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

CHUNGUANG SUN *

Abstract. We describe the issues involved in the design and implementation of efficient parallel algorithms for solving sparse linear least squares problems on distributed-memory multiprocessors. We consider both the QR factorization method due to Golub and the method of corrected semi-normal equations due to Björck. The major tasks involved are sparse QR factorization, sparse triangular solution and sparse matrix-vector multiplication. The sparse QR factorization is accomplished by a parallel multifrontal scheme recently introduced. New parallel algorithms for solving the related sparse triangular systems and for performing sparse matrix-vector multiplications are proposed. The arithmetic and communication complexities of our algorithms on regular grid problems are presented. Experimental results on an Intel iPSC/860 machine are described.

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. 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 for solving (1) due to Golub [9] is to reduce A to an upper triangular matrix R by an orthogonal matrix Q

$$Q^t A = \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{2}$$

and transform b by forming $Q^t b$. The solution to (1) is then obtained by solving the triangular system $Rx = c$, where c is the first N components of $Q^t b$. 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 and sparse, it is often too expensive to store Q in the main memory.

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

To avoid the explicit formation of $A^t A$, the semi-normal equations $R^t Rx = A^t b$ can be used to handle new right hand sides without storing Q, where R is computed as in (2). Björck [1] has shown that the error in the computed solution for the method of semi-normal equations(SNE) is about the same as for the method of normal equations. Furthermore, Björck [1] has proposed that a correction step or an iterative refinement step be added to the SNE method as follows.

$$\left\{ \begin{array}{c} r \leftarrow b - Ax, \\ R^t R \triangle x = A^t r, \\ x \leftarrow x + \triangle x. \end{array} \right. \tag{3}$$

* Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY 14853-3801.
This research was partially supported by the Cornell Theory Center, which receives major funding from the National Science Foundation and the IBM corporation, with additional support from New York State and members of its Corporate Research Institute.

This approach is referred to as the method of corrected semi-normal equations(CSNE). Björck [1] has shown that the CSNE method 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 new parallel algorithms which are based on the QR factorization method, the SNE method and the CSNE method for solving sparse linear least squares problems on distributed-memory multiprocessors. The major tasks involved are sparse QR factorization, sparse triangular solution and sparse matrix-vector multiplication. The sparse QR factorization is accomplished by a parallel multifrontal scheme described in [20]. A new parallel multifrontal algorithm for solving the related sparse triangular systems is proposed. The storage scheme for R in the numerical factorization is also used in the sparse triangular solution. A parallel approach for performing sparse matrix-vector multiplications in the SNE method and the CSNE method is introduced. The arithmetic and communication complexities of our algorithms on regular grid problems are presented. Numerical experiments on an Intel iPSC/860 machine are described.

In §2, multifrontal method for sparse QR factorization is reviewed. A new parallel sparse triangular solution algorithm is described in §3. A parallel sparse matrix-vector multiplication scheme is introduced in §4. Complexity results on regular grid problems are presented in §5. Experimental results are described in §6. Finally, concluding remarks are contained in §7.

2. Parallel multifrontal sparse QR factorization. Typically, the first step in a direct method for solving a sparse linear least squares problem is to compute a sparse QR factorization. Multifrontal method is shown to be effective for sparse QR factorization in [6, 11, 14, 15]. Parallel implementation of the multifrontal method is extensively discussed in literature(e.g. [2, 16, 20]). The parallel multifrontal sparse QR factorization algorithm proposed in [20] is used in this work to perform parallel sparse QR factorization. The parallel multifrontal sparse QR factorization algorithm consists of a symbolic phase and a numeric phase. The symbolic phase is done in a serial manner while the numeric phase is processed in parallel. Supernodal tree is used to organize the numeric computation. The reader is referred to [20] for details.

A supernode consists of a set of adjacent rows in R. Let $S = \{i, i+1, \cdots, j\}$ be a supernode consisting of rows $i, i+1, \cdots, j$ of R. S is associated with an upper trapezoidal matrix F called the *frontal matrix* of S. Without loss of generality, we assume that F is a dense $n \times n$ upper triangular matrix, where n is the number of nonzeros in row i of R. The first $|S| = j-i+1$ rows of F are rows $i, i+1, \cdots, j$ of R and are, therefore, referred to as the *factor rows* of S. The matrix consisting of the factor rows is referred to as the factor matrix of S.

A sparse QR factorization is decomposed into a set of computational tasks according to the supernodal tree structure. A task is to form the frontal matrix of a supernode. Computation starts with the leaves of the tree and progresses toward the root of the tree. Disjoints subtrees are processed in parallel. The computational tasks are statically mapped onto node processors before numerical factorization starts. The proportional mapping scheme [17] is used to map a general sparse problem and the subtree-to-subcube mapping [8] is used to map a regular grid problem.

If a task associated with a supernode S is entirely mapped to a processor π, then the frontal matrix of S is entirely allocated to π. If a task associated with a supernode S is partitioned among a set of p processors, then the frontal matrix F of the supernode S is partitioned among the set of processors by the equal-volume partitioning strategy introduced in [20]. In other words, the rows of F are partitioned into p blocks such that every block consists of a set of adjacent rows of F and contains approximately $|F|/p$ numerical values of F. In our implementation, the storage scheme for R used in the numerical factorization is also used in the sparse triangular solution.

3. Parallel multifrontal sparse triangular solution. Solving dense triangular systems efficiently on distributed-memory multiprocessors is a challenging task as shown in [3, 10, 12, 13] because of the sequential data dependency and the small amount of arithmetic work involved. It is even more challenging to solve sparse triangular systems efficiently on distributed-memory multiprocessors since the sparsity consideration makes the complicated techniques used in the parallel dense triangular solution algorithms difficult to be applied to solving sparse triangular systems. Sparse triangular solution in the context of solving sparse and square linear systems is discussed in [5, 19, 21].

In this section we describe a parallel multifrontal algorithm for solving $R^tRx = c$, a crucial step for the SNE or CSNE method. A sparse forward elimination or back substitution is decomposed into a sequence of solutions of dense trapezoidal systems according to the supernodal structure of R. The overall sparse triangular solution is accomplished by processing all dense trapezoidal systems in a multifrontal framework. The advantages of the multifrontal method such as data locality and dense matrix computations are effectively exploited in the sparse triangular solution.

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 sequence* is defined as a subset of $\mathcal{N} = \{1, 2, \cdots, N\}$ in increasing order. Obviously, S_a, S_f and S_u defined above are index sequences. A sparse vector is represented as a dense vector and an index sequence. While the dense vector contains the numerical values of the non-zero entries of the sparse vector, the index sequence 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 sequence. Let $I = \{i_1, \cdots, i_\alpha\}$ and $J = \{j_1, \cdots, j_\beta\}$ be index sequences. We assume that I is a subset of J. The *scatter* of a sparse vector $[v, I]$ to another sparse vector $[w, J]$ is defined as

$$[w, J] \leftarrow [w, J] \oplus [v, I], \tag{4}$$

where $w_h = w_h + v_k$ if $i_k = j_h$ for $1 \le k \le \alpha$. The *gather* of a sparse vector $[w, J]$ to another sparse vector $[v, I]$ is defined as

$$[v, I] \leftarrow [v, I] \lhd [w, J], \tag{5}$$

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

3.1. Sparse forward elimination. 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 supernode tree, the system $T^tz = d$ is solved as follows.

$$T_f^tz = d_f \text{ and } d_u = d_u - T_u^tz. \tag{6}$$

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 supernode 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 $e(C)$, the effect vector of C, can be described by the *scatter* operation defined in (4):

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

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 (6). The computed 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 supernode 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 elimination algorithm is presented in Fig. 1, where $parent(S)$ is the parent of S.

---

for each supernode S in a postorder **do**
  $[d_f, S_f] \leftarrow [d_f, S_f] \lhd [c, \mathcal{N}]$, $d_u = 0$;
  if $proc(S) = \{\pi\}$ **then**
    for each child C of S **do** $[d, S_a] \leftarrow [d, S_a] \oplus [e(C), C_u]$;
    solve $T^t z = d$;
    if S is not a root and $\pi \neq scatter(parent(S))$ **then**
      send $e(S)$ to $scatter(parent(S))$;
  else if $\pi \in proc(S)$ **then**
    if $\pi = scatter(S)$ **then**
      for each child C of S **do**
        if $\pi \neq store\_e(C)$ **then** receive $e(C)$;
        $[d, S_a] \leftarrow [d, S_a] \oplus [e(C), C_u]$;
      end for
    end if
    solve $T^t z = d$ in parallel;
    if $\pi = store\_e(S)$ and S is not a root and $\pi \neq scatter(parent(S))$ **then**
      send $e(S)$ to $scatter(parent(S))$;
  end if
end for

FIG. 1. *A parallel multifrontal sparse forward elimination algorithm on a processor π.*

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, a vector of length m, is the solution vector to the lower trapezoidal system of S computed in the sparse forward elimination phase. Partition T and y as

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

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 (5):

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

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. 2.

---

for each supernode S in the reverse order of a postorder **do**
    if $proc(S) = \{\pi\}$ **then**
        if S is not a root **then**
            if $\pi \neq store\_s(J)$ **then** receive y_J, where $J = parent(S)$;
            $[y_u, S_u] \leftarrow [y_u, S_u] \lhd [y_J, J_a]$;
        end if
        solve $Ty = z$;
    else if $\pi \in proc(S)$ **then**
        if $\pi = gather(S)$ **then**
            if S is not a root **then**
                if $\pi \neq store\_s(J)$ **then** receive y_J, where $J = parent(S)$;
                $[y_u, S_u] \leftarrow [y_u, S_u] \lhd [y_J, J_a]$;
            end if
        end if
        solve $Ty = z$ in parallel;
        if $\pi = store\_s(S)$ **then**
            for each child C of S **do**
                if $\pi \neq gather(C)$ **then** send y to $gather(C)$;
        end if
    end if
end for

FIG. 2. *A parallel multifrontal sparse back substitution algorithm on a processor π.*

3.3. Parallel solution of dense triangular systems. The parallel multifrontal approach described above is a general scheme that can be used to solve sparse triangular systems arising from multifrontal solution of sparse linear least squares problems and sparse symmetric positive definite systems on distributed-memory multiprocessors. For any implementation of the parallel multifrontal sparse triangular solution algorithm, how the dense triangular systems $T^tTy = d$ are solved in parallel would have major impact on the performance of the implementation, and the parallel solution of $T^tTy = d$ is dictated by the storage scheme for T. Now we look at a particular implementation based on the storage scheme described in [20] for solving $T^tTy = d$ in parallel.

Consider the parallel solution of $T^tz = d$ first. Let $p = |proc(S)|$. If $p = 1$, the system $T^tz = d$ is solved by one processor and no communication is involved. If $p > 1$, let T be partitioned into k blocks as

$$T = \begin{pmatrix} T_0 \\ T_1 \\ \vdots \\ T_{k-1} \end{pmatrix}, \tag{7}$$

where $k \leq p$. Assume that the number of rows in T_i and the number of elements in the first row of T_i are δ_i and λ_i, respectively. Let T_i be mapped to processor $\pi_i (0 \leq i < k)$. The system $T^tz = d$ is solved by processors $\pi_0, \pi_1, \cdots, \pi_{k-1}$ in a serial manner. Let $\pi_0 = scatter(S)$—i.e., processor π_0 is responsible for obtaining the vector d. Processor $\pi_i (0 < i < k)$ waits for a message of length λ_i from processor π_{i-1}, performs its portion of the forward elimination and sends a vector of length λ_{i+1} to processor π_{i+1} if $i < k-1$. The details are straightforward and they are omitted.

Now consider the parallel solution of $Ty = z$. If $p = 1$, the system $Ty = z$ is solved by one processor and no communication is involved. If $p > 1$, let T be partitioned as in (7). Let $\pi_{k-1} = gather(S)$—i.e., processor π_{k-1} is responsible for obtaining y_u and then sending y_u to other processors. First, a processor receives y_u and subtracts from its right hand side elements the contribution of the solution components in y_u. Second, processor π_i receives δ_j solution components from every processor π_j such that $0 \leq j < i$ and subtracts from its right hand side elements the contribution of the received δ_j solution components. Then processor π_i performs its portion of the back substitution, obtains δ_i solution components and send them to every processor π_j such that $i < j < k$. Once processor π_i sends out the δ_i solution components, processors $\pi_{i+1}, \pi_{i+2}, \cdots, \pi_{k-1}$ updates their local right hand side elements concurrently. This parallel dense triangular solution algorithm is straightforward and its details are omitted.

If we rearrange the storage scheme for T after the parallel sparse QR factorization, a different and possibly more efficient dense triangular solution algorithm described in [3, 10, 12, 13] can be used. In fact, the parallel cyclic dense triangular solution algorithm described in [12] is used to solve dense triangular systems arising from parallel solution of sparse symmetric positive definite systems discussed in [19].

4. Parallel sparse matrix-vector multiplication. To complete the parallel implementation of the SNE method or the CSNE method for solving sparse linear least squares problems, either A^tb or $r = b - Ax$ and A^tr should be formed in parallel, where x is the computed solution obtained in the previous iteration. The algorithm for computing $A^t(b - Ax)$ is dictated by the storage scheme for A. We assume that the columns of A have already been permuted by a suitable fill-reducing ordering.

In our approach the matrix A is distributed among processors by rows. If the j^{th} row of R is mapped to processor π, then all rows from A whose first nonzeros are in column j are mapped to processor π. If the i^{th} row of A is assigned to processor π, the corresponding component b_i of the right hand side vector b is also assigned to processor π. Let A_π denote the submatrix consisting of all rows of A assigned to processor π and b_π the vector consisting of all components of b assigned to processor π.

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 made available on every processor. Processor π computes $r_\pi = b_\pi - A_\pi x$, where r_π is the portion of the residual vector r stored in processor π. Since $A^t r = \sum_{\pi=0}^{p-1} A_\pi^t r_\pi$, the product $A^t r$ can be obtained by a global summation after processor π forms $A_\pi^t r_\pi$ locally for $0 \le \pi < p$. The product $A^t b = \sum_{\pi=0}^{p-1} A_\pi^t b_\pi$ can be similarly computed.

5. Analysis of regular grid problems. The theoretical analysis of the computation and communication complexities of the parallel sparse QR factorization algorithm on regular grid problems is provided in [20]. In this section, we present the computation and communication complexities of the parallel sparse triangular algorithm described in §3 on regular grid problems. In particular, our analysis focuses upon the implementation discussed in §3.3. Either a multiplicative operation or an additive operation is considered a flop. Communication volume is defined to be the total number of elements communicated among processors including both elements sent and elements received. Assume that an $n \times n$ upper triangular matrix F is partitioned into p blocks as

$$F = \begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{p-1} \end{pmatrix} \tag{8}$$

using the equal-volume partitioning strategy [20]. The block F_i is mapped to processor $\pi_i (0 \le i < p)$. Assume that the number of rows in F_i and the number of elements in the first row of F_i are δ_i and λ_i, respectively, for $0 \le i < p$. Then

$$\frac{\lambda_i(\lambda_i + 1)}{2} \approx \frac{n(n+1)}{2p}(p-i).$$

Hence

$$\lambda_i \approx \sqrt{\frac{p-i}{p}}n \quad \text{and} \quad \delta_i \approx \frac{\sqrt{p-i} - \sqrt{p-i-1}}{\sqrt{p}}n.$$

The system $F^t z = d$ is solved by p processors in a serial manner. The arithmetic complexity for solving $F^t z = d$ on p processors is n^2. The communication volume for solving $F^t z = d$ is

$$2\sum_{i=1}^{p-1} \lambda_i \le C(n,p), \quad \text{where} \quad C(n,p) = \sqrt{2}(p-1)n. \tag{9}$$

The arithmetic complexity for solving $Fy = z$ on p processors is

$$\sum_{i=0}^{p-1} \frac{\delta_i(\delta_i + 1)}{2} + \sum_{i=0}^{p-2} \delta_i \delta_{i+1}.$$

The above expression is derived from the fact that once processor π_i sends out the δ_i solution components, processors $\pi_{i+1}, \pi_{i+2}, \cdots, \pi_{p-1}$ updates their local right hand side elements concurrently. We have shown that the above expression is bounded by $W(n,p) = \phi(p)n^2/p + O(n)$, where

$$\phi(p) = \frac{1}{2}(1 + \sum_{i=0}^{p-2}(1 + 2\sqrt{(p-i-2)(p-i-1)} - 2\sqrt{(p-i-2)(p-i)})). \tag{10}$$

It is easy to show that ϕ is a non-descending function with very slow growth rate, $\phi(2) = 1$ and $\phi(p) < 6$ if $p \leq 10^6$. The communication volume for solving $Fy = z$ is

$$2\sum_{i=1}^{p-1} i\delta_i \leq C(n,p). \tag{11}$$

Inequalities (9) and (11) above can be easily established.

Consider the parallel solution of $T^t z = d$, where T is a submatrix consisting of the first $m \leq n$ rows of F. Assume that T is partitioned among $j \leq p$ processors $\pi_0, \pi_1, \cdots, \pi_{j-1}$. The arithmetic complexity for solving $T^t z = d$ is $2nm - m^2$. The communication volume is $2\sum_{i=1}^{j-1} \lambda_i \leq C(n,p)$. For the parallel solution of $Ty = z$, the arithmetic complexity is bounded by $W(n,p)$ and the communication volume is $2\sum_{i=1}^{j-1} i\delta_i + 2(j-1)\sum_{i=j}^{p-1} \delta_i \leq C(n,p)$.

Now we present the complexity analysis of our overall parallel sparse triangular solution algorithm on regular grid problems. Consider a $k \times k$ two-dimensional 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.

TABLE 1
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	$2p/4^j$

Consider the supernodal elimination tree corresponding to a $k \times k$ 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 1, 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 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 $p = 4^q (q \geq 3)$. The number of processors assigned to a supernode at level i is $p/2^{i-1}$ for $1 \leq i \leq 2q$. A subtree rooted at a supernode in level $2q$ is entirely mapped to one processor. The reader is referred to [18] for details on the characterization of the supernodal elimination tree associated with a regular grid.

Let μ 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 elimination are $\sum_{i=1}^{2l-1} w_i$ and $\sum_{i=1}^{2q-1} c_i$, respectively, where $w_i = 2n_i m_i - m_i^2$ and $c_i = \sqrt{2}(p_i - 1)n_i\mu + 4(n_i - m_i)\mu$. The term $4(n_i - m_i)\mu$ in c_i represents the possible communication between the processor assigned a supernode S on level i and the two processors assigned the children of S. Similarly, the arithmetic and communication complexities of the sparse back substitution are $\sum_{i=1}^{2l-1} \overline{w}_i$ and $\sum_{i=1}^{2q-1} \overline{c}_i$, respectively, where $\overline{w}_i = \phi(p_i)n_i^2/p_i + O(n_i)$ and $\overline{c}_i = \sqrt{2}(p_i - 1)n_i\mu + 4n_i\mu$. The term $4n_i\mu$ in $\overline{c}_i$ represents the possible communication between the processor assigned a supernode S on level i and the two processors assigned the children of S. The complexity results are tabulated in Table 2, where FE represents the sparse forward elimination phase while BS represents the sparse back substitution phase.

TABLE 2
Arithmetic and communication complexities

	arithmetic	communication
FE	$\frac{125}{24}k^2 + O(k)$	$\frac{545\sqrt{2}}{224}kp\mu + O(k\mu)$
BS	$\frac{99}{4}\frac{k^2\phi(p)\log p}{p} - 42\frac{k^2\phi(p)}{p} + O(k)$	$\frac{545\sqrt{2}}{224}kp\mu + O(k\mu)$

6. Experimental results. Our parallel algorithms have been tested on a 32-node iPSC/860 machine for a set of problems including regular grid problems and practical problems with irregular sparsity structure. The practical problems include sparse least squares problems arising from particle methods for modelling turbulent combustion and a large problem from the Bramley's test set. 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 [4]. The general sparse problem is ordered by minimum degree ordering [7].

TABLE 3
Characteristics of the test problems

problem	M	N	$\|A\|$	$\|R\|$
GRID100	39204	10000	156816	297953
GRID200	158404	40000	633616	1481803
GRID250	248004	62500	992016	2459325
GRID20.3	54872	8000	438976	1298284
GRID24.3	97336	13824	778688	2806944
GRID27.3	140608	19683	1124864	4665657
GRID28.3	157464	21952	1259712	5445708
NIMBUS	23871	1325	181972	136356

Our experiment is to solve these sparse linear least squares problems by the QR factorization method, the SNE method and the CSNE method. The characteristics of the test problems are shown in Table 3, 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. "GRIDk.3" represents a three-dimensional $k \times k \times k$ grid.

All programs are written in C and double-precision floating-point arithmetic is used. No assembler code other than calls to the Basic Math Library provided by Intel is used. 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. We have obtained the running times and communication statistics for all test problems. To save space, only the results for regular grid problems on 32 processors are presented.

TABLE 4
Message count per processor ($p = 32$)

Test Problem	Tri Sol	Total No. Msgs max	min	avg	No. Msgs Sent max	min	No. Msgs Recv max	min
GRID100	FE	14	2	6.94	6	1	9	1
	BS	55	2	34.12	37	1	41	1
GRID200	FE	14	2	7.00	6	1	9	1
	BS	56	2	35.88	37	1	42	1
GRID250	FE	14	2	7.06	6	1	9	1
	BS	57	2	37.69	37	1	43	1

In Table 4, messages communicated by each processor are tabulated. The "max", "min", and "avg" under the "Total No. Msgs" are the maximum number, minimum number and average number of messages communicated by a processor including both messages sent and messages received, respectively. The "max" and "min" under the "No. Msgs Sent" are the maximum number and minimum number of messages sent by a processor, respectively. The "max" and "min" under the "No. Msgs Recv" are with respect to number of messages received by a processor. The average number of messages sent by a processor is the same as the average number of messages received by a processor, and is also equal to half of the average number of messages communicated by a processor. Note that the number of messages per processor for regular grid problems increases very slowly as the problem size increases.

TABLE 5
Communication volume per processor($p = 32$).

Test Problem	Tri Sol	Total Volume max	min	avg	Volume Sent max	min	Volume Recv max	min
GRID100	FE	10376	728	4937.50	4376	424	6256	304
	BS	12160	424	3382.00	10976	16	3880	16
GRID200	FE	20856	1456	9986.00	8800	848	12552	608
	BS	24280	848	6883.50	21832	40	7808	40
GRID250	FE	26064	1808	12548.50	11000	1056	15680	752
	BS	30360	1056	8669.00	27264	56	9768	56

In Table 5, volume of messages communicated by each processor is shown. The total volume communicated by a processor is defined as the total number of floating-point numbers communicated by that processor. The volume sent by a processor is defined as the total number of floating-point numbers sent by a processor. The volume received is similarly defined. The "max", "min", and "avg" under the "Total Volume" are the maximum, minimum and average volume communicated by a processor including both messages sent and messages received, respectively. The "max" and "min" under the "Volume Sent" are the maximum and minimum volume sent by a processor, respectively. The "max" and "min" under the "Volume Recv" are with respect to volume received by a processor. The average volume sent by a processor is the same as the average volume received by the processor, and is also equal to half of the average volume communicated by the processor.

TABLE 6

Performance of our parallel algorithms on an iPSC/860

problem	p	qr_time	fe_time	bs_time	mv_time	gs_time
GRID100	1	16.569	0.429	0.392	0.169	0
	2	8.397	0.217	0.202	0.086	0.033
	4	4.408	0.119	0.108	0.044	0.101
	8	2.557	0.073	0.066	0.022	0.192
	16	1.519	0.055	0.044	0.012	0.263
	32	0.962	0.054	0.038	0.006	0.292
GRID200	8	13.896	0.282	0.243	0.089	0.670
	16	8.313	0.189	0.147	0.046	1.056
	32	5.112	0.155	0.106	0.025	1.195
GRID250	32	8.622	0.223	0.148	0.040	1.965
GRID20.3	8	30.735	0.176	0.115	0.055	0.130
	16	18.348	0.159	0.083	0.028	0.211
	32	11.325	0.173	0.073	0.014	0.234
GRID24.3	16	49.109	0.301	0.146	0.048	0.390
	32	30.026	0.309	0.120	0.024	0.480
GRID27.3	32	56.232	0.435	0.162	0.036	0.640
GRID28.3	32	68.832	0.497	0.181	0.040	0.721
NIMBUS	1	67.964	0.087	0.071	0.159	0
	2	38.380	0.055	0.046	0.090	0.005
	4	29.335	0.056	0.048	0.086	0.009
	8	18.215	0.061	0.048	0.063	0.028
	16	11.349	0.069	0.053	0.041	0.031
	32	7.719	0.094	0.072	0.027	0.047

The running times on a 32-node Intel iPSC/860 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 elimination, 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 $A^t(b-Ax)$. A running time is obtained by measuring the time spent on each processor and taking the maximum time spent on a processor.

The "mflops" in Table 7 is the execution rate defined as the number of megaflops performed per second during numerical factorization. The "QR" and "SNE" in Table 7 represent the running times for the QR factorization method and the SNE method, respectively. The "C_STEP" shows the running time for a correction step in the CSNE method. The time for the QR factorization method includes "qr_time", the time for forming Q^tb and "bs_time". The time for the SNE method includes "qr_time", "fe_time", "bs_time", "mv_time" and "gs_time". Since the time spent on the local portion of the matrix-vector product A^tb is less than the "mv_time" and the difference between the two is negligible, we use the "mv_time" for calculating the running time for the SNE method. The running time for a correction step is the sum of "fe_time", "bs_time", "mv_time" and "gs_time". Suppose that the CSNE method performs k correction steps. The running time for the CSNE method is the time for the SNE method plus k times the running time for a correction step.

Due to insufficient storage space on node processors, some test problems can not be run on small number of processors. If a test problem can be run on p processors, its timing results on p

TABLE 7
Performance of the QR factorization, SNE and CSNE methods on an iPSC/860

problem	p	mflops	QR	SNE	C_STEP
GRID100	1	3.968	17.671	17.559	0.990
	2	7.830	8.947	8.935	0.538
	4	14.917	4.691	4.780	0.372
	8	25.715	2.712	2.910	0.353
	16	43.287	1.613	1.893	0.374
	32	68.350	1.027	1.352	0.390
GRID200	8	36.059	14.524	15.180	1.284
	16	60.276	8.664	9.751	1.438
	32	98.020	5.259	6.593	1.481
GRID250	32	110.950	8.878	10.998	2.376
GRID20.3	8	45.605	31.302	31.211	0.476
	16	76.394	18.618	18.829	0.481
	32	123.769	11.457	11.819	0.494
GRID24.3	16	82.604	49.560	49.994	0.885
	32	135.103	30.284	30.959	0.933
GRID27.3	32	142.792	56.597	57.505	1.273
GRID28.3	32	144.114	69.249	70.271	1.439
NIMBUS	1	8.615	69.112	68.281	0.317
	2	15.253	39.061	38.576	0.196
	4	19.936	29.837	29.534	0.199
	8	31.805	18.577	18.415	0.200
	16	50.738	11.569	11.543	0.194
	32	73.124	7.969	7.959	0.240

processors are reported. 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 efficiency of the overall sparse triangular solution including both the forward elimination and back substitution is demonstrated by the fact that the triangular solution time is a small fraction of the execution time of the highly efficient numerical factorization phase. The sparse back substitution is more efficient than the sparse forward elimination. This is consistent with the complexity results in §5, To order to accelerate the sparse forward elimination, the storage scheme for R can be rearranged after numerical factorization so that efficient dense triangular solution kernels can be used. Since rearranging the storage scheme for R incurs extra communication cost, whether to rearrange the storage scheme for R depends on the number of sparse triangular solutions required after the numerical factorization. As the number of processors increases, the global summation time for obtaining the matrix-vector product A^tb or $A^t(b - Ax)$ increases.

The numerical accuracy of our parallel implementations of the QR factorization method, the SNE method and the CSNE method is illustrated in Table 8. 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 8, $\delta x = x - \bar{x}$, where $\bar{x}$ is the computed solution. The three rows of results for the CSNE method represent the results of three iterative refinement steps. Matstoms [15] indicates that maximal accuracy is often achieved within 1–3 iterative refinement steps.

TABLE 8
Numerical accuracy (p=32)

problem	method	$\|\delta x\|_1$	$\|\delta x\|_2/\|x\|_2$	$\|\delta x\|_\infty$
GRID250	QR	1.1240e-09	8.6181e-16	4.1211e-13
	SNE	1.1791e-09	8.7503e-16	4.7606e-13
	CSNE	9.5973e-11	8.4482e-17	5.6843e-14
		4.4885e-11	5.6563e-17	2.8422e-14
		2.6744e-11	4.3478e-17	2.8422e-14
GRID28.3	QR	1.8351e-09	1.3565e-14	1.5241e-12
	SNE	1.7747e-09	1.3276e-14	1.7444e-12
	CSNE	1.0618e-11	7.9244e-17	1.0658e-14
		3.4079e-12	4.3365e-17	7.1054e-15
		1.3469e-12	2.7390e-17	7.1054e-15

The three parallel approaches for solving sparse linear least squares have their strength and weakness. The QR factorization method is numerically backward stable and is the most efficient method on large number of processors. However, in case that the orthogonal matrix Q is too large to fit in the main memory, the right hand side must be available before the sparse QR factorization is performed or the orthogonal matrix Q must be written into the secondary storage at increased I/O cost. The SNE method and the CSNE method avoid the disadvantage of the QR factorization method. However, the SNE method is not numerically stable. The CSNE method is, in general, as accurate as the QR factorization method, but it can't handle problems with widely different row norms or the "stiff" problems. Furthermore, the CSNE method is less efficient than the QR factorization method, especially so on large number of processors.

7. Concluding remarks. We have described three efficient parallel approaches based on the QR factorization method, the SNE method and the CSNE method for solving large and sparse linear least squares problems on distributed-memory multiprocessors. The required parallel sparse QR factorization has been discussed in [20]. Central to our algorithms is a new parallel multifrontal algorithm for solving the related sparse triangular systems. The three approaches have their advantages and disadvantages. In practice, the choice among the three approaches depends on the availability of the right hand sides, the size of the primary storage space and the desired accuracy for the computed solutions. Since the parallel sparse QR factorization is the dominate phase in all three approaches and relatively long messages are communicated among processors in our algorithm for parallel sparse QR factorization [20], the proposed parallel approaches for solving sparse linear least squares problems are especially suitable for machines which communicate long messages much more efficiently than they communicate short messages such as the IBM SP1 system. Implementations on the IBM SP1 system are currently being pursued.

Acknowledgements. I would like to thank Professor Thomas F. Coleman for his helpful comments on the manuscript. Numerical experiments were performed on a 32-node Intel iPSC/860 machine in the Computer Science Department, Cornell University.

REFERENCES

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

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

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

[4] J. A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[5] J. A. George, M. T. Heath, J. W. H. Liu, and E. G. Y. Ng, *Solution of sparse positive definite systems on a hypercube*, J. Comp. Appl. Math, 27 (1989), pp. 129–156.

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

[7] ———, *The evolution of the minimum degree algorithm*, SIAM Review, 31 (1989), pp. 1–19.

[8] J. A. George, J. W. H. Liu, and E. G. Y. Ng, *Communication results for parallel sparse Cholesky factorization on a hypercube*, Parallel Computing, 10 (1989), pp. 287–298.

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

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

[11] 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.

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

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

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

[15] P. Matstoms, *The Multifrontal Solution of Sparse Linear Least Squares Problems*, PhD thesis, Linköping University, Sweden, 1991.

[16] 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.

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

[18] C. Sun, *Some Applications of Clique Trees to Sparse Cholesky Factorizations*, PhD thesis, Pennsylvania State University, University Park, PA, 1991.

[19] ———, *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.

[20] ———, *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.

[21] E. Zmijewski, *Sparse Cholesky Factorization on a Multiprocessor*, PhD thesis, Cornell University, Ithaca, NY, 1987.