New Parallel Algorithms for Direct Solution of Sparse Linear Systems: Part I - Symmetric Coefficient Matrix

Kartik Gopalan     C. Siva Ram Murthy *

Department of Computer Science and Engineering
Indian Institute of Technology
Madras 600 036, India
e-mail: murthy@iitm.ernet.in

Abstract

In this paper, we propose a new parallel bidirectional algorithm, based on Cholesky factorization, for the solution of sparse symmetric system of linear equations. Unlike the existing algorithms, the numerical factorization phase of our algorithm is carried out in such a manner that the entire back substitution component of the substitution phase is replaced by a single step division. Since there is a substantial reduction in the time taken by the repeated execution of the substitution phase, our algorithm is particularly suited for the solution of systems with multiple \( b \)-vectors. The effectiveness of our algorithm is demonstrated by comparing it with the existing parallel algorithm, based on Cholesky factorization, using extensive simulation studies.

In Part II of this paper, we propose a new parallel bidirectional algorithm, based on \( LU \) factorization, for the solution of general sparse system of linear equations, having non-symmetric coefficient matrix.

Key Words: Linear Equation, Sparse Symmetric System, Cholesky Factorization, Parallel Algorithm, Bidirectional Scheme, Multiprocessor.

1 Introduction

The problem of solving large sparse systems of linear equations arises in various applications such as finite element analysis, power system analysis, and circuit simulation for VLSI CAD.

*Author for correspondence.
In this paper, we consider the problem of solving sparse symmetric system of linear equations of the form $Ax = b$, where $A$ is a sparse symmetric coefficient matrix of dimension $N \times N$, and $x$ and $b$ are $N$-vectors. 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 [1, 2, 7, 11, 15, 21], 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 [6, 12, 14].

In Part I of this paper, 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 bidirectional algorithm, based on $LU$ factorization, for the solution of general sparse system of linear equations, is presented in Part II of this paper. The application of the novel concept of bidirectional elimination to dense linear systems can be found in [19, 20].
The rest of the paper is organized as follows. In section 2, we present the bidirectional sparse Cholesky factorization algorithm for sparse symmetric matrices. In section 3, we present the bidirectional algorithm for the substitution phase which does not have a back substitution component. In section 4, we develop a bidirectional heuristic algorithm for ordering on the lines of the popular nested dissection ordering algorithm [4, 5] for sparse symmetric matrices. In section 5, we describe a symbolic factorization algorithm which sets up data structures required by the bidirectional Cholesky factorization phase. In section 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 7, we conclude this paper with some observations about possible future improvements to the bidirectional scheme.

2 The Bidirectional Sparse Cholesky Factorization (B-SCF) 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.

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 \textit{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 \( A = U^TU \). We name this process as \textit{factorization in backward direction}. The bidirectional Cholesky factorization of the coefficient matrix \( A \) proceeds as follows.

- \textit{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 \( \left\lceil \frac{N}{2} \right\rceil \) pivot columns, to obtain a lower trapezoidal matrix \( L_0 \), as shown in figure 1, in which only
Figure 1: The progression of BSCF algorithm for $N = 4$
the sub-diagonal entries in columns 1 to \( \left\lceil \frac{N}{2} \right\rceil \) are present. Concurrently, we factorize
A_1 in backward direction, through pivot columns N to \( \left\lceil \frac{N}{2} + 1 \right\rceil \), to obtain an upper
trapezoidal matrix L_1, as shown in figure 1, in which only the super-diagonal elements
in columns N to \( \left\lceil \frac{N}{2} + 1 \right\rceil \) 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 si-
multaneous Cholesky factorization in both forward and backward directions and thus
doubling the number of sub-matrices for \( \log N \) times. Finally we end up with N sub-
matrices of order 1 \( \times \) 1.

The bidirectional Cholesky factorization algorithm described above produces a tree of trape-
zoids of multipliers (i.e., L matrices). In the substitution phase, which is described in
section 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.

### 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

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

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 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). The forward parent of node \( i \), is defined as

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

Similarly, the backward parent of node \( i \), is defined as

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

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 4).

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

### 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 [18]. The following elementary tasks are considered for the BSCF algorithm.

- \( f\text{divide}(i, s) \) divides by \( \sqrt{A_{x0}[i, i]} \) every nonzero element of the sub-diagonal part of the \( i \)th column of sub-matrix \( A_{x0} \).

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

• \( b_{\text{modify}}(i, \text{vector}, s) \) subtracts the contents of \( \text{vector} \) from the \( i \)th column of a sub-matrix \( A_{x1} \), at stage \( s \in \{1 \cdots \log N\} \). \( \text{vector} \) 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.

• \( F^{(s)}_i \) denotes the set of all columns with indices smaller than \( i \) that modify the \( i \)th column in the forward direction at stage \( s \).

• \( B^{(s)}_i \) denotes the set of all columns with indices greater than \( i \) that modify the \( i \)th column in the backward direction at stage \( s \).

These data structures are generated during the symbolic factorization phase. This phase is described in section 5. In the remaining part of this section, we describe the implementation of BSCF algorithm on a message passing multiprocessor.

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, the multiple of the modifying column can be calculated at the processor storing column \( i \) itself and the resulting vector can be 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 \( F^{(s)}_j \) or \( B^{(s)}_j \)). 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 1 below, we incorporate the above idea in the BSCF algorithm and present the \( \text{fan-in} \) BSCF algorithm. The set \( \text{List}_{myid} \) is the set of columns stored in processor \( P_{myid} \). Each processor maintains the sparse vectors \( f\text{Update}_j \) and \( b\text{Update}_j \) for \( 1 \leq j \leq N \). If column \( i \) is to modify column \( j \) in forward direction at stage \( s \) then, after performing \( f\text{divide}(i, s) \) operation, the processor \( P_{myid} \), which stores the column \( i \), adds an appropriate
multiple of column $i$ to the vector $f_{Update_j}$. 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 $f_{Update}$ vector is sent to the processor storing the column $j$. Similar mechanism operates for factorization in backward direction.

**Algorithm 1** (*The parallel fan-in BSCF algorithm*)

begin

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

par begin

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 $f_{Update_i} := 0$;

while $List \neq \phi$ do

if $\exists i \in List$ such that $f_{divide}(j, s)$ 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

$f_{divide}(i, s)$;

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

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

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

send a message of the form $(j, f_{Update}_j, s)$

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
parts of the columns of the symmetric matrix $A_{x0}$ are stored*)

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

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

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

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

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 $A_{x1}[j, \bar{i}] \neq 0$ do

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

(*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 1. Let the number of processors $p = 2^d$
(as in hypercube multiprocessors) and $N = 2^n \ (n, d \in \mathcal{N}, \text{ the set of natural numbers}).
Figure 2: Progression of the BSCF algorithm for $p = 4$ and $N = 16$ (four columns are stored in each processor).
Assume that we map the equations on the processors in a block wrap manner (as shown in figure 2). 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 2.

In this figure, we also 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.

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 substitution), 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.

3.1 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 [17]. 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 $x_i = b_0[i]/L_0[i,i]$ and modifies
Figure 3: The progression of substitution phase for $N = 4$
the remaining elements of $b_0$-vector as $b_0[j] = b_0[j] - L_0[j,i] \cdot x_i$ 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). 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 $b_1[j] = b_1[j] - L_1[j,i] \cdot x_i$ 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).

- **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).

- **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.

### 3.2 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 [6], parallel schemes for solving sparse triangular systems resulting from regular Cholesky factorization. Their scheme is an adaptation of the corresponding dense algorithm proposed by Romine and Ortega in [24] and it uses the following inner product form to carry out forward factorization.

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

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 \( b_{x_0} \) from left to right across the trapezoid \( L_{x_0} \), each element \( b_{x_0}[i] \) 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 \( b_{x_0} \). If column \( i \) belongs to the first half of the matrix \( A_{x_0} \) then, after subtracting the complete inner product of row \( i \) in \( L_{x_0} \) from \( b_{x_0}[i] \), the processor storing the column \( i \) computes \( x_i = b_{x_0}[i] / L_{x_0}[i, i] \). 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 \( L_{x_0} \) from \( b_{x_0}[i] \), two copies of the element \( b_{x_0}[i] \), namely \( b_{x_00}[i] \) and \( b_{x_01}[i] \), are made for modification at the next stage of the BS algorithm. Similar mechanism operates while updating a vector \( b_{x_1} \) with a trapezoid \( L_{x_1} \) in the backward direction. The complete details of the BS algorithm are given below.

**Algorithm 2** (* The bidirectional substitution algorithm *)

```
begin
  for \( s := 1 \) to \( \log N \) do
    parbegin
      Forward_modify(List_mid, s);
      Backward_modify(List_mid, s);
    parend
  end

procedure Forward_modify(List, s)
begin
  Let \( b_{x_0} \) be the forward copy of the b-vector to be modified
  by trapezoid \( L_{x_0} \) 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 \( L_{x_0} \) do
      receive message \((i, t)\) having partial inner product \( t \) from processor \( P_j \);
      \( b_{x_0}[i] := b_{x_0}[i] - t \);
    if column \( i \) belongs to the first half of \( L_{x_0} \) then
      \( x_i := b_{x_0}[i] / L_{x_0}[i, i] \);
      for all \( j \) such that \( L_{x_0}[j, i] \neq 0 \) do
        \( t_j := t_j + x_i \times L_{x_0}[j, i] \);
```

14
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 
  $b_{x0}[i] := b_{x0}[i]$;
  $b_{x01}[i] := b_{x0}[i]$;
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 
        $t_j := t_j + x_i \times L_{x1}[j,i]$;
        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 
end

As in the case of the BSCF algorithm, a special situation arises when $p = 2^d$ and $N = 2^n$ ($n,d \in \mathbb{N}$). 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 $L_{ind}^Tx = y$. 

15
In this manner, we avoid executing excessive number of floating point operations when all the remaining computations are restricted to occur within individual processors.

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

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. Various techniques for the ordering phase can be found in [3, 4, 5, 16, 22].

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 [4, 5]. 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 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 $7 \times 7$ 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
Figure 4: Dissection of a $7 \times 7$ grid by separators during nested dissection

Figure 5: The nested dissection tree for a $7 \times 7$ grid
again to the two separated halves. Figure 4 shows the manner in which the separators (S1 to S15) divide a $7 \times 7$ grid. The recursive division of the grid yields a tree structure of separators and nodes as shown in figure 5. 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 $1 \times 1$, $1 \times 2$, $2 \times 1$ or $2 \times 2$. 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 6 and the forward and backward elimination trees resulting from this ordering are shown in figure 7. As seen from figure 7, although the forward tree is short and wide, 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.

![Figure 6: Ordering of a $7 \times 7$ grid using regular nested dissection ordering](image-url)
Figure 7: The forward and backward elimination trees for a $7 \times 7$ grid obtained using regular nested dissection ordering.

Figure 8: The colouring of tree nodes in bidirectional nested dissection ordering.
Figure 9: Ordering of a $7 \times 7$ grid using bidirectional nested dissection ordering

- **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 5.

- **Step 2**: At each level of the nested dissection tree, label approximately half of the tree nodes as white and the other half as black, as shown in figure 8.

- **Step 3**: While numbering the grid points, proceed as follows.

  1. Keep two counts - `whiteCount` initialized to 1 and `blackCount` initialized to $k \times k$.
  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. Apply the above step to all grid points of each node at level 0 followed by each node at level 1 and so on up to the root.

The ordering obtained from this scheme is shown in figure 9 and the corresponding forward and backward elimination trees are shown in figure 10. As seen in this figure, although the forward elimination tree is not as short and wide as in the case of regular nested dissection
Figure 10: The forward and backward elimination trees for a $7 \times 7$ grid obtained using bidirectional nested dissection ordering
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 which allocates memory and sets up the appropriate data structures prior to the BSCF algorithm.

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 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 3 (*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

22
Forward_SF(col,s);
for col := N downto 1 do
  Backward_SF(col,s);
end

procedure Forward_SF(col,s)
begin
  Let $A_{x_0}$ be the forward sub-matrix that contains column $col$ at stage $s$;
  if $col$ belongs to the first half of $A_{x_0}$ then
    Calculate $fparent(col, A_{x_0})$ using definition given in section 3.2.2;
    if $fparent(col, A_{x_0})$ belongs to the first half of $A_{x_0}$ then
      $Colstruct(A_{x_0}, fparent(col, A_{x_0})) :=$
      $Colstruct(A_{x_0}, fparent(col, A_{x_0})) \cup Colstruct(A_{x_0}, col);$
    for all $j$ such that $j$ belongs to second half of $A_{x_0}$ and $A_{x_0}[col,j] \neq 0$ do
      $Colstruct(A_{x_0}, j) := Colstruct(A_{x_0}, j) \cup Colstruct(A_{x_0}, col);$
    for all $j$ such that $j \in Colstruct(A_{x_0}, col)$ do
      $F_{j}^{(s)} := F_{j}^{(s)} \cup \{col\};$
  else
    $Colstruct(A_{x_00}, col) := Colstruct(A_{x_0}, col);$
    for all $j \in Colstruct(A_{x_0}, col)$ do
      $Colstruct'(A_{x_01}, j) := Colstruct'(A_{x_01}, j) \cup \{col\};$
  end

procedure Backward_SF(col,s)
begin
  Let $A_{x_1}$ be the backward sub-matrix that contains column $col$ at stage $s$;
  if $col$ belongs to the second half of $A_{x_1}$ then
    Calculate $bparent(col, A_{x_1})$ using definition given in section 3.2.2;
    if $bparent(col, A_{x_1})$ belongs to the second half of $A_{x_1}$ then
      $Colstruct'(A_{x_1}, bparent(col, A_{x_1})) :=$
      $Colstruct'(A_{x_1}, bparent(col, A_{x_1})) \cup Colstruct'(A_{x_1}, col);$
    for all $j$ such that $j$ belongs to first half of $A_{x_1}$ and $A_{x_1}[col,j] \neq 0$ do
      $Colstruct'(A_{x_1}, j) := Colstruct'(A_{x_1}, j) \cup Colstruct'(A_{x_1}, col);$
    for all $j$ such that $j \in Colstruct'(A_{x_1}, col)$ do
      $B_{j}^{(s)} := B_{j}^{(s)} \cup \{col\};$
  else

23
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 [8, 10, 13]. While the former scheme by George et.al. requires the information about the elimination tree structure apriori, the latter scheme by 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 \( s \).
\[ ffparent(i, s) = \min \{ j \mid j \in \text{first half of } A_{x0} \text{ and } j \in Colstruct(A_{x0}, i) \} \cdot \]

- \( fbparent(i, s) \) denotes the false backward parent of a column \( i \) in the sub-matrix \( A_{x1} \) being factorized in the backward direction at stage \( s \).
\[ fbparent(i, s) = \max \{ j \mid j \in \text{second half of } A_{x1} \text{ and } j \in Colstruct'(A_{x1}, i) \} \cdot \]

The details of this algorithm are described below.

**Algorithm 4** (*The parallel bidirectional symbolic factorization*)

begin
for \( s := 1 \) to \( \log N \) do

parbegin
Forward\_SF(List\_myid,s);
Backward\_SF(List\_myid,s);

end.

\textbf{procedure} Forward\_SF(List,s)
\begin{itemize}
\item for each \textit{i} \in List \textbf{do}
\begin{itemize}
\item Let \textit{Ax}_0 be the forward sub-matrix to which column \textit{i} belongs at stage \textit{s};
\item \textit{dummy\_parent} := last node of sub-matrix \textit{Ax}_0;
\item Determine the false forward parent \textit{ffparent}(i, s);
\item \textit{send} \textit{ffparent}(i, s) to processor containing \textit{dummy\_parent};
\item if \textit{i} = \textit{dummy\_parent} then
\begin{itemize}
\item receive \textit{ffparent}(j, s) from each column \textit{j};
\item broadcast forward fet \textit{T_{ff}} constructed from received \textit{ffparent} information;
\end{itemize}
\item receive forward fet \textit{T_{ff}} broadcast from \textit{dummy\_parent};
\end{itemize}
\item Let the children of column \textit{i} in \textit{T_{ff}} be \textit{CHLD}(i);
\item (*initialise the expected and accumulated weights for node \textit{i}*)
\item \textit{exp\_wt}(i) := | \textit{CHLD}(i) |; \textit{acc\_wt}(i) := 0;
\item \textit{first}(i) := true;
\item if column \textit{i} is a true leaf of \textit{T_{ff}} and column \textit{i} is in first half of sub-matrix \textit{Ax}_0 then
\begin{itemize}
\item \textit{send} \textit{Col\_struct}(\textit{Ax}_0, i) to \textit{ffparent}(i, s) with weight 1;
\item \textit{send} \textit{Col\_struct}(\textit{Ax}_0, i) with weight 0 to all nodes \textit{j} in second half of \textit{Ax}_0 such that \textit{j} \in \textit{Col\_struct}(\textit{Ax}_0, i);
\end{itemize}
\item repeat
\begin{itemize}
\item receive a message \textit{S} intended for column \textit{i};
\end{itemize}
\item Let the message be from processor storing column \textit{j} with weight \textit{w};
\item if column \textit{i} is in first half of sub-matrix \textit{Ax}_0 then
\begin{itemize}
\item \textbf{case} type of \textit{S}
\begin{itemize}
\item attach or ordinary:
\begin{itemize}
\item \textit{Col\_struct}(\textit{Ax}_0, i) := \textit{Col\_struct}(\textit{Ax}_0, i) \cup \textit{Col\_struct}(\textit{Ax}_0, j);
\item \textit{acc\_wt} := \textit{acc\_wt} + \textit{w};
\end{itemize}
\item if \textit{j} \in \textit{CHLD}(i) then delete \textit{j} from \textit{CHLD}(i);
\item if (\textit{i} \text{ CHLD}(i) = 0 \text{ and } \textit{acc\_wt}(i) \geq \textit{exp\_wt}(i)) then
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
```plaintext
ffparent(i, s) := k where k = min(Construct(Ax0, 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 Construct(Ax0, i) to ffparent(i) with weight wt;
    send Construct(Ax0, i) to all nodes j in second half of Ax0
    such that j ∈ Construct(Ax0, i) with weight 0;
detach:
    delete j from CHLD(i);
else
    case type of S
    attach or ordinary:
    if j ∈ Construct(Ax0, i) then
        Construct(Ax0, i) := Construct(Ax0, i) ∪ Construct(Ax0, j);
detach:
    if i = dummy_parent then
        delete j from CHLD(i);
    if (| CHLD(i) = 0 |) then
        broadcast forward phase over message;
until S is forward phase over message;
for each i ∈ List do
    if column i is in second half of sub-matrix then
        Construct(Ax00, i) := Construct(Ax0, i);
    for all j such that Ax0[j, i] ≠ 0 do
        Construct'(Ax01, j) := Construct(Ax01, j) ∪ i;
end

procedure Backward_SF(List, s)
begin
    for each i ∈ List do
        Let Ax1 be the backward sub-matrix to which column i belongs at stage s;
```
\texttt{dummy\_parent} := \text{last node of sub-matrix} \ A_{x1};

Determine the false backward parent \texttt{fbparent}(i, s);

\textit{send} \texttt{fbparent}(i, s) \textit{to processor containing} \texttt{dummy\_parent};

\textbf{if} \texttt{i} = \texttt{dummy\_parent} \textbf{then}

\begin{itemize}
  \item [receive] \texttt{fbparent}(j, s) \textit{from each column} \ j;
  \item [broadcast] backward fet \ T_{fb} \textit{constructed from received}
  \texttt{fbparent} \textit{information};
  \item [receive] backward fet \ T_{fb} \textit{broadcast from} \texttt{dummy\_parent};
\end{itemize}

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

\texttt{exp\_wt}(i) := | \textit{CHLD}(i) |; \texttt{acc\_wt}(i) := 0;

\texttt{first}(i):=true;

\textbf{if} column \texttt{i} \textit{is a true leaf of} \ T_{fb} \textbf{and} column \texttt{i} \textit{is in second half}
\textit{of sub-matrix} \ A_{x1} \textbf{then}

\begin{itemize}
  \item [send] \textit{Colstruct}'(A_{x1}, i) \textit{to} \texttt{fbparent}(i, s) \textit{with weight} 1;
  \item [send] \textit{Colstruct}'(A_{x1}, i) \textit{with weight} 0 \textit{to all nodes} \ j \ \textit{in first half}
  \textit{of sub-matrix} \ A_{x1} \textit{such that} \ j \in \textit{Colstruct}'(A_{x1}, i);
\end{itemize}

\textbf{repeat}

\textit{receive} a message \texttt{S} \textit{intended for column} \texttt{i};

Let the message be from processor storing column \texttt{j} \textit{with weight} \texttt{w};

\textbf{if} column \texttt{i} \textit{is in second half of sub-matrix} \ A_{x1} \textbf{then}

\textbf{case} type of \texttt{S}

\begin{itemize}
  \item [attach or ordinary:]
    \begin{align*}
    \textit{Colstruct}'(A_{x1}, i) & := \textit{Colstruct}'(A_{x1}, i) \cup \textit{Colstruct}'(A_{x1}, j); \\
    \texttt{acc\_wt} & := \texttt{acc\_wt} + \texttt{w};
    \end{align*}

  \item [if] \texttt{j} \in \textit{CHLD}(i) \textbf{then} \textit{delete} \texttt{j} \textit{from} \textit{CHLD}(i);

  \item [if] (| \textit{CHLD}(i) = 0 |) \textbf{and} (\texttt{acc\_wt}(i) \geq \texttt{exp\_wt}(i)) \textbf{then}
    \texttt{fbparent}(i, s) := k \ \text{where} \ k = \max(\textit{Colstruct}'(A_{x1}, i));

  \item [if] \texttt{fbparent}(i) \textit{has changed} \textbf{then}
    \textit{send} a detach message to old parent;

  \item [if] \texttt{first}(i) \textbf{then}
    \begin{align*}
    \texttt{wt} & := \texttt{acc\_wt}(i) - \texttt{exp\_wt}(i) + 1; \\
    \texttt{exp\_wt}(i) & := 0; \\
    \texttt{first}(i) & :=\text{false};
    \end{align*}

  \item [else]
    \begin{align*}
    \texttt{wt} & := \texttt{w}; \\
    \textit{send} \textit{Colstruct}'(A_{x1}, i) \textit{to} \texttt{fbparent}(i) \textit{with weight} \texttt{wt};
    \end{align*}
\end{itemize}
send Colstruct'(A_{x_1}, i) to all nodes j in first half of sub-matrix such that j ∈ Colstruct'(A_{x_1}, i) with weight 0;
detach:
delete j from CHLD(i);
else

   case type of S
 attach or ordinary:
      if j ∈ Colstruct'(A_{x_1}, i) then
   Colstruct'(A_{x_1}, i) := Colstruct'(A_{x_1}, i) ∪ Colstruct'(A_{x_1}, j);
detach:
      if i = dummy_parent then
   delete j from CHLD(i);
      if (|CHLD(i)| = 0) then
   broadcast backward phase over message;
until S is backward phase over message;
for each i ∈ List do
   if column i is in first half of sub-matrix then
      for all j such that A_{x_1}[j, i] ≠ 0 do
   Colstruct(A_{x_10}, j) := Colstruct'(A_{x_10}, j) ∪ i;
   Colstruct'(A_{x_11}, i) := Colstruct'(A_{x_11}, i);
end

6 Experimental Results and Performance Analysis

To evaluate the performance of the entire bidirectional scheme presented in this paper, 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 machines 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 4.
• Symbolic factorization: The sequential bidirectional symbolic factorization algorithm described in section 5.
• Numerical factorization: The parallel BSCF algorithm described in section 2.
• **Substitution**: The parallel BS algorithm described in section 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 [4].

• **Symbolic factorization**: The sequential symbolic factorization algorithm presented in [8].

• **Numerical factorization**: The parallel fan-in algorithm given in [1].

• **Substitution**: The elimination tree based forward and back substitution algorithms given in [14].

Mapping of columns onto processors is an important issue. For the bidirectional scheme, we have used the *block wrap around mapping* using gray code [23] whereas for the regular algorithm we have used the *subtree-to-processor* mapping [9] 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 11, 12, 13, and 14 show the comparison of the measured speedups of the two schemes for various values of the above parameters.

As mentioned earlier in section 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 a 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 11(a), 12(a), 13(a), and 14(a) compare the pseudo-speedup of the bidirectional algorithm with the speedup of the regular algorithm for the first three phases put together. The figures 11(b), 12(b), 13(b), and 14(b) compare the pseudo-speedup of the bidirectional algorithm with the speedup of the regular algorithm for the substitution phase alone. Figures 11(c), 12(c), 13(c), and 14(c) plot the actual speedups of bidirectional and
Figure 11: Speedups obtained for bidirectional algorithm versus regular algorithm for a $16 \times 16$ grid (i.e., $N = 256$) with $C/E = 50$
Figure 12: Speedups obtained for bidirectional algorithm versus regular algorithm for a $16 \times 16$ grid (i.e., $N = 256$) with $C/E = 100$
(a) factorization

(b) substitution

(c) solving multiple b-vectors with 8 processors

Figure 13: Speedups obtained for bidirectional algorithm versus regular algorithm for a $32 \times 32$ grid (i.e., $N = 1024$) with $C/E = 50$
Figure 14: Speedups obtained for bidirectional algorithm versus regular algorithm for a $32 \times 32$ grid (i.e., $N = 1024$) with $C/E = 100$
regular algorithms for all the four phases put together versus the number of $b$-vectors for which substitution phase is repeatedly executed. In figure 11(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 12(c) $p = 8$ and $k = 16$, in figure 13(c) $p = 32$ and $k = 32$, and in figure 14(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.

7 Conclusions

In this paper, 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.

In Part II of this paper, we present the bidirectional algorithm for solution of general sparse linear systems. We describe the important differences with the symmetric coefficient matrix case, that arise in the ordering technique, the symbolic factorization phase, and message passing during numerical factorization phase.

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.

34
Acknowledgements  This work is supported by the Indian National Science Academy, and the Department of Science and Technology.

References


