PARALLEL SOLUTION OF SPARSE LINEAR LEAST SQUARES PROBLEMS ON DISTRIBUTED-MEMORY MULTIPROCESSORS[1]

Chunguang Sun
Advanced Computing Research Institute
Cornell Theory Center
Cornell University
Ithaca, NY 14853-3801
csun@cs.cornell.edu

Tech. Report CTC95TR212, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY, May 1995.

Abstract

This paper studies the solution of large-scale sparse linear least squares problems on distributed-memory multiprocessors. The method of corrected semi-normal equations is considered. New block-oriented parallel algorithms are developed for solving the related sparse triangular systems. The arithmetic and communication complexities of the new algorithms applied to regular grid problems are analyzed. The proposed parallel sparse triangular solution algorithms together with a block-oriented parallel sparse QR factorization algorithm result in a highly efficient block-oriented approach to the parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. Performance of the block-oriented approach is demonstrated empirically through an implementation on an IBM Scalable POWERparallel system SP2. The largest problem solved has over two million rows and more than a quarter million columns. The execution speed for the numerical factorization of this problem achieves over 3.7 gigaflops per second on an IBM SP2 machine with 128 processors.

Key words. parallel algorithms, sparse matrix, orthogonal factorization, multifrontal method, least squares problems, triangular solution, distributed-memory multiprocessors

AMS subject classifications. 65F05, 65F50

Abbreviated title: Parallel Solution of Sparse Least Squares Problems

[1] This work was supported in part by the Cornell Theory Center which receives funding from members of its Corporate Research Institute, the National Science Foundation (NSF), the Advanced Research Projects Agency (ARPA), the National Institutes of Health (NIH), New York State, and IBM corporation.

PARALLEL SOLUTION OF SPARSE LINEAR LEAST SQUARES PROBLEMS ON DISTRIBUTED-MEMORY MULTIPROCESSORS †

CHUNGUANG SUN ‡

Abstract. This paper studies the solution of large-scale sparse linear least squares problems on distributed-memory multiprocessors. The method of corrected semi-normal equations is considered. New block-oriented parallel algorithms are developed for solving the related sparse triangular systems. The arithmetic and communication complexities of the new algorithms applied to regular grid problems are analyzed. The proposed parallel sparse triangular solution algorithms together with a block-oriented parallel sparse QR factorization algorithm result in a highly efficient block-oriented approach to the parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. Performance of the block-oriented approach is demonstrated empirically through an implementation on an IBM Scalable POWERparallel system SP2. The largest problem solved has over two million rows and more than a quarter million columns. The execution speed for the numerical factorization of this problem achieves over 3.7 gigaflops per second on an IBM SP2 machine with 128 processors.

Key words. parallel algorithms, sparse matrix, orthogonal factorization, multifrontal method, least squares problems, triangular solution, distributed-memory multiprocessors

1. Introduction. The numerical solution of large and sparse linear least squares problems

$$\min_x \|Ax - b\|_2 \tag{1}$$

lies at the heart of many challenging computational problems frequently arising in geodetic survey, photogrammmetry, tomography, structural analysis, surface fitting, numerical optimization, etc. Let A be an $M \times N$ matrix with full column rank. The QR factorization method [7] for solving (1) first computes the QR factorization

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}. \tag{2}$$

Then the solution to (1) is obtained by solving the triangular system $Rx = c$, where c is the first N components of $Q^t b$. The matrix R is referred to as the upper triangular factor of A. This method is numerically backward stable. However, the orthogonal matrix Q needs to be retained if right hand sides are not yet available when A is factorized. If A is large, it is often too expensive to store Q in the main memory.

The method of normal equations is to solve $A^tAx = A^tb$. This method is simple but exhibits potential numerical instability because of the loss of information in forming A^tA explicitly and the fact that the condition number of A^tA is the square of that of A.

To avoid the explicit formation of A^tA, the semi-normal equations $R^tRx = A^tb$ can be used to handle new right hand sides without storing Q. Björck [2] has shown that the method of semi-normal equations(SNE) is numerically unstable and proposed that a correction step or an iterative refinement step be added to the SNE method as follows:

$$\begin{cases} r \leftarrow b - Ax, \\ R^tR\triangle x = A^tr, \\ x \leftarrow x + \triangle x. \end{cases} \tag{3}$$

† This work was supported in part by the Cornell Theory Center which receives funding from members of its Corporate Research Institute, the National Science Foundation (NSF), the Advanced Research Projects Agency (ARPA), the National Institutes of Health (NIH), New York State, and IBM corporation.

‡ Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY 14853-3801(`e-mail:csun@cs.cornell.edu, phone:607-254-8863, fax:607-254-8888`).

Björck [2] has shown that this new method, the method of corrected semi-normal equations(CSNE), is in general as accurate as the QR factorization method. However, it is not always backward stable and may not be accurate for problems with widely different row norms. The numerical stability of the method of normal equations and the SNE method is far less satisfactory than that of the CSNE method.

In this paper we propose a new block-oriented approach to the parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. Our approach is based on the CSNE method. Major tasks involved are sparse QR factorization, sparse triangular solution and sparse matrix-vector multiplication. The sparse QR factorization is computed by a block-oriented parallel multifrontal algorithm recently introduced in [19]. New block-oriented parallel multifrontal algorithms are presented for solving the related sparse triangular systems. The proposed parallel sparse triangular solution algorithms use efficient parallel dense triangular solvers as their numerical kernels. The storage scheme for R in the parallel sparse triangular solution is exactly the same as that in the parallel sparse QR factorization. Performance of the new parallel sparse triangular solution algorithms applied to regular grid problems is analyzed and their arithmetic and communication complexities are presented.

The parallel algorithms for sparse triangular solution and sparse matrix-vector multiplication together with a parallel sparse QR factorization algorithm proposed in [19] result in a highly efficient block-oriented approach to the parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. Since the QR factorization method for solving linear least squares problems involves only QR factorization and triangular solution, the parallel algorithms presented in this paper can be obviously used to implement the QR factorization method in parallel as well.

In §2, multifrontal sparse QR factorization is reviewed. New parallel sparse triangular solution algorithms are proposed in §3. Block-oriented parallel dense triangular solvers are described in §4. Complexity results for regular grid problems are presented in §5. Experimental results obtained on an IBM SP2 machine are discussed in §6. Finally, concluding remarks are contained in §7.

2. Parallel multifrontal sparse QR factorization. Typically, the first task in a direct method for solving sparse linear least squares problems is to compute a sparse QR factorization. The multifrontal method has proved to be effective for sparse QR factorization [6, 9, 12, 14]. Parallel implementations of multifrontal sparse QR factorization have been discussed in [3, 15, 19]. Sparse QR factorization involves the following steps:

1. Find a permutation matrix P such that AP has a sparse upper triangular factor R.
2. Determine the symbolic structure of R.
3. Perform numerical factorization—i.e., compute the numerical values of the nonzeros of R.

Step 1 and step 2 constitute the symbolic phase of the sparse QR factorization, and step 3 is the corresponding numeric phase. Assume that the columns of A have been permuted by P determined in step 1. Since A has full column rank, A^tA is symmetric and positive definite. Let LL^t be the Cholesky factorization of A^tA. Then L^t is equal to the upper triangular factor of A, apart from possible sign differences in the rows. The *elimination tree* [13] of R is defined to be the structure with N nodes $\{1, 2, \cdots, N\}$ such that node j is the parent of node i if and only if

$$j = \min\{k > i \mid r_{ik} \neq 0\}.$$

Note that a node in the elimination tree corresponds to a row in R. The reordering and symbolic factorization algorithms described in [5] can be applied to A^tA to accomplish step 1 and step 2 stated above.

Let $\mathcal{R}_j$ denote the structure of row j of R—i.e., the set of column indices of nonzeros in row j of R. A *supernode* is defined to be a maximal set of contiguous rows $\{i, i+1, \cdots, j\}$ in R such that $\mathcal{R}_l = \{l\} \cup \mathcal{R}_{l+1}$ and l is the only child of $l+1$ in the elimination tree which is postordered for $i \le l < j$.

We illustrate the main ideas of multifrontal sparse QR factorization by a model problem. Consider a $k \times k$ regular grid with $(k-1)^2$ small squares. Associated with each square is a set of four equations involving the four variables at the corners of the square. The assembly of these equations results in a sparse overdetermined system of equations $Ax = b$, where A is a $4(k-1)^2$ by k^2 matrix. This model problem is motivated by the finite element method. A 3×3 regular grid with nested dissection ordering is shown in Fig. 1. The sparsity structures of A and its triangular factor R corresponding to the 3×3 grid are shown in Fig. 2. The elimination tree and the supernodal elimination tree are illustrated in Fig. 3, where **1**, **2**, $\cdots$, **7** represent the supernodes of R.

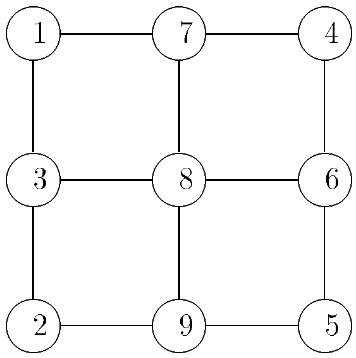

FIG. 1. *A 3×3 regular grid with nested dissection ordering*

In general, a supernode consists of a set of contiguous rows in R. Let $S = \{i, i+1, \cdots, j\}$ be a supernode consisting of rows $i, i+1, \cdots, j$ of R. The supernode S is associated with an $n \times n$ upper triangular matrix F_S called the *frontal matrix* of S, where $n = |\mathcal{R}_i|$. The first $|S| = j - i + 1$ rows of F_S are rows $i, i+1, \cdots, j$ of R and are referred to as the *factor rows* of S. The upper trapezoidal matrix formed by the factor rows is referred to as the *factor matrix* of S. The upper triangular matrix U_S formed by the non-factor rows of F_S is referred to as the *update matrix* of S. Let A_S denote the submatrix consisting of all rows of A whose first nonzeros are in columns $i, i+1, \cdots, j$ with zero columns removed.

For the example discussed above, let $F_{\mathbf{i}}$ and $U_{\mathbf{i}}$ be the frontal matrix and the update matrix of supernode $\mathbf{i}$ for $\mathbf{1} \le \mathbf{i} \le \mathbf{7}$. Consider how $F_{\mathbf{3}}$ is computed. The matrix $F_{\mathbf{1}}$ is formed as below:

$$A_{\mathbf{1}} = \begin{pmatrix} a_{11} & a_{13} & a_{17} & a_{18} \\ a_{21} & a_{23} & a_{27} & a_{28} \\ a_{31} & a_{33} & a_{37} & a_{38} \\ a_{41} & a_{43} & a_{47} & a_{48} \end{pmatrix} = Q_1 F_{\mathbf{1}} = Q_{\mathbf{1}} \begin{pmatrix} r_{11} & r_{13} & r_{17} & r_{18} \\ & u_{23} & u_{27} & u_{28} \\ & & u_{37} & u_{38} \\ & & & u_{48} \end{pmatrix}, \tag{4}$$

where $Q_{\mathbf{1}}$ is an orthogonal matrix. The first row of $F_{\mathbf{1}}$ is the first row of R. The matrix $F_{\mathbf{2}}$ is similarly obtained. Let $\overline{U}_{\mathbf{1}}$ denote the extension of $U_{\mathbf{1}}$ according to the sparsity structure of $F_{\mathbf{3}}$—i.e.,

$$\overline{U}_{\mathbf{1}} = \begin{pmatrix} u_{23} & u_{27} & u_{28} & 0 \\ & u_{37} & u_{38} & 0 \\ & & u_{48} & 0 \\ & & & 0 \end{pmatrix}. \tag{5}$$

$$\begin{array}{c}
\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\
\begin{bmatrix}
x & & x & & & & x & x & \\
x & & x & & & & x & x & \\
x & & x & & & & x & x & \\
x & & x & & & & x & x & \\
 & x & x & & & & & x & x \\
 & x & x & & & & & x & x \\
 & x & x & & & & & x & x \\
 & x & x & & & & & x & x \\
 & & & x & & x & x & x & \\
 & & & x & & x & x & x & \\
 & & & x & & x & x & x & \\
 & & & x & & x & x & x & \\
 & & & & x & x & & x & x \\
 & & & & x & x & & x & x \\
 & & & & x & x & & x & x \\
 & & & & x & x & & x & x
\end{bmatrix}
\end{array}
\qquad
\begin{array}{c}
\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\
\begin{bmatrix}
x & & x & & & & x & x & \\
 & x & x & & & & & x & x \\
 & & x & & & & x & x & x \\
 & & & x & & x & x & x & \\
 & & & & x & x & & x & x \\
 & & & & & x & x & x & x \\
 & & & & & & x & x & x \\
 & & & & & & & x & x \\
 & & & & & & & & x
\end{bmatrix}
\end{array}$$

FIG. 2. *Sparsity structures of a sparse matrix A and its upper triangular factor R*

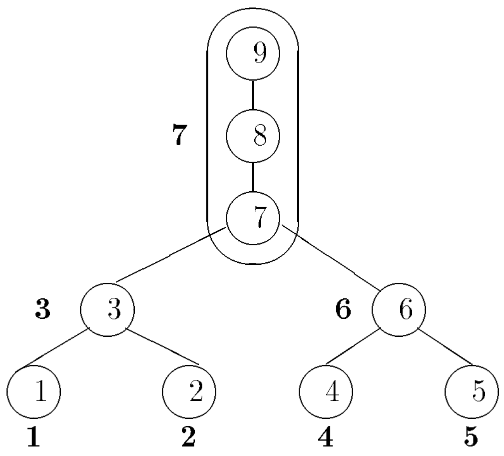

FIG. 3. *The elimination tree and the supernodal elimination tree of R in Fig.2*

The matrix $\overline{U}_2$ can be similarly defined. The frontal matrix F_3 is formed as follows:

$$\left(\begin{array}{c} \overline{U}_1 \\ \overline{U}_2 \\ \overline{A}_3 \end{array} \right) = Q_3 \left(\begin{array}{c} F_3 \\ 0 \end{array} \right), \tag{6}$$

where Q_3 is an orthogonal matrix and $\overline{A}_3$ is the extension of A_3 according to the sparsity structure of F_3. Note that $\overline{A}_3$ is actually empty in this example.

In summary, a multifrontal sparse QR factorization is decomposed into a set of computational tasks based on the supernodal elimination tree of R. A task is to form the frontal matrix of a supernode and obtain a set of rows of R associated with the supernode. Computation starts with the leaves of the tree and progresses toward the root of the tree. Disjoints subtrees can be processed independently. A serial algorithm for multifrontal sparse QR factorization is illustrated in Fig. 4.

---

for each supernode S in a topological ordering **do**

let the children of S be $X, Y, \cdots, Z$.

$$\left(\begin{array}{c} \overline{U}_X \\ \overline{U}_Y \\ \vdots \\ \overline{U}_Z \\ \overline{A}_S \end{array} \right) = Q_S \left(\begin{array}{c} F_S \\ 0 \end{array} \right), \tag{7}$$

where Q_S is an orthogonal matrix.

end for

FIG. 4. *A serial multifrontal QR factorization algorithm*

In a parallel setting, the computational tasks are mapped onto node processors by the proportional mapping scheme [16]. Assume four processors $\{\mu_0, \mu_1, \mu_2, \mu_3\}$ are used for solving the problem discussed above. Then the task of forming F_7 is partitioned among the four processors. Computation of F_3 is partitioned among μ_0 and μ_1. Computation of F_6 is partitioned among μ_2 and μ_3. The frontal matrices F_1, F_2, F_4 and F_5 are computed by μ_0, μ_1, μ_2 and μ_3, respectively.

If a task associated with a supernode S is entirely mapped to a processor μ, then the frontal matrix F_S is entirely allocated to μ. If a task associated with a supernode S is partitioned among a set of p processors, then the frontal matrix F_S is partitioned among the set of processors by the *equal-row partioning scheme* or the *equal-volume partitioning scheme* introduced in [19]. Briefly, the rows of F_S are partitioned into $k \geq p$ blocks such that every block consists of a set of contiguous rows of F_S. When the equal-row partioning scheme is used, every block contains n/k rows of F_S. When the equal-volume partitioning scheme is used, every block contains approximately $|F_S|/k$ numerical values of F_S.

The symbolic phase of the sparse QR factorization involves graph-theoretic computations which are difficult to be parallelized. Efficient and practical parallel algorithms for the symbolic phase are not yet available. Fortunately, the symbolic phase usually takes a small portion of the overall execution time. Therefore, the symbolic phase is performed in a serial manner. In contrast, the numeric phase dominates the execution time for solving a sparse linear least squares problem and must be processed in parallel. Our parallel implementation of the numeric phase is based on a block-oriented parallel multifrontal sparse QR factorization algorithm proposed in [19]. At the heart of our parallel sparse QR factorization is the parallel implementation of the numerical kernel (7) in Fig. 4, which is described in detail in [19].

3. Parallel multifrontal sparse triangular solution. In this section we describe a new parallel multifrontal algorithm for solving the sparse triangular systems $R^tRx = c$, a crucial step for the CSNE method. A sparse forward substitution or sparse back substitution is decomposed into a sequence of solutions of dense trapezoidal systems. The overall sparse triangular solution is accomplished by processing all dense trapezoidal systems in a multifrontal framework. The main advantages of the multifrontal method such as data locality and dense matrix computations are fully exploited in the sparse triangular solution. The storage scheme for R in the parallel sparse triangular solution is exactly the same as that in the parallel sparse QR factorization. No data redistribution is needed.

Let S denote a supernode consisting of rows $i_1, i_2, \cdots, i_m$ of R, where $i_{j+1} = i_j+1$ for $1 \le j < m$. Let $S_a = \{i_1, \cdots, i_m, i_{m+1}, \cdots, i_n\}$ denote the nonzero column indices of row i_1 of R in increasing order. Let $S_f = \{i_1, \cdots, i_m\}$ and $S_u = \{i_{m+1}, \cdots, i_n\}$. The variables associated with S_f and S_u are referred to as the *factor variables* and the *update variables* of S, respectively.

An *index vector* is defined as a vector of integers in increasing order. Clearly, S_a, S_f and S_u defined above are index vectors. A sparse vector is represented as a dense vector and an index vector. While the dense vector contains the numerical values of the non-zero entries of the sparse vector, the index vector contains the indices of the corresponding non-zero entries. We use $[val, ind]$ to denote a sparse vector, where val is the dense vector and ind is the corresponding index vector. Let $I = \{i_1, \cdots, i_\alpha\}$ and $J = \{j_1, \cdots, j_\beta\}$ be two index vectors and $I \subset J$. The *scatter* of a sparse vector $[v, I]$ to another sparse vector $[w, J]$ is defined as

$$[w, J] \leftarrow [w, J] \oplus [v, I], \tag{8}$$

where $w_h = w_h + v_k$ if $i_k = j_h$ for $1 \le k \le \alpha$. The other elements of w are not changed. The *gather* of a sparse vector $[w, J]$ to another sparse vector $[v, I]$ is defined as

$$[v, I] \leftarrow [v, I] \lhd [w, J], \tag{9}$$

where $v_k = w_h$ if $i_k = j_h$ for $1 \le k \le \alpha$.

3.1. Sparse forward substitution. Assume that supernode S is defined as above. The lower trapezoidal system associated with S is written as $T^tz = d$, where T is the factor matrix of S, z is a vector of length m to be determined, and d is a vector of length n. Partition T and d as

$$T = (T_f \;\; T_u) \;\text{ and }\; d = \begin{pmatrix} d_f \\ d_u \end{pmatrix},$$

where T_f is an $m \times m$ upper triangular matrix and d_f contains the first m components of vector d. The vector d_f is initialized with values of the right hand side vector c—i.e., $d_j = c_{i_j}$ for $1 \le j \le m$. The vector d_u is initialized to zero. If S is a leaf in the supernodal elimination tree, the system $T^tz = d$ is solved as follows:

$$T_f^t z = d_f \text{ and } d_u = d_u - T_u^t z. \tag{10}$$

The resulting vector d_u represents the effects on the update variables of S by solving the factor variables of S. The vector d_u is called the *effect vector* of S and is denoted by $e(S)$. If S is an interior supernode in the supernodal elimination tree, the vector d needs to be updated by the effect vectors from all children of S. Let C be a child of S. The update on the vector d by $e(C)$ can be described by the *scatter* operation defined in (8):

$$[d, S_a] \leftarrow [d, S_a] \oplus [e(C), C_u].$$

After d has been updated by effect vectors from all children of S, the associated trapezoidal system $T^t z = d$ is solved as in (10). The effect vector $e(S) = d_u$ represents the accumulated effects on the update variables of S by solving all factor variables in the subtree rooted at S.

Let $proc(S)$ be the set of processors among which the frontal matrix of S is partitioned. We assume that one processor in $proc(S)$ is responsible for scattering the effect vectors of the children of S into the right hand side vector of the lower trapezoidal system associated with S and this processor is denoted as $scatter(S)$. We also assume that one processor in $proc(S)$ stores $e(S)$ after the lower trapezoidal system associated with S is solved and this processor is denoted as $store\_e(S)$. The parallel multifrontal sparse forward substitution algorithm is presented in Fig. 5, where $\mathcal{N} = \{1, 2, \cdots, N\}$ and $parent(S)$ is the parent of S.

```
for each supernode S in a topological ordering do
    [d_f, S_f] ← [d_f, S_f] ◁ [c, N], d_u = 0;
    if proc(S) = {μ} then
        for each child C of S do [d, S_a] ← [d, S_a] ⊕ [e(C), C_u];
        solve T^t z = d;
        if S is not a root and μ ≠ scatter(parent(S)) then
            send e(S) to scatter(parent(S));
    else if μ ∈ proc(S) then
        if μ = scatter(S) then
            for each child C of S do
                if μ ≠ store_e(C) then receive e(C);
                [d, S_a] ← [d, S_a] ⊕ [e(C), C_u];
            end for
        end if
        solve T^t z = d in parallel;
        if μ = store_e(S) and S is not a root and μ ≠ scatter(parent(S)) then
            send e(S) to scatter(parent(S));
    end if
end for
```

FIG. 5. *A parallel multifrontal sparse forward substitution algorithm on a processor μ.*

3.2. Sparse back substitution. The upper trapezoidal system associated with supernode S is described as $Ty = z$, where T is the factor matrix of S, y is a vector of length n, and z is the solution vector to the lower trapezoidal system of S computed in the sparse forward substitution phase. Partition T and y as

$$T = (T_f \quad T_u) \quad \text{and} \quad y = \begin{pmatrix} y_f \\ y_u \end{pmatrix},$$

where T_f is an $m \times m$ upper triangular matrix and y_f contains the first m components of vector y. If S is not a root, the vector y_u is determined from the solution vector to the upper trapezoidal system associated with the parent of S. This is accomplished by the *gather* operation defined in (9):

$$[y_u, S_u] \leftarrow [y_u, S_u] \lhd [y_J, J],$$

where $J = parent(S)$ and y_J is the solution to the upper trapezoidal system associated with J.

We assume that one processor in $proc(S)$ stores the solution vector to the upper trapezoidal system of S and this processor is denoted as $store\_s(S)$. We also assume that one processor in $proc(S)$ is responsible for gathering solution components corresponding to the update variables of the supernode S from the solution vector to the upper trapezoidal system of the parent of S and this processor is denoted as $gather(S)$. The parallel multifrontal sparse back substitution algorithm is described in Fig. 6.

```
for each supernode S in the reverse order of a topological ordering do
    if proc(S) = {μ} then
        if S is not a root then
            if μ ≠ store_s(J) then receive y_J, where J = parent(S);
            [y_u, S_u] ← [y_u, S_u] ◁ [y_J, J_a];
        end if
        solve Ty = z;
    else if μ ∈ proc(S) then
        if μ = gather(S) then
            if S is not a root then
                if μ ≠ store_s(J) then receive y_J, where J = parent(S);
                [y_u, S_u] ← [y_u, S_u] ◁ [y_J, J_a];
            end if
        end if
        solve Ty = z in parallel;
        if μ = store_s(S) then
            for each child C of S do
                if μ ≠ gather(C) then send y to gather(C);
        end if
    end if
end for
```

FIG. 6. *A parallel multifrontal sparse back substitution algorithm on a processor μ.*

4. Block-oriented parallel dense triangular solution algorithms. A number of parallel algorithms have been designed for solving dense triangular systems of linear equations on distributed-memory multiprocessors [1, 4, 8, 10, 11, 17]. Most of the previous works on parallel triangular solution have assumed that a triangular matrix is distributed to processors by **single** columns or rows.

A crucial step in the parallel implementation of the CSNE method is to solve $R^t R x = c$. In the parallel multifrontal method described in §3, the solution of $R^t y = c$ or $Rx = y$ is decomposed into a number of solutions of dense triangular systems. As shown in Fig. 5 and Fig. 6, the performance of the parallel sparse triangular solution is determined by the parallel dense triangular solvers used for accomplishing line 15 in Fig. 5 and line 15 in Fig. 6. In our approach, a triangular matrix is distributed to a set of processors by **blocks** of columns or rows. This block-oriented data-to-processor mapping is used in our parallel sparse QR factorization and is fixed prior to the parallel sparse triangular solution phase.

In this section, we propose new parallel algorithms for solving dense triangular systems. We demonstrate that our block-oriented parallel triangular solvers achieve significant improvement in performance over the conventional non-block approach. We apply our block-oriented parallel triangular solvers to the multifrontal solution of $R^tRx = c$ and obtain a highly efficient parallel implementation of the CSNE method for solving sparse linear least squares problems.

4.1. Parallel solution of dense lower triangular systems. Let L be an $n \times n$ lower triangular matrix. Assume that X and B are vectors of length n. Consider the parallel solution of $LX = B$ on a set of p processors $\{\mu_0, \mu_1, \ldots, \mu_{p-1}\}$. We partition $LX = B$ as follows:

$$\begin{bmatrix} L_{0,0} & & & \\ L_{1,0} & L_{1,1} & & \\ \vdots & \vdots & \ddots & \\ L_{m,0} & L_{m,1} & \cdots & L_{m,m} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_m \end{bmatrix} = \begin{bmatrix} B_0 \\ B_1 \\ \vdots \\ B_m \end{bmatrix},$$

where $L_{i,j}$ $(i > j)$ is an $h_i \times h_j$ rectangular block, and $L_{i,i}$ is an $h_i \times h_i$ lower triangular block. The blocks X_i and B_i are vectors of length h_i. Note that $n = \sum_{i=0}^{m} h_i$.

As fixed by our parallel sparse QR factorization [19], a dense lower triangular system $LX = B$ arising in solving the sparse lower triangular system $R^t y = c$ is partitioned into blocks of columns. Specifically, the lower triangular matrix L is partitioned as $L = [L_{*,0} \quad L_{*,1} \quad \ldots \quad L_{*,m}]$, where

$$L_{*,i} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ L_{i,i} \\ \vdots \\ L_{m,i} \end{bmatrix}$$

for $0 \leq i \leq m$. The block of columns $L_{*,i}$ and vector B_i are assigned to the same processor which is denoted by $map[i]$. The block mapping vector map is available on every participating processor. A block-oriented parallel algorithm for solving $LX = B$ is shown in Fig. 7, where $myblocks$ is a set of blocks assigned to a processor, and fan_in$(V, map[i])$ assigns the sum $W = \sum_\mu V$ to $map[i]$.

Two strategies for partitioning a triangular matrix on p processors are used in our parallel sparse QR factorization. One is the *equal-row partitioning scheme* in which an $n \times n$ lower triangular matrix is partitioned into k blocks each of which contains $s = n/k$ columns. On the IBM SP2 machine, $s = \lceil 2n/(7p) \rceil$ produces optimal performance for the parallel sparse QR factorization.

Another strategy is the *equal-volume partitioning scheme* in which an $n \times n$ lower triangular matrix is partitioned into k blocks and the i^{th} block is an $\rho_i \times \gamma_i$ lower trapezoidal block, where

$$\rho_i \approx \sqrt{\frac{k-i}{k}}n \quad \text{and} \quad \gamma_i \approx \frac{\sqrt{k-i} - \sqrt{k-i-1}}{\sqrt{k}}n. \tag{11}$$

On the IBM SP2 machine, $k = n/\lceil 2n/(7p) \rceil$ produces optimal performance for the parallel sparse QR factorization.

$$
\begin{array}{l}
\textbf{for } i = 0 \textbf{ to } m \textbf{ do} \\
\quad V = 0; \\
\quad \textbf{for } j \in myblocks,\ j < i \textbf{ do} \\
\quad\quad V = V + L_{i,j} X_j; \\
\quad W = \text{fan\_in}(V,\ map[i]); \\
\quad \textbf{if } i \text{ in } myblocks \textbf{ then} \\
\quad\quad X_i = L_{i,i}^{-1}(B_i - W); \\
\textbf{end for}
\end{array}
$$

FIG. 7. *A block-oriented parallel dense forward substitution algorithm*

Performance results of our parallel lower triangular solver with equal-row partitioning scheme are shown in Fig. 8. Since a block wrap mapping is used in the parallel sparse QR factorization, the same block wrap mapping is used here—i.e., $map[i] = \mu_j$, where $j = i \bmod p$. Block sizes equal to $1, 2, \cdots, 50$ are examined. The special case $s = 1$ gives the worst performance. For fixed p and n, the running time initially decreases rapidly as the block size increases. Once the best performance is achieved, the running time remains relatively constant as the block size increases. Hence it is not necessary to determine the optimal block size exactly.

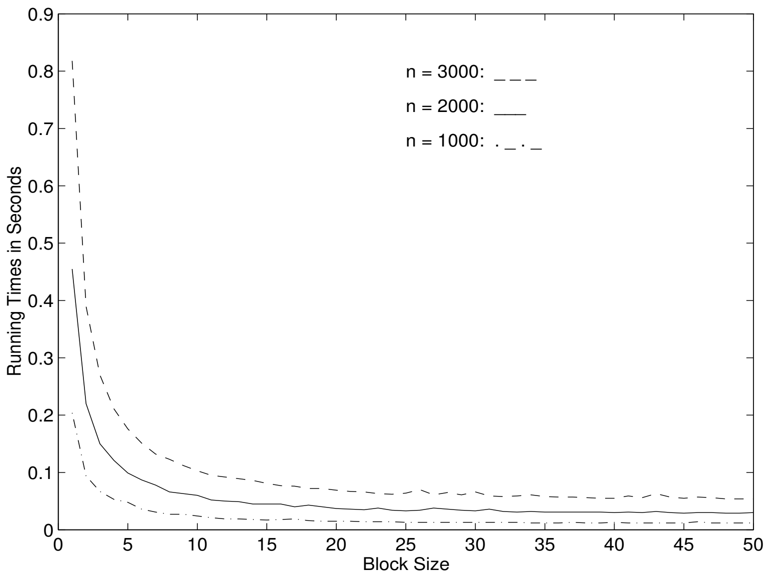

FIG. 8. *Performance of a parallel dense lower triangular solver on an IBM SP2($p = 16$)*

For $p = 16$ and $n = 1000, 2000, 3000$, the block sizes used in the parallel sparse QR factorization are $18, 36$ and 54, respectively. They are in the range of block sizes which give optimal or nearly optimal performance for our parallel dense lower triangular solver. The same pattern is observed for other values of p and n. In other words, the block partitions which produce the optimal performance for the parallel sparse QR factorization also produce the optimal or nearly optimal performance for the block-oriented dense lower triangular solver. Comparison with the cyclic algorithm [10] is shown in Table 1, where "time" is the execution time and "mflops" is the number of mega flops executed per second.

The above discussions on the parallel lower triangular solver with equal-row partitioning scheme also apply to the parallel lower triangular solver with equal-volume partitioning scheme. Details on the equal-volume partitioning scheme are omitted.

TABLE 1
Performance comparison of two lower triangular solvers on an IBM SP2(p=16)

	Cyclic		Block		
n	time	mflops	s	time	mflops
1000	0.077	12.994	18	0.016	64.078
2000	0.132	30.200	36	0.031	129.136
3000	0.181	49.713	54	0.053	171.269

We now analyze the complexity of the parallel lower triangular solver. Consider an $n \times n$ lower triangular matrix which is partitioned into k blocks with block size $s = n/k$. For simplicity, we assume that n is a multiple of k. Each step of the algorithm requires no more than $\lceil i/p \rceil$ products of $s \times s$ matrix and $s \times 1$ vector and solution of an $s \times s$ lower triangular system. Therefore, the algorithm requires no more than

$$\sum_{i=0}^{k-1} (\lceil \frac{i}{p} \rceil 2s^2 + s^2) = 2s^2 \sum_{i=1}^{k-1} \lceil \frac{i}{p} \rceil + ks^2$$

floating-point operations. Let $k = qp + r$, where q and r are non-negative integers with $0 < r < p$. It has been shown in [17] that

$$\sum_{i=0}^{k-1} (\lceil \frac{i}{p} \rceil = \frac{1}{2p}(k^2 + kp - 2k + pr + 2r - 2p - r^2).$$

Since $n = ks$ and $s \leq n/p$, it can be easily derived that the number of arithmetic operations required by the algorithm is bounded above by $4n^2/p$.

Each step of the algorithm requires a fan-in operation. The number of floating-point numbers communicated by a processor is bounded above by $k(p-1)s = n(p-1)$.

4.2. Parallel solution of dense upper triangular systems. Let U be an $n \times n$ upper triangular matrix. Assume that X and B are vectors of length n. Consider the parallel solution of $UX = B$ on a set of p processors $\{\mu_0, \mu_1, \ldots, \mu_{p-1}\}$. We partition $UX = B$ as follows:

$$\begin{bmatrix} U_{0,0} & U_{0,1} & \cdots & U_{0,m} \\ & U_{1,1} & \cdots & U_{1.m} \\ & & \ddots & \vdots \\ & & & U_{m,m} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_m \end{bmatrix} = \begin{bmatrix} B_0 \\ B_1 \\ \vdots \\ B_m \end{bmatrix},$$

where $U_{i,j}$ $(i < j)$ is an $n_i \times n_j$ rectangular block and $U_{i,i}$ is an $n_i \times n_i$ upper triangular block. The blocks X_i and B_i are vectors of length n_i.

Again, as fixed by our parallel sparse QR factorization [19], a dense upper triangular system $UX = B$ arising in solving the sparse upper triangular system $Rx = y$ is partitioned into blocks of rows. Specifically, the upper triangular matrix U is partitioned as

$$U = \begin{bmatrix} U_{0,*} \\ U_{1,*} \\ \vdots \\ U_{m,*} \end{bmatrix},$$

where $U_{i,*} = [0 \ \ldots 0 \ U_{i,i} \ \ldots \ U_{i,m}]$ for $0 \le i \le m$. The block of rows $U_{*,i}$ and vector B_i are assigned to the processor $map[i]$. A block-oriented parallel algorithm for solving $UX = B$ is shown in Fig. 9, where fan_out($X_j, map[j]$) broadcasts the vector X_j to other processors.

---

for $j = m$ **to** 0 **step** -1 **do**
  if $j \in myblocks$ **then**
    $X_j = U_{j,j}^{-1} B_j$;
  fan_out(X_j, $map[j]$);
  for $i \in myblocks$, $i < j$ **do**
    $B_i = B_i - U_{i,j} X_j$;
end for

FIG. 9. *A block-oriented parallel dense back substitution algorithm*

Performance results of our parallel upper triangular solver with equal-row partitioning scheme are shown in Fig. 10. As in the lower triangular solver, a block wrap mapping is used. Block sizes equal to $1, 2, \cdots, 50$ are examined. Discussions and conclusions on the parallel dense upper triangular solver are the same as those on the parallel dense lower triangular solver, and they are omitted. Comparison with the cyclic algorithm [10] is shown in Table 2. It can be easily shown that the arithmetic and communication complexities for the upper triangular solver are the same as those for the lower triangular solver.

TABLE 2
Performance comparison of two upper triangular solvers on an IBM SP2(p=16)

	Cyclic		Block		
n	time	mflops	s	time	mflops
1000	0.082	12.154	18	0.009	114.352
2000	0.135	29.656	36	0.021	190.849
3000	0.194	46.444	54	0.032	280.444

5. Analysis of regular grid problems. The complexity analysis of our parallel sparse QR factorization algorithm on regular grid problem is provided in [19]. In this section, we present the arithmetic and communication complexities of the parallel sparse triangular algorithms described in §3 on regular grid problem.

Consider the supernodal elimination tree corresponding to a $k \times k$ regular grid with nested dissection ordering, where $k = 2^l - 1$ and l is a positive integer. Let $\tau_j = 2^{l-j} - 1$. The characterization of a heavest path in the supernodal elimination tree is given in Table 3, where n_i is the size of the frontal matrix on the heavest path at level i, m_i is the number of factor rows in the

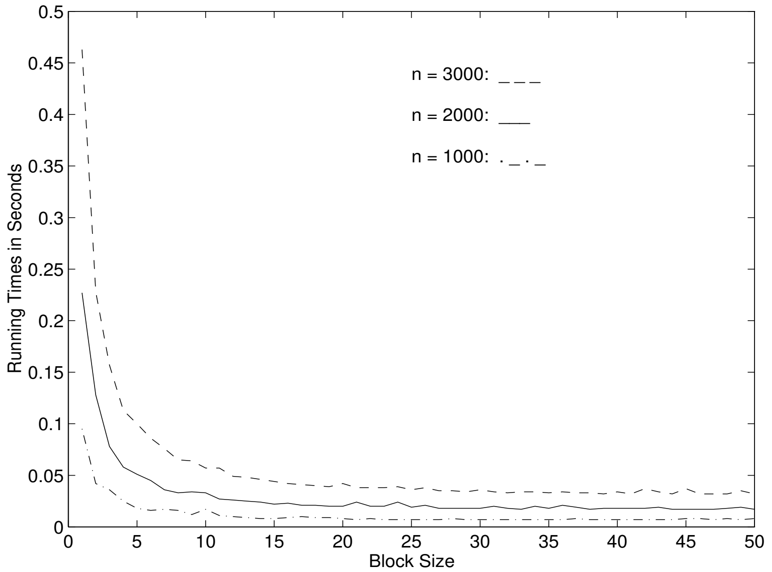

FIG. 10. *Performance of a parallel dense upper triangular solver on an IBM SP2($p = 16$)*

TABLE 3
Characterization of a heaviest path

level i	n_i	m_i	p_i
1	τ_0	τ_0	p
2	$\tau_1 + \tau_0$	τ_1	$p/2$
3	$\tau_1 + \tau_0$	τ_1	$p/4$
4	$6\tau_2 + 4$	τ_2	$p/8$
$2j + 1(j \geq 2)$	$5\tau_j + 4$	τ_j	$p/4^j$
$2j(j \geq 3)$	$7\tau_j + 6$	τ_j	$2p/4^j$

frontal matrix and p_i is the number of processors assigned to a supernode at level i. The root is considered to be at level 1. The total number of levels is $2l - 1$. Assume that the total number of processors $p = 4^d (d \geq 3)$. The number of processors assigned to a supernode at level i is $p/2^{i-1}$ for $1 \leq i \leq 2d$. A subtree rooted at a supernode at level $2d$ is entirely mapped to one processor.

Let ρ denote the ratio of the time for transmitting one floating-point number from one processor to another processor to the time for one floating-point operation. Then the arithmetic and communication complexities of the sparse forward substitution are $\sum_{i=1}^{2l-1} w_i$ and $\sum_{i=1}^{2d-1} c_i$, respectively, where

$$w_i < 2n_i^2/p_i \quad \text{and} \quad c_i \leq (p_i - 1)n_i\rho + 2(n_i - m_i)\rho.$$

The term $2(n_i - m_i)\rho$ in c_i represents the possible cost for communicating two effect vectors. The overall complexity results are given in Table 2. Notice that the complexity results of the sparse back substitution are the same as those of the sparse forward substitution.

6. Experimental results. To complete the parallel implementation of the CSNE method, $A^t b$ and $A^t(b - Ax)$ must be formed in parallel. The algorithm for computing $A^t(b - Ax)$ is dictated by the storage scheme for A. In our implementation, the matrix A is distributed among processors by rows. If the j^{th} row of R is mapped to processor μ, then all rows from A whose first nonzeros are in column j are mapped to processor μ. If the i^{th} row of A is assigned to processor μ, the

Table 4
Arithmetic and communication complexities

	Arithmetic	Communication
Forward Substitution	$99\frac{k^2 \log p}{p} - 168\frac{k^2}{p} + O(k)$	$\frac{545}{112}kp\rho + O(k\rho)$
Back Substitution	$99\frac{k^2 \log p}{p} - 168\frac{k^2}{p} + O(k)$	$\frac{545}{112}kp\rho + O(k\rho)$

corresponding component b_i of the right hand side vector b is also assigned to processor μ. Let A_μ denote the submatrix consisting of all rows of A assigned to processor μ and b_μ the vector consisting of all components of b assigned to processor μ.

We assume that there are p processors in the system which are numbered $0, 1, \cdots, p-1$. The solution x computed in the previous iteration is available to all processors. Processor μ computes $r_\mu = b_\mu - A_\mu x$, where r_μ is the portion of the residual vector r stored on processor μ. Since $A^t r = \sum_{\mu=0}^{p-1} A_\mu^t r_\mu$, the product $A^t r$ can be obtained by a global summation after processor μ forms $A_\mu^t r_\mu$ locally for $0 \le \mu < p$. The product $A^t b = \sum_{\mu=0}^{p-1} A_\mu^t b_\mu$ can be similarly computed.

Our parallel algorithms have been tested on an IBM SP2 machine for a collection of large-scale structured and unstructured problems. We report our experimental results for regular grid problems, problems arising from particle methods for modelling turbulent combustion and an unstructured problem RAEFSKY3. The sparse least squares problems for modelling turbulent combustion correspond to three-dimensional $k \times k \times k$ grids. There are a number of particles associated with each cubic element. A particle corresponds to an equation involving the eight variables at the corners of the cubic element. Assume that a cubic element contains eight particles. The assembly of the equations corresponding to all particles results in a sparse overdetermined system of equations $Ax = b$, where A is an $8(k-1)^3$ by k^3 sparse matrix. The regular grid problems and the three-dimensional grid problems are ordered by nested dissection ordering. The problem RAEFSKY3 is one of the large sparse matrices maintained by Tim Davis and can be obtained by anonymous ftp from `ftp.cis.ufl.edu`. It is unsymmetric and has general sparsity structure. The problem RAEFSKY3 is ordered by minimum degree ordering.

Table 5
Characteristics of the test problems

Problem	M	N	$\|A\|$	$\|R\|$	Megaflops
RAEFSKY3	21,200	21,200	1,488,768	12,175,976	30,217
GRID300	3,57,604	90,000	1,430,416	3,734,104	1,655
GRID500	996,004	250,000	3,984,016	11,709,081	7,530
CUBE27	140,608	19,683	1,124,864	4,665,657	8,029
CUBE40	474,552	64,000	3,796,416	24,626,969	83,889
CUBE50	941,192	125,000	7,529,536	62,819,416	322,646
CUBE60	1,643,032	216,000	13,144,256	133,958,505	954,870
CUBE65	2,097,152	274,625	16,777,216	186,600,885	1,547,438

Our experiment is to solve these sparse linear least squares problems by the CSNE method. The characteristics of the test problems are shown in Table 5, where M is the number of rows, N the number of columns, $|A|$ the number of nonzeros in matrix A, $|R|$ the number of nonzeros in factor R. "GRIDk" represents a $k \times k$ grid. "CUBEk" represents a three-dimensional $k \times k \times k$ grid. "Megaflops" is the actual number of megaflops performed in the multifrontal sparse QR factorization of A.

Table 6
Running times in seconds of the parallel algorithms on an IBM SP2

problem	p	qr_time	mflops	fe_time	bs_time	mv_time	gs_time	speed_up
RAEFSKY3	4	140.688	166	0.339	0.283	0.055	0.030	
	8	83.264	314	0.206	0.166	0.030	0.036	
	16	49.561	576	0.137	0.115	0.017	0.041	
	32	29.321	1031	0.114	0.111	0.010	0.063	
GRID300	1	44.152	37	0.990	0.836	0.209	0.0	1.00
	2	22.232	74	0.512	0.435	0.115	0.071	1.99
	4	11.390	145	0.273	0.230	0.058	0.128	3.88
	8	6.465	256	0.152	0.132	0.029	0.135	6.47
	16	3.778	438	0.104	0.086	0.029	0.170	11.69
	32	2.191	755	0.097	0.061	0.020	0.195	20.15
GRID500	4	43.052	175	0.761	0.603	0.170	0.339	
	8	25.099	300	0.424	0.373	0.093	0.360	
	16	13.862	543	0.266	0.210	0.069	0.398	
	32	7.764	970	0.184	0.147	0.050	0.473	
CUBE27	1	145.048	55	0.447	0.386	0.136	0.0	1.00
	2	73.013	110	0.231	0.198	0.064	0.015	1.99
	4	38.591	208	0.125	0.105	0.035	0.027	3.76
	8	19.910	403	0.072	0.062	0.018	0.033	7.29
	16	12.572	639	0.052	0.043	0.010	0.040	11.54
	32	7.137	1125	0.042	0.041	0.006	0.044	20.32
CUBE40	8	211.053	397	0.325	0.269	0.055	0.104	
	16	122.984	682	0.196	0.164	0.034	0.119	
	32	68.699	1221	0.134	0.121	0.018	0.131	
	64	39.611	2118	0.119	0.124	0.012	0.143	
CUBE50	32	255.945	1261	0.304	0.262	0.038	0.380	
	64	149.961	2152	0.234	0.254	0.024	0.286	
	128	99.758	3234	0.268	0.594	0.025	0.713	
CUBE60	64	424.921	2247	0.402	0.371	0.039	1.041	
	128	255.275	3740	0.455	0.457	0.029	2.379	
CUBE65	128	415.517	3724	0.757	0.812	0.039	2.693	

All programs are written in C and no assembler code is used. Double-precision floating-point arithmetic is employed. A flop is either a multiplicative or an additive operation. The number of flops performed by a processor is obtained by counting the actual number of flops performed by that processor.

The running times on an IBM SP2 machine are shown in Table 6. The "qr_time", "fe_time", "bs_time", "mv_time" are the running times in seconds for numerical factorization, forward substitution, back substitution, and the local portion of the matrix-vector product $A^t(b - Ax)$, respectively. The "gs_time" is the time spent on global summation for computing the matrix-vector product $A^t(b - Ax)$. The last column represents the speed-ups for those problems which are small enough to be run on a single processor. The SP2 nodes used in our experiments are "thin" nodes which are roughly equivalent to RS/6000 model 390. A thin node has 66.7 MHz clock speed, 64 Kbytes data cache, 64 bit memory bus and 128 Mbytes of memory space.

A running time is obtained by measuring the time spent on each processor and taking the maximum time spent on a processor. Due to insufficient storage space on node processors, some test problems can not be run on small number of processors. The running time for a problem on one processor is the time spent by the best serial algorithm we have for that problem on one processor.

The performance reults shown in Table 6 are obtained with the equal-volume partitioning scheme. The performance of the equal-row partitioning scheme is slightly worse than that of the equal-volume partitioning scheme. The efficiency of the overall sparse triangular solution including both forward substitution and back substitution is demonstrated by the fact that the triangular solution time is a very small fraction of the execution time of the highly efficient numerical factorization phase.

The numerical accuracy of the CSNE method is illustrated in Table 7. The numerical values of our test matrices are randomly generated values in (-1,1). The true solution x is given as $x_i = 2.0 + (i-1)/1000.0$ for $(1 \leq i \leq N)$ and the right hand side b is set to Ax. In Table 7, $\delta x = x - \bar{x}$, where $\bar{x}$ is the computed solution. The four rows of results for each problem represent the solution of the semi-normal equations and results of three iterative refinement steps. In practice, maximal accuracy is often achieved within 1–3 iterative refinement steps [14].

TABLE 7
Numerical accuracy

problem	$\|\delta x\|_1$	$\|\delta x\|_2/\|x\|_2$	$\|\delta x\|_\infty$
	1.6723e-09	5.2781e-16	2.9843e-13
GRID300	1.1723e-10	6.6784e-17	4.2633e-14
	3.6702e-11	3.5425e-17	2.8422e-14
	1.8918e-11	2.5067e-17	2.8422e-14
	1.4873e-10	8.9103e-16	9.5923e-14
CUBE27	6.4135e-12	6.7688e-17	7.1054e-15
	1.3531e-12	2.9442e-17	3.5527e-15
	3.6526e-13	1.4910e-17	3.5527e-15

7. Concluding remarks. We have described an efficient block-oriented approach to the parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. Our approach is based on the method of corrected semi-normal equations. The required parallel sparse QR factorization is discussed in [19]. Central to our approach are highly efficient block-oriented parallel multifrontal algorithms for solving the related sparse triangular systems. The idea of solving sparse triangular systems in a multifrontal manner is also considered in the context of solving systems of sparse linear equations [18]. The parallel sparse QR factorization algorithm [19] and the parallel sparse back substitution algorithm described in section 3 can also be used to implement the QR factorization method [7] for solving sparse linear least squares problems as discussed in [20].

Acknowledgements. I would like to thank Professor Thomas F. Coleman for many discussions relating to this work and for his helpful comments on the manuscript.

REFERENCES

[1] R. H. Bisseling and J. G. G. van de Vorst, *Parallel triangular system solving on a mesh network of transputers*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 787–799.

[2] Å. Björck, *Stability analysis of the method of seminormal equations for linear least squares problems*, Linear Algebra Appl., 88/89 (1987), pp. 31–48.

[3] E. Chu and J. A. George, *Sparse orthogonal decomposition on a hypercube multiprocessor*, SIAM J. Mat. Anal. Appl., 11 (1990), pp. 453–465.

[4] S. C. Eisenstat, M. T. Heath, C. S. Henkel, and C. H. Romine, *Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 589–600.

[5] J. A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[6] ———, *Householder reflections versus Givens rotations in sparse orthogonal decomposition*, Linear Algebra and its Appl., 88/89 (1987), pp. 223–238.

[7] G. Golub, *Numerical methods for solving linear least squares problems*, Numer. Math., 7 (1965), pp. 206–216.

[8] M. T. Heath and C. H. Romine, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 558–588.

[9] J. G. Lewis, D. J. Pierce, and D. K. Wah, *Multifrontal Householder QR factorization*, Tech. Report ECA-TR-127, Boeing Computer Services, Seattle, WA, November 1989.

[10] G. Li and T. F. Coleman, *A parallel triangular solver for a hypercube multiprocessor*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 485–502.

[11] ———, *A new method for solving triangular systems on distributed-memory message-passing multiprocessors*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 382–396.

[12] J. W. H. Liu, *On general row merging schemes for sparse Givens transformations*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 1190–1211.

[13] ———, *The role of elimination trees in sparse factorization*, SIAM J. Mat. Anal. Appl., 11 (1990), pp. 134–172.

[14] P. Matstoms, *Sparse QR Factorization with Applications to Linear Least Squares Problems*, PhD thesis, Linköping University, Sweden, 1994.

[15] P. E. Plassmann, *Sparse Jacobian estimation and factorization on a multiprocessor*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., SIAM, Philadelphia, 1990, pp. 152–179.

[16] A. Pothen and C. Sun, *A mapping algorithm for parallel sparse Cholesky factorization*, SIAM J. Sci. Comput., 14 (1993), pp. 1253–1257.

[17] C. H. Romine and J. M. Ortega, *Parallel solution of triangular systems of equations*, Parallel Computing, 6 (1988), pp. 109–114.

[18] C. Sun, *Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors*, Tech. Report CTC92TR102, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, Aug. 1992.

[19] ———, *Parallel sparse orthogonal factorization on distributed-memory multiprocessors*, Tech. Report CTC93 TR162, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, Dec. 1993. (Revised, To appear in SIAM Journal on Scientific Computing).

[20] ———, *Parallel multifrontal solution of sparse linear least squares problems on distributed-memory multiprocessors*, Tech. Report CTC94TR185, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, July 1994.