SPARSE JACOBIAN ESTIMATION AND FACTORIZATION ON A MULTIPROCESSOR[*]

PAUL E. PLASSMANN[†]

Abstract. In this paper we present algorithms and experimental results for the estimation and QR factorization of large, sparse Jacobians on a message-passing multiprocessor. The gist of this work is the development of paradigms for the efficient solution of the "inner loop" of a nonlinear optimization algorithm: the estimation of the Jacobian, its factorization, and the solution of the resulting trust-region problem. A parallel sparse QR factorization based on the global row reduction algorithm is introduced. We emphasize the commonality between row partitions that allow for the efficient parallel factorization of the Jacobian and its estimation. We also note that the interprocessor communication structure constructed for the QR factorization can be used to solve an associated trust-region problem. Finally, experimental results obtained on the Intel iPSC/2 are presented.

1. Introduction.

To solve many nonlinear optimization problems it is necessary to estimate and factor the Jacobian of a nonlinear function $F : \mathbf{R}^n \mapsto \mathbf{R}^m$, with $m \geq n$. In large scale optimization problems this Jacobian is often sparse and the efficient solution of these problems depends on the utilization of this structure. In this paper we will consider the problem of developing efficient algorithms for the "inner loop" of a nonlinear optimization algorithm: the estimation of the Jacobian, its QR factorization, and the solution of the resulting trust-region problem on a distributed memory computer.

The parallel QR factorization algorithm presented in this paper is based on the concept of row merge heaps introduced by Liu [10] as a means of reducing the incidental fill incurred during a sequential row-oriented QR factorization. In the parallel algorithm, the leaves of this row merge tree are partitioned by determining a special set of vertices in the row merge heap. These vertices, called foundation vertices, yield an initial row distribution, or assignment of rows to processors. Thus, the computation of the upper trapezoidal matrices associated with these foundation vertices is entirely local to each processor. To handle the interprocessor communication that is required to further reduce these matrices, we introduce the global row reduction algorithm. This algorithm, which attempts to minimize interprocessor communication, is described in section 2.

Related to the symbolic factorization phase of the global row reduction algorithm are several algorithmic problems which we discuss in section 3. We note that the row merge heap can be efficiently computed in parallel by an almost linear time forest merging algorithm. The concept of a foundation vertex in the row merge heap is introduced, and we give a characterization of the set of row partitions that can be represented by foundation vertices. An algorithm is presented for computing a set of foundation vertices which in turn determines a suitable initial row partition. This algorithm is a heuristic which seeks to minimize the required interprocessor communication while balancing the amount of local work required of each processor. We discuss how the required communication is computed, stored, and can be reutilized during the subsequent factorization of matrices with the same nonzero structure. As an aside, we note that an additional problem posed by the symbolic

[*] Presented at the Mathematical Sciences Institute workshop on Large-Scale Numerical Optimization, Cornell University, October 19-20, 1989.

[†] Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439; previously, Center for Applied Mathematics, 305 Sage Hall, Cornell University, Ithaca, NY 14853. Research partially supported by the Computational Mathematics Program of the National Science Foundation under grant DMS-8706133 and by the U.S. Army Research Office through the Mathematical Sciences Institute, Cornell University.

factorization is to develop a consistent means for processors to determine locally the set of processors involved at each stage of the factorization.

For these algorithms experimental results obtained on the Intel iPSC/2 hypercube are presented in section 4. We find that the global row reduction algorithm can incur slightly more incidental fill than the sequential algorithm. However, this extra work is offset by the elimination of some of the interprocessor communication that would be required by nonlocal merging of the upper trapezoidal matrices. In addition, it seems that this approach is more amenable to different load balancing schemes and can be employed with any column ordering heuristic.

To solve nonlinear least-squares problems, the Levenberg-Marquardt approach requires the solution of a sequence of trust-region problems [11]. In section 5 we show that the global row reduction algorithm can be modified to solve the matrix problem that arises in solving these trust-region problems. This modification allows the use of the interprocessor communication structure that is generated for the QR factorization without requiring an additional symbolic factorization step. In section 6 we show how the elements of the Jacobian can be estimated in a natural manner using the initial row distribution. We note that in some instances an intersection graph coloring may not be adequate and a multicoloring or full matrix method must be used.

2. A Parallel Sparse QR Factorization Algorithm. The sparse QR factorization algorithm described in this section is based on the concept of the *row merge heap* first introduced by Liu [10]. In this context, let A be a sparse $m \times n$ matrix, $m \geq n$, of full column rank. We wish to factor A into the matrix product QR, where Q is an orthogonal $m \times m$ matrix and R is an upper triangular $m \times n$ matrix. This factorization is realized by computing a sequence of elementary orthogonal transformations $Q_1^T, Q_2^T \ldots, Q_K^T$ that reduces A to upper triangular form. It is usually impractical to explicitly form Q; instead, these transformations are applied to both sides of the linear system $Ax \overset{\text{L.S.}}{=} b$. The resulting upper triangular system $Rx \overset{\text{L.S.}}{=} Q^T b$ can then be solved for x by back substitution.

The sparsity structure of the upper triangular matrix R is determined by an ordering of the columns of A. Mathematically R is equal to the Cholesky factor of $A^T A$, but it is known that the sparsity structure predicted by the symbolic Cholesky factorization of $A^T A$ may overestimate the sparsity structure of R. However, if A is reordered into block upper triangular (Dulmage-Mendelsohn) form, it can be shown [2] that for each diagonal block these two structures are equal (assuming no accidental numerical cancellation). Equivalently, if the bipartite graph representation of A has the strong Hall property, then the two structures are equal; we assume this to be the case for the remainder of this paper.

Given the equivalence of the nonzero structure of the orthogonal and Cholesky factors, a column ordering can be chosen using existing heuristics for reduction of fill in factoring symmetric positive definite systems [7, 8]. Hence, the resulting nonzero structure of R will be reasonably sparse and therefore one expects that the computation required to obtain R will be reduced with respect to column orderings that produce more fill.

The amount of computation required to reduce A to upper triangular form can vary dramatically depending on what type of orthogonal transformations are used, and in what manner they are applied. Sequences of Givens rotations have been shown to be very effective in performing this reduction [6, 8]. Consider the Givens rotation $G_{ij}^{(k)}$ which operates on rows r_i and r_j of the matrix to zero the k-th element of row r_j. If $S(r_i)$ and $S(r_j)$ are the sorted lists of nonzeros in the rows before the application of the Givens rotation, then the structures after the rotations, $S(r_i^{'})$ and $S(r_j^{'})$, are given by

$$
\begin{array}{lrcl}
(1) \qquad & S(r_i^{'}) & = & S(r_i) \cup S(r_j) \\
& S(r_j^{'}) & = & (S(r_i) \cup S(r_j)) \setminus \{k\} \quad .
\end{array}
$$

As a result, during the factorization process rows with leading nonzeros in the same column inherit the structure of the rows they are merged with.

2.1. The Row Merge Heap. This inheritance of structure can be compactly represented by the *row merge heap* data structure [10]. For a given column ordering, the row merge heap, $H(A)$, is constructed via the following rules:

1. The leaves of H, $r_1, r_2, \ldots, r_m$, represent the rows of A. The interior vertices of H are labeled by the columns, $c_1, c_2, \ldots, c_n$, of A.
2. Each vertex x in H has an associated structure (a sorted list) $S(x)$. For the leaves of H, the structures $S(r_i)$ are just the initial nonzero structures of the rows of A. In addition, a vertex x is a child of the vertex $c_k \Leftrightarrow S(x) = \{k, \ldots\}$.
3. The formula $S(c_k) = (\bigcup_{\{\text{children } x\}} S(x)) \cap \{k+1, \ldots, n\}$ gives the structure of an interior vertex c_k.

For example, consider the matrix structure shown in Figure 1. If the rows and columns

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & \times & & \times & & & \times \\
2 & \times & & \times & & & \\
3 & \times & & & & & \times \\
4 & & & \times & & & \times \\
5 & & & \times & & & \times \\
6 & & & \times & \times & & \\
7 & & \times & & \times & \times & \\
8 & & \times & & & \times & \\
9 & & \times & & \times & \times & \\
10 & & & & & \times & \times \\
11 & & & & & \times & \\
12 & & & & & \times & \times
\end{array}
$$

FIG. 1. *An example of a matrix nonzero structure.*

of the matrix are ordered as shown in the figure, then by using the rules described above one can construct the row merge heap shown in Figure 2.

Based upon the row merge heap, Liu suggests a sequential sparse QR factorization algorithm. In this algorithm, a binary splitting $H^{'}$ of the row merge heap is generated. A binary splitting of the row merge heap is constructed by adding interior vertices to the heap until each vertex has no more than two children. A binary splitting of the heap in Figure 2 is shown in Figure 3. Associated with each vertex in the binary heap $H^{'}$ is an *essentially full* upper trapezoidal matrix. A sparse matrix is essentially full if, after all zero rows and columns of the matrix are removed, the resulting matrix can be permuted

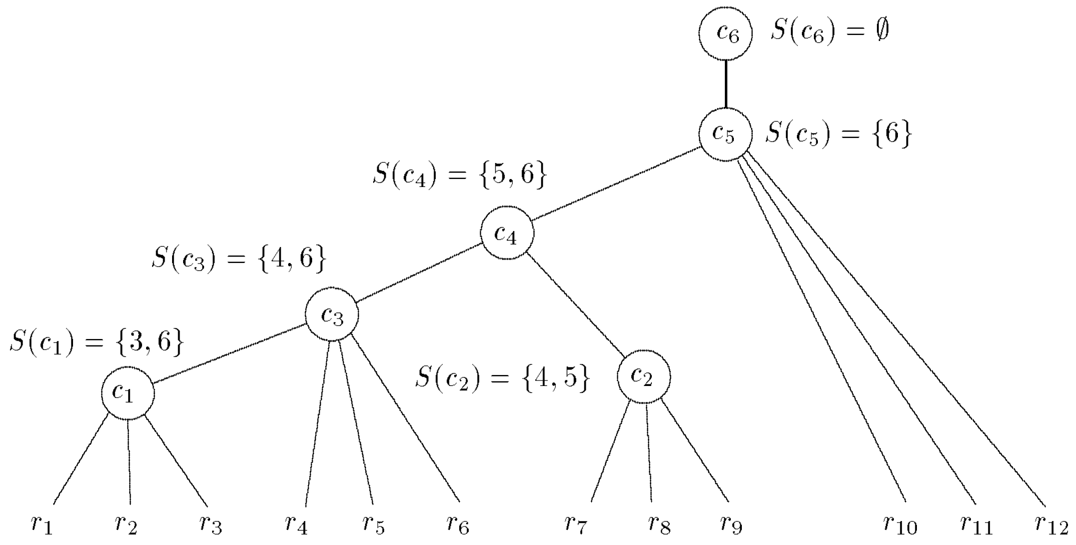
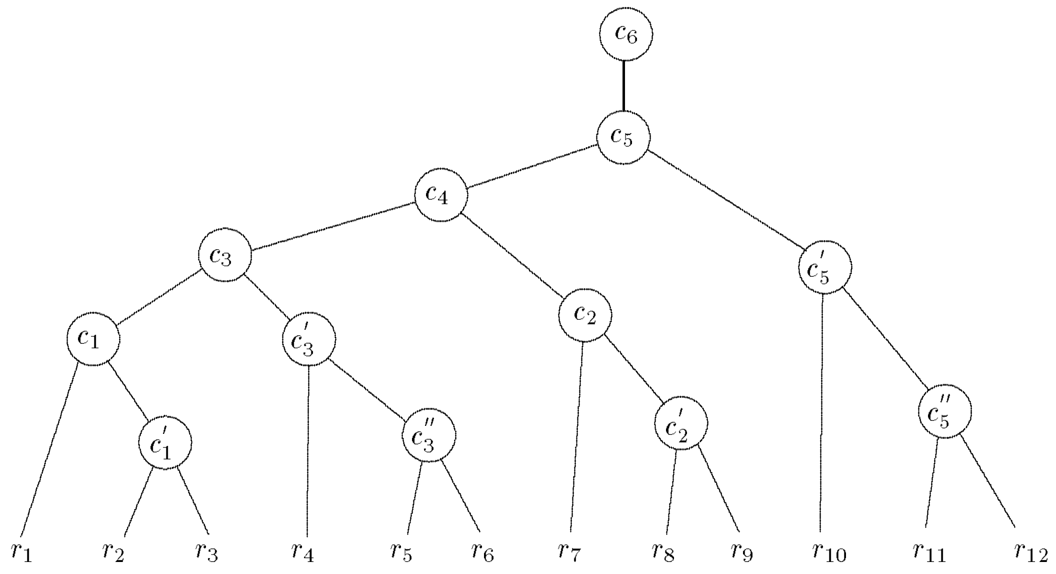


FIG. 2. *The row merge heap constructed from the matrix in Figure 1.*



c_6
c_5
c_4
c_3
c_5'
c_1
c_3'
c_2
c_1'
c_3''
c_2'
c_5''
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12}

FIG. 3. *A binary splitting of the row merge heap in Figure 2.*

to form a dense upper trapezoidal matrix. In the sequential algorithm, the vertices of the binary heap are visited in a postorder traversal of the heap. The upper trapezoidal matrix associated with each vertex is generated by merging the matrices corresponding to the two children of that vertex. In general, these matrices may not be essentially full, however, if the number of rows involved in the merge exceeds the number of resulting columns, then the resulting matrix is essentially full [10]. Thus to generate the factor R, the row merge heap is "evaluated" from the leaves of the row merge tree to the root. The top row of the matrix associated with vertex c_k yields the k-th row of the matrix R.

To show how these merges work in the context of the example shown in Figure 1, denote by $T(v)$ the upper trapezoidal matrix associated with a vertex v in the heap. For the splitting shown in Figure 3, the matrices corresponding to the vertices $c_1^{'}$ and c_1 are shown in Figure 4. We denote a nonzero in the matrix with $\times$, a zero that was filled in

$$T(r_1) \;=\; \begin{array}{cccccc} \times & 0 & \times & 0 & 0 & \times \end{array} \qquad T(c_1^{'}) \;=\; \begin{array}{cccccc} \times & 0 & \times & 0 & 0 & f \\ * & 0 & f & 0 & 0 & \times \end{array}$$

$$R_{1*} \;=\; \begin{array}{cccccc} \times & 0 & \times & 0 & 0 & \times \end{array} \qquad T(c_1) \;=\; \begin{array}{cccccc} * & 0 & \times & 0 & 0 & \times \\ & & * & 0 & 0 & \times \end{array}$$

FIG. 4. *The structures of the matrices involved in computing* $T(c_1)$.

during the merging of the two child matrices of the vertex by f, and denote by $*$ a nonzero that was eliminated during the merging. When the matrix corresponding to a column vertex is formed, the top row of the matrix is a completed row of the factor R which is then removed from the matrix and placed in storage allocated for the factor. In Figure 4 the completed first row of R is denoted by R_{1*}. Also note that the nonzero structure of $T(c_1)$ is given by $S(c_1)$, that the matrices are essentially full, and that the structure of R_{1*} is given by $S(c_1) \cup \{1\}$.

2.2. The Global Row Reduction Algorithm. When considering a parallel algorithm based on the row merge heap concept, it is natural to consider a data distribution that preserves the "row-oriented" character of the algorithm. Thus, in this section, we assume that the matrix is distributed to the processors by rows. In addition, since m can be much greater than n, it is reasonable to prefer a row to a column distribution since then the computational efficiencies would probably depend on m/p instead of n/p. In this section we contrast two possible parallel row-oriented algorithms for computing the QR factorization of a sparse matrix: the nonlocal merge algorithm and the global row reduction algorithm.

Given that the rows of the matrix are distributed to the processors, the row merge heap displays the computational dependency of final rows of the factor R upon the initial rows of A. For example, in Figure 3 the vertex c_1 has descendant rows r_1, r_2, and r_3 and, consequently, one uses only these rows to compute the first row of R. This observation suggests an approach for several possible parallel algorithms. Suppose we have 4 processors. Then rows 1-3 could be assigned to processor 1, rows 4-6 to processor 2, rows 7-9 to processor 3, and rows 10-12 to the fourth processor. Based on the row merge heap shown in Figure 3, the computation of the upper trapezoidal matrices associated with the vertices c_1, c_3', c_2, and c_5' can all be done locally (on the processor to which the rows were initially assigned). This computation can be done using Liu's sequential algorithm. After these local results are computed, some sort of communication is required between processors 1 and 2 to compute the matrices associated with c_3, and then between processors 1, 2, and 3 for c_4, and so forth.

One possibility would be to continue with the binary merge algorithm, perform exactly the same computation that is done in the sequential algorithm, but now perform the merges on data that is not always local. Denote this algorithm as the *nonlocal merge* algorithm. Since each step requires the merging of two upper trapezoidal sparse matrices, the approach we might consider is a generalization of a dense upper triangular matrix merging algorithm, originally proposed in the context of solving positive definite trust region problems [5]. In this nonlocal merging algorithm, the rows of the two upper trapezoidal matrices are wrapped onto an embedded ring of processors using the structure of the parent in the binary merge. This algorithm is discussed in more detail elsewhere [13].

A different approach to the problem of interprocessor communication is the *global row reduction* algorithm. In this algorithm the redistribution of rows required by the binary merges is avoided, instead a global reduction of rows is executed to compute each row of R. Each processor maintains a set of upper trapezoidal matrices; among these matrices local merges are done whenever possible. However, if interprocessor communication is required, say to compute the k-th row of R, then the processors communicate according to a *reduction tree* E_k. The reduction tree is a rooted tree whose vertices represent a subset of the processors. The edges of this tree represent communication between these processors, and the order in which a parent processor communicates with its children is specified.

Based on the reduction tree E_k, a processor receives rows (in a particular order) from its children. The processor merges each of these rows to eliminate the nonzero in column k, and then sends the resulting row back to the processor that originally sent it. After the rows from its children processors have been processed, the processor sends its row to its parent in the reduction tree and waits for the row to be returned with the nonzero in column k eliminated. Following the global row reduction, the processor at the root of the reduction tree contains the computed row R_{k*}. Each processor does any possible local merging and then participates in the next global row reduction for which it contains a nonzero. Figure 5 presents a description of the global row reduction algorithm.

An example of a reduction tree is shown in Figure 6 involving 6 processors: a, b, c, d, e, and *root*. We denote the structures of the leading row in the upper trapezoidal matrix at each processor by A, B, C, D, E, and *Root*. The reduction tree is "evaluated" from the leaves up to the root; the edge numbers represent the order in which a vertex reduces the rows sent by its children. One can think of the order in which the rows are processed as a sort of "parallel postorder" traversal of the reduction tree. For example, in Figure 6 processor c first receives a row from processor a, computes and then applies a Givens rotation to the two rows to zero the first element. Processor c then sends the resulting row back to a and acts on the row sent from processor b. The row sent back to processor a inherits the structure of the row at processor c, hence $A' = (A \cup C) \setminus \{k\}$. At the same time, processor *root* can receive and process the row sent by processor e. Processor e then receives its modified row with the new structure $E' = (E \cup Root) \setminus \{k\}$. The row reduction

```
$L_v$ = List of upper triangular matrices, initially the rows
    assigned to processor $v$;

**Proc** $(v)$ : {program for processor $v$}
    **For** $k = 1, \ldots, n$ **do**
        Perform local merges on matrices in $L_v$ with leading nonzero $k$;
        **If** global reduction required for $T \in L_v$
            **For** each child $u$ of $v$ in $E_k$ (in order given) **do**
                Receive row $r$ from processor $u$;
                Merge $r$ with the top row of $T$;
                Send $r$ back to child processor $u$;
            **enddo**
            **If** $v = root$ **then**
                Store top row of $T$ in data structure for $R_{k*}$;
            **else**
                Send top row of $T$ to parent processor of $v$ in $E_k$;
                Receive top row of $T$ from parent processor;
                Bring $T$ back to upper triangular form;
            **endif**
        **endif**
    **enddo**
```

FIG. 5. *The global row reduction algorithm.*

continues until R_{k*} has been computed at processor *root*, and each processor has received its modified row.

Consider the resulting situation if the same six processors shown in Figure 6 had been involved at step k of the nonlocal merge algorithm. In this case the result would be one upper trapezoidal matrix with nonzero structure $S^{'} = (A \cup B \cup C \cup D \cup E \cup Root) \setminus \{k\}$ wrapped onto the six processors. Consequently, all these processors would be involved in the future reduction steps given by the nonzeros in $S^{'}$. In the global row reduction algorithm, not all of these processors are involved in every step since each processor does not inherit the union of the structures. More work has been done by the nonlocal merge algorithm because we are left with only one upper trapezoidal matrix distributed across six processors, as opposed to the global row reduction algorithm which leaves six sparser upper trapezoidal matrices, each with a different structure on a different processor. However, the experimental results presented in Section 4 show that over all the reduction steps, more intermediate fill can be generated by the global row reduction algorithm, leading to more total arithmetic work. The advantage of the global row reduction algorithm is an increase in fine-grain parallelism and a decrease in the interprocessor communication.

Implicit in this approach is an assignment of the rows of R to processors. For a more explicit example of the algorithm consider the matrix in Figure 1, its row merge heap shown in Figure 3, and the initial distribution of rows to processors described above. Suppose processor 1 is assigned the first row of R, processor 3 the second row, and processor 2 the third and fourth rows of R. Then Figure 7 depicts the interprocessor communication of rows required to compute R_{4*} given one possible reduction tree. In the figure we denote a nonzero that has just been eliminated by a $*$ and a zero that has just been filled in by an f.

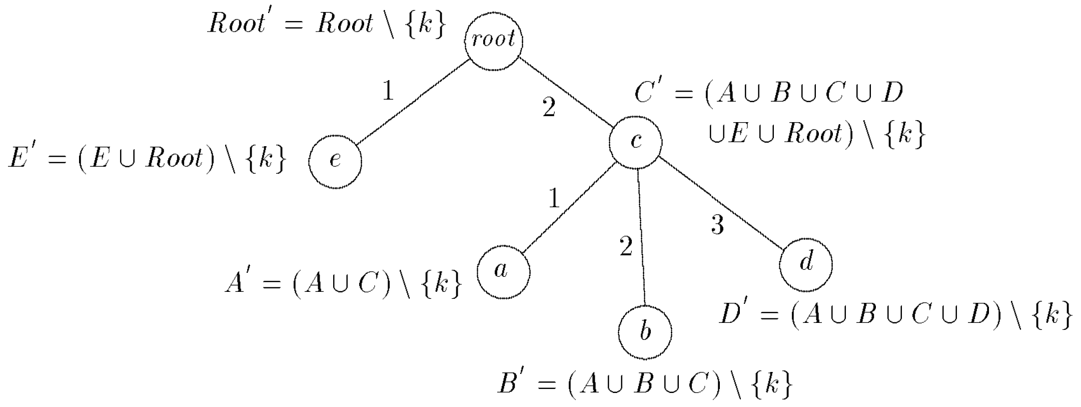

$Root' = Root \setminus \{k\}$
root
1
2
$E' = (E \cup Root) \setminus \{k\}$
e
$C' = (A \cup B \cup C \cup D \cup E \cup Root) \setminus \{k\}$
c
1
2
3
$A' = (A \cup C) \setminus \{k\}$
a
d
b
$D' = (A \cup B \cup C \cup D) \setminus \{k\}$
$B' = (A \cup B \cup C) \setminus \{k\}$

FIG. 6. *An example of a global row reduction tree.*

The dashed box and an arrow represent the sending of the row in the box to the indicated processor. Note the local reductions that are done after a row is returned to a processor: on processor 1 in diagram (c), and on processor 3 in diagram (d). It is also interesting to note that only processors 3 and 4 are required to communicate in the computation of R_{5*} since processors 1 and 2 do not inherit a nonzero in this location. This example demonstrates that fewer processors may be involved at each step of the factorization with the global row reduction algorithm than with the nonlocal merge algorithm. With the nonlocal merge algorithm all four processors are involved in the processor ring to compute $T(c_5)$. Of course, in this example there are not enough nonzeros in $S(c_5)$ to make it completely around the ring, however all the processors would be involved in the redistribution of the rows, and would also be involved in any subsequent matrix merges.

3. Combinatorial Problems in Computing the QR Factorization. In this section we describe several algorithmic results related to the symbolic factorization phase of the global row reduction algorithm. First, we show that the problem of determining the row merge heap in parallel can be done by a simple extension of an elimination forest merging algorithm [16]. Then we rephrase the problem of finding an initial row partition as a problem in finding a special set of vertices, called a foundation, in the row merge heap. Given that the row partition is generated in this manner, it has been shown that the interprocessor communication required by the global row reduction algorithm can be computed by performing a certain vertex elimination upon a quotient graph [13]. The advantage of this approach is that the quotient graph is a much smaller structure than a representation of the entire nonzero structure of the matrix, yet it captures the essential information necessary to determine the interprocessor communication during the symbolic phase of the sparse factorization.

3.1. Parallel Computation of the Row Merge Heap. Based on the row merge heap rules presented earlier, one can construct a sequential algorithm for this computation [10]. This column-driven algorithm is shown in Figure 8. Suppose that for the parallel algorithm we are given an initial assignment of the rows to processors described by the p-partition $\Pi = \{R_1, R_2, \ldots, R_p\}$ of the rows of the matrix. It is possible to execute the row merge heap algorithm locally on each processor with the rows assigned to that processor. For example, processor a can compute the heap structure H_a from the row set R_a using the sequential row merge heap algorithm. The resulting structure will not necessarily be a connected heap, but rather a forest of heaps. This forest has leaves R_a, and a structure

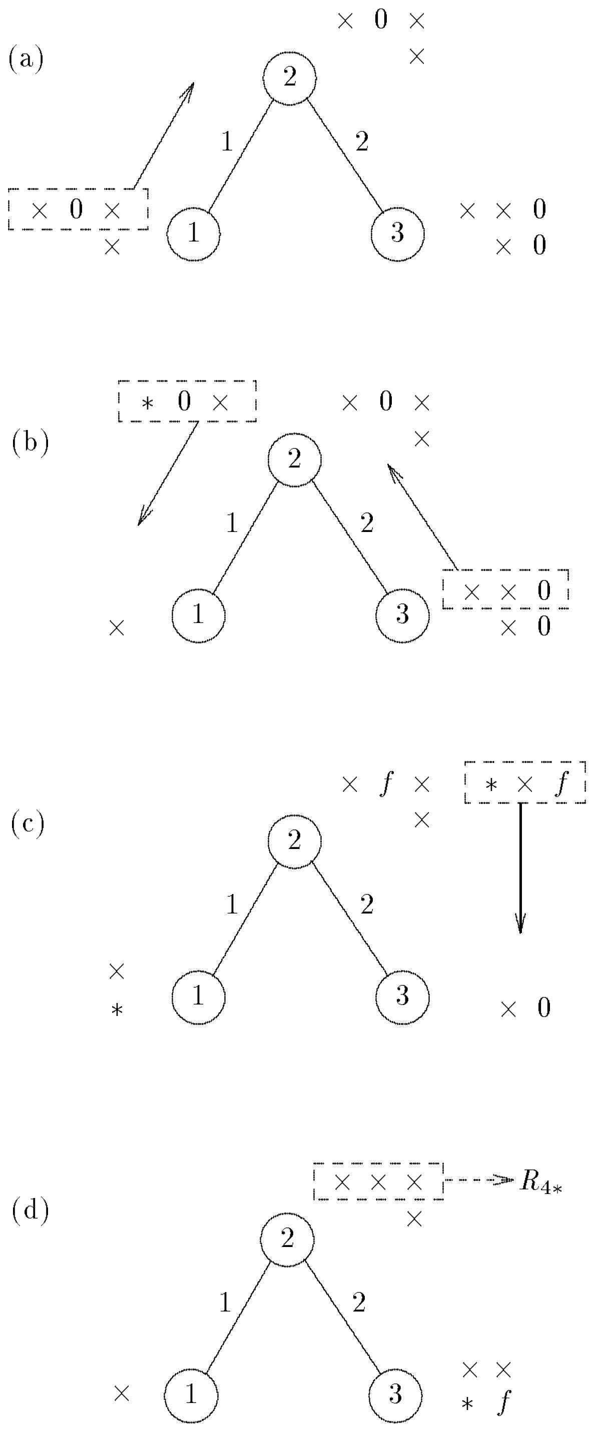


FIG. 7. *Computing R_{4*} with the global row reduction algorithm for the matrix in Figure 1.*

```
Initialize trees $T_i$, $i = 1, \ldots, m$, with
    $T_i = \{r_i\}$ and $S(T_i) = S(r_i)$;
**For** $k = 1, \ldots, n$ **do**
    Find all trees, say $T_1, \ldots, T_s$, with $S(root(T_i)) = \{k, \ldots\}$;
    Form a new tree with root $c_k$ by linking the above trees by
        their roots to $c_k$;
    Set $S(c_k) = (\bigcup_{j=1}^{s} S(root(T_j)) \setminus \{k\}$;
**enddo**
```

FIG. 8. *A sequential algorithm to compute the row merge heap.*

related to the complete row merge heap $H(A)$. Following the work of Zmijewski [16] on the parallel merging of elimination forests, an algorithm can be developed for merging these heap forests to obtain the entire row merge heap. Consider the algorithm shown in Figure 9 which takes two heap forests, H_a and H_b, and merges them to produce the heap forest H_{ab}. This merged heap forest is equivalent to the heap forest generated by the row merge heap algorithm on the row set $R_a \cup R_b$.

```
{Compute: $H_{ab}$ = merge($H_a, H_b$)}
$H_{ab}$ = the disconnected set of vertices $\{c_1, \ldots, c_n\} \cup R_a \cup R_b$;
**For** $k = 1, \ldots, n$ **do**
    **For** $i = a, b$ **do**
        **For** all children $v$ of $k$ in $H_i$ **do**
            Find $u$ = root($v$) in $H_{ab}$;
            **If** $(u \neq k)$ Link the tree rooted at $u$ to $k$ in $H_{ab}$;
        **enddo**
    **enddo**
**enddo**
```

FIG. 9. *An algorithm to merge two heap forests.*

For example, consider the matrix structure presented in Figure 1 and the resulting row merge heap in Figure 2. If we partition the rows of that matrix into the two sets $R_a = \{1, 2, 3, 7, 8, 9\}$ and $R_b = \{4, 5, 6, 10, 11, 12\}$, then the corresponding heap forests H_a and H_b are shown in Figure 10. If we use the heap forest merge algorithm to merge these two structures, at the start of the last pass through the k loop of the algorithm, with $k = 6$, we have obtained the partial row merge heap $H_{ab}^{(5)}$ shown in Figure 11. The child of c_6 in H_a is c_3. The root of the tree containing c_3 in $H_{ab}^{(5)}$ is c_5, thus we link c_5 to c_6 in H_{ab}. Notice that c_6 has two children in H_b, c_4 and c_5. But after we have added c_6 to H_{ab}, the root of the tree containing these vertices is c_6, hence there is nothing left to be done.

A proof of correctness for the heap forest merge algorithm is similar to the proof presented by Zmijewski [16] and we will not present it here. Note that this merging algorithm requires only the disjoint set primitives *find* and *link*, hence, as Zmijewski pointed out in the elimination forest case, an implementation using path compression and set union [15] obtains a nearly linear running time. However, there are several important differences between these two algorithms.

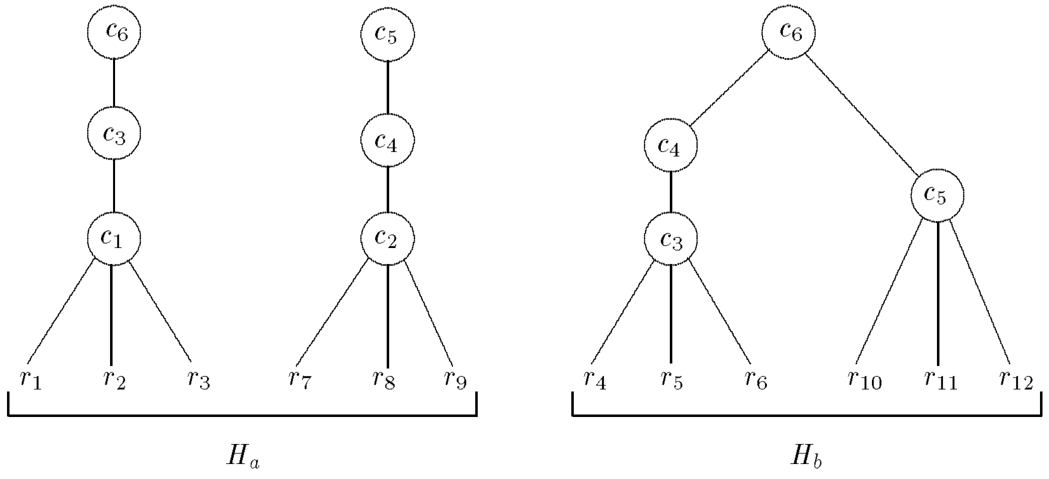

c_6
c_3
c_1
c_5
c_4
c_2
c_6
c_4
c_3
c_5
r_1 r_2 r_3 r_7 r_8 r_9
r_4 r_5 r_6 r_{10} r_{11} r_{12}
H_a
H_b

FIG. 10. *The heap forests H_a and H_b.*

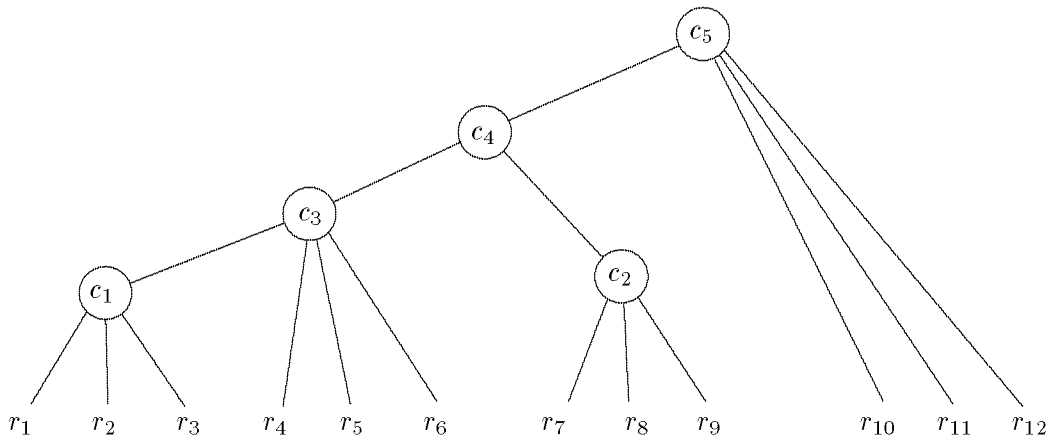

c_5
c_4
c_3
c_2
c_1
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12}

FIG. 11. *The partially constructed heap $H_{ab}^{(5)}$.*

First, note that finding the root of a row vertex r in the partially constructed heap forest is trivial: it is simply itself. Hence, each of these finds takes only constant time. Suppose that the rows are evenly distributed, so that each processor is assigned no more than $\lceil m/p \rceil$ rows. The row merge heap is constructed by recursively performing merges in $\log(p)$ time stages. Since the rows are evenly distributed, the first step in the recursion merges at most $2\lceil m/p \rceil$ row vertices, the next at most $4\lceil m/p \rceil$ row vertices, up to the last step which merges all m row vertices. Thus the total time required for just merging the row vertices is $O(m)$. Otherwise, $O(n)$ finds and links of the column vertices are required for each merge, hence these operations require a running time of $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of Ackerman's function. The total running time of the merging algorithm over the $\log(p)$ stages is then $O(m + n\log(p)\alpha(n))$. By the same argument, the amount of interprocessor communication is just $O(m+n\log(p))$. Hence, for $m \geq n\log(p)$ the running time of the heap forest merge algorithm is essentially linear.

Note that the parallel algorithm does not compute the vertex structures along with the row merge heap as described in the row merge heap algorithm. There is a difference between the information contained in the elimination tree used in the sparse Cholesky factorization and the row merge heap used here. The elimination forest construction is a more compact representation of the information required for the factorization because the nonzero structure of the rows of the Cholesky factor can be easily computed from the elimination tree and the original matrix [9, 14]. In the row merge heap, the structure of a vertex is only a subset of the set of its ancestors. This means that the row merge heap, without the vertex structures, is insufficient to exactly determine the nonzero structures of the upper trapezoidal matrices associated with the vertices, or the nonzero structure of the final matrix factor R. However, the row partition algorithm presented in the next section requires only the row merge heap without the vertex structures, hence the heap forest merge algorithm is sufficient for this task. The vertex structures are necessary for the allocation of space, which can be done after the determination of a row partition, during the symbolic factorization phase.

3.2. Determining a Row Partition. The interprocessor communication required during the factorization of the matrix is determined by the distribution of rows to processors. Formally, let $\mathcal{R}$ be the set of row indices $\{1, \ldots, m\}$ and let $\Pi = \{R_1, R_2, \ldots, R_p\}$ be a p–partition of $\mathcal{R}$, where row $r \in R_j \Leftrightarrow$ row r is assigned to processor j. For the purposes of the next section, it is necessary to assume that Π is generated by a special set of vertices $v_1, \ldots, v_p$ of $\hat{H}(A)$, a splitting of the row merge heap. We will call this set of vertices a *foundation* of the partition Π if it has the property: $r \in R_j \Leftrightarrow$ row r is a descendant of v_j in $\hat{H}(A)$.

For example, consider the row merge heap shown in Figure 2. For the row partition $R_1 = \{1, 2, 3\}$, $R_2 = \{4, 5, 6\}$, $R_3 = \{7, 8, 9\}$, and $R_4 = \{10, 11, 12\}$ we can split vertices c_3 and c_5 and obtain the new heap shown in Figure 12. Thus, we obtain the founding vertices $v_1 = c_1$, $v_2 = \hat{c}_3$, $v_3 = c_2$, and $v_4 = \hat{c}_5$ for this partition.

Not all p–partitions of $\mathcal{R}$ can be obtained from a foundation in a row merge heap. However, a characterization of allowable partitions can be derived. First, a *splitting* $\hat{H}$ of the row merge heap H is obtained by starting with H and adding interior vertices according to the following rule. Given a vertex w, with children $w_1, \ldots, w_K$, one can take a subset of k of these vertices, with $1 < k < K$, delete their edges to w and reattach them to the added vertex $\hat{w}$, and then insert $\hat{w}$ as a child of w. To determine how to generate a suitable splitting of the heap we need the following definitions. Let $desc(v)$ be the set of row indices descendant from vertex v in the heap or heap splitting $\tilde{H}$. Call a vertex $v \in \tilde{H}$ *unique* to

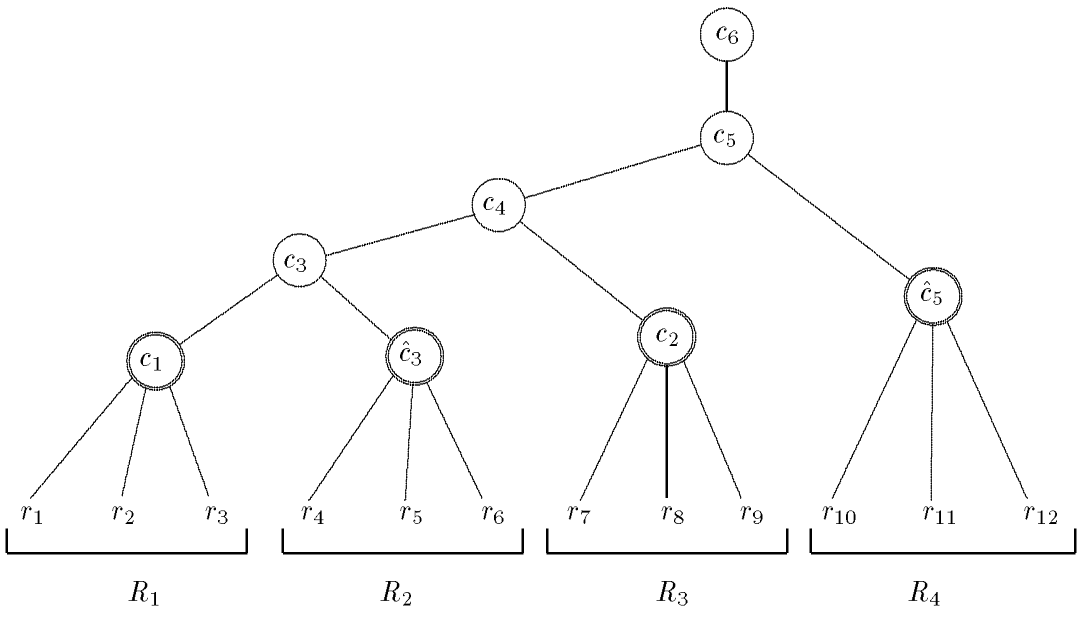


FIG. 12. *A set of foundation vertices for a row partition.*

the set R_i if $desc(v) \subseteq R_i$. Likewise, a vertex v is said to *dominate* R_i if $R_i \subseteq desc(v)$. Finally, we say a vertex v *founds* R_i if v is both unique to and dominates R_i. A set of vertices $\{v_1, \ldots, v_p\}$ that defines the p–partition $\{R_1, \ldots, R_p\}$ is then equivalent to a set of foundation vertices. These definitions are illustrated in Figure 13. Figure 14 illustrates

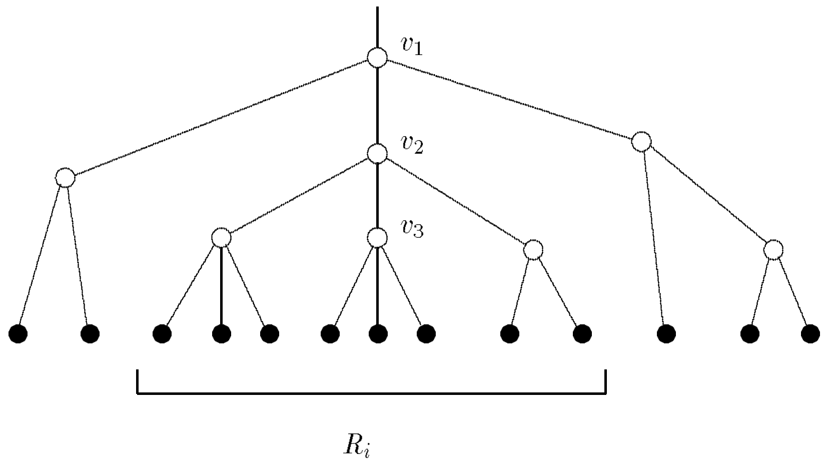


FIG. 13. *Vertices v_1 and v_2 dominate R_i, vertices v_2 and v_3 are unique to R_i, vertex v_2 founds R_i.*

how a founding vertex $\hat{w}$ can be created by a splitting of the heap.

Let the *minimum dominating vertex* for R_i be the vertex w_i in H that dominates the set R_i and has no descendant in H that also dominates R_i. The vertex w_i is clearly unique (in the usual sense). Then the set of partitions that give rise to a founding set of vertices can be characterized by the following theorem.

THEOREM 1. *Let $\{w_1, \ldots, w_p\}$ be the set of minimum dominating vertices in the row merge heap H corresponding to the p–partition $\Pi = \{R_1, \ldots, R_p\}$. Π allows a splitting $\hat{H}$ of H with founding vertices $\{v_1, \ldots, v_p\}$ $\Leftrightarrow$ each child v of w_i, for $1 \leq i \leq p$, is either unique to the set R_i or $desc(v) \cap R_i = \emptyset$.*

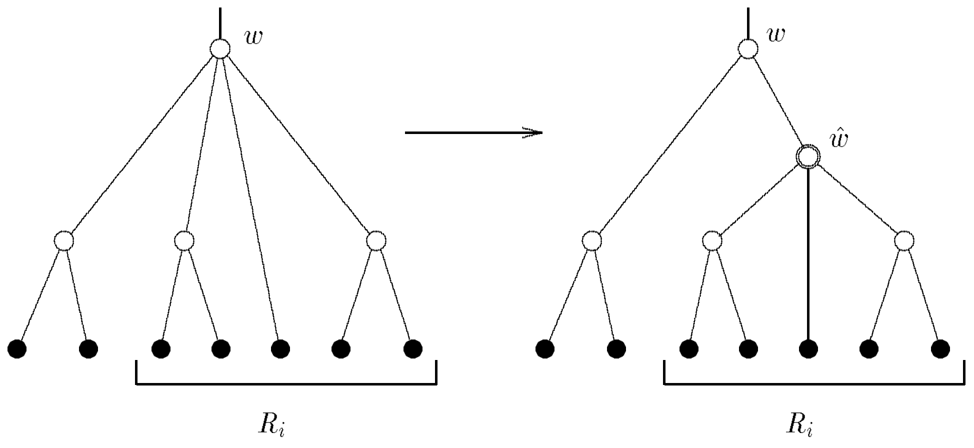


FIG. 14. *In this case vertex w can be split to form the founding vertex $\hat{w}$ for R_i.*

Proof: ($\Rightarrow$) For each i, $1 \le i \le p$, consider v_i, the founding vertex for R_i. There are two cases, based on whether v_i is a vertex in H. In the first case, assume v_i is in H, then v_i is both a dominating vertex and is unique to the set R_i. If v_i has more than one child, then it is the minimum dominating vertex w_i and each child must be unique to R_i. Otherwise, v_i has a descendant in H which is the minimum dominating vertex w_i. The children of this vertex must be unique to R_i, satisfying the conditions of the lemma. For the second case, assume that v_i is not in H. Then v_i is the closest descendant of some original vertex w in H. Since v_i is unique to the set R_i, then all of the descendants of v_i are also unique to R_i. Since v_i dominates R_i, we note that any child v of w in H with $desc(v) \cap R_i \ne \emptyset$ must be a descendant of v_i in $\hat{H}$. Thus, prior to the splitting of H, any descendant of v_i that was a child of w was also unique to R_i. Likewise, since v_i dominates R_i, then w also dominates R_i. Since by our definition a split vertex must have more than one child, no descendant of v_i can dominate R_i. Hence, w must also be the minimum dominating vertex in H for R_i, and each child v of w in H is either unique to R_i or has $desc(v) \cap R_i = \emptyset$.

($\Leftarrow$) Let w_i be the minimum dominating vertex for R_i. If w_i is unique to R_i then w_i qualifies as a foundation vertex. Otherwise, w_i must have more than one child, say vertices $u_1, \ldots, u_k$, that are unique to R_i. We can construct a new vertex v_i whose parent is w and whose children are $u_1, \ldots, u_k$. For any other child v of w we have that $desc(v) \cap R_i = \emptyset$, hence v_i dominates R_i and is a foundation vertex for R_i. Repeating this construction for each set R_i in the partition generates the desired splitting $\hat{H}$ and a set of foundation vertices for the partition Π. □

Given a row merge heap H we must address the practical problem of determining a suitable partition Π by which to distribute the rows of the matrix. There are two competing aims in finding such a partition: first, to minimize the amount of interprocessor communication, and second, to equalize the amount of work assigned to each processor. The first aim can be realized by looking for a set of founding vertices as "high" up the row merge heap as possible, in this way the processors do not have to communicate until late in the factorization. To satisfy the second aim we have to estimate the amount of work required to construct the upper triangular matrix associated with the founding vertex of a partition.

Consider a vertex v in H, let the function $\tilde{w}(v)$ be the amount of work required to compute its associated upper triangular matrix. Note that if v has more than one child it is certainly the case that

$$\tilde{w}(v) > \sum_{\text{children } x} \tilde{w}(x) \quad . \tag{2}$$

Unfortunately, it is difficult to compute $\tilde{w}(v)$, and since the function is not additive it is hard to imagine a recursive scheme for dividing the work evenly between processors. As one possible approximation consider the function $w(v) = \{$the number of leaves descendent from $v\}$, which is additive, and thus obeys the formula

$$w(v) = \sum_{\text{children } x} w(x) \quad . \tag{3}$$

Using any additive approximation for the vertex weights, the row partition algorithm shown in Figure 15 can be used to recursively search the row merge heap to obtain the row partition sets. The algorithm is initialized by specifying a small nonnegative parameter α. As each partition set is constructed by the algorithm, its size is limited to be at most $1 + \alpha$ times the average size of the remaining partition sets. The global variable *avg* maintains this average size which is computed as the total remaining weight divided by the number of partition sets left to be constructed. The vertices of the row merge heap are explored in a depth first search (DFS) order starting at the root of the heap. Based on the weight of the vertex under consideration it is either accepted by itself to define a partition set, or added to a partially completed partition set, or split further by a recursive call to the procedure *split*.

At the completion of the algorithm each set P_i, for $i = 1, \ldots, p$, contains a set of column vertices whose row vertex descendants yield the row partition set R_i. Some additional postprocessing of these sets must be done to convert them into a set of foundation vertices; this additional work is straightforward and we will not discuss it here. However, note that the final result of the algorithm is not always a set of p foundation vertices. Instead, some number $p^{'} \geq p$ foundation vertices are constructed, and one or more of these vertices is assigned to each processor. After all possible local reductions have been completed, each of these vertices will correspond to an essentially full upper trapezoidal matrix in the matrix list at a processor, and the global row reduction process can begin. Note that as α decreases to zero, the number of foundation vertices assigned to each processor will tend to increase since we are demanding a more even distribution of rows to processors.

3.3. Reduction of the Set of Candidate Processors. Once a row partition is determined, this partition can be broadcast and the rows of the matrix redistributed to the correct processors. At this point it is convenient to make use of a column permutation that maintains the row merge heap structure, but makes its reduction a little easier. We note that the computations required to reduce the row merge heap are invariant with respect to column permutations that preserve the heap structure. Therefore we can reorder the columns so that the descendants of all of the foundation vertices are ordered before any of the foundation vertices. Given this reordering, let k_{min} denote the column index of the minimum valued foundation vertex, then the computation of all rows R_{k*} for $k < k_{min}$ are local to the processor to which the row is assigned. Once row reductions between processors begin, the nonzero structure of the upper triangular matrices at each processor changes in a nontrivial way based on the reduction trees. Thus, the set of processors that participate at each stage of the factorization is not easily computable.

In the symbolic factorization stage space is allocated for the intermediate storage of the upper triangular matrices and for the matrix factor R. Recall that a reduction tree,

```
begin {Do a DFS of $H$ }
    $P_k = \emptyset,\ k = 1, \ldots, p$;
    $avg = m/p$;
    $k = 1$;
    $v = n$; {start at root of $H$}
    split($v$);
end

procedure split(vertex $v$)
    begin
        For each child $u$ of $v$ do
            if $w(u) > (1 + \alpha)avg$ {weight too large}
                split($u$);
            else if $w(u) \geq avg$ {weight alone acceptable}
                $P_{k+1} = P_k$;
                $P_k = \{u\}$;
                $k = k + 1$;
                update $avg$;
            else if $w(u) + w(P_k) > (1 + \alpha)avg$ {weight too large}
                split($u$);
            else {weight small enough...}
                $P_k = P_k \cup \{u\}$;
                $w(P_k) = w(P_k) + w(u)$;
                If $w(P_k) \geq avg$ {...and large enough}
                    $k = k + 1$;
                    update $avg$;
                endif
            endif
        enddo
    end
```

FIG. 15. *A row partition algorithm.*

say E_k, is used to organize the row reductions among processors to produce R_{k*}, the k-th row of R. The reduction trees must also be computed during the symbolic factorization. The approach used is to generate a *candidate* set $X^{(k)}$ of processors at each stage k of the symbolic factorization. This candidate set has two properties: (1) the set $X^{(k)}$ contains all the processors that are in the reduction tree E_k, and (2) each processor can easily compute $X^{(k)}$ based on local information. Ideally, one would like to generate a candidate set as close to the correct set of processors as possible in order to increase the efficiency of the symbolic factorization. Several possible algorithms for constructing these candidate sets based on a quotient graph vertex elimination model have been discussed elsewhere [13]. Another simple possibility would be to let the candidate set be the set of all processors which are assigned a row in the set $desc(c_k)$ of the row merge heap. For the discussion that follows we assume that a candidate set can be easily computed, and that it satisfies the two properties given above.

Given the set of candidate processors $X^{(k)}$, in Figure 16 we present an algorithm for generating the reduction tree during the symbolic factorization stage of the global row reduction algorithm. The basic scheme used in the hypercube implementation of the global row reduction algorithm is to compute the reduction tree in two steps. First, we compute a minimum depth spanning tree (MDST), as if one was doing a sparse gather operation, rooted at the processor assigned the completed row R_{k*}. As information is passed up the tree, we eliminate nodes that do not participate in the row reduction. Note that the computed MDST is not necessarily unique.

```
Proc $(v)$ : {program for processor $v$}
    $t\_root = \emptyset$;
    If $(S(v) = \{k, \ldots\})$ $t\_root = v$;
    For each child $u$ of $v$ in MDST do
        Receive $(S(u'), u')$ from processor $u$;
        If $(S(u') = \{k, \ldots\})$ then
            If $(t\_root = \emptyset)$ then
                $t\_root = u'$;
            Else
                $S(t\_root) = merge(S(t\_root), S(u'))$;
                $S(u') = S(t\_root) \setminus \{k\}$;
                Send $S(u')$ to processor $u'$;
            endif
        endif
    enddo
Send $(S(t\_root), t\_root)$ to the parent of $v$ in MDST;
If $(t\_root = v)$ Receive $S(v)$;
```

FIG. 16. *An algorithm to generate E_k from the candidate set $X^{(k)}$.*

In this algorithm, $S(v)$ is the structure of the upper trapezoidal matrix at processor v. The variable *t_root* is the root of a subtree of the reduction tree E_k being computed. If $S(v)$ has a leading nonzero in position k, then *t_root* is set to v and structures are merged into $S(v)$ as they are received. Otherwise, some descendant w of v in the MDST is promoted to *t_root*, and processor v computes any required mergings into $S(w)$. After messages have been received from all the children of v in the MDST, the computed structure $S(t\_root)$ is sent to the parent of processor v.

To store the required interprocessor communication each processor need only store its parent and children in the reduction tree. This information can be stored contiguously in a vector along with a set of pointers. For example, suppose that in the final reduction tree E_k processor v receives rows from processors $u_1, u_2, \ldots, u_s$, (in the order listed), and then sends its resulting top row to processor w. Then we append the processor values $u_1, u_2, \ldots, u_s, w$ contiguously to this storage vector, note that the pointer $head(k)$ points to the location storing u_1, and set the pointer $head(k+1)$ to the location following w. The first interprocessor communication by processor v takes place at some step k_v with $k_v \geq k_{min}$. Since each processor communicates with no more than $\log(p)$ processors at each step, the length of this communication storage vector is no longer than $(n - k_{min} + 1)(\log(p) + 1)$ and will usually be much shorter. A vector of pointers of length $n - k_{min} + 2$ is also required.

4. Experimental Results for the QR Factorization of a Sparse Matrix.

Presented in this section are experimental results obtained with an implementation of the global row reduction algorithm for sparse QR factorizations. These algorithms were implemented on a 32-node Intel iPSC/2 hypercube with 4.5 MBytes of memory per node in Green Hills Fortran-386 Fortran and run under version R3.2 of the iPSC operating system. Given the matrix A and some initial distribution of the matrix rows onto the processors, the major steps in this parallel implementation are the following.

1. Determine a column ordering to reduce the fill in $\mathit{Cholesky}(A^TA)$. In our experiments a simple parallel implementation of the minimum-degree algorithm [7] was used to determine the column ordering.
2. Swap columns based on the column ordering determined. Since a row mapping is used, this is a local operation.
3. Determine a "preliminary" row merge heap by the local column-driven algorithm, shown in Figure 8, based on the set of rows R_i assigned to the processor. This local computation is followed by the global heap forest merging algorithm, shown in Figure 9, to determine the complete row merge heap.
4. Use the row partition algorithm, shown in Figure 15, on one processor to determine a row partition Π, and broadcast the result.
5. Swap rows among processors based on the computed row partition.
6. Renumber the column vertices so that the local merging is done in a postorder traversal of the row merge heap, and globally, all vertices corresponding to local computation are labeled before the foundation vertices.
7. Construct the quotient graph based on the partition Π and the row merge heap. Since the foundation vertices are labeled after them, column vertices local to a processor do not need to be included in the quotient graph. (This step is not discussed in this paper, but is presented in reference [13]. Briefly, the quotient graph is a compact way to represent the structures of the upper trapezoidal matrices at each processor during the symbolic factorization. The advantage of this approach is that the size of the quotient graph is proportional to the number of partitions, not the number of rows, and hence can be maintained easily by each processor.)
8. The symbolic factorization phase:
   (a) Use a quotient graph elimination algorithm to determine the candidate set at each step of the factorization (see reference [13] for a lengthy discussion on possible quotient graph elimination models).
   (b) Reduce the minimum depth spanning tree generated from the candidate set, using the algorithm shown in Figure 16, to obtain the the reduction trees and store the result.
   (c) Determine the intermediate storage required for the upper triangular matrices.
   (d) Determine the storage required for the matrix factor R.
9. Perform the numerical factorization.

For these experiments four sparse matrix structures were used in the construction of test matrices. A description of the matrix structures used is given below. Note that these matrices range from being very structured to random, and as a result the densities of the matrix factors progressively increases. For the test matrices used, the number of initial nonzeros was comparable.

1. *Grid 1:* This matrix is constructed from the 9-point difference operator on a square $k \times k$ grid. Each row structure is then repeated four times to form a $k^2 \times 4k^2$ matrix.

2. *Grid 2:* This matrix is constructed from a finite element $k \times k$ grid problem. There is a matrix column associated with each vertex in the grid and a matrix row with each square element. In the row corresponding to an element, there is a nonzero for each of the four vertices that define the element. This structure is repeated four times to obtain a $(k-1)^2 \times 4k^2$ matrix.
3. *Banded:* This matrix is obtained by starting with a square matrix with a nonzero diagonal and a band of width ten percent of the dimension of the matrix. The structure inside the band is random, and this matrix structure is repeated four times.
4. *Random:* This matrix is constructed by starting with a random square matrix with nonzero diagonal, and repeating the structure four times.

Table 1 summarizes the results obtained from the implementation on hypercubes of 8, 16, and 32 processors. Shown in this table are the nonzero densities of the original test matrix A and of the resulting factor R. The efficiency shown is an "effective" time to perform one Givens operation for just the numerical factorization stage of the factorization relative to the sequential binary row merge heap algorithm of Liu. This efficiency is computed using the following values. Let t_{parallel} be the execution time of the numerical factorization on the hypercube, p be the number of processors used, and γ the time required to perform one Givens update (operation) on one processor. In addition, let N_L be the number of operations required by Liu's sequential row merge heap algorithm. Then the effective efficiency is computed by the formula

$$\text{efficiency} = \frac{p\, t_{\text{parallel}}}{\gamma\, N_L} \quad . \tag{4}$$

Also listed is $t_{\text{fact}}/t_{\text{symb}}$, the ratio of the time required to perform the numerical factorization to the time required to perform the symbolic factorization. The final column of Table 1 is a measure of the computational imbalance, which is computed as the ratio of the maximum number of operations performed by any processor to the average number of operations per processor.

TABLE 1
A summary of experimental results for sparse systems.

Problem	Density A (R)	p	Efficiencies	$t_{\text{fact}}/t_{\text{symb}}$	Comp. Imbalance
Grid 1	2.1%	8	3.56	1.38	1.18
(13456 nonzeros)	(18.4%)	16	5.15	1.65	1.31
400×1600		32	7.70	1.72	1.28
Grid 2	0.83%	8	5.26	0.65	1.83
(7056 nonzeros)	(6.7%)	16	6.76	0.43	2.26
484×1764		32	7.23	0.32	2.07
Banded	2.2%	8	3.88	2.19	1.15
(13964 nonzeros)	(32.9%)	16	5.18	2.16	1.38
400×1600		32	6.64	1.89	1.32
Random	2.0%	8	3.91	6.15	1.30
(12952 nonzeros)	(61.2%)	16	3.18	3.92	1.63
400×1600		32	3.54	3.55	1.33

Interpreting the effective efficiency is treacherous because not all the perceived inefficiencies are the result of idle processors waiting for messages, load balancing problems, or communication overhead. A significant part of this inefficiency is due to the overhead in handling sparse data structures and is, therefore, very dependent on the implementation itself. Even in a good sequential implementation, the overhead in manipulating the sparse data structures will account for most of the observed execution time. Thus, a data structure overhead of one to two times the actual time spent in numerical computation is not unreasonable, but the actual amount would depend on both the problem solved and the implementation.

As noted earlier, the global row reduction algorithm can be doing more numerical work than the sequential row merge heap algorithm because it can allow more incidental fill. That this additional fill occurs is shown in Figure 17 which shows the ratio of the number of operations required by the global row reduction algorithm to the number of operations required by the sequential row merge heap algorithm. This ratio is plotted for the four test problems as a function of the number of processors used for a fixed problem size. Note that for the more structured problems this ratio increases as the number of processors increase, thus the decreasing efficiencies seen for these problems is at least partially due to this additional work.

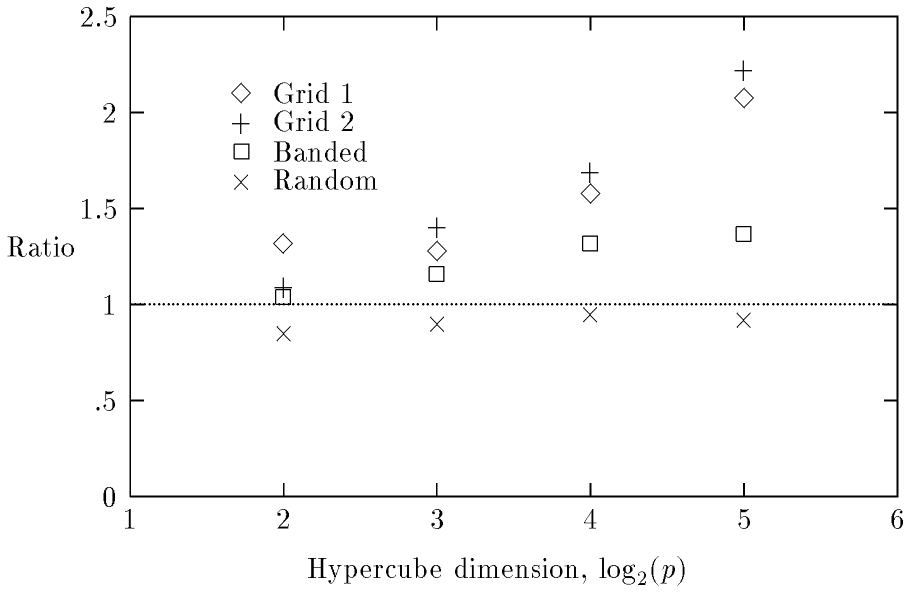


FIG. 17. *A comparison of the number of operations required.*

To give some idea of the improvement in computational speed that can be afforded by this parallel implementation we have plotted in Figure 18 the execution times of the various phases of the algorithm for the same banded problem used in Table 1. For this problem good improvements in the execution time are observed as the number of processors is increased. Also note that, in this case, the symbolic factorization phase consistently takes about one half the time required by the numerical factorization. In this problem, as in the other test problems, the amount of time required to compute a minimum-degree ordering and reorder the matrix columns was minimal.

As noted in steps 7 and 8(a), associated with the global row reduction algorithm are a number of algorithmic problems related to a determination of the interprocessor communication at each stage of the factorization. There are a number of useful characterizations

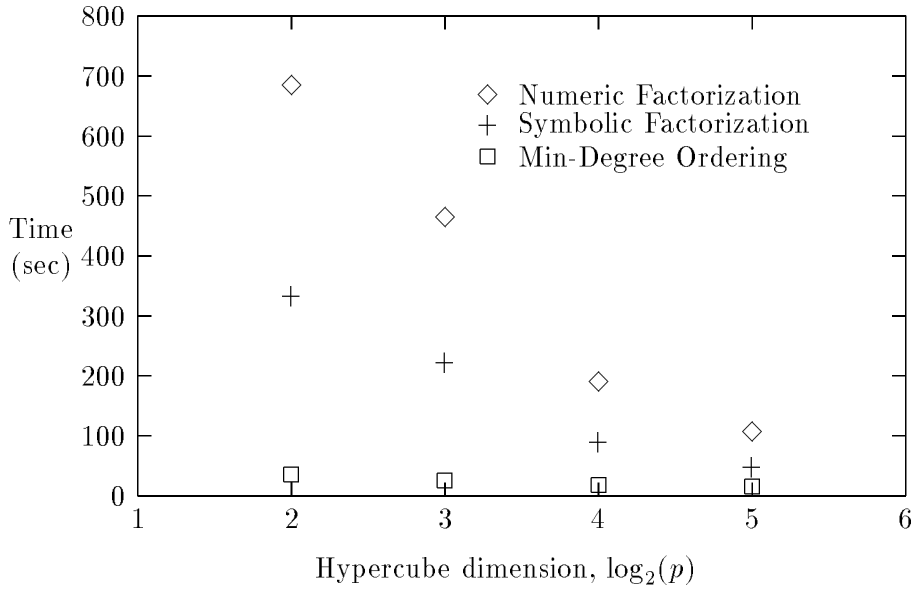


FIG. 18. *Times required for various phases of the factorization.*

that have been developed to streamline the symbolic factorization phase of the algorithm [13]. It is likely that a better implementation of the symbolic factorization phase would significantly decrease the execution time of this phase from the times presented here.

Overall, these results are preliminary, but encouraging. The concept of the row merge heap algorithm is effective on medium grain parallel machines like the iPSC/2 because it decomposes a large sparse problem into a number of smaller dense problems. The existence of these smaller dense problems allows the attainment of a reasonable ratio of work to communication overhead. However, a strict parallel implementation of Liu's sequential row merge heap algorithm can require more interprocessor communication than is necessary. Thus, taking a different approach, the global row reduction algorithm has been introduced as a means of reducing the amount of required interprocessor communication.

5. A Sparse R-S Reduction Algorithm.

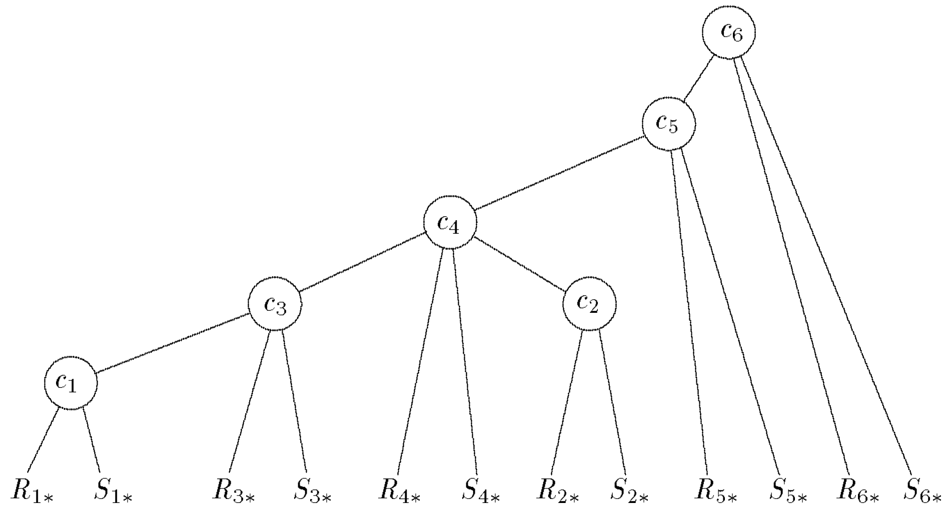
The solution of the trust-region problem involved in the Levenberg-Marquardt approach to solving nonlinear least-squares problems requires the repeated reduction of a system of the form

$$\begin{bmatrix} R \\ \lambda^{1/2} I \end{bmatrix} \tag{5}$$

to upper triangular form [11]. In this system R is the upper triangular factor obtained from the QR factorization of the Jacobian and λ is the current estimate of the Levenberg-Marquardt parameter. Since the lower half of the system is diagonal, it is clear that the structure of this resulting upper triangular matrix has the same nonzero structure as the original matrix R. However, it might be less clear that essentially the same algorithm that was used to compute the QR factorization and originally obtain R can be used to reduce this system to upper triangular form. For brevity, we denote the lower matrix in the above system as S and refer to this matrix problem as the R-S reduction.

A quick observation is that the column vertex dependence of the row merge heap generated by the structure of the matrix in equation (5) is the same as that of the original matrix. However, the leaves of the heap are different. If we let R_{k*} be the k-th row of R

and S_{k*} be the k-th row of the diagonal matrix, then column vertex c_k in the row merge has just these two rows as leaf descendants. For our example system, with the row merge heap first shown in Figure 2, we obtain the heap shown in Figure 19.



c_6
c_5
c_4
c_3
c_2
c_1
R_{1*} S_{1*} R_{3*} S_{3*} R_{4*} S_{4*} R_{2*} S_{2*} R_{5*} S_{5*} R_{6*} S_{6*}

FIG. 19. *The R-S reduction row merge heap for the example matrix.*

For simplicity, assume that we have one founding vertex assigned to each processor v. Denote by $\hat{S}^{(k)}(v)$ the structure of the upper triangular matrix at processor v at the start of step k of the R-S reduction, and by $S^{(k)}(v)$ the corresponding structure during the global row reduction algorithm. Let k_v be the first column index for which processor v must participate in a global row reduction step. Consider the case when $k < k_v$ and R_{k*} is assigned to processor v, here we also assign S_{k*} to processor v. For this case we observe that both $S^{(k+1)}(v)$ and $\hat{S}^{(k+1)}(v)$ equal $S(R_{k*}) \setminus \{k\}$. But for $k \geq k_v$ and R_{k*} assigned to processor v, we observe that $S^{(k+1)}(v) \subseteq S(R_{k*}) \setminus \{k\}$. The problem is that if we keep both S_{k*} and R_{k*} at processor v, then we have $\hat{S}^{(k+1)}(v) = S(R_{k*}) \setminus \{k\}$, and we will not be able to use the reduction trees generated by the global row reduction algorithm.

The following modification to the algorithm solves this problem. Let σ_k be the processor corresponding to last numbered child of $root(E_k)$, where E_k is the reduction tree for step k of the global row reduction algorithm. Following the computation of the QR factorization, we have R_{k*} residing at processor $root(E_k)$. But for $k \geq k_{root(E_k)}$ we assign S_{k*} to $root(E_k)$ and move the row R_{k*} to processor σ_k. Since σ_k is the last processor to merge a row with $root(E_k)$ in the global row reduction algorithm, we have $S^{(k+1)}(\sigma_k) = S(R_{k*})\setminus\{k\}$. But note that $S^{(k)}(\sigma_k) \subseteq S(R_{k*})$. Hence, in the R-S reduction algorithm we do not merge R_{k*} into the upper triangular matrix at processor σ_k until after all of its children in E_k are processed (to maintain the structures of the children). Then R_{k*} can be merged into the upper triangular matrix at processor σ_k, and the top row of this new matrix sent to $root(E_k)$. The completed row of the reduced R-S matrix is computed and stored at processor $root(E_k)$, and the row sent from processor σ_k is returned with the nonzero in column k eliminated. Now the structure of the upper triangular matrix at processor $root(E_k)$ has vertex k deleted, hence $\hat{S}^{(k+1)}(root(E_k)) = \hat{S}^{(k)}(root(E_k)) \setminus \{k\}$. Likewise, on processor σ_k we have $\hat{S}^{(k+1)}(\sigma_k) = S(R_{k*}) \setminus \{k\} = S^{(k+1)}(\sigma_k)$. Since the structures on the children processors are also maintained, we have the result that if, for each processor v, $\hat{S}^{(k)}(v) = S^{(k)}(v)$, then $\hat{S}^{(k+1)}(v) = S^{(k+1)}(v)$. Therefore, with this modification we have the same structure at each processor that was present during the global row reduction algorithm, and the reduction trees that were generated for the global row reduction algorithm suffice to specify the interprocessor communication required for the R-S reduction.

6. Parallel Estimation of a Sparse Jacobian. In this section we consider the problem of the parallel estimation of a sparse Jacobian $J(x)$ by the forward difference equation

$$J(x)d \cong F(x+d) - F(x) \quad . \tag{6}$$

We assume that the function evaluation can only be done as a "black box," (*i.e.* that the component functions are highly interrelated, or for some reason it is inefficient to separate the evaluation of each of the component functions). Thus we are presented with a subroutine which when given a point x, produces the function value $F(x)$. We will also assume that each function evaluation takes approximately the same amount of time to compute.

On a sequential machine the goal is to minimize the total number of function evaluations necessary to estimate $J(x)$. But on a parallel machine the saving of one function evaluation may not make any difference; it may only mean that one more processor is idle while the other processors are busy computing function values at other differencing points. These processors will have to remain idle since the factorization of the Jacobian cannot begin until the matrix has been estimated.

If ρ_{max} is the maximum number of nonzeros in a row of the Jacobian, we know that at least that many function evaluations are required to estimate $J(x)$ [4]. Thus an optimal parallel algorithm with p processors would take at least $\lceil \frac{\rho_{max}}{p} \rceil$ times the time required for one function evaluation. A realistic goal might then be to demand an algorithm that estimates the Jacobian with no more than $K = \lceil \frac{\rho_{max}}{p} \rceil p$ function evaluations, since reducing the number of function evaluations from this amount would not improve the running time of the optimization algorithm. Or if we overlap the function evaluation involved in the step acceptance computation with the Jacobian estimation, as suggested by Byrd, Schnabel, and Shultz [1] and Coleman and Li [3], we may require the estimation to be done with one less evaluation.

On a variety of test problems it has been shown by Coleman and Moré [4] that intersection graph coloring algorithms often generate a set of differencing vectors which requires very close to the optimal number of function evaluations. Also, the incidence degree order (IDO) coloring algorithm, which they find to be very effective, has essentially the same structure as the minimum-degree ordering algorithm. Therefore, this algorithm fits very naturally into the framework established for initializing the QR factorization. A possible problem that can arise is when the coloring algorithm determines a coloring that requires some $K' > K$ function evaluations. In this case we would like to be able to turn to an alternate approach which does not exceed the limit of K function evaluations. The alternatives we consider are the multicoloring methods and full matrix methods.

The full matrix method was originally proposed by Newsam and Ramsdell [12]. Consider some $\rho \geq \rho_{max}$ difference vectors combined into a $n \times \rho$ matrix D and the corresponding ρ function differences combined into a $m \times \rho$ matrix ΔF. If $S(J)$ is the nonzero structure of the Jacobian, we can approximate the Jacobian by solving the minimization problem

$$min \ \{ \|AD - \Delta F\|_F \ \ s.t. \ \ A \in S(J) \,\} \quad . \tag{7}$$

This problem decomposes into m independent linear least squares problems which can be rephrased in terms of $S(i)$, the nonzero structure of row i, as

$$min \ \{ \|D^T_{S(i)} A^T_{S(i)} - \Delta F^T_i \|_2 \} \quad , \tag{8}$$

where ΔF_i is the i-th row of ΔF, and $A_{S(i)}$ are the nonzeros of the i-th row of A. If each matrix $D_{S(i)}$ is of full rank (*i.e.* rank $|S(i)|$), then equation (8) has a unique solution with which the i-th row of the Jacobian can be estimated.

One can think of equation (8) as a general framework for all of these Jacobian estimation algorithms. These techniques can be classified, based on the structure of the matrices $D_{S(i)}$, into three types:

1. *coloring methods*, $D_{S(i)}$ can be permuted to a diagonal matrix,
2. *multicoloring methods*, $D_{S(i)}$ can be permuted to an upper triangular matrix, and
3. *full matrix methods*, $D_{S(i)}$ is dense and of full rank, and must be factored to solve for each row of the Jacobian.

Consider the case when the the coloring algorithm determines a coloring that requires some $K' > K$ function evaluations. First we note that, in this situation, a multicoloring approach can save at least one function evaluation.

THEOREM 2. *If we have a coloring of cardinality $K' > \rho_{max}$, then a multicoloring can evaluate $J(x)$ with $\hat{K}$ function evaluations, where $K' - 1 \geq \hat{K} \geq \rho_{max}$.*

Proof: We are given that the coloring partitions the columns of the Jacobian into the groups $C_1, C_2, \ldots, C_{K'}$ with $K' > \rho_{max}$. We will show how to construct a multicoloring which allows for the solution of the Jacobian with $K' - 1$ function evaluations. Let the differencing vectors generated by the coloring be $d^{(i)} = \sum_{j \in C_i} \tau e_j$, and construct from them the $K' - 1$ multicoloring differencing vectors $d^{(i,i+1)} = d^{(i)} + d^{(i+1)}$, for $i = 1, \ldots, K' - 1$. We note that equation (6) forms the $K' - 1$ equations

$$\sum_j J_{kj} d^{(i,i+1)} = f_k(x + d^{(i,i+1)}) - f_k(x) \quad . \tag{9}$$

We will show that these equations are always equivalent to an upper triangular system.

Let v denote the k-th row of J. Since we have a K'-coloring, there are at most some set of indices $j_1, j_2, \ldots, j_{K'}$ of v, where $j_i \in C_i$, that are nonzero. Thus, the above equations are equivalent to a system of the form:

$$\begin{aligned} v_{j_1} + v_{j_2} &= \Delta f^{(1,2)} \\ v_{j_2} + v_{j_3} &= \Delta f^{(2,3)} \\ &\vdots \\ v_{j_{K'-1}} + v_{j_{K'}} &= \Delta f^{(K'-1,K')} \quad . \end{aligned} \tag{10}$$

Since $\rho_{max} < K'$, at most $K' - 1$ of the v_{j_i} are nonzero, hence the above equations can always be rearranged into a solvable upper triangular system. □

A problem with both the multicoloring and full matrix approaches is that the upper triangular matrices or matrix factors used to solve for each row of the Jacobian must be stored. This additional storage is of size $O(\sum \rho_i^2)$, where ρ_i is the number of nonzeros in row i of the Jacobian. This additional storage can be prohibitive. But suppose we have two rows, r_1 and r_2, with structures satisfying $S(r_2) \subseteq S(r_1)$. We note that the matrix system used to solve for r_1 can also be used to solve for r_2. To extend this idea, suppose we have t full matrix systems with structures $S_1, S_2, \ldots, S_t$. If for each row r_i of J there is a set S_j such that $S(r_i) \subseteq S_j$, we can then use these matrix systems to solve for each row of the Jacobian. To construct these systems requires $\hat{K} = max|S_j|$ function evaluations, and these matrices can be stored in $O(\sum |S_j|^2)$ space.

Suppose the columns of the Jacobian are ordered in a minimum-degree ordering to reduce the fill in the upper triangular factor R during the QR factorization stage. Then we find that the k-th vertex chosen by the ordering algorithm had minimum degree in the partially eliminated graph $G^{(k)}(J^T J)$. We observe that after the first $k-1$ vertices have been eliminated, the adjacency list of vertex k is just the structure $S(c_k)$ in the row merge heap. Therefore it makes sense to propose the set of foundation vertices $v_1, v_2, \ldots, v_p$ as the generators of the sets $S_1, S_2, \ldots, S_t$ discussed above. But the structure $S(v)$ of vertex v in the row merge heap is only a subset of the structure required for the full matrix method because it does not include column vertices that were eliminated earlier. So we define the new structure $S^*(v)$ by the formula

$$S^*(v) = \bigcup_{r \in desc(v)} S(r) \quad , \tag{11}$$

where $desc(v)$ is the set of rows which are descendants of vertex v in the row merge heap.

If we use the foundation vertices as the generators of a full matrix Jacobian approximation algorithm, we require $max|S^*(v_i)|$ function evaluations to be able to solve for every row of $J(x)$. For the test problems described in the previous section we show in Table 2 these values as compared to the size of colorings generated by the IDO ordering. We also show the number of function evaluations required by a multicoloring generated from the structures $S^*(v_i)$. This multicoloring was generated by a greedy assignment of colors to the indices of the structures while maintaining an upper triangular matrix solution.

TABLE 2
Comparison of Jacobian estimation schemes based on the foundation vertices.

Problem	$\rho_{\max}$	IDO Coloring	p	$max\|S^*(v_i)\|$	Multi-Coloring
Grid 1			8	25	27
(13456 nonzeros)	9	14	16	25	27
400×1600			32	15	18
Grid 2			8	9	12
(7056 nonzeros)	4	6	16	9	10
484×1764			32	9	10
Banded			8	27	28
(13964 nonzeros)	14	29	16	30	30
400×1600			32	27	29
Random			8	74	79
(12952 nonzeros)	28	43	16	67	68
400×1600			32	63	64

Clearly the sizes of the matrix systems generated by this method are too large. Suppose we demand a full matrix solution that requires only $\hat{K}$ function evaluations, but we have a vertex v in the row merge heap with $|S^*(v)| > \hat{K}$. We can always replace v with its children, $u_1, \ldots, u_s$, and then continue this process recursively until we have a set of new vertices, $w_1, \ldots, w_t$, with $max|S^*(w_i)| \leq \hat{K}$. This process will terminate since $\rho_{max} \leq \hat{K}$.

In Table 3 we show the results of this approach on the test problems discussed earlier. We show both the average number and the maximum number of full matrix systems per processor, given that the initial vertices on processor i are its foundation vertices. In this case we have set $\hat{K}$ equal to the cardinality of the IDO coloring given in Table 2. In the last column is shown the amount of storage required for these full matrices as a fraction of the storage required for the Jacobian matrix.

TABLE 3
Results for the modified full matrix Jacobian estimation scheme.

Problem	Systems/Processor		Storage
	Avg.	Max.	Required
Grid 1	8	15	1.80
(13456 nonzeros)	5	9	1.75
400×1600	3	5	1.90
Grid 2	11	18	1.08
(7056 nonzeros)	7	12	1.18
484×1764	3	7	1.19
Banded	32	38	7.02
(13964 nonzeros)	12	20	5.06
400×1600	8	12	7.40
Random	9	13	4.82
(12952 nonzeros)	5	11	5.28
400×1600	3	4	6.75

7. Summary and Discussion. Most of the computation required in the solution of large nonlinear optimization problems is contained within the "inner loop" of the optimization algorithm: the estimation of the Jacobian, its factorization, and the solution of a trust-region problem (or another globalization strategy). Thus, in this paper we have concentrated on developing parallel algorithms for solving these specific tasks for problems in which the Jacobian is sparse.

We have introduced a parallel sparse QR factorization based on the global row reduction algorithm. The algorithm has the interesting feature that it is equivalent to the sequential row merge heap algorithm local to a processor, but when interprocessor communication is required it attempts to minimize this communication in exchange for some additional incidental fill. For the test problems considered, this additional fill was found to be nominal, and the required interprocessor communication was shown not to dominate the arithmetic work. Determining exactly what communication is required is computed during a symbolic factorization in which a sequence of candidate sets of processors are reduced to a set of reduction trees. The reduction tree data structure compactly describes the required interprocessor communication during the numeric factorization. The heuristic presented for doing this reduction orders the children of a processor in the reduction tree by the order in which it receives messages from them. The advantage of this approach is that it achieves local load balancing, which should be reflected by a more even distribution of work during the numeric factorization stage. An interesting topic for further research would be to explore other approaches for generating the reduction tree from the candidate set, especially methods that take previous row reductions between processors, and their inherited sparsity structure, into account. Also, the minimum depth spanning tree used in the reduction tree algorithm is not unique; aspects of this choice should be explored.

We note that since the nonzero structure of the Jacobian is fixed, the setting up of these communication data structures need only be done once. Thus the time required for this computation is amortized over the number of iterations of the outer loop required to solve the nonlinear optimization problem. We have also seen that a slight modification of the global row reduction algorithm allows for the solution of a trust-region problem using the same communication structure generated for the QR factorization.

The initial row distribution for the global row reduction algorithm was shown to be equivalent to the determination of a set of foundation vertices in a split form of the row merge heap. A characterization of a row partition in terms of foundation vertices was given. Based on the row merge heap, a heuristic for generating a good initial row partition was presented that attempts to minimize the interprocessor communication during the factorization while balancing the workload among the processors. A disadvantage of this algorithm is its assumption that the vertex weights are additive. For the test problems considered, this approximation yielded acceptable partitions. However, better approximation schemes for these weights is a topic worthy of further study.

An algorithm based directly on the original binary row merge heap approach, such as the nonlocal merge algorithm briefly discussed in this paper, seems to implicitly require well-balanced recursive decomposition of the row merge heap in order to be effective; for example, when the column ordering is done with the nested dissection heuristic. Problems with this approach include the fact that a good separator of the intersection graph does not necessarily correspond to an even distribution of rows between the two separated components of the intersection graph. Also, the nested dissection approach is not as effective as a minimum-degree ordering on some sparse problems, hence a "general purpose" algorithm such as the global row reduction algorithm is essential for these problems.

Thus, a significant advantage of the global row reduction algorithm is that it can be employed with any initial row distribution and is, in this sense, independent of the fill-reducing heuristic that was used in ordering the columns of the matrix. For example, the algorithm can handle the problem of "splinters" in the row merge heap (*i.e.* small subtrees rooted very high in the row merge heap). In the nonlocal merge algorithm, where a row partition is derived from a recursive decomposition of the binary row merge heap, such subtrees can result in a disparity in the work assigned to processors. This problem can be taken care of by the global row reduction algorithm, since each processor maintains a list of upper triangular matrices. The small amount of work represented by a splinter can be easily incorporated into any one of these lists.

Finally, we note that two other research groups, A. Pothen and P. Raghavan, and E. Chu and A. George, have also recently developed parallel algorithms and hypercube implementations for the sparse QR factorization. These algorithms seem to be essentially different than the global row reduction algorithm presented here, but, of course, they address many similar concerns.

Acknowledgements. The work reported in this paper was partially completed with the assistance of computing facilities of the Advanced Computing Facility at the Cornell Center for Theory and Simulation in Science and Engineering, which is supported by the National Science Foundation and New York State. The author would also like to thank one referee for very detailed and helpful comments.

REFERENCES

[1] R. Byrd, R. Schnabel, and G. Shultz, *Parallel quasi-Newton methods for unconstrained optimization*, Tech. Rep., Department of Computer Science, University of Colorado at Boulder, 1988.

[2] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert, *Predicting fill for sparse orthogonal factorization*, Journal of the Association for Computing Machinery, 33 (1986), pp. 517–532.

[3] T. F. Coleman and G. Li, *Solving systems of nonlinear equations on a message-passing multiprocessor*, Tech. Rep. CS–87–887, Computer Science Department, Cornell University, 1987.

[4] T. F. Coleman and J. J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM Journal on Numerical Analysis, 20 (1983), pp. 187–209.

[5] T. F. Coleman and P. Plassmann, *Solution of nonlinear least-squares problems on a multiprocessor*, Tech. Rep. CS–88–923, Computer Science Department, Cornell University, 1988.

[6] W. Gentleman, *Row elimination for solving sparse linear systems and least squares problems*, in Conference in Numerical Analysis, Lecture Notes in Mathematics 506, G. Watson, ed., Springer-Verlag, 1975, pp. 122–133.

[7] A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[8] J. George and M. Heath, *Solution of sparse least squares problems using Givens rotations*, SIAM Journal on Numerical Analysis, 34 (1980), pp. 69–83.

[9] J. W.-H. Liu, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Transactions on Mathematical Software, 12 (1986), pp. 127–148.

[10] ———, *On general row merging schemes for sparse Givens transformations*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 1190–1211.

[11] J. J. Moré, *The Levenberg-Marquardt algorithm: Implementation and theory*, in Lecture Notes in Mathematics, No. 630–Numerical Analysis, G. Watson, ed., Springer-Verlag, 1978, pp. 105–116.

[12] G. Newsam and J. Ramsdell, *Estimation of sparse Jacobian matrices*, Tech. Rep. TR-17-81, Aiken Computation Laboratory, Harvard University, 1981.

[13] P. E. Plassmann, *The Parallel Solution of Nonlinear Least-Squares Problems*, Cornell University Ph.D. thesis, 1990.

[14] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Transactions on Mathematical Software, 8 (1982), pp. 256–276.

[15] R. Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.

[16] E. Zmijewski, *Sparse Cholesky factorization on a multiprocessor*, Tech. Rep. 87–856, Cornell University, 1987.