New Parallel Algorithms for Direct Solution of Sparse Linear Systems : Part II - Non-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 Part I of this this paper, we proposed a new parallel *bidirectional algorithm*, based on *Cholesky factorization*, for the solution of sparse *symmetric* system of linear equations. In 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. As with the sparse symmetric systems, 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. 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 our algorithm is demonstrated by comparing it with the existing parallel algorithm, based on LU factorization, using extensive simulation studies.

Key Words: Linear Equation, General Sparse System, LU Factorization, Parallel Algorithm, Bidirectional Scheme, Multiprocessor.

[*]Author for correspondence.

1 Introduction

The problem of solving *general sparse systems of linear equations* arises in various problems in structural engineering, chemical engineering, and nuclear physics. In this paper, 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. 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 [2, 1, 23, 6, 4, 10, 15], not much progress has been made in the case of substitution phase [6, 11, 14] due to the limited amount of parallelism inherent in this phase.

In Part I of this paper, we developed a *bidirectional algorithm* that is suitable for the solution of sparse symmetric linear systems with multiple b-vectors. In this paper 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 described in Part I; the description is not repeated here.

It is known that for sparse non-symmetric problems, pivoting is necessary to ensure numerical stability during numerical factorization phase. In this paper, 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 [21, 22].

The rest of the paper is organized as follows. In section 2, we present the bidirectional sparse factorization algorithm based on LU factorization for general sparse matrices. In section 3, we develop a bidirectional heuristic algorithm which produces a reordered coefficient matrix suitable for numerical factorization phase. In section 4, we look at a symbolic factorization algorithm which sets up data structures required by the numerical factorization phase. In section 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 6, we conclude the paper with some observations about possible future improvements to the bidirectional scheme.

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. The basic concept behind the bidirectional factorization algorithm is the same as that presented in section 2.1 of Part I. In this section, we 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.

2.1 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 3).

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

2.2 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. 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 2 in Part I), 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 *)

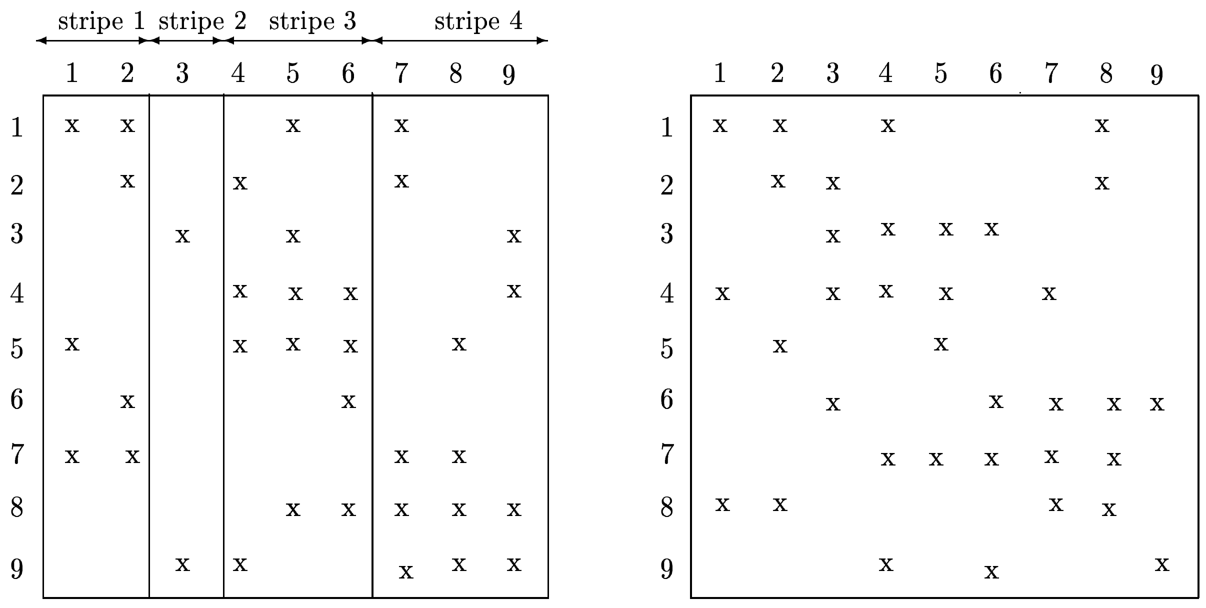
```
begin
    for $s := 1$ to $\log N$ do
        parbegin
            Forward_factorize($List_{myid}$,$s$);
            Backward_factorize($List_{myid}$,$s$);
        **parend**
**end**

**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 2.3 of Part I, 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 2 of Part I). 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.

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. Various techniques for the ordering phase can be found in [3, 12, 16, 17, 18, 19, 24].

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 [5], is applied to the coefficient matrix A. This is followed by application of Liu's scheme of elimination tree rotation [18, 19] 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.



(a) 9 x 9 striped sparse matrix

(b) 9 x 9 alternate stripe reordered matrix

Figure 1: Ordering of a 9×9 matrix using alternate stripe reordering.

The ordering resulting from the above scheme is, however, not suited for the BSF algorithm due to reasons given below. Recall that in section 2.1 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 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 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 4 of Part I. The latter method can be alternatively viewed as consisting of two stages - (i) applying the regular nested dissection method [3] 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 paper 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 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. 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 numewrical factorization phase, parallelizing this phase does not yeild significant improvements in the overall performance. For the case of regular symbolic factorization, parallel algorithms have been described in [7, 9, 13].

5 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 the SPARC Classic machines 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 3.
- *Symbolic factorization* : The sequential bidirectional symbolic factorization algorithm described in section 4.
- *Numerical factorization* : The parallel fan-out BSF algorithm described in section 2.
- *Substitution* :The parallel BS algorithm described in section 3 of Part I.

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

- *Ordering* : The fill reducing minimum degree ordering [5] followed by Liu's elimination tree rotation scheme [18].
- *Symbolic factorization* : The sequential symbolic factorization algorithm presented in [7].
- *Numerical factorization* : The parallel fan-out algorithm given in [2, 15].
- *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 [25] whereas for the regular algorithm we have used the *subtree-to-processor* mapping [8] based on elimination tree.

For the purpose of simulation we used three test matrices, described in table 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 point operation (50 and 100). Figures 2, 3, and 4 show the comparison of the measured speedups of the two schemes for various values of the above parameters.

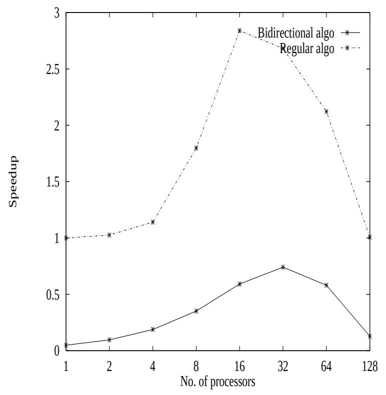
Table 1: Matrices from Harwell-Boeing collection

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.

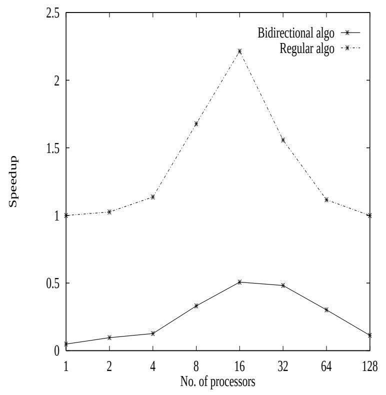
As mentioned earlier in section 1, the first three phases, namely ordering, symbolic factorization, and numerical factorization, are executed only once and the substitution phase is repeatedly executed for each one of the different b-vectors. The output of the factorization phase of the bidirectional 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 2(a), 2(d), 3(a), 3(d), 4(a), and 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 2(b), 2(e), 3(b), 3(e), 4(b), and 4(e) compare the pseudo-speedup of the bidirectional algorithm with the speedup of the regular algorithm for the substitution phase alone. The figures 2(c), 2(f), 3(c), 3(f), 4(c), and 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 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 2(f), 3(c), 3(f), 4(c), and 4(f). These 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.

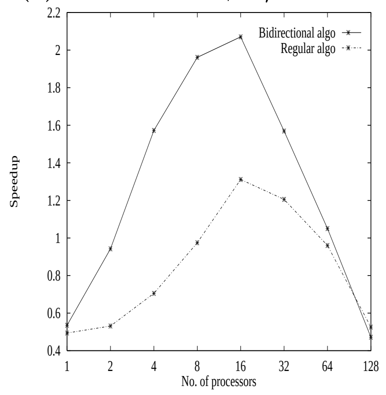
Figures 5(a), (b), and (c), and figures 6(a), (b), and (c) compare the pseudo-speedup of the bidirectional factorization phase with two different reorderings of each of the coefficient matrices - one obtained using the ASR heuristic proposed in section 3 and the other obtained using Liu's scheme [18]. The graphs clearly indicate that BSF algorithm gives higher speedup



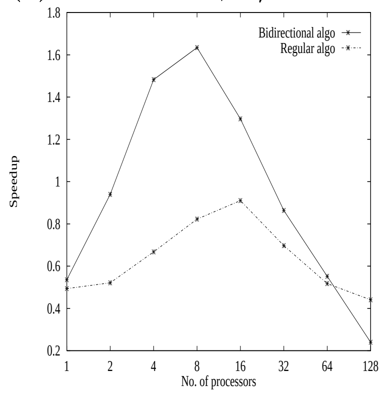
(a) factorization, C/E=50



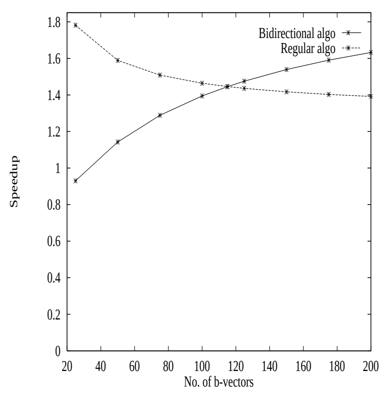
(d) factorization, C/E=100



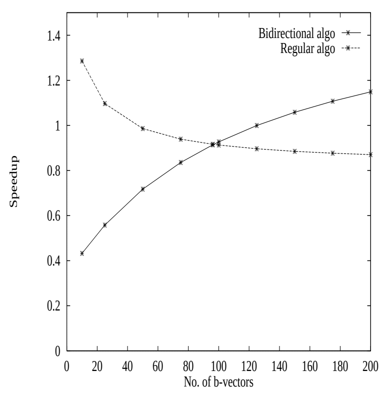
(b) substitution, C/E=50



(e) substitution, C/E=100

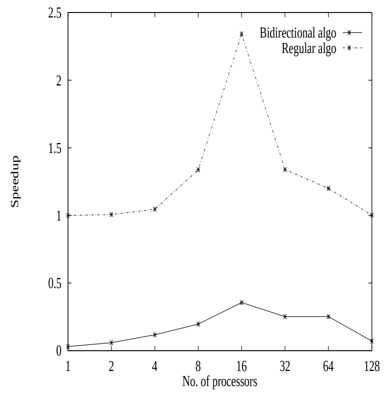


(c) solving multiple b-vectors with 16 processors, C/E=50

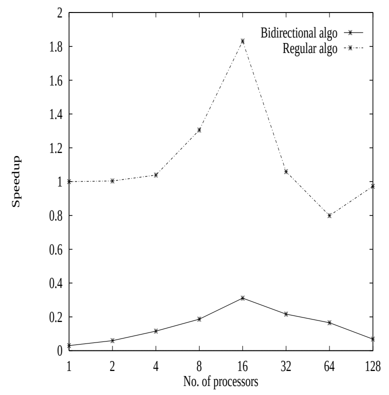


(f) solving multiple b-vectors with 8 processors, C/E=100

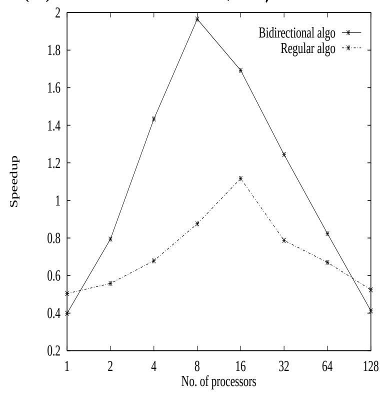
Figure 2: Speedups obtained for bidirectional algorithm versus regular algorithm for WILL199.



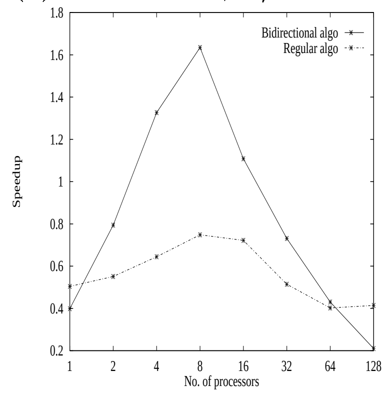
(a) factorization, C/E=50



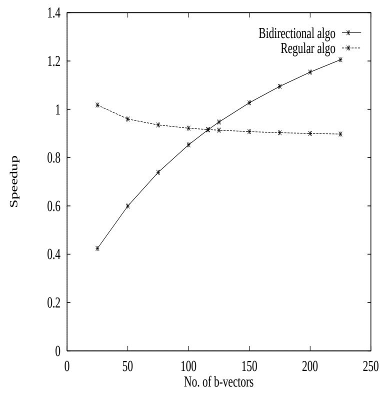
(d) factorization, C/E=100



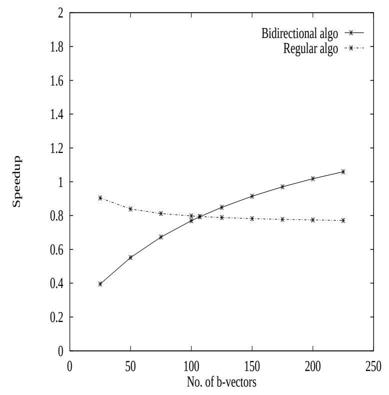
(b) substitution, C/E=50



(e) substitution, C/E=100

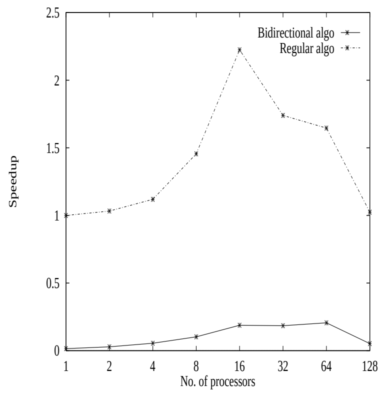


(c) solving multiple b-vectors with 8 processors, C/E=50

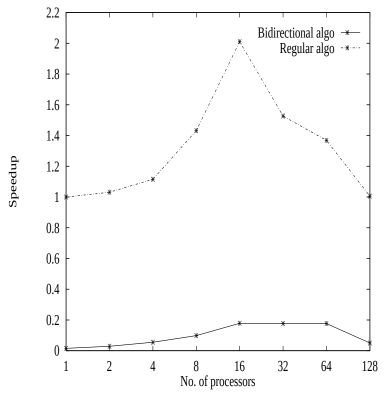


(f) solving multiple b-vectors with 8 processors, C/E=100

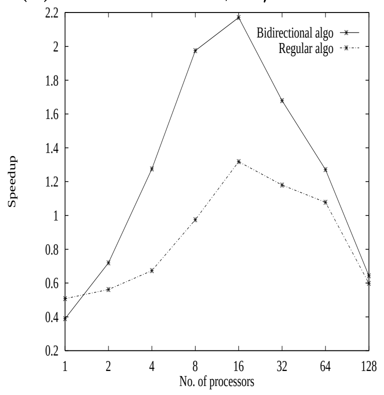
Figure 3: Speedups obtained for bidirectional algorithm versus regular algorithm for GRE216A.



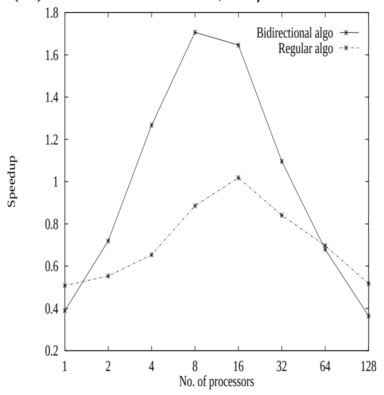
(a) factorization, C/E=50



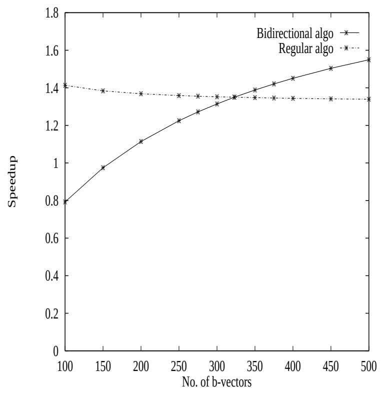
(d) factorization, C/E=100



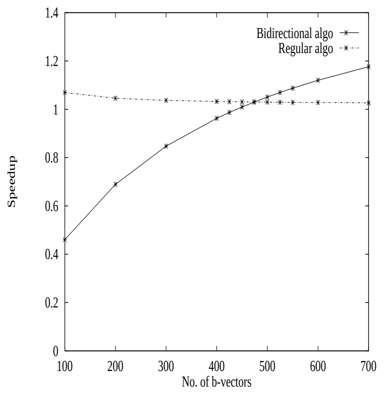
(b) substitution, C/E=50



(e) substitution, C/E=100

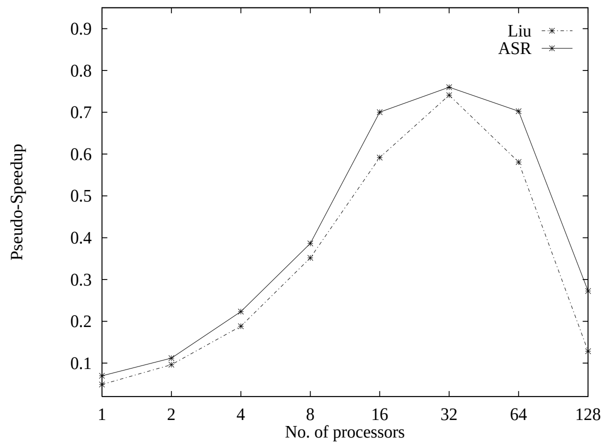


(c) solving multiple b-vectors with 16 processors, C/E=50

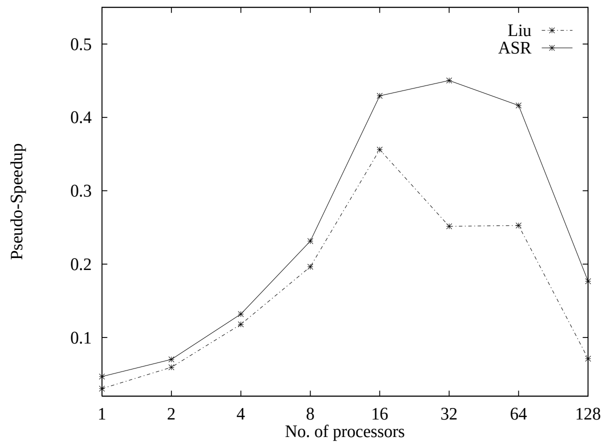


(f) solving multiple b-vectors with 8 processors, C/E=100

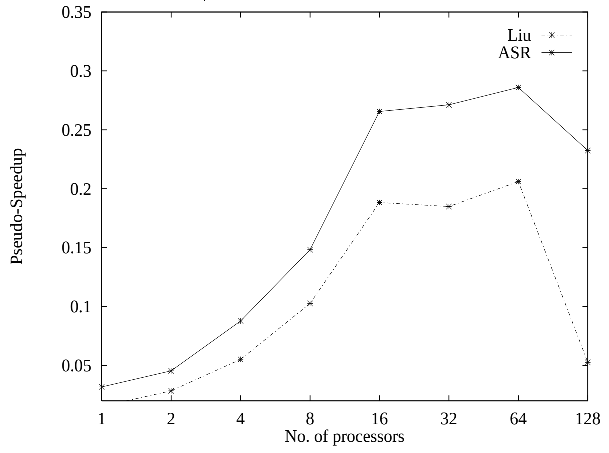
Figure 4: Speedups obtained for bidirectional algorithm versus regular algorithm for GRE343.



(a) WILL199 matrix

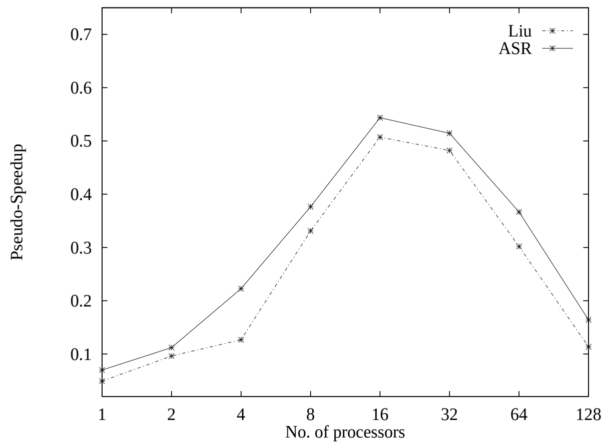


(b) GRE216A matrix

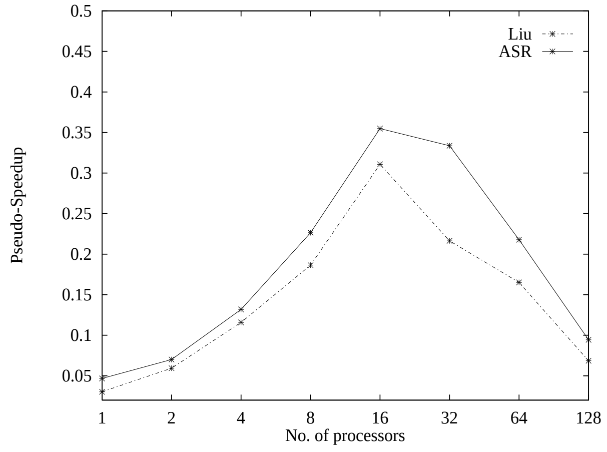


(c) GRE343 matrix

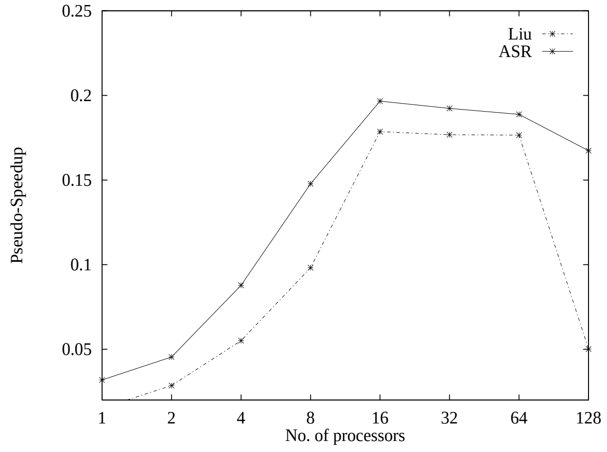
Figure 5: Pseudo-speedups obtained for bidirectional factorization with matrices reordered by ASR method versus those reordered by Liu's rotation method. $C/E = 50$.



(a) WILL199 matrix



(b) GRE216A matrix



(c) GRE343 matrix

Figure 6: Pseudo-speedups obtained for bidirectional factorization with matrices reordered by ASR method versus those reordered by Liu's rotation method. $C/E = 100$.

when the coefficient matrix is reordered using the ASR heuristic rather than with Liu's scheme.

6 Conclusions

In this paper, 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.

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

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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

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

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