This paper appears in
SIAM Journal on Scientific Computing, Vol 17, No. 3, pp. 666-685, May 1996.

PARALLEL SPARSE ORTHOGONAL FACTORIZATION ON DISTRIBUTED-MEMORY MULTIPROCESSORS[1]

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

Abstract

In this paper, we propose a new parallel multifrontal algorithm for the orthogonal factorization of large sparse matrices on distributed-memory multiprocessors. We explore the use of block partitioning schemes in parallel sparse orthogonal factorization. Our block-oriented parallel algorithm for sparse orthogonal factorization achieves high performance by incurring strictly less communication overhead than the conventional non-block algorithm, maintaining relatively balanced load distribution among processors and accelerating the parallel numerical kernel via increased cache utilization. We analyze the performance of our parallel algorithm and present its arithmetic and communication complexities for regular grid problems. We report the experimental results of an implementation of our parallel algorithm on an Intel iPSC/860 machine. Through our theoretical analysis and experimental results, we demonstrate that our new block-oriented algorithm outperforms the conventional non-block algorithm impressively.

Key words. parallel algorithms, sparse matrix, orthogonal factorization, multifrontal method, block partitioning scheme, distributed-memory multiprocessors

AMS subject classifications. 65F05, 65F50

Abbreviated title: Parallel Sparse Orthogonal Factorization

[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 SPARSE ORTHOGONAL FACTORIZATION ON DISTRIBUTED-MEMORY MULTIPROCESSORS [†]

CHUNGUANG SUN [‡]

Abstract. In this paper, we propose a new parallel multifrontal algorithm for the orthogonal factorization of large sparse matrices on distributed-memory multiprocessors. We explore the use of block partitioning schemes in parallel sparse orthogonal factorization. Our block-oriented parallel algorithm for sparse orthogonal factorization achieves high performance by incurring strictly less communication overhead than the conventional non-block algorithm, maintaining relatively balanced load distribution among processors and accelerating the parallel numerical kernel via increased cache utilization. We analyze the performance of our parallel algorithm and present its arithmetic and communication complexities for regular grid problems. We report the experimental results of an implementation of our parallel algorithm on an Intel iPSC/860 machine. Through our theoretical analysis and experimental results, we demonstrate that our new block-oriented algorithm outperforms the conventional non-block algorithm impressively.

Key words. parallel algorithms, sparse matrix, orthogonal factorization, multifrontal method, block partitioning scheme, distributed-memory multiprocessors

AMS subject classifications. 65F05, 65F50

Abbreviated title: Parallel Sparse Orthogonal Factorization

1. Introduction. Let A be a sparse $m \times n$ matrix with full column rank. The QR factorization of A is expressed as

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix},$$

where Q is an $m \times m$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. Usually Q is not formed explicitly.

A row merging scheme and a general row merging scheme for sparse QR factorization are proposed in [8] and [19], respectively. Sequential algorithms for sparse QR factorization are also considered in [11, 17, 22, 23]. Parallel algorithms are described in [5, 24, 27] for implementing the numeric phase of the general row merging scheme [19] on MIMD distributed-memory multiprocessors. A parallel algorithm is discussed in [16] for implementing the numeric phase of the row merging scheme [8] on SIMD architecture. Previously-proposed parallel sparse QR factorization algorithms [5, 16, 24, 27] are based on the non-block or single row partition of the dense matrices involved in the numerical computation. Block-oriented approach to parallel sparse QR factorization has not been explored before.

In this paper we consider a block-oriented approach to parallel sparse QR factorization and describe a new parallel multifrontal algorithm for sparse QR factorization on distributed-memory multiprocessors. We propose block schemes for partitioning the upper trapezoidal matrices involved in the numerical computation. We explore a spectrum of block partitioning schemes ranging from the conventional single row partition to the block partition with maximal block size. Our block-oriented approach results in an efficient parallel multifrontal algorithm for sparse QR factorization with low communication costs, relatively balanced load distribution among processors and a high performance parallel numerical kernel due to the increased cache utilization.

[†] 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(csun@cs.cornell.edu).

Both A and R are distributed to processors by rows. The supernodal structure of R is fully exploited. The performance of our parallel algorithm with a special block partition applied to regular grid problems is analyzed, and the arithmetic and communication complexity results of our analysis are presented. Experimental results of an implementation of our parallel algorithm on a 32-node Intel iPSC/860 machine are obtained for a collection of sparse matrix problems including regular grid problems and a general sparse problem with irregular sparsity structure. Our block-oriented approach to parallel sparse QR factorization achieves significant improvement in performance over the conventional non-block approach described in [5].

In §2, a multifrontal sparse QR factorization scheme is reviewed. A new parallel multifrontal algorithm for sparse QR factorization is proposed in §3. The complexity analysis of our algorithm for regular grid problems is presented in §4. Experimental results are discussed in §5. Finally, concluding remarks are contained in §6.

2. A multifrontal sparse QR factorization algorithm. We briefly review a multifrontal approach to sparse QR factorization in this section. We refer the reader to [11, 17, 19] for further details. Let $A^TA = LL^T$, where L is the Cholesky factor of A^TA. It is well-known that R^T is equal to L, apart from possible sign differences in the rows. Therefore, the elimination tree of A^TA can be defined in terms of the sparsity structure of R. Specifically, j is the parent of i if $r_{ij} (i < j)$ is the leading off-diagonal nonzero in the i^{th} row of R. For simplicity, it is assumed that A^TA is irreducible and the corresponding elimination tree is indeed a tree. A node in the elimination tree corresponds to a row in R. Let $G(A^TA)$ denote the adjacency graph of A^TA. The steps involved in sparse QR factorization are described in Fig. 1.

1. Find a column ordering P_1 for A such that $(AP_1)^T(AP_1)$ has a sparse Cholesky factor.
2. Compute the elimination tree of $(AP_1)^T(AP_1)$ from $G(A^TA)$ and P_1. Number the nodes of the elimination tree in postorder. Let the postorder be P_2 and set $P = P_1P_2$.
3. Determine the symbolic structure of R from $G(A^TA)$ and P, where $(AP)^T(AP) = R^TR$.
4. Compute the supernodal elimination tree of R from the elimination tree and the symbolic structure of R.
5. Perform numerical factorization by processing the supernodes in a topological ordering.

FIG. 1. *Sparse QR factorization*

Step 1 is done by applying a symmetric ordering such as the nested dissection ordering [7] or the minimum degree ordering [12] to $G(A^TA)$. Step 2 can be accomplished by using an algorithm described in [18]. The symbolic factorization step is discussed in [9]. There are several slightly different definitions of supernode [1, 2, 3, 10, 20]. Any of those definitions can be used in our approach. In an implementation of our algorithm, the fundamental supernode [2] is used. Let $\mathcal{R}_j$ denote the structure of row j of R—i.e., the set of nonzero column indices in row j of R. A fundamental supernode is a maximal set of contiguous rows $\{i, i+1, \cdots, k\}$ in R such that $\mathcal{R}_j = \{j\} \cup \mathcal{R}_{j+1}$ for $i \le j < k$ and j is the only child of $j+1$ in the elimination tree which has been postordered. The first four steps are referred to as the *symbolic phase* of the sparse QR factorization and Step 5 is referred to as the *numeric phase* of the sparse QR factorization.

A supernode K is associated with an upper trapezoidal matrix F_K called the *frontal matrix* of K. The sparsity structure of F_K is given by $\mathcal{R}_i$, where i is the least-numbered row in K. The number of columns in F_K is $|\mathcal{R}_i|$ and the number of rows in F_K is at most $|\mathcal{R}_i|$. Liu [19] has shown that F_K can be considered as a dense matrix for all practical purposes. Our parallel algorithm described in §3 works correctly regardless whether F_K is upper trapezoidal or upper triangular. For simplicity, we assume that F_K is a dense upper triangular matrix. The first $|K|$ rows of F_K are rows $i, i+1, \cdots, i+|K|-1$ in R and they are called the *factor rows* of F_K. The upper triangular matrix obtained by removing the factor rows of F_K is called the *update matrix* of K and is denoted by U_K.

Let A_j denote the submatrix consisting of all rows of A whose first nonzeros are in column j of A. The number of nonzero columns in A_j is at most $|\mathcal{R}_j|$. Let $children(K)$ be the set of children of K in the supernodal elimination tree. The numeric phase of a sequential multifrontal sparse QR factorization algorithm is described in Fig. 2. The storage for R is allocated before the numeric phase starts. The assembly of an update matrix U_C into F_K in line 3 can be done by a matrix *extend-add* operation introduced in [20], where C is any child of K. Merging U_D into F_K in line 5, reducing A_j to an upper trapezoidal matrix T_j in line 7 and merging T_j into F_K in line 8 can be accomplished by either Householder transformations or Givens rotations. In our implementation, U_D or T_j is merged into F_K by Givens rotations and A_j is reduced to T_j by Householder transformations. It is clear that this multifrontal approach mainly involves dense matrix computations. Furthermore, the update matrices can be handled by a stack data structure, so data locality is achieved. These desirable features are also present in our parallel algorithm.

1 **for** each supernode K in a topological ordering **do**
2   allocate space for U_K and initialize F_K to zero;
3   assemble U_C into F_K and discard U_C, where $C \in children(K)$;
4   **for** each $D \in children(K) \setminus \{C\}$ **do**
5   merge U_D into F_K and discard U_D;
6   **for** each $j \in K$ **do**
7   transform A_j to an upper trapezoidal matrix T_j;
8   merge T_j into F_K;
9   **end for**
10 **end for**

FIG. 2. *The numeric phase of a sequential multifrontal sparse QR factorization algorithm*

3. A parallel multifrontal sparse QR factorization algorithm. Several parallel algorithms have been described in [5, 24, 27] for implementing the numeric phase of sparse QR factorization on distributed-memory multiprocessors.

- In the algorithm discussed in [5], a frontal matrix F is mapped to a ring of processors by assigning the rows of F to the processors in a wrap-around manner. An update matrix U which needs to be merged into F is shifted among the ring of processors in a circular fashion. Before each shift the diagonal elements of U are annihilated against the frontal matrix F by Givens rotations and U becomes a smaller upper triangular matrix. This shift is continued until U is completely merged into F. This approach is referred to as the *row-oriented approach.*

- In the approach described in [24], a global row reduction algorithm is introduced to compute a frontal matrix. Parallel computation of the tree structure required for the multifrontal method is also discussed. In this approach, the amount of arithmetic work may increase as the number of processors increases.

- A parallel algorithm is discussed in [27] for implementing the sparse QR factorization scheme proposed in [11] in parallel. In this scheme, a frontal matrix is computed by row-oriented Householder transformations. Before the parallel computation of a frontal matrix is started, the relevant update matrices are redistributed among processors to ensure relatively balanced load distribution. As expected, this redistribution improves load balancing by incurring additional communication overhead.

A drawback common to all of these algorithms is that frontal matrices are computed by communicating **single** rows—i.e., a message contains only one row or one transformed row of some frontal matrix. This can result in excessive number of small messages communicated among processors. Because of the message start-up time, the overhead associated with a short message is proportionally higher than that associated with a long message.

To reduce communication costs, a block approach is suggested as a future research topic in [5]. A frontal matrix is partitioned into blocks with same number of rows. If F is an $h \times h$ upper triangular matrix and is mapped to p processors, block size equal to h/p^2 is recommended in [5]. As shown in §5, the communication volume of this block approach is still relatively high. To the best of our knowledge, experimental results of the row-oriented approach and the block-oriented approach described in [5] have not been reported in the literature. In §5, performance results of these two approaches will be discussed.

3.1. An overall framework for parallel sparse QR factorization. We propose a new parallel algorithm for the numeric phase of the multifrontal sparse QR factorization described in §2. The symbolic phase of the sparse QR factorization is done sequentially. After the symbolic structure of R and the corresponding supernodal elimination tree are determined, the arithmetic work associated with each subtree is estimated. The supernodal elimination tree is mapped onto processors by a proportional mapping scheme described in [26]. The root of the supernodal elimination tree is partitioned among all processors. For a supernode that has already been mapped to a set of processors, each subtree rooted at a child of that supernode is allocated a subset of processors whose size is proportional to the workload associated with that subtree.

We consider the formation of a frontal matrix as a computational task. The supernodal elimination tree is the precedence graph among the computational tasks. Computation proceeds from leaves to root. Initially each processor is working on its own subtrees. Later on processors cooperate to compute the frontal matrix of a partitioned supernode. When independent subtrees are being processed, no communication among processors is needed.

The overall parallel algorithm for the multifrontal sparse QR factorization is described in Fig. 3. The key operation in this algorithm is to merge an upper triangular matrix into another one in parallel and it will be discussed in detail in the next subsection.

3.2. A parallel numerical kernel. Let V and W be two $h \times h$ upper triangular matrices. Consider merging V into W by Givens rotations on a set of p processors $\{\mu_0, \mu_1, \ldots, \mu_{p-1}\}$—i.e., transforming the matrix $\begin{bmatrix} W \\ V \end{bmatrix}$ into an upper triangular form by Givens rotations. Let V and W be identically partitioned into $k (k \leq h)$ blocks by rows as

$$V = \begin{bmatrix} V_0 \\ V_1 \\ \vdots \\ V_{k-1} \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_{k-1} \end{bmatrix}.$$

```
for each supernode K in a topological ordering do
    if F_K is entirely mapped to μ then
        allocate space for U_K and initialize F_K to zero;
        assemble U_C into F_K and discard U_C, where C ∈ children(K);
        for each D ∈ children(K) \ {C} do
            merge U_D into F_K and discard U_D;
        for each j ∈ K do
            merge rows in A_j into F_K;
    else
        if F_K is partially mapped to μ then
            allocate space for μ's portion of U_K;
            initialize μ's portion of F_K to zero;
            assemble U_C into F_K in parallel and discard μ's portion of U_C,
                where C ∈ children(K);
            for each D ∈ children(K) \ {C} do
                merge U_D into F_K in parallel;
                discard μ's portion of U_D;
            end for
            merge relevant rows from A into F_K in parallel;
        end if
    end if
end for
```

FIG. 3. *A parallel multifrontal sparse orthogonal factorization algorithm on a processor μ.*

Blocks V_i and W_i are assigned to processor μ_j, where $j = i \bmod p$. The block V_i is merged into blocks $W_i, W_{i+1}, \ldots, W_{k-1}$, successively.

For a given number of processors, how the upper triangular matrices are partitioned determines the performance of merging V into W. Since there are p processors, the number of blocks should be at least p. Clearly, a block must contain at least one row. Therefore, $p \leq k \leq h$. The total number of possible block partitions for an $h \times h$ upper triangular matrix on p processors is

$$\sum_{k=p}^{h} \binom{h-1}{k-1} = \sum_{k=p}^{h} \frac{(h-1)!}{(k-1)!(h-k)!}.$$

For reasonable h and p, it is impractical to determine an optimal block partition by examining all possible block partitions. We consider two strategies for partitioning upper triangular matrices into blocks.

Equal-row partitioning scheme. First, we consider partitioning upper triangular matrices V and W into blocks with equal number of rows. This scheme partitions an $h \times h$ matrix into $k = h/s$ blocks, where $s (1 \leq s \leq h/p)$ is the block size or the number of rows in a block. This scheme is referred to as the *equal-row partitioning scheme*. The special cases $s = 1$ and $s = h/p^2$ are considered in [5].

The number of blocks k determines the arithmetic work distribution of merging V into W. The larger the number of blocks, the more balanced the arithmetic work distribution. The worst and the best arithmetic work distributions are achieved for $k = 1$ and $k = h$, respectively. To maintain load balance, the number of blocks should be large—i.e., the block size should be small.

Let α be the startup time for a message and β the transfer time per floating-point number. The total communication cost on all processors is

$$\sum_{i=1}^{k-1} i(\alpha + \beta(s(h-is) - (s-1)s/2)) \approx \frac{\alpha}{2}k^2 + \frac{\beta h^2}{6}k - \frac{\beta h^2}{4}.$$

Clearly, the communication cost is proportional to the number of blocks k. To keep the communication cost low, the number of blocks should be small—i.e., the block size should be large.

An optimal block size minimizes the combined effect of the arithmetic work distribution and the communication cost—i.e., the overall execution time. To obtain an optimal block size or a block size that is close to be optimal, it is necessary to examine how the algorithmic performance of merging V into W is dependent upon the block size.

Experiments are performed on an Intel iPSC/860 machine to investigate the relationship between the algorithmic performance and the block size used. Our results show that the special case $s = 1$ gives the worst performance. For fixed p and h, the running time initially decreases rapidly as the block size increases. Once the best performance is achieved, the running time increases slowly as the block size increases. The running times are virtually indistinguishable at the neighborhood of the optimal block size for given h and p. Therefore, it is not necessary to determine the optimal block size exactly. As an illustration, experimental results for $p = 32$ and $h = 600, 700, 800$ are shown in Fig. 4. Block sizes equal to $1, 2, \cdots, 40$ are used. The same kind of relationship is observed for other values of p and h.

TABLE 1
Optimal block sizes and corresponding performance results on an Intel iPSC/860 machine

	$p = 8$			$p = 16$			$p = 32$		
h	s	time	mflops	s	time	mflops	s	time	mflops
100	5	0.057	18.875	4	0.048	22.414	2	0.053	20.299
200	10	0.220	37.735	6	0.177	46.902	4	0.155	53.559
300	16	0.546	50.691	8	0.423	65.432	5	0.353	78.407
400	20	1.106	58.954	10	0.839	77.716	6	0.668	97.610
500	25	1.929	65.775	13	1.292	98.204	7	1.123	112.982
600	30	3.074	71.147	15	2.079	105.197	8	1.615	135.421
700	35	4.545	76.277	18	3.061	113.257	9	2.300	150.731
800	40	6.505	79.448	20	4.329	119.382	11	3.090	167.251
900	44	8.996	81.712	22	5.835	125.978	12	4.189	175.479

The optimal block sizes for $p = 8, 16$ and 32 are shown in Table 1 under column "s". The columns under "time" and "mflops" show the corresponding running times and floating-point execution rates in mega flops per second, respectively. Floating-point operations are performed in double-precision arithmetic. Since the dense submatrices involved in the numerical kernel of our parallel sparse QR factorization algorithm are of order 100 to 900, the range of h shown in Table 1 is between 100 and 900. Let $s(h, p)$ denote an optimal block size for merging two $h \times h$ upper triangular matrices on p processors. Table 1 shows that optimal block sizes for $p = 8, 16, 32$ can be expressed as

(1) $$s(h, p) = \lceil 2h/(5p) \rceil.$$

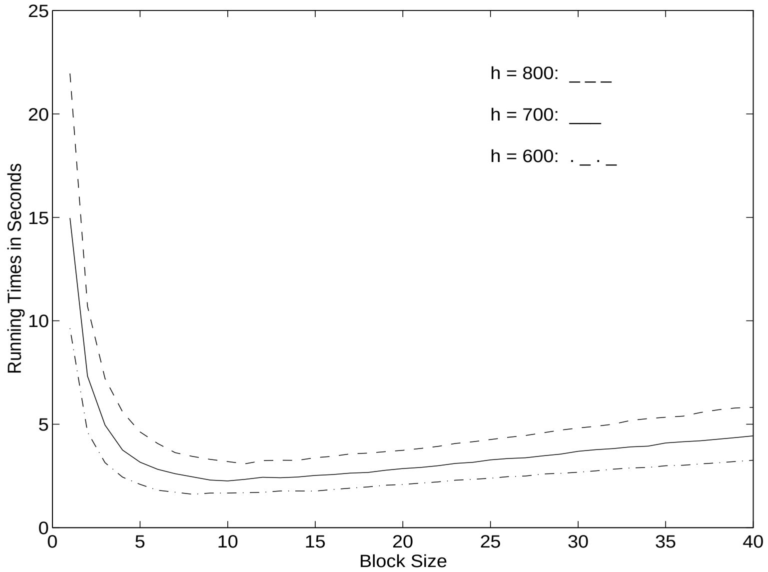

FIG. 4. *Equal-Row Partitioning Scheme with $p = 32$ and $h = 600, 700, 800$ on an iPSC/860*

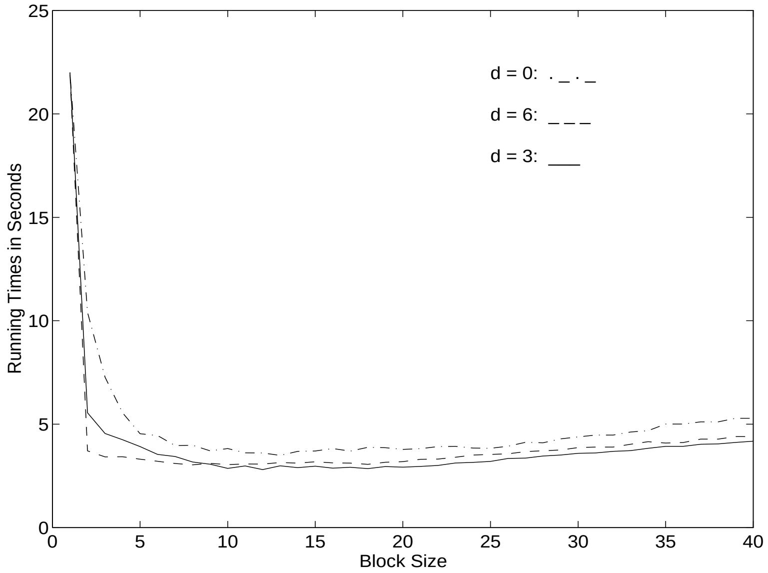

FIG. 5. *Revised Equal-Volume Partitioning Scheme with $p = 32$, $h = 800$ and $d = 0, 3, 6$ on an iPSC/860*

Experimental results for $p = 2, 4$ have also been obtained. Optimal block sizes for $p = 2, 4$ can be expressed as

$$s(h, p) = \lceil h/(3p) \rceil. \tag{2}$$

Revised equal-volume partitioning scheme. Because of the triangular shape of the matrices, the equal-row partitioning scheme may result in load imbalance since the blocks become smaller and smaller toward the end. To correct this, we propose schemes which partition the upper triangular matrices V and W into blocks with non-decreasing number of rows. The *equal-volume partitioning scheme* introduced in [30] is one of such schemes. This scheme partitions V and W identically into k blocks by rows. Each block consists of a set of contiguous rows and contains about $h(h+1)/(2k)$ elements. Assume that the number of rows and the number of columns in the i^{th} block are denoted by r_i and c_i, respectively. Then

$$\frac{c_i(c_i+1)}{2} \approx \frac{h(h+1)}{2}\left(\frac{k-i}{k}\right). \tag{3}$$

$$c_i \approx \sqrt{\frac{k-i}{k}}\,h \quad \text{and} \quad r_i \approx \frac{\sqrt{k-i}-\sqrt{k-i-1}}{\sqrt{k}}\,h. \tag{4}$$

Notice that $\sum_{i=0}^{k-1} r_i = h$. It is easy to show $r_i \leq r_{i+1}$ for $0 \leq i < k-1$.

Now we propose a revised equal-volume partitioning scheme. Let k be the expected number of blocks to be partitioned. The revised equal-volume partitioning scheme actually partitions V and W into $l (l \leq k)$ blocks such that the i^{th} block has $r_i + d$ $(d \geq 0)$ rows, where r_i is defined as in (4) and $d \geq 0$. This revised scheme is precisely the equal-volume partitioning scheme when $d = 0$. Clearly, $r_i + d \leq r_{i+1} + d$ for $0 \leq i < l-1$. The number of blocks l can be determined from $\sum_{i=0}^{l-1}(r_i + d) = h$. Our empirical results indicate that the optimal value for d is 3. Experimental results on 32 processors for $h = 800$ and $d = 0, 3, 6$ are shown in Fig. 5. In the remainder of this paper, $d = 3$ is assumed for the revised equal-volume partitioning scheme.

Depending on the number of processors used, the equal-row partitioning scheme with block size specified by expression (1) or (2) is referred to as the equal-row partitioning scheme with optimal block partition. Similarly, the revised equal-volume partitioning scheme with the expected number of blocks to be partitioned given by $h/s(h,p)$ is referred to as the revised equal-volume partitioning scheme with optimal block partition. The performance results of these two schemes with optimal block partition are compared in Fig. 6. The revised equal-volume partitioning scheme is, in general, more efficient than the equal-row partitioning scheme.

Row-oriented kernel vs. column-oriented kernel. Assume that V_i and W_j are 4×5 upper trapezoidal blocks. Two ways of merging V_i into W_j are shown below:

$$\begin{bmatrix} W_j \\ V_i \end{bmatrix} \Rightarrow \begin{bmatrix} x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ 1 & 2 & 3 & 4 & x \\ & 5 & 6 & 7 & 8 \\ & & 9 & 10 & 11 \\ & & & 12 & 13 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ 1 & 2 & 4 & 7 & x \\ & 3 & 5 & 8 & 11 \\ & & 6 & 9 & 12 \\ & & & 10 & 13 \end{bmatrix}.$$

The numbers show the order in which the elements of V_i are annihilated. These two approaches correspond to merging V_i into W_j by rows and by columns, respectively. If the block size—i.e.,

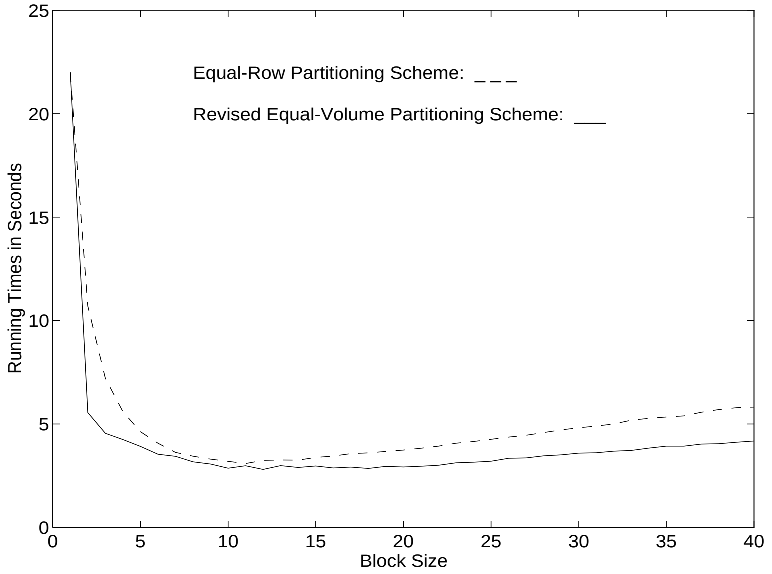

FIG. 6. *Comparison of the two partitioning schemes with $p = 32$ and $h = 800$ on an iPSC/860*

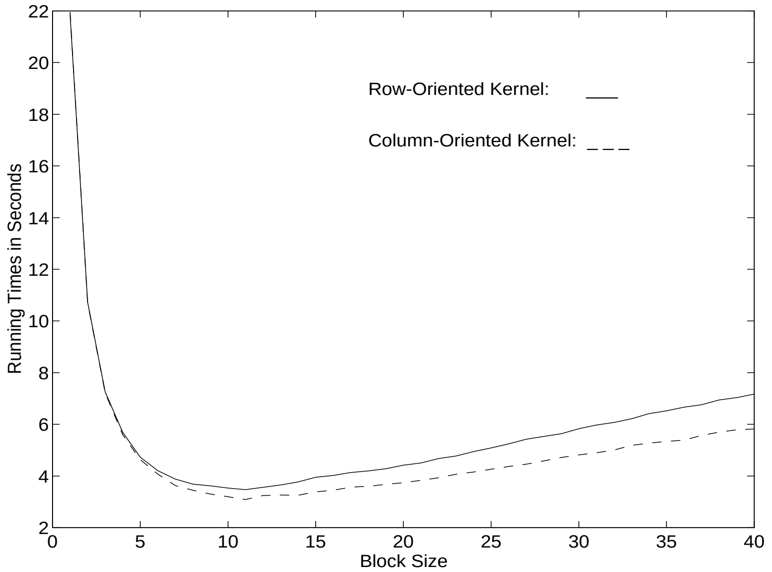

FIG. 7. *Comparison of the row-oriented and the column-oriented kernels with $p = 32$ and $h = 800$ on an iPSC/860*

the number of rows in V_i is reasonably large, the column-oriented kernel is more efficient than the row-oriented kernel. The reason is that in the column-oriented kernel a rectangular submatrix of V_i is rotated into a row in W_j so this row in W_j is repetitively accessed. Therefore, the cache is better utilized and, consequently, the performance of the numerical kernel is improved. This is confirmed by the empirical results shown in Fig. 7. As the block size increases, a row in W_j is repetitively accessed more frequently in the column-oriented kernel. Therefore, the cache utilization of the column-oriented kernel gets better and it becomes increasingly more efficient than the row-oriented kernel as the block size increases.

3.3. Parallel computation of a frontal matrix. In order to compute a frontal matrix F_K in parallel, we have designed parallel algorithms for assembling the update matrix of a child of K into F_K, merging the update matrix of a child of K into F_K and merging relevant rows from A into F_K. These algorithms correspond to lines 13, 16 and 19 in Fig. 3, respectively.

To simplify our discussion, we take the 7×7 nine-point grid with nested dissection ordering as an example. A portion of its supernodal elimination tree consists of supernodes $K = \{19, 20, 21\}$, $C = \{7, 8, 9\}$ and $D = \{16, 17, 18\}$. Supernodes C and D are the only children of supernode K. The frontal matrices F_K, F_C and F_D are illustrated in Fig. 8, where the row numbers displayed with a frontal matrix show its sparsity structure.

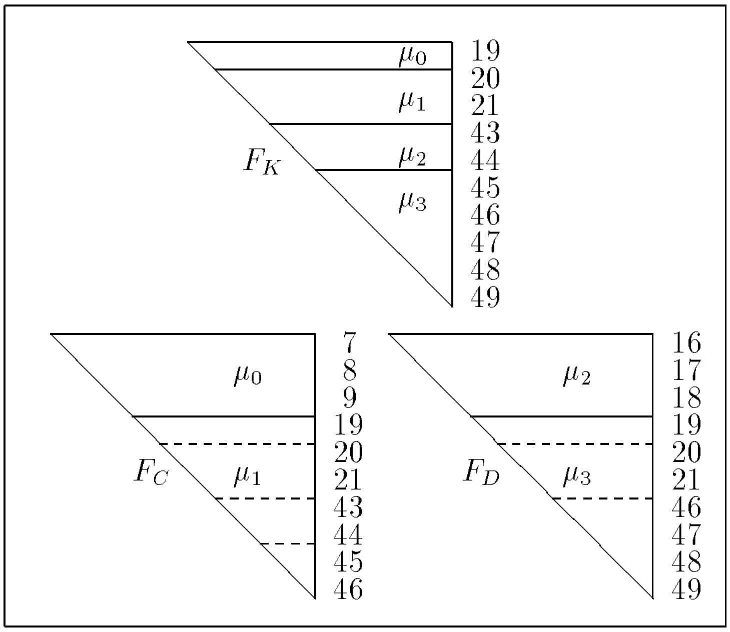

FIG. 8. *Parallel computation of a frontal matrix*

We assume that F_K is partitioned among four processors $\{\mu_0, \mu_1, \mu_2, \mu_3\}$, F_C is partitioned among μ_0 and μ_1, and F_D is partitioned among μ_2 and μ_3, respectively. The solid lines inside the triangles in Fig. 8 show one way of partitioning F_K, F_C and F_D into blocks.

Suppose that the frontal matrices F_C and F_D have already been computed. The update matrix U_C is the second block of F_C and is assigned to μ_1. Similarly, the update matrix U_D is the second block of F_D and is assigned to μ_3. Based on the sparsity structure of U_C, U_D and F_K, the update matrices U_C and U_D are further partitioned into sub-blocks shown by the dash lines in Fig. 8.

Suppose U_C has already been assembled into F_K. Now we consider merging U_D into F_K. A sub-block B of U_D is sent to a processor involved for computing F_K if the processor owns a block B_K of F_K such that the row numbers of B are contained in the row numbers of B_K. For instance, the second sub-block of U_D owned by μ_3 is sent to μ_1 which owns the second block of F_K.

Once a sub-block B is received by a processor, it is conceptually extended to a temporary block by the matrix *extend-add* operation introduced in [20] and this temporary block is then rotated into a block B_K owned by the processor. The result of rotating B into B_K is another upper trapezoidal block $\hat{B}$. If $\hat{B}$ is not empty, it is sent to the processor which owns the block just below the block B_K in F_K. Notice that the number of rows in $\hat{B}$ is less than or equal to the number of rows in B since the block B_K may contain zero rows. Eventually, the newly-created block $\hat{B}$ is merged into the remaining blocks of F_K, successively.

In order to complete the computation of F_K, a row from A whose first nonzero is in column j of A such that $i \leq j < i + |K|$ must be rotated into F_K, where i is the first row in K. The rows in the submatrix A_j, defined in §2, form a rectangular block with at most $|\mathcal{R}_j|$ nonzero columns. This block is rotated into F_K starting with a block B_K of F_K such that row $j - i$ of F_K is in block B_K. The resulting block is rotated into the remaining blocks of F_K, successively.

4. Analysis of regular grid problems. In this section, we present the complexity analysis of our parallel sparse QR factorization algorithm based on a special equal-volume partitioning scheme for regular grid problems. We compare our results with those provided in [5]. To be consistent with the results in [5], we count only the number of multiplications in our analysis.

We focus on the equal-volume partitioning scheme in which an $h \times h$ frontal matrix distributed among p processors is partitioned into p blocks. It seems to be very difficult or impossible to obtain the complexity results of an arbitrary equal-row or equal-volume partitioning scheme. The analytical results obtained for a special case of the equal-volume partitioning scheme would shed light on the performance of an arbitrary equal-row or equal-volume partitioning scheme from a theoretical angle.

Our goal is to obtain an upper bound on the overall execution time of our parallel sparse QR factorization algorithm. There are two levels of analysis. In the outer level, the supernodal elimination tree is mapped onto node processors by the subtree-to-subcube mapping scheme [13]. We identify a heavest path of the supernodal elimination tree. The heavest path consists of a sequence of computational tasks each of which is the computation of a frontal matrix. Careful exploration of the heavest path leads to an upper bound on the execution time of our algorithm. In the inner level, we investigate the cost of computing a frontal matrix in parallel. The processors assigned for the task start execution at the same time. We identify a processor μ that has the most work to do and finishes its share of the task last. We characterize the sequence of subtasks executed by the processor μ. We show that there is no or very little gap between the subtasks executed by μ. Therefore, the time complexity of computing a frontal matrix in parallel is simply the sum of the time complexities of subtasks executed by μ.

The communication model assumed in our analysis is briefly described here. Consider the transmission of a message from one processor μ to another processor $\hat{\mu}$. The completion of the **send** operation on μ does not imply that the message has been received by $\hat{\mu}$, only that the message has been sent and the buffer space for the message on μ can be reused. Once the corresponding **receive** is issued by $\hat{\mu}$, processor $\hat{\mu}$ waits until the message arrives in the specified buffer. An example of this model is the communication model used on an Intel iPSC/2 or Intel iPSC/860 machine [15]. In this model, the cost of transmitting a message is charged to the receiver. Let τ 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 multiplication.

4.1. Analysis of the parallel numerical kernel. Consider two $h \times h$ upper triangular matrices V and W which are identically partitioned by our equal-volume partitioning scheme into

p blocks as

$$V = \begin{bmatrix} V_0 \\ V_1 \\ \vdots \\ V_{p-1} \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_{p-1} \end{bmatrix}.$$

Blocks V_i and W_i are assigned to processor μ_i, $0 \leq i < p$. The number of rows and the number of columns in the i^{th} block are defined as in (4) in §3.2 except that k is replaced by p.

The total number of multiplications for merging V into W is $2h(h+1)(h+2)/3$. The tasks involved in merging V into W are illustrated in Fig. 9, where $M(V_i, W_i)(0 \leq i < p)$ is the task for merging V_i into W_i, $M(V_i, V_j)(i < j)$ is the task for merging the transformed block V_i into the transformed block W_j, and $C(V_i, \mu_j, \mu_k)$ is the task for receiving the transformed block V_i from processor μ_j by processor μ_k, where $k = j + 1 \bmod p$. For simplicity, p is assumed to be 4 in Fig. 9. Clearly, the last processor has the most work to do and finishes its execution last.

$$\begin{tikzcd}
\mu_0 & \mu_1 & \mu_2 & \mu_3 \\
M(V_0, W_0) & M(V_1, W_1) \arrow[d] & M(V_2, W_2) \arrow[d] & M(V_3, W_3) \arrow[d] \\
& C(V_0, \mu_0, \mu_1) \arrow[d] & C(V_1, \mu_1, \mu_2) \arrow[d] & C(V_2, \mu_2, \mu_3) \arrow[d] \\
& M(V_0, W_1) & M(V_1, W_2) \arrow[d] & M(V_2, W_3) \arrow[d] \\
& & C(V_0, \mu_1, \mu_2) \arrow[d] & C(V_1, \mu_2, \mu_3) \arrow[d] \\
& & M(V_0, W_2) & M(V_1, W_3) \arrow[d] \\
& & & C(V_0, \mu_2, \mu_3) \arrow[d] \\
& & & M(V_0, W_3)
\end{tikzcd}$$

FIG. 9. *Tasks for merging V into W on 4 processors*

Now we examine the tasks assinged to μ_{p-1} in detail. Since $|V_i| \approx |V_{i-1}|$. the arithmetic work for $M(V_i, W_i)$ is about the same as the arithmetic work for $M(V_{i-1}, W_{i-1})$. Processors μ_2 and μ_3 start execution at the same time and they finish $M(V_2, W_2)$ and $M(V_3, W_3)$ at about the same time. Then μ_3 waits for the transmission of the transformed block V_2 from μ_2 to μ_3. Therefore, there is no gap or little gap between $M(V_3, W_3)$ and $C(V_2, \mu_2, \mu_3)$. Obviously, there is no gap between $C(V_2, \mu_2, \mu_3)$ and $M(V_2, W_3)$.

It can be easily shown that the arithmetic work in $M(V_i, W_j)$ is slightly greater than the arithmetic work in $M(V_{i-1}, W_{j-1})$. Once $M(V_2, W_3)$ is finished, the desired message is ready to be received. Hence, there is no gap between $M(V_2, W_3)$ and $C(V_1, \mu_2, \mu_3)$. Similar arguments can be extended to the rest of the tasks executed by μ_3. Therefore, there is no gap or little gap among the tasks executed by μ_3. The discussion for this example is equally valid for a general p.

Since the total number of messages communicated by μ_{p-1} is only $p-1$, the start-up costs for the $p-1$ messages are ignored and the cost for the task $C(V_i, \mu_{p-2}, \mu_{p-1})$ is estimated by the communication volume—i.e., the number of floating-point numbers communicated by the task $C(V_i, \mu_{p-2}, \mu_{p-1})$.

The above analysis shows that the arithmetic and communication complexities for merging V into W on p processors can be obtained by considering the communication volume and the arithmetic work on the last processor μ_{p-1}. Let ℓ be the number of rows in V_{p-1} or W_{p-1}. Then $\ell \approx h/\sqrt{p}$.

Since the total number of rows received by the last processor is $h-\ell$ and each row is of length at most ℓ, the total number of numerical values received by the last processor is bounded above by $(h-\ell)\ell$. Therefore, an upper bound on the communication cost for merging two $h \times h$ upper triangular matrices on p processors is

$$comm(h,p) = \frac{h^2\tau}{\sqrt{p}} - \frac{h^2\tau}{p}.$$

The work performed by the last processor is bounded above by the arithmetic work required for merging an $(h-\ell)\times\ell$ matrix into an $\ell\times\ell$ upper triangular matrix plus the arithmetic work for merging two $\ell\times\ell$ upper triangular matrices. Therefore, an upper bound on the arithmetic work for merging two $h \times h$ upper triangular matrices on p processors is

$$work(h,p) = 2\frac{h^3}{p} - \frac{4}{3}\frac{h^3}{p\sqrt{p}} + 2\frac{h^2}{\sqrt{p}} + \frac{4}{3}\frac{h}{\sqrt{p}}.$$

4.2. Analysis of the parallel sparse QR factorization algorithm. Now we analyze the performance of our parallel sparse QR factorization algorithm applied to regular grid problems with nested dissection ordering [7]. Consider a $k \times k$ 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 sparse matrix.

The supernodal elimination tree of a grid problem is a complete binary tree and is mapped onto node processors by the subtree-to-subcube mapping scheme [13]. The frontal matrices are partitioned by our special equal-volume partitioning scheme. A frontal matrix F_K is computed by merging the two update matrices U_1 and U_2 of the children of K. The size of the update matrix of a supernode is at most the size of the frontal matrix of its parent. Assume that the frontal matrix F_K is of order h and it is mapped to p processors. One update matrix U_1 is assembled into F_K and the other update matrix U_2 is merged into F_K. Before U_2 can be merged into F_K, it needs to be appropriately distributed among the p processors. This is equivalent to a parallel assembly of U_2 into F_K. After U_2 is properly distributed among the p processors, U_2 is merged into F_K as in the dense case.

The arithmetic work for merging U_2 into F_K is bounded above by the arithmetic work for merging two $h \times h$ upper triangular matrices in parallel since the size of U_2 is at most the size of F_K. Therefore, an upper bound on the arithmetic work for computing a frontal matrix F_K of order h on p processors is given by $work(h,p)$.

The worst scenario in an assembly operation is that a processor receives $h(h+1)/2p$ numerical values. The communication cost for computing a frontal matrix F_K of order h on p processors is bounded above by the communication cost for assembling U_1 and U_2 into F_K on p processors and the communication cost for merging two $h \times h$ upper triangular matrices on p processors. Therefore, an upper bound on the communication cost for computing a frontal matrix F_K of order h on p processors is

$$\frac{2h(h+1)\tau}{2p} + comm(h,p) = \frac{h^2\tau}{\sqrt{p}} + \frac{h\tau}{p}.$$

Consider the supernodal elimination tree corresponding to a $k \times k$ grid with nested dissection ordering, where $k = 2^t - 1$ and t is a positive integer. Let $m_j = 2^{t-j} - 1$. The characterization of a heavest path in the supernodal elimination tree is given in Table 2, where n_i is the size of the frontal matrix on the heavest path at level i. The root is considered to be at level 1. The reader is referred to [28] for details on the characterization of the supernodal elimination tree associated with a regular grid.

TABLE 2
Characterization of a heaviest path

level i	n_i
1	m_0
2	$m_1 + m_0$
3	$m_1 + m_0$
4	$6m_2 + 4$
$2j+1 (j \geq 2)$	$5m_j + 4$
$2j (j \geq 3)$	$7m_j + 6$

Assume that p is a power of 4. The upper bounds for the arithmetic and communication costs of our algorithm with the special block partition and the conventional row-oriented algorithm are presented in Table 3, where $c_1 = (343 + 250\sqrt{2})/3\sqrt{2}$, $c_2 = (8801\sqrt{2} - 6202)/21\sqrt{2}$, $c_3 = (103 + 72\sqrt{2})/4\sqrt{2}$, and $c_4 = (49+50\sqrt{2})/\sqrt{2}$. The results of the row-oriented algorithm are taken from [5]. Under a reasonable assumption that p is less than the number of columns k^2, the communication complexity of our algorithm with the special block partition is much better. The overall time complexity of our algorithm with the special block partition is $1229k^3/4p + O(k^3 \log p/p^{3/2})$ and that of the row-oriented algorithm is $146k^3/3p + 146k^3\tau/12p + O(k^3\tau/p^{3/2})$. If $\tau \geq 22$, the overall complexity of our algorithm with the special block partition is better. Dunigan [6] has shown that the values of τ for iPSC/2 and iPSC/860 are roughly 59 and 1000, respectively. Note that the communication cost dominates the arithmetic cost in the row-oriented approach.

TABLE 3
Arithmetic and communication complexities

algorithm	arithmetic	communication
special block partition	$\frac{1229}{4}\frac{k^3}{p} - c_1\frac{k^3 \log p}{p^{3/2}} - c_2\frac{k^3}{p^{3/2}} + O(\frac{k^2}{\sqrt{p}})$	$c_3\frac{k^2\tau}{\sqrt{p}} - c_4\frac{k^2\tau}{p} + O(\frac{k\tau \log p}{\sqrt{p}})$
row-oriented partition	$\frac{146}{3}\frac{k^3}{p} + 31\frac{k^2 \log k}{p} + O(\frac{k^2}{p})$	$\frac{146}{12}\frac{k^3\tau}{p} - \frac{371}{12}\frac{k^3\tau}{p^{3/2}} + O(k^2\tau \log p)$

The results shown in Table 3 are the complexity results of the algorithms which are based on two extreme partitioning schemes in which an $h \times h$ frontal matrix distributed among p processors is partitioned into h and p blocks, respectively. Therefore, it can be expected that the arithmetic complexity of our algorithm based on an arbitrary equal-row or equal-volume partitioning scheme is between the complexity bounds of the row-oriented algorithm and our algorithm with the special block partition. Similarly, the communication complexity of our algorithm based on an arbitrary equal-row or equal-volume partitioning scheme is between the complexity bounds of our algorithm with the special block partition and the row-oriented algorithm.

5. Experimental results. Our parallel multifrontal sparse QR factorization algorithm has 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 large and sparse linear least squares problems arising from particle methods for modelling turbulent combustion [25] and a large problem NIMBUS from the Bramley's test set used in [21]. The sparse linear least squares problems for modelling turbulent combustion correspond to three-dimensional $k \times k \times k$ twenty-seven point 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 problems for modelling turbulent combustion are ordered by the nested dissection ordering [7]. The general sparse problem is ordered by the minimum degree ordering [12].

TABLE 4
Characteristics of the test problems

problem	m	n	$\|A\|$	$\|R\|$
GRID100	39204	10000	156816	297953
GRID200	158404	40000	633616	1481803
GRID255	258064	65025	1032256	2569386
TC20	54872	8000	438976	1298284
TC24	97336	13824	778688	2806944
TC27	140608	19683	1124864	4665657
NIMBUS	23871	1325	181972	136356

The characteristics of the test problems are shown in Table 4, where m is the number of rows, n the number of columns, $|A|$ number of nonzeros in matrix A, and $|R|$ number of nonzeros in factor R. "GRIDk" represents a $k \times k$ nine-point grid. "TCk" represents a problem for modelling turbulent combustion and corresponds to a three-dimensional $k \times k \times k$ twenty-seven point grid. The QR factorization method described in [14] is used to solve these sparse linear least squares problems on an Intel iPSC/860 machine. The sparse QR factorization is computed by our parallel multifrontal approach described in §3. Both the equal-row partitioning scheme and the revised equal-volume partitioning scheme are used. The orthogonal transformations are applied to the right hand side vector b and then discarded. The required parallel sparse back substitution algorithm is described in [29, 31]. The column-oriented kernel is used in our implementation.

Let h be the order of a frontal matrix that is assigned to p processors. We explore four instances of the equal-row partitioning scheme with block sizes equal to 1, h/p^2, $s(h,p)$ as defined in §3.2 and h/p, respectively. For convenience, the versions of our parallel sparse QR factorization algorithm, which use these four instances of the equal-row partitioning scheme in their kernels, are referred to as algorithm ERP1, ERP2, ERP and ERPP, respectively. We explore two instances of the revised equal-volume partitioning scheme with block sizes corresponding to $s(h,p)$ as defined in §3.2 and h/p, respectively. The versions of our parallel sparse QR factorization algorithm, which use these two instances of the revised equal-volume partitioning scheme in their kernels, are referred to as algorithm EVP and EVPP, respectively. Note that ERP1 is the row-oriented approach proposed in [5] and ERP2 is only suggested as a future research topic in [5]. ERP and EVP are algorithms with optimal block partition.

We evaluate the practical performance of these six parallel sparse QR factorization algorithms on a 32-node Intel iPSC/860 machine. All algorithms are implemented in C and double-precision

TABLE 5
Message count per processor ($p = 32$)

Test Problem	Fact Alg	Total # Msgs max	min	avg	# Msgs Sent max	min	# Msgs Recv max	min
GRID200	ERP1	50838	27344	37824.2	25513	13685	25466	13659
	ERP2	5972	5309	5657.6	3002	2648	3005	2650
	ERP	464	327	390.94	239	161	238	159
	ERPP	113	17	67.56	60	5	54	12
	EVP	284	175	226.25	146	86	144	85
	EVPP	94	17	59.62	48	6	46	11
GRID255	ERP1	83440	43165	61365.2	41841	21568	41599	21596
	ERP2	9188	8351	8817.9	4606	4159	4610	4173
	ERP	465	324	387.4	238	161	236	156
	ERPP	111	17	67.3	59	5	55	12
	EVP	331	178	254.4	169	91	170	87
	EVPP	97	17	64.0	54	6	49	11

floating-point arithmetic is used. A dense Givens rotation is implemented as a call to the subroutine `drot` in the highly optimized Basic Math Library provided by Intel. No assembler code other than calls to `drot` is used. The compilation optimization level used is -O4.

We have obtained the running times, communication statistics, and load balance results of the six algorithms for all of our test problems. To save space, the communication statistics and load balance results are presented only for regular grid problems GRID200 and GRID255 on 32 processors. The relative performance of the six algorithms for regular grid problems on smaller number of processors is the same as that on 32 processors. The relative performance of the six algorithms for other problems is similar to that for these two regular grid problems.

In Table 5, messages communicated per processor are tabulated. The column under "Fact Alg" shows the algorithm used in the numerical factorization. The "max", "min", and "avg" under the "Total # 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 "# Msgs Sent" are the maximum number and minimum number of messages sent by a processor, respectively. The "max" and "min" under the "# Msgs Recv" are with respect to the 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 equal to half of the average number of messages communicated by a processor. For both the equal-row partitioning scheme and the revised equal-volume partitioning scheme, the number of messages required decreases as the block size increases. The number of messages required by EVP is less than that required by ERP since EVP partitions a frontal matrix into less number of blocks than ERP does. Similarly, EVPP communicates less number of messages than ERPP.

In Table 6, volume of messages communicated per processor is shown. The total volume communicated by a processor is defined as the total number of floating-point double-precision numbers communicated by that processor. The volume sent by a processor is defined as the total number of floating-point double-precision numbers sent by the processor. The volume received is similarly defined. The last column in Table 6 is the average length of all messages obtained by dividing the total communication volume by total number of messages. For both the equal-row partitioning scheme and the revised equal-volume partitioning scheme, the required communication volume decreases as the block size increases.

TABLE 6
Communication volume per processor (scaled by 10^{-3}, $p = 32$).

Test Problem	Fact Alg	Total Volume max	min	avg	Volume Sent max	min	Volume Recv max	min	avg msg length
GRID200	ERP1	3562.0	2054.0	2781.6	1783.4	1023.5	1778.7	1028.3	0.074
	ERP2	795.6	734.5	762.9	401.0	362.9	394.7	367.9	0.135
	ERP	261.7	197.6	228.7	132.8	96.7	128.8	99.9	0.585
	ERPP	139.9	23.1	94.3	71.7	7.9	68.2	15.2	1.396
	EVP	250.0	180.2	208.6	126.4	86.0	123.7	90.0	0.922
	EVPP	137.3	38.7	100.7	70.2	14.9	67.7	23.8	1.690
GRID255	ERP1	7349.4	4294.3	5751.2	3678.5	2142.0	3670.9	2148.7	0.094
	ERP2	1504.6	1419.4	1460.6	756.0	704.7	750.9	711.5	0.166
	ERP	416.0	311.0	364.3	211.4	152.3	204.6	158.8	0.940
	ERPP	228.6	40.3	154.1	117.2	11.7	111.4	28.6	2.291
	EVP	411.9	302.1	350.9	208.4	147.1	203.5	149.8	1.380
	EVPP	221.0	69.2	166.3	115.5	24.4	108.5	42.8	2.599

Consider Table 5 and Table 6 together. Clearly, ERP1 and ERP2 require excessive number of messages and excessive communication volume. For an $h \times h$ frontal matrix partitioned among p processors, the block size for ERP2 is $s = h/p^2$. For reasonable large problem, s is too small. For instance, the root frontal matrix of TC27 is 729×729 and is partitioned among 32 processors. Therefore, s is less than one and is set to one in our experiment. The order of the six algorithms in decreasing communication requirements is ERP1, ERP2, ERP, EVP, ERPP and EVPP. EVPP and EVP are better than ERPP and ERP in terms of communication costs, respectively. The four algorithms ERP, ERPP, EVP and EVPP communicate relatively long messages. This feature is desirable for distributed-memory multiprocessors since the overhead associated with short messages is relatively higher than that associated with long messages.

TABLE 7
Load distribution among processors ($p = 32$)

Problem	Fact Alg	Max/Avg	Min/Avg
GRID200	ERP1	1.219	0.717
	ERP2	1.290	0.726
	ERP	1.260	0.707
	ERPP	1.514	0.428
	EVP	1.245	0.731
	EVPP	1.313	0.540
GRID255	ERP1	1.230	0.720
	ERP2	1.310	0.705
	ERP	1.273	0.690
	ERPP	1.538	0.412
	EVP	1.254	0.700
	EVPP	1.323	0.531

Load balance results are shown in Table 7. "Max/Avg" is the ratio of the maximum number of floating-point operations performed by a processor to the average number of floating-point operations performed by a processor during the numerical factorization. "Min/Avg" is the ratio of the

Table 8

Performance of the numeric phase of the parallel sparse QR factorization algorithms on an iPSC/860

test problem	no. procs	ERP2 time	ERP2 mflops	ERP time	ERP mflops	EVP time	EVP mflops
GRID100	1	17.165	3.971	17.165	3.971	17.165	3.971
	2	8.651	7.878	8.750	7.789	8.780	7.763
	4	4.450	15.316	4.460	15.281	4.487	15.189
	8	2.790	24.428	2.479	27.493	2.487	27.404
	16	1.760	38.724	1.589	42.892	1.446	47.133
	32	1.384	49.245	1.115	61.125	0.932	73.128
GRID200	8	13.567	37.780	13.048	39.283	13.074	39.205
	16	8.754	58.552	7.807	65.655	7.389	69.369
	32	6.689	76.628	4.992	102.678	4.270	120.039
GRID255	32	12.042	85.448	8.124	126.657	7.134	144.234
TC20	8	26.646	53.094	25.941	54.537	24.261	58.314
	16	21.686	65.238	15.454	91.546	13.747	102.913
	32	16.663	84.904	10.123	139.756	8.665	163.272
TC24	16	47.042	86.840	40.040	102.026	34.431	118.646
	32	38.662	105.662	25.608	159.525	21.316	191.645
TC27	32	63.256	127.571	43.766	184.38	35.946	224.494
NIMBUS	1	72.336	8.210	72.336	8.210	72.336	8.210
	2	40.319	14.824	40.277	14.840	40.160	14.883
	4	42.263	15.137	42.624	15.008	39.960	16.009
	8	32.383	20.145	32.660	19.974	29.697	21.967
	16	32.430	20.317	25.247	26.098	23.632	27.881
	32	39.677	16.883	23.874	28.058	22.362	29.955

Table 9

Performance of the numeric phase of the parallel sparse QR factorization algorithms on an iPSC/860

test problem	no. procs	ERP1 time	ERP1 mflops	ERPP time	ERPP mflops	EVPP time	EVPP mflops
GRID100	1	17.165	3.971	17.165	3.971	17.165	3.971
	2	9.620	7.085	8.810	7.736	8.802	7.743
	4	6.699	10.174	4.599	14.819	4.583	14.871
	8	5.500	12.392	2.692	25.318	2.661	25.612
	16	4.069	16.750	1.642	41.507	1.540	44.256
	32	3.072	22.186	1.056	64.541	1.019	66.884
GRID200	8	31.591	16.225	14.760	34.727	14.036	36.518
	16	24.191	21.188	8.845	57.950	8.149	62.899
	32	17.712	28.939	5.436	94.291	5.072	101.058
GRID255	32	34.295	30.003	9.740	105.643	8.831	116.517

minimum number of floating-point operations performed by a processor to the average number of floating-point operations performed by a processor during the numerical factorization. A floating-point operation or 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 during the numerical factorization. Roughly, the order of the six algorithms from the least balanced workload distribution to the most balanced workload distribution is ERPP, EVPP, ERP2, ERP, EVP and ERP1. The difference in workload distribution among ERP2, ERP, EVP and ERP1 is relatively small. However, the difference between these four algorithms and ERPP or EVPP is fairly pronounced. EVPP and EVP produce better load distribution than ERPP and ERP, respectively.

The running times of the three algorithms ERP2, ERP and EVP for all test problems on a 32-node Intel iPSC/860 are shown in Table 8. To save space, the running times of the three algorithms ERP1, ERPP and EVPP are shown in Table 9 for the regular grid problems only. The column under "time" is the running time for the numerical factorization in seconds which is obtained by measuring the time spent on each processor and taking the maximum time spent on a processor. All processors start at the same time. The "mflops" is the execution rate defined as the number of mega flops performed per second during the numerical factorization. The "time" in Table 8 or Table 9 includes the time for applying the Givens rotations to the right hand side vector b for all six algorithms—i.e., the time for $Q^T b$ is included in the numerical factorization time.

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 $p = 2^i (0 \leq i \leq 5)$ processors, its timing results on p processors are reported. The running time for a problem on one processor is the time spent by the best serial sparse QR factorization algorithm we have for that problem on one processor.

From Table 8 and Table 9 we observe that the order of the six algorithms in increasing efficiency is ERP1, ERP2, ERPP, EVPP, ERP and EVP. These timing results are in agreement with the communication and workload distribution statistics shown in Table 5, Table 6 and Table 7. Because of the excessive communication requirements, ERP1 and ERP2 are significantly worse than the other four algorithms in performance. ERPP and EVPP are inferior to ERP and EVP. Since EVP requires less number of messages and less communication volume than ERP, and EVP produces more balanced workload distribution than ERP, EVP is more efficient than ERP. Similarly, EVPP is more efficient than ERPP. It is worth noting that EVPP looks promising. One of the advantages of EVPP or ERPP is that it is not necessary to determine the optimal block partitions for a new machine. The performance of all six algorithms for the general sparse problem NIMBUS is worse than that for regular grid problems because of the unbalanced supernodal elimination tree associated with NIMBUS. Speedups for problem GRID100 are shown in Table 10.

Table 10
Speedups on problem GRID100

no. procs	ERP1	ERP2	ERP	ERPP	EVP	EVPP
2	1.78	1.98	1.96	1.95	1.96	1.95
4	2.56	3.86	3.85	3.73	3.82	3.75
8	3.12	6.15	6.92	6.38	6.90	6.45
16	4.22	9.75	10.80	10.45	11.87	11.15
32	5.59	12.40	15.39	16.25	18.42	16.84

6. Concluding remarks. We have described an efficient block-oriented approach to parallel multifrontal sparse QR factorization on distributed-memory multiprocessors. We have proposed novel block schemes for partitioning frontal matrices and explored a spectrum of block partitioning schemes ranging from the conventional single row partition to the block partition with maximal block size. Our block-oriented approach results in an efficient parallel sparse QR factorization algorithm with low communication costs, relatively balanced arithmetic work distribution among processors and a fast parallel numerical kernel due to the enhanced cache utilization. We have presented both complexity results of our parallel algorithm for regular grid problems and experimental results of an implementation of our parallel algorithm on an Intel iPSC/860 machine. We have demonstrated that our new block-oriented algorithm achieves significant improvement in performance over the conventional non-block algorithm [5].

Our approach is especially suitable for those machines which communicate long messages much more efficiently than they communicate short messages such as the IBM SP1 system since relatively long messages are used in our algorithm. An implementation of our parallel sparse QR factorization algorithm on an IBM SP1 system is currently being pursued. We are also extending our algorithm to compute the QR factorization of a sparse matrix with rank deficiency in parallel.

In the implementation of our parallel sparse QR factorization algorithm, the orthogonal matrix Q is not computed since it is often too expensive to store Q in the main memory when A is large. A sparse linear least squares problem $\min_x \|Ax - b\|$ is solved by applying the orthogonal transformations to the right hand side vector b during the sparse QR factorization. This approach is numerically backward stable. However, the right hand side vector must be available before A is factorized. The method of corrected semi-normal equations(CSNE) proposed by Björck [4] can be used to handle new right sides without storing Q. Björck [4] 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 "stiff" problems. A parallel multifrontal algorithm for solving sparse linear least squares problems on distributed-memory multiprocessors based on the CSNE method is described in [31].

Acknowledgements. I would like to thank Professor Thomas F. Coleman for many discussions relating to this work and for his helpful comments and suggestions on the manuscript. I would also like to thank the referees for their constructive comments and suggestions. Numerical experiments were performed on a 32-node Intel iPSC/860 machine in the Computer Science Department, Cornell University.

REFERENCES

[1] C. Ashcraft, *A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems*, Tech. Report ECA-TR-51, Boeing Computer Services, Seattle, WA, 1987.

[2] C. Ashcraft and R. G. Grimes, *The influence of relaxed supernode partitions on the multifrontal method*, ACM Transactions on Mathematical Software, 15 (1989), pp. 291–309.

[3] C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, Int. J. Supercomputer Appl., 1 (1987), pp. 10–30.

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

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

[6] T. H. Dunigan, *Performance of the Intel iPSC/860 hypercube*, Tech. Report 11491, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

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

[8] J. A. George and M. T. Heath, *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra and its Appl., 34 (1980), pp. 69–83.
[9] J. A. George and J. W. H. Liu, *An optimal algorithm for symbolic factorization of symmetric matrices*, SIAM J. Comput., 9 (1980), pp. 583–593.
[10] ———, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.
[11] ———, *Householder reflections versus Givens rotations in sparse orthogonal decomposition*, Linear Algebra and its Appl., 88/89 (1987), pp. 223–238.
[12] ———, *The evolution of the minimum degree algorithm*, SIAM Review, 31 (1989), pp. 1–19.
[13] 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.
[14] G. Golub, *Numerical methods for solving linear least squares problems*, Numer. Math., 7 (1965), pp. 206–216.
[15] Intel, *iPSC/2 and iPSC/860 User's Guide*, Intel SSD, Beaverton, OR, 1991.
[16] S. G. Kratzer, *Massively parallel sparse matrix computations*, Tech. Report SRC-TR-90-008, Supercomputing Research Center, Bowie, MD, February 1990.
[17] 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.
[18] J. W. H. Liu, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Trans. on Math. Software, 12 (1986), pp. 127–148.
[19] ———, *On general row merging schemes for sparse Givens transformations*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 1190–1211.
[20] ———, *The multifrontal method for sparse matrix solution: theory and practice*, SIAM Review, 34 (1992), pp. 82–109.
[21] S. Lu and J. L. Barlow, *Parallel computation of orthogonal factors of sparse matrices*, in Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1993, pp. 486–490.
[22] P. Matstoms, *The Multifrontal Solution of Sparse Linear Least Squares Problems*, PhD thesis, Linköping University, Sweden, 1991.
[23] D. J. Pierce and J. G. Lewis, *Sparse rank revealing QR factorization*, Tech. Report MEA-TR-193, Boeing Computer Services, Seattle, WA, May 1992.
[24] 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.
[25] S. B. Pope, *Particle methods for turbulent combustion.* private communication, 1991.
[26] A. Pothen and C. Sun, *A mapping algorithm for parallel sparse Cholesky factorization*, SIAM J. Sci. Comput., 14 (1993), pp. 1253–1257.
[27] P. Raghavan, *Distributed Sparse Matrix Factorization: QR and Cholesky Decompositions*, PhD thesis, Pennsylvania State University, University Park, PA, 1991.
[28] C. Sun, *Some Applications of Clique Trees to Sparse Cholesky Factorizations*, PhD thesis, Pennsylvania State University, University Park, PA, 1991.
[29] ———, *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.
[30] ———, *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).
[31] ———, *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.