus those reordered by Liu's rotation method. $C/E = 50$.

figures clearly indicate that with increasing number of b-vectors, the speedup obtained from our bidirectional scheme steadily becomes higher than that obtained from the regular scheme. Increasing the C/E ratio causes a decrease in the magnitude of speedup obtained.

Figures 4.5(a), (b), and (c) compare the pseudo-speedup of the bidirectional factorization phase with two different reorderings of each of the coefficient matrices - one obtained using the ASR heuristic proposed in section 4.3 and the other obtained using Liu's scheme [38]. The graphs clearly indicate that BSF algorithm gives higher speedup when the coefficient matrix is reordered using the ASR heuristic rather than with Liu's scheme.

4.6 Conclusions

In this chapter, we have proposed a new bidirectional algorithm for direct solution of general sparse system of linear equations. This scheme generates a series of trapezoidal factor matrices during the factorization phase due to which the substitution phase has only one forward substitution component. Unlike the regular substitution algorithms, it does not possess a back substitution component in the substitution phase. Thus the bidirectional algorithm is well suited for situations where the system of equations has to be solved for multiple b-vectors. We have demonstrated the effectiveness of the bidirectional algorithm by comparing it with the regular methods for solving general sparse systems. Further work is possible in the direction of incorporating partial pivoting in the present parallel bidirectional scheme. This will call for modification of the bidirectional symbolic factorization method since, the structure of the filled sub-matrices at each stage of factorization will depend not only on the structure of coefficient matrix A, but also on the row interchanges that occur due to partial pivoting. Also, as in the sparse symmetric case, the amount of parallelism can be increased by using $2p$ processors, instead of p processors, for handling the forward and backward operations on separate processors.

Chapter 5

Conclusions

In this thesis, we have addressed the problem of solving three important classes of sparse linear systems - (i) block tridiagonal linear systems, (ii) sparse symmetric linear systems, and (iii) general sparse linear systems. In the first class, we have proposed an improved mapping of *cyclic elimination* (CE) algorithm onto hypercube multiprocessors which achieves desirable mapping through judicious use of the concept of data replication. For the second and third classes of problems, we have proposed new *bidirectional algorithms* which, due to the absence of back-substitution component in the substitution phase, are very well suited for solving multiple b-vector systems. Most of the existing parallel algorithms for solving sparse linear systems attempt to parallelize their good sequential counterparts. This approach has not borne fruit, since the basic goal of a good sequential algorithm i.e., minimizing the total operation count, conflicts with the basic goal of a good parallel algorithm, which is maximizing the number of concurrent sub-problems. By exploiting the higher degree of parallelism available in the problem itself, the new algorithms proposed in our work achieve better performance than the traditional algorithms.

5.1 Summary

In chapter 2, we have proposed an improved mapping of the cyclic elimination algorithm for the solution of the block-tridiagonal linear systems onto hypercube multiprocessors. Unlike the previous mapping schemes, our improved mapping uses the concept of data replication to achieve only neighbouring processor communication at all stages of processing. Our improved mapping scheme is shown to be effective by comparing it with the existing mapping of the *cyclic reduction* (CR) algorithm onto hypercubes using both analytical and simulation methods. The comparison shows that as the number of

processors increases, our improved mapping steadily overtakes the existing mapping of the CR algorithm in terms of speedup. Two significant features of our algorithm are that, the computational load is balanced among all processors at all stages of the algorithm and secondly, much of the communication gets overlapped with the computation giving an overall better performance.

In chapter 3, we have proposed a new *bidirectional* algorithm for the direct solution of sparse symmetric system of linear equations. This scheme generates a series of trapezoidal factor matrices during the factorization phase due to which the substitution phase has only one forward substitution component and, unlike the regular substitution algorithms, it does not possess a back-substitution component. For the numerical factorization phase, we have proposed a *fan-in bidirectional sparse Cholesky factorization* (BSCF) algorithm. For the substitution phase, we have proposed a *bidirectional substitution* algorithm in which the b-vector gets modified by the tree of trapezoids produced during the factorization phase. For the ordering phase, we have proposed a *bidirectional nested dissection* algorithm which produces orderings suited to parallel factorization using BSCF algorithm. Further, we have developed *bidirectional symbolic factorization algorithm* which sets up the appropriate data structures required during the BSCF algorithm.

In chapter 4, we have addressed the problem of solving general sparse linear systems using the bidirectional scheme. For the factorization phase, we have developed a *fan-out bidirectional sparse factorization* (BSF) algorithm based on LU factorization. The bidirectional algorithm for the substitution phase is the same as that for the sparse symmetric case. In the ordering phase, we have proposed an *alternate stripes reordering* algorithm which produces orderings suited to parallel factorization using BSF algorithm. We have also developed a *bidirectional symbolic factorization algorithm* for setting up the appropriate data structures required during the BSF algorithm.

In order to demonstrate the effectiveness of the two bidirectional schemes presented in chapters 3 and 4, we have conducted extensive simulation studies on the performance of these algorithms on hypercube multiprocessors. We have compared the speedups obtained from the entire bidirectional scheme for solving the sparse symmetric linear systems with those obtained from the regular Cholesky factorization based schemes.

Similarly, we have compared the speedups obtained from the entire bidirectional scheme for solving the general sparse linear systems with those obtained from the regular LU factorization based schemes. The results indicate that, when solving for multiple b-vectors, the speedups obtained from the bidirectional schemes steadily overtake those obtained from the regular schemes, as the number of b-vectors for which the system is solved increases.

5.2 Suggestions for Future Work

Further work can be done in the following directions.

- In chapter 1, the degree of parallelism in the improved mapping of cyclic elimination algorithm onto hypercube multiprocessors can be controlled by switching over to the sequential algorithm for solving block-tridiagonal systems at a stage earlier than $\log N$. Determining the optimal stage k, at which this switching should occur is an open problem.
- In the bidirectional algorithms for solving sparse linear systems in chapters 3 and 4, further concurrency can be exploited by assigning the computation of forward and backward factorization phases to separate processors. This will mean using twice the number of processors currently being considered.
- In chapter 4, the bidirectional algorithms presented for solving general sparse linear systems can be modified to include pivoting which is widely considered to be crucial for ensuring the stability.

Bibliography

[1] J.C.Agui and J.Jimenez, *A binary tree implementation of a parallel distributed tridiagonal solver*, Parallel Computing, Vol. 21, No. 2, 1995, pp. 233-241.

[2] G.Alaghband and H.Jordan, *Multiprocessor sparse L/U decomposition with controlled fill-in*, Technical Report 85-48, ICASE, NASA Langey Research Center, Hampton, VA 1985.

[3] P.Amodio, *Optimised cyclic reduction for the solution of linear tridiagonal systems on parallel computers*, Computers and Mathematics with Applications, Vol. 26, No. 3, 1993, pp. 45-53.

[4] C.Ashcraft, S.C.Eisenstat and J.W.H.Liu, *A fan-in algorithm for distributed sparse numerical factorization*, SIAM J. Sci. Stat. Comput., Vol. 11, No. 3, 1990, pp. 593-599.

[5] C.Ashcraft, S.C.Eisenstat, J.W.H.Liu, and A.H.Sherman, *A comparison of three column based distributed sparse factorization schemes*, Technical Report YALEU/DCS/RR-810, Yale University, New haven, CT, 1990.

[6] D.P.Bertsekas and J.N.Tsitsiklis, *Parallel and Distributed Computation - Numerical Methods*, Prentice-Hall, Engelwood Cliffs, New Jersy, 1989.

[7] B.L.Buzbee, G.H.Golub and C.W.Nielson, *On direct methods for solving Poisson's equations*, SIAM J. Numer. Anal., Vol. 7, 1970, pp. 627-655.

[8] J.M.Conroy, *Parallel nested dissection*, Parallel Computing, Vol. 16, 1990, pp. 139-156.

[9] I.S.Duff and J.K.Reid, *Multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Soft., Vol. 9, May 1983, pp. 302-325.

[10] A.George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., Vol. 10, No. 2, 1973, pp. 345-363.

[11] A.George and E.Ng, *Parallel sparse Gaussian elimination with partial pivoting*, Annals of Operations Research, Vol. 22, 1990, pp. 219-240.

[12] A.George and J.W.H.Liu, *The evolution of minimum degree ordering algorithm*, SIAM Review, Vol. 31, No. 1, 1989, pp. 1-19.

[13] A.George, M.T.Heath, J.W.H.Liu and E.Ng, *Computer Solution of Large Sparse Positive Definite Systems* Prentice Hall, Englewood Cliffs, NJ, 1981.

[14] A.George, M.T.Heath, J.W.H.Liu and E.Ng, *Solution of sparse positive definite systems on hypercube*, J. Comput. Applied Math., Vol. 27, 1989, pp. 129-156.

[15] A.George, M.T.Heath, J.W.H.Liu and E.Ng, *Sparse Cholesky factorization on a local memory multiprocessor*, SIAM J. Sci. Stat. Comput., Vol. 9, No. 2, 1988, pp. 327-340.

[16] A.George, M.T.Heath, E.Ng and J.W.H.Liu, *Symbolic Cholesky factorization on local memory multiprocessor*, Parallel Computing, Vol. 5, 1987, pp. 85-95.

[17] A.George, J.W.H.Liu and E.Ng, *Communication results for parallel sparse Cholesky factorization on hypercube*, Parallel Computing, Vol. 10, No. 3, 1989, pp. 287-298.

[18] J.R.Gilbert and H.Hafsteinsson, *Parallel symbolic factorization for sparse linear systems*, Parallel Computing, Vol. 14, 1990, pp. 151-162.

[19] G.H.Golub, and C.F.V.Loan, *Matrix Computations : Second Edition*, John Hopkins University Press, Baltimore, MD, 1989.

[20] M.T.Heath, E.Ng and B.W.Peyton, *Parallel algorithms for sparse linear systems*, SIAM Review, Vol. 33, 1991, pp. 420-460.

[21] D.Heller, *A survey of parallel algorithms in numerical linear algebra*, SIAM Review, Vol. 20, No. 4, Oct. 1978, pp. 740-777.

[22] C.W.Ho, *Fast Parallel Algorithms Related to Chordal Graphs*, Ph.D. Thesis, Institute of Computer and Decision Sciences, National Tsing Hua University, Hsinchu, Taiwan, Republic of China, 1988.

[23] C.T.Ho and S.L.Johnsson, *Optimizing tridiagonal solvers for alternating direction methods on boolean cube multiprocessors*, SIAM J. Sci. Stat. Comput., Vol. 11, No. 3, May 1990, pp. 563-592.

[24] R.Hockney, *A fast direct solution of Poisson's equation using Fourier analysis*, J. ACM, Vol. 12, 1965, pp. 95-113.

[25] R.W.Hockney and C.R.Jesshope, *Parallel Computers*, Adam Hilger Ltd, 1981.

[26] J.Jess and H.Kees, *A data structure for parallel L/U decomposition*, IEEE Trans. on Computers, C-31, 1982, pp. 231-239.

[27] S.L.Johnsson, *Odd-even Cyclic Reduction on Ensemble Architecture and the Solution of Tridiagonal Systems of Equations*, Technical Report DCS-RR339, Department of Computer Science, Yale University, New Haven, CT, 1984.

[28] P.S.Kumar, M.K.Kumar and A.Basu, *A parallel algorithm for elimination tree computation and symbolic factorization*, Parallel Computing, Vol. 18, 1992, pp. 849-856.

[29] P.S.Kumar, M.K.Kumar and A.Basu, *Parallel algorithms for sparse triangular system solution*, Parallel Computing, Vol. 19, 1993, pp. 187-196.

[30] V.Kumar, A.Grama, A.Gupta, and G.Karypis, *Introduction to Parallel Computing - Design and Analysis of Algorithms*, Benjamin/Cummings Publishing Company Inc., 1994.

[31] S.Lakshmivarahan and S.K.Dhall, *Analysis and Design of Parallel Algorithms - Arithmetic and Matrix Problems*, McGraw- Hill Publishing Company, 1990.

[32] C.E.Leiserson and T.G.Lewis, *Orderings for parallel sparse symmetric factorization*, In Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing, 1987, pp. 27-32.

[33] T.G.Lewis, B.W.Peyton, and A.Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Stat. Comput., Vol. 10, 1989, pp. 1146-1173.

[34] G.Li and T.F.Coleman, *A new method for solving triangular systems on distributed memory message passing multiprocessors*, SIAM J. Sci. Stat. Comput., Vol. 10, 1989, pp. 382-396.

[35] W.Y.Lin and C.L.Chen, *A parallel algorithm for solving tridiagonal linear systems on distributed memory multiprocessors*, Intl. J. High Speed Comput., Vol. 6, No. 3, 1994, pp. 375-386.

[36] J.W.H.Liu, *Computational models and task scheduling for parallel sparse Cholesky factorization*, Parallel Computing, Vol. 3, 1986, pp. 327-342.

[37] J.W.H.Liu, *Role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. App., Vol. 11, 1990, pp. 134-172.

[38] J.W.H.Liu, *Reordering sparse matrices for parallel elimination*, Parallel Computing, Vol. 11, 1989, pp. 73-91.

[39] J.W.H.Liu, *Equivalent sparse matrix reordering by elimination tree rotations*, SIAM J. Sci. Stat. Comput., Vol. 9, 1988, pp. 424-444.

[40] J.W.H.Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Review, Vol. 34, 1992, pp. 82-109.

[41] J.W.H.Liu, *Modification of minimum degree algorithm by multiple elimination*, ACM Trans. Math. Soft., Vol. 11, 1985, pp. 141-153.

[42] K.N.B.Murthy, *New Algorithms for Parallel Solution of Linear Equations on Distributed Memory Multiprocessors*, Ph.D. Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Madras, India, 1995.

[43] K.N.B.Murthy and C.S.R.Murthy, *A new Gaussian elimination based algorithm for parallel solution of linear equations*, Computers and Mathematics with Applications, Vol. 29, No. 7, 1995, pp. 39-54.

[44] E.Ng, *Parallel direct solution of sparse linear systems*, Parallel Supercomputing: Methods, Algorithms and Applications, John Wiley and Sons Ltd., 1989.

[45] J.M.Ortega and R.G.Voigt, *Solution of partial differential equations on vector and parallel computers*, SIAM Review, Vol. 27, No. 2, June 1985, pp. 149-240.

[46] R.P.Pargas, *Parallel solution of elliptic partial differential equations on a tree machine*, Ph.D. Thesis, University of North Carolina, Chapel Hill, 1982.

[47] F.Peters, *Parallel pivoting algorithms for sparse symmetric matrices*, Parallel Computing, Vol. 1, 1984, pp. 99-110.

[48] E.M.Reingold, J.Nievergelt, and N.Deo, *Combinatorial Algorithms : Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ, 1977.

[49] C.H.Romine and J.M.Ortega, *Parallel solution of triangular systems of equations*, Parallel Computing, Vol. 6, 1988, pp. 109-111.

[50] A.H.Sameh and D.J.Kuck, *On stable parallel linear system solvers*, J. ACM, Vol. 25, No. 1, January 1978, pp. 81-91.

[51] G.Spaletta and D.J.Evans, *The parallel recursive decoupling algorithm for solving tridiagonal linear equations*, Parallel Computing, Vol. 19, January 1993, pp. 563-576.

[52] H.S.Stone, *Parallel tridiagonal equation solvers*, ACM Trans. Math. Soft., Vol. 1, No. 4, 1975, pp. 289-307.

Publications from this Work

"An Improved Mapping of Cyclic Elimination onto Hypercubes using Data Replication", submitted to *Journal of Parallel Algorithms and Applications*.

"New Algorithms for Direct Solution of Sparse Linear Systems: Part I - Symmetric Coefficient Matrix", under preparation.

"New Algorithms for Direct Solution of Sparse Linear Systems: Part II - Nonsymmetric Coefficient Matrix", under preparation.