A PARALLEL NONLINEAR LEAST-SQUARES SOLVER: THEORETICAL ANALYSIS AND NUMERICAL RESULTS

THOMAS F. COLEMAN* AND PAUL E. PLASSMANN†

Abstract. The authors recently proposed a new parallel algorithm, based on the sequential Levenberg-Marquardt method, for the nonlinear least-squares problem. The algorithm is suitable for message-passing multiprocessor computers.

In this paper we provide a parallel efficiency analysis and report our computational results. Our experiments were performed on an Intel iPSC/2 multiprocessor with 32 nodes: we present experimental results comparing our parallel algorithm with sequential MINPACK code executed on a single processor. These experimental results show that essentially full efficiency is obtained for problems where the row size is sufficiently larger than the number of processors.

Key words. hypercube computer, Levenberg-Marquardt, nonlinear least-squares, message-passing multiprocessor, parallel algorithms, QR-factorization, trust-region algorithms

AMS(MOS) subject classifications. 65H10, 65F05, 65K05, 65K10, 90C30

---

* Computer Science Department and Center for Applied Mathematics, Cornell University, Ithaca, New York 14853. Research partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-86ER25013.A000.

† 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.

1. Introduction. Let $F : \mathbf{R}^n \mapsto \mathbf{R}^m$, with $m \geq n$, be a continuously differentiable function. The nonlinear least-squares problem is to find a local minimum of the function

$$\psi(x) = \frac{1}{2}\|F(x)\|_2^2 = \frac{1}{2}\sum_{i=1}^{m} f_i^2(x), \tag{1.1}$$

where f_i is the i-th component of F. Recently, Coleman and Plassmann [CP89] proposed a parallel implementation of the well-known Levenberg-Marquardt algorithm [L44, M63] for solving this problem when the Jacobian of $F(x)$ is dense [1]. In this paper we present a theoretical analysis of our parallel method as well as experimental results obtained on an Intel iPSC/2 hypercube. The experimental results are obtained on a hypercube multiprocessor; however, we feel that the algorithm is not limited to this architecture. In fact, all that is required of the multiprocessor interconnection topology is support of a ring embedding and means for efficient gather and broadcast operations.

There are three main computational tasks that need to be addressed in a parallel implementation of the Levenberg-Marquardt algorithm [2]:

1. Evaluation or approximation of the Jacobian matrix $J(x)$.
2. The QR-factorization of $J(x)$,

   $$J = Q\begin{bmatrix} R \\ 0 \end{bmatrix}.$$

   where Q in an orthogonal matrix and R is upper-triangular, in order to solve the least-squares problem

   $$\begin{bmatrix} R \\ 0 \end{bmatrix} s \overset{\text{L.S.}}{=} -Q^T F. \tag{1.2}$$

3. The computation of the Levenberg-Marquardt parameter λ_*, and vector s_*, satisfying,

   $$(J^T J + \lambda_* D^T D)s_* = -J^T F, \tag{1.3}$$

   such that $\|Ds_*\|_2 = \Delta$, where D is a diagonal scaling matrix and Δ is a positive scalar representing the "trust region" size. Computationally, this means solving least-squares systems of the form,

   $$\begin{bmatrix} R \\ 0 \\ \lambda^{1/2} D \end{bmatrix} s(\lambda) \overset{\text{L.S.}}{=} -\begin{bmatrix} Q^T F \\ 0 \end{bmatrix}, \tag{1.4}$$

   for different values of λ.

We address these computational issues in the remainder of the paper. In Section 2 we summarize the issues involved with respect to the parallel finite-difference approximation of the Jacobian matrix. In Section 3 we summarize the row-oriented parallel QR factorization proposed in [CP89], provide a new complexity analysis, and present numerical results. A theoretical analysis of the parallel algorithm for determining the Levenberg-Marquardt parameter is given in Section 4 along with numerical results. Finally, we present experimental results for the entire method and conclusions in Section 5.

2. Parallel Approximation of the Jacobian. It is often the case that the number of rows of the Jacobian is much larger than the number of columns. For the QR factorization stage this suggests a row-oriented method, where the rows of the Jacobian are distributed to processors. This data distribution achieves a better load balancing than a column-oriented method, and results in an algorithm whose efficiency depends on the ratio m/p rather than n/p, where p is the number of processors. Experience has shown that computational costs involved in the QR factorization

---

[1] Plassmann [P90] considers the large sparse case.

[2] For a more detailed description of the Levenberg-Marquardt algorithm, including step acceptance and convergence criteria, we refer the interested reader to the excellent article by Moré [M78].

stage often dominate the Jacobian approximation stage. Thus, we have chosen to pursue a row-oriented QR factorization algorithm [3]. We would like to take advantage of this data distribution in approximating the Jacobian whenever possible. Let I_i, $i = 1, \ldots, p$, be a partition of the rows of J, where I_i is the set of row indices assigned to processor i and let $F_{I_i}(x) = \{f_j(x) \mid j \in I_i\}$ be the corresponding function blocks. We say that the function F is *block separable* if there exists a partition of the rows such that the evaluation of each function block is computationally independent.

Suppose the function is block separable relative to the partition I_i, $i = 1, \ldots, p$ and let $J_{I_i}^T(x)$ be the set of the rows of the Jacobian estimated at the point x. The j-th column of the Jacobian can be estimated in parallel by having each processor compute its block of row components according to the formula

$$J_{I_i}^T(x) e_j \cong \frac{F_{I_i}(x + \tau e_j) - F_{I_i}(x)}{\tau}. \tag{2.1}$$

However, often the evaluation of $F(x)$ is not completely separable; there may be some amount of redundant computation due to common factors that must be computed for each partition of the function $F_{I_i}(x)$, $i = 1, \ldots, p$. If this redundant computation is inexpensive relative to communication cost entailed by using a column-oriented scheme, then we consider this computational overhead tolerable. All of the test problems considered in the experimental section fall into this category. Otherwise, if the redundant computation required by such a partition of the rows is deemed too expensive, a column-oriented approach to approximating $J(x)$ must be adopted.

A subtle problem occurs when n/p is small, the evaluation of $F(x)$ is expensive and not separable, and therefore the estimation of the Jacobian is computationally expensive relative to the QR factorization. Suppose a step $s^{(k)}$ is to be considered at the k-th iteration of the algorithm; $F(x^{(k)} + s^{(k)})$ must be evaluated to determine if it meets certain acceptance criteria. When this computation is relatively expensive and not separable, and therefore must be done on one processor, then the remainder of the processors will remain idle during this computation. This can result in detrimental effects on the efficiency of the entire implementation. Byrd, Schnabel, and Shultz [BSS88] and Coleman and Li [CL87] note that this problem can be alleviated somewhat by guessing, based on the previous iteration, whether the proposed point will be accepted. If acceptance is assumed, the Jacobian at $x^{(k)} + s^{(k)}$ can begin to be approximated by idle processors. If we guess that the proposed iterate will not be accepted, then idle processors could evaluate the function at some additional points which might fare better with the acceptance criteria. These ideas were not implemented in our code, but could easily be added. Nevertheless, for $n/p \gg 1$, the computation required to estimate the Jacobian will always dominate these isolated function evaluations.

3. A Parallel Row-Oriented Householder QR Algorithm. In this section we analyze and experiment with the parallel row-oriented Householder QR factorization proposed in [CP89]. We show that this algorithm is more efficient than previous hybrid (Householder/Givens) factorization algorithms. The efficiency of the parallel QR factorization used to solve (1.2) is of paramount importance because a completely new approximation to the Jacobian is computed for each iteration. Consequently, a full QR factorization is also required. For the test problems considered in this paper, we find that the QR factorization is always a major (and sometimes the dominant) computational cost. An additional advantage of this algorithm is that, unlike the hybrid scheme, it produces the same Householder vectors that would be produced by a standard sequential Householder QR algorithm. This property is advantageous in situations where the same system must be solved for multiple right hand sides. Finally, we show that column pivoting can be introduced into the algorithm with only a slight increase in the computation and communication complexity. In our implementation column pivoting is important because the QR factorization can then be used to estimate matrix rank.

[3] If a column-oriented Jacobian approximation scheme is used, one must convert this column-oriented data distribution into a row-oriented distribution for the QR factorization stage [JH88, MVV87, SS85]. Of course, for sufficiently large n this problem can be avoided by using a column-oriented QR factorization algorithm. We did not take such an approach because it is usually the case that $m \gg n$. However, there exist good column-oriented QR factorization algorithms [B88, CG88, M87], and in Section 4 we describe an efficient column-oriented algorithm for determining the Levenberg-Marquardt parameter.

Column-oriented methods have dominated the work on parallel QR algorithms; however, two row-oriented algorithms have been considered previously [CP86, PR87]. These two algorithms are very similar: to reduce each column of the matrix a reduction involving only data local to each processor is performed, followed by a global reduction requiring communication between the processors. The reduction of rows local to a processor yields one row per processor with a nonzero in the column being reduced to upper triangular form. This approach has the advantage that all these reductions and matrix updates will be local to the processors, and with the wrap mapping [4] of rows the computational load will be well-balanced. Following this local stage is a global stage: a minimum-depth spanning tree is embedded in the hypercube, rooted at the processor where the nonzero for the column under consideration should reside. Rows are communicated up this tree and the leading nonzero is annihilated by a Givens rotation with respect to the parent's row. These rows are then updated with this rotation and the result communicated back to the child. The hypercube topology allows this global reduction process to take place in $\log(p)$ steps. Of these two algorithms the one presented by Pothen and Raghavan [PR87] seems to be the most efficient since Householder reductions, as opposed to Givens, are used in the local stage.

Our algorithm is computationally more efficient than the hybrid approach: the full Householder vector is calculated and the intermediate Givens reductions are avoided. However, our challenge is to obtain the same communication complexity as the hybrid approach. We meet this challenge by noticing that computation of the Householder vector and the subsequent rank-one update to the matrix can be combined to halve the number of messages that seem to be required at first glance.

To review the algorithm given in [CP89], consider the QR factorization of an $m \times n$ matrix A. At step j of the factorization the first $j-1$ rows of R and the Householder vectors have been computed; we need only consider the $(m-j+1) \times (n-j+1)$ lower right submatrix of A, denoted by $A^{(j)}$, with columns $a_k^{(j)}$, $k = j, \ldots, n$. The Householder transformation, $P^{(j)}$, to reduce the first column of $A^{(j)}$, $a_j^{(j)}$, is

$$P^{(j)} = [I - 2\frac{v_{(j)} v_{(j)}^T}{v_{(j)}^T v_{(j)}}], \tag{3.1}$$

where $v_{(j)} = a_j^{(j)} \pm \|a_j^{(j)}\|_2 e_j$. To determine $a_k^{(j+1)}$, $k = j+1, \ldots, n$, we need to compute the corresponding rank one update to $A^{(j)}$:

$$\begin{aligned} a_k^{(j+1)} &= a_k^{(j)} - \frac{2}{v_{(j)}^T v_{(j)}} v_{(j)}^T a_k^{(j)} v_{(j)} \\ &= a_k^{(j)} - \alpha_k^{(j)} v_{(j)}, \end{aligned} \tag{3.2}$$

with $\alpha_k^{(j)}$ defined as shown. Let *leader* designate the processor that holds row j. Note that $v_{(j)}$ agrees with $a^{(j)}$ except in the first component. Therefore, the portions of the inner product $v_{(j)}^T a_k^{(j)}$ local to each processor are just $a_j^{(j)T} a_k^{(j)}$ except on *leader* where $a^{(j)}$ and $v_{(j)}$ differ in the first component. We can take advantage of this fact and combine the communication to compute $v_{(j)}$ with the communication required for the rank-one update to the remainder of the matrix. An outline of the resulting algorithm is given as Algorithm 3.1. For this description we use the notation $[a_j^{(j)}]_{I_i}$ to represent the subvector of $a_j^{(j)}$ with components given by the index set I_i. The $\tilde{\alpha}$ vector is a work vector used in the computation of $\|a_j^{(j)}\|_2$ and the constants $\alpha_k^{(j)}$, $k = j+1, \ldots, n$.

In Figure 3.1 we exhibit the efficiencies of this algorithm compared to the hybrid algorithm described by Pothen and Raghavan in [PR87] as a function of the number of rows. (The data points in the figure are experimental results obtained on a 32 node iPSC/2 hypercube with 4.5 Mbytes of memory per node. All the experimental results presented in this paper are from implementations done on this machine.) The dotted lines in the figure are plots of a theoretical model of the efficiencies of the algorithms that will be presented later in this section. For this plot the number of

[4] The specific row-oriented distribution we consider is a wrapping of rows onto processors. Thus, if the processors on the ring are numbered $0, 1, 2, \ldots, p-1$, then row k of the Jacobian is assigned to processor $(k-1) mod(p)$.

Index Set: I_i {set of row indices assigned to processor i}

Proc (i) : {program for processor i}
  For $j = 1, ..., n$ **do**
    If (i =*leader*) Delete $\{j\}$ from I_i;
    Compute dot products for $k = j, \ldots, n$
      $\tilde{\alpha}_k = [a_j^{(j)}]_{I_i}^T [a_k^{(j)}]_{I_i}$;
    Combine $[\tilde{\alpha}_j, \ldots, \tilde{\alpha}_n]$ using *gather-sum*;
    If (i = *leader*) **then**
      Compute first component of $v^{(j)}$ and the coefficients
        $[\alpha_{j+1}^{(j)}, \ldots, \alpha_n^{(j)}]$ and *broadcast* the result;
    endif
    Update columns, $k = j+1, \ldots, n$
      $[a_k^{(j+1)}]_{I_i} = [a_k^{(j)}]_{I_i} - \alpha_k^{(j)} [v^{(j)}]_{I_i}$;
  enddo

ALGORITHM 3.1
A Parallel Row-Oriented Householder QR Algorithm

columns is fixed at 100. The efficiencies shown were calculated by dividing the time taken by an efficient sequential implementation of the algorithm run on one processor by the product of the time taken by the parallel implementation and the number of processors used. In this case our parallel implementations were compared with the MINPACK QR factorization subroutine QRFAC executed on a single processor of the hypercube and the efficiencies shown were computed from the execution times of these programs. In Table 3.1 we show some representative execution times for our implementations of the hybrid algorithm (Hybrid) and Algorithm 3.1 as compared to the sequential QR factorization program (Single Processor).

There is a subtle point in solving (1.2): the orthogonal matrix Q does not need to be saved if the right hand side of equation is updated along with the rows of the Jacobian. To achieve this goal the right hand side is treated like an additional column to the matrix J and distributed across the processors in the same wrap mapping and updated along with the corresponding rows of the Jacobian.

Column pivoting can be added to Algorithm 3.1. The column norms of the matrix A are initialized at the beginning. They are updated after each stage of the computation to obtain the column norms of $A^{(j)}$. For example, suppose at stage j the column norms $\|a_k^{(j)}\|_2$, $k = 1, \ldots, n$, are known by *leader*. The column of maximum norm, k_{max}, is determined by *leader* and the result is broadcast. Columns j and k_{max} are then interchanged by all processors. After stage j of the algorithm the updated norms can be obtained from the formula:

$$\|a_k^{(j+1)}\|_2 = \|a_k^{(j)}\|_2 \left(1 - \left(\frac{[a_k^{(j+1)}]_j}{\|a_k^{(j)}\|_2}\right)^2\right)^{\frac{1}{2}}, \tag{3.3}$$

for $k = j+1, \ldots, n$. The results are then sent to the next leader (*i.e.* the next processor on the ring) for stage $j+1$ of the QR algorithm. Note that numerical cancellation is a potential problem in computing these norm updates. However, circumstances that would result in this problem can be monitored and the suspect column norm can be recomputed. A standard way to monitor for numerical cancellation is to keep track of the products of the multiplicative factors in (3.3) that have been obtained since the last explicit calculation of the column norm. When this product is sufficiently less than one, then there is the possibility of cancellation error, and the column norm is recomputed. In our implementation the recomputation is done by broadcasting a special notifier to the other processors instead of the column pivot. The required column norms are then recomputed and the result gathered at *leader*. Our observation has been that recomputation of the column norms is rarely required and therefore does not significantly affect the efficiency of the algorithm.

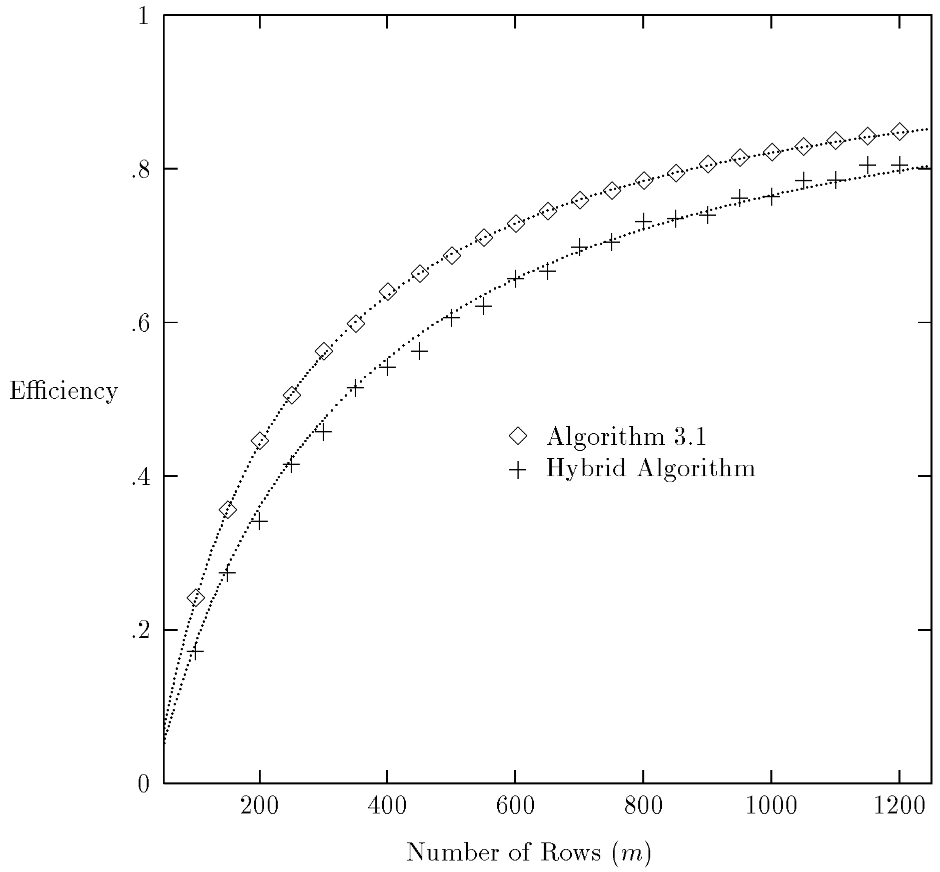

FIG. 3.1. *Efficiencies of Algorithm 3.1 and Hybrid on the iPSC/2 ($n = 100$, $p = 32$)*

Another potential concern for numerical stability might be the possibility of overflow from the way the $\tilde{\alpha}_k$ are computed in Algorithm 3.1. We note that these partial sums can be scaled by the most recent approximation to the column norms available to all the processors. We did not find it necessary to include this scaling in our implementation.

Figure 3.2 exhibits a graph comparing the efficiencies of Algorithm 3.1 with and without pivoting. In Table 3.2 we include some representative times from these experiments. The efficiencies are again computed by comparing the running times of the parallel algorithms to running times of the MINPACK QR subroutine QRFAC on a single processor. As before these results were obtained on a 32 node iPSC/2 hypercube with the number of columns fixed at 100. The data points are the experimental results and the dotted curves are theoretical approximations to these efficiencies that we will now describe.

The efficiencies observed for the row-oriented Householder algorithm can be explained by a simple model for the communication overhead involved and consideration of computational imbalances between the processors. The efficiency is computed by the formula

$$\text{efficiency} = \frac{t_{\text{seq}}}{p\ t_{\text{parallel}}}, \tag{3.4}$$

where t_{seq} is the execution time of the sequential algorithm and t_{parallel} is the execution time of the parallel algorithm on p processors. The parallel execution time can be considered to consist of three parts: (1) the optimal time, t_{seq}/p, (2) the computational imbalance relative to the optimal distribution of work, t_{comp}, and (3) the communication overhead demanded by the parallel algorithm, t_{comm}. Hence, we have that

$$t_{\text{parallel}} = \frac{t_{\text{seq}}}{p} + t_{\text{comp}} + t_{\text{comm}} \tag{3.5}$$

TABLE 3.1
Execution Times of Algorithm 3.1 and Hybrid on the iPSC/2 Hypercube

Execution Times (sec), without Pivoting					
n	m	Single Processor	p	Hybrid	Algorithm 3.1
100	200	31.60	8	5.17	4.86
			16	3.60	3.08
			32	2.98	2.27
100	400	69.44	8	9.89	9.64
			16	5.86	5.45
			32	4.11	3.46
100	800	145.28	8	19.41	19.20
			16	10.59	10.22
			32	6.38	5.84
100	1600	299.86	8	39.21	39.07
			16	20.47	20.15
			32	11.25	10.76
200	400	246.85	8	35.02	34.12
			16	20.44	19.01
			32	14.10	11.77
200	800	543.14	8	73.01	72.18
			16	39.09	37.82
			32	23.01	21.12
400	400	775.93	8	113.11	109.32
			16	66.47	61.04
			32	46.81	37.97

and (3.4) can be rewritten as

$$\text{efficiency} = \frac{1}{1 + \frac{t_{\text{comm}} + t_{\text{comp}}}{t_{\text{seq}}} p}. \tag{3.6}$$

The sequential execution time of the Householder QR algorithm measured in old-style flops is

$$t_{\text{seq}} = n^2(m - n/3). \tag{3.7}$$

In the discussion that follows we use equation (3.7) to define the length of time we consider to be one flop. On the iPSC/2 this time was experimentally determined to be 19.15μsec. However, this definition can be tricky since, for example, an add, multiply, or divide can take varying lengths of time to execute depending on how the code is written. Consequently, some of the coefficients in the following formulae had to be obtained experimentally and are not simple multiples of the above defined flop.

To approximate the computational imbalance we consider two dominant terms. The first is due to the variation of the number of rows assigned to the processors, and the second term is due to the idle time of processors during the computation of the α's in Algorithm 3.1. Work is not quite equally distributed to the processors with the row wrapping. On the average, half the processors are assigned an extra row; hence, the remaining processors are idle during the portion of the Householder update corresponding to this extra row. The Householder update to a row of length k requires $2k$ flops, resulting in a computational imbalance over the entire factorization of $\frac{1}{2}\sum_{k=1}^{n} 2k = n^2/2$ flops. The total idle time of processors during the accumulations of sums in the computation of an $\tilde{\alpha}$-vector of length k is approximately $(\log(p)-1)k\tau_{\text{add}}$ where τ_{add} is the time required for an add. Summing this expression from $k = 1$ to n yields $\frac{1}{2}(\log(p) - 1)n^2\tau_{\text{add}}$. Finally, the processor *leader* requires some length of time, say β_1, to compute each element of the α-vector and time β_2 per element to update the column norms. These computations result in a total imbalance of $(\beta_1 + \beta_2)n^2/2$. Combining these contributions yields an approximation for t_{comp}, in flops, of

$$t_{\text{comp}} \approx (\beta_1 + \beta_2 + (\log(p) - 1)\tau_{\text{add}} + 1)n^2/2. \tag{3.8}$$

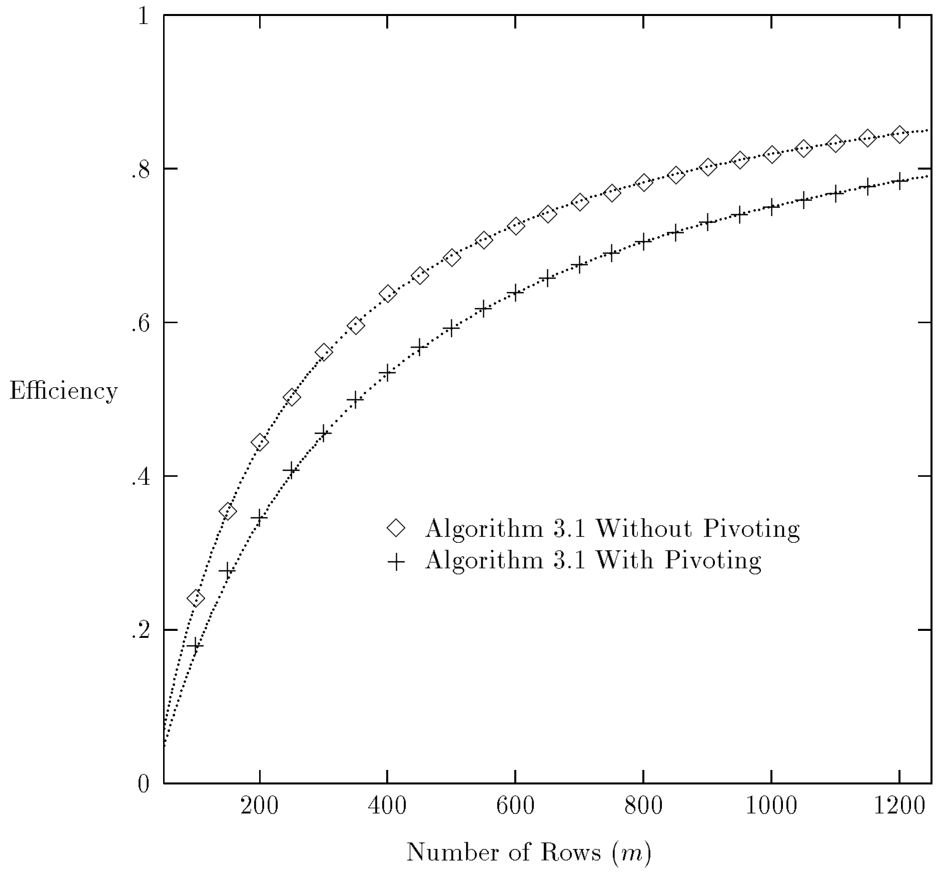

FIG. 3.2. *Efficiencies of Algorithm 3.1 With and Without Column Pivoting ($n = 100$, $p = 32$)*

In our implementation, the times β_1 and β_2 were determined to be 22.4μsec and 110.6μsec and τ_{add} was found to be 11.2μsec.

The communication overhead for Algorithm 3.1 includes the time required for the accumulation and broadcast of the α-vectors. This overhead is $\sum_{k=1}^{n} 2\log(p)\tau(k)$, where $\tau(k)$ is the time required to send a double precision vector of length k between neighboring processors. An additional time of $n\log(p)\tau(1) + \sum_{k=1}^{n}\tau(k)$ is required to broadcast the pivot and transfer the column norms. Combining these two terms yields an approximation to the communication overhead of

$$t_{\text{comm}} \approx (2\log(p) + 1)T(n) + n\log(p)\tau(1), \tag{3.9}$$

where we define $T(n)$ to be $\sum_{k=1}^{n}\tau(k)$.

For the iPSC/2 the function $\tau(k)$ is, fortunately, empirically simple to describe; the cost function is essentially linear over large ranges of vector lengths k. Experimentally, we determined that a good approximation to this cost function is given by

$$\begin{aligned} \tau(k) &\approx \tau_1 + \gamma_1 k \quad , \quad 1 \le k \le 12 \\ \tau(k) &\approx \tau_2 + \gamma_2 k \quad , \quad 13 \le k. \end{aligned} \tag{3.10}$$

The start-up times, τ_1 and τ_2, were determined to be 378μsec and 702μsec. The incremental costs, γ_1 and γ_2, are 1.19μsec/value and 2.87μsec/value. With these coefficients, the term $T(n)$ in (3.9) can be approximated, for $n \ge 13$, by

$$T(n) \approx \gamma_2 n^2/2 + \tau_2 n + 78(\gamma_1 - \gamma_2) + 12(\tau_1 - \tau_2). \tag{3.11}$$

After substituting these coefficients into the equations for t_{comp} and t_{comm}, (3.6) was plotted along with the experimental results for Algorithm 3.1 in Figures 3.1, 3.3, and 3.4. To model the efficiency of the row-oriented Householder algorithm without pivoting we need only eliminate the

Table 3.2
Execution Times of Algorithm 3.1 with Pivoting on the iPSC/2 Hypercube

Execution Times (sec), with Pivoting				
n	m	Single Processor	p	Algorithm 3.1
100	200	32.36	8	5.62
			16	3.82
			32	3.01
100	400	70.60	8	10.44
			16	6.22
			32	4.21
100	800	147.10	8	20.13
			16	11.03
			32	6.61
100	1600	303.11	8	40.07
			16	21.01
			32	11.56
200	400	249.97	8	36.88
			16	21.65
			32	14.36
200	800	547.77	8	75.15
			16	40.57
			32	23.77
400	400	785.60	8	119.11
			16	70.66
			32	47.52

β_2 term from the equation for t_{comp} and also the communication overhead due to pivoting in the equation for t_{comm}. The resulting modeling function is plotted in Figures 3.1 and 3.2.

Finally, to model the hybrid algorithm we note that only t_{comp} must be modified from the analysis of the efficiency of Algorithm 3.1. Instead of accumulating sums as in Algorithm 3.1, the hybrid algorithm performs a nonlocal binary reduction of rows by Givens rotations. The binary reduction of rows of length k by Givens rotations entails a total idle time for the processors of approximately $(\log(p)-1)k\tau_{\text{Givens}}$, where τ_{Givens} is the time required to apply a Givens rotation to a 2-vector. For the iPSC/2, τ_{Givens} was measured to be approximately 37.3μsec. Summing this value from $k=1$ to n yields total time $\frac{1}{2}(\log(p)-1)n^2\tau_{\text{Givens}}$. Including the term for the differing number of rows assigned to processors we have that computational imbalance for the hybrid algorithm measured in flops is

$$t_{\text{comp}}^{(\text{hybrid})} \approx ((\log(p)-1)\tau_{\text{Givens}}+1)n^2/2. \tag{3.12}$$

Substituting this expression into (3.6) along with the expression for t_{comm} without pivoting we obtain the efficiency modeling function plotted in Figure 3.1.

4. A Parallel Implementation of the Levenberg-Marquardt Algorithm. To determine the Levenberg-Marquardt parameter, the matrix in (1.4) must be reduced to upper triangular form. This reduction is computationally intensive: $n(n+1)/2$ Givens rotations and the corresponding row updates, or $O(n^3)$ flops. Note that the work required in this reduction is independent of m, the number of rows. Algorithm 4.1 details a parallel method to accomplish this reduction. In the algorithmic description let S represent storage for an upper triangular matrix which is initially set equal to the matrix $\sqrt{\lambda}I$ in (1.4). Remember that the rows of R and S are wrapped onto an embedded ring of processors as described in Section 1.

Algorithm 4.1 proceeds in n stages which have been indexed by $j=0,\ldots,n-1$ in the description. At stage j of Algorithm 4.1 the superdiagonal of S that is a distance j from the main diagonal is eliminated by Givens rotations. After n stages, the upper triangular matrix S has been completely zeroed and the updated upper triangular matrix R is still wrapped onto the processors

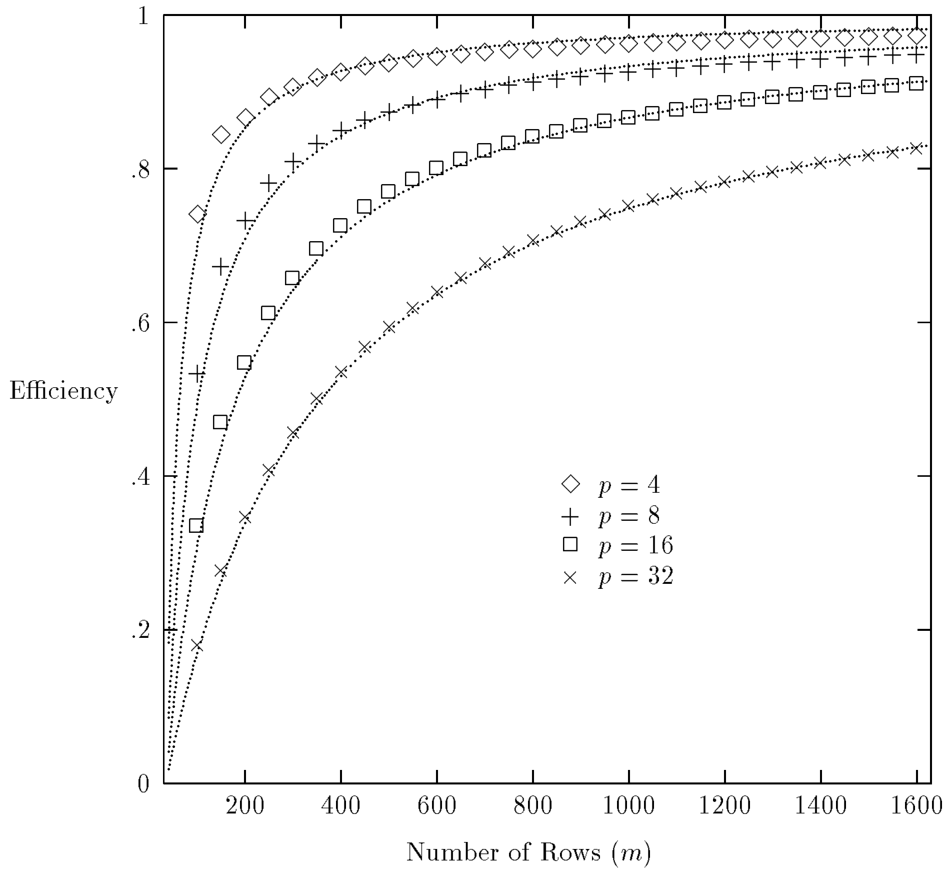

FIG. 3.3. *Efficiencies of Algorithm 3.1 With Pivoting* ($n = 100$)

in the same manner as at the start of the algorithm. As the leading nonzero of each row of S is eliminated and the corresponding rows updated, the rows of S move around the embedded ring in a systolic manner. Although the work at each stage is not completely balanced, the processor doing the most work rotates around the ring. This imbalance is somewhat offset by the required communication. Experimental results of the efficiency of this algorithm as a function of the number of columns are presented as data points in Figure 4.1. Also plotted in the figure are the modeling functions for these efficiencies that we will develop below.

Similar to the analysis of the QR factorization algorithms, we can model the observed efficiencies of Algorithm 4.1. First note that the total sequential work, measured in flops, is given by the formula

$$\begin{aligned} t_{\text{seq}} &\approx \sum_{k=1}^{n} 4\left(\frac{k^2}{2}\right) \\ &\approx \frac{2}{3}n^3 \qquad (4.1) \end{aligned}$$

since the application of each Givens rotation to a 2-vector requires 4 flops. As before, we use the above equation to define the length of time we consider to be a flop. On the iPSC/2 this time was determined to be 10.9μsec.

At each step of the outer loop in Algorithm 4.1 there is a processor assigned the longest rows relative to other processors. Each of these rows differs from the average row length by $p/2$ elements. Since each processor has approximately k/p rows at step $n-k$, we have that the computational imbalance is bounded by

$$\begin{aligned} t_{\text{comp}}^{(\text{row})} &\approx \sum_{k=1}^{n} 4\left(\frac{k}{p}\right)\left(\frac{p}{2}\right) \\ &\approx n^2. \qquad (4.2) \end{aligned}$$

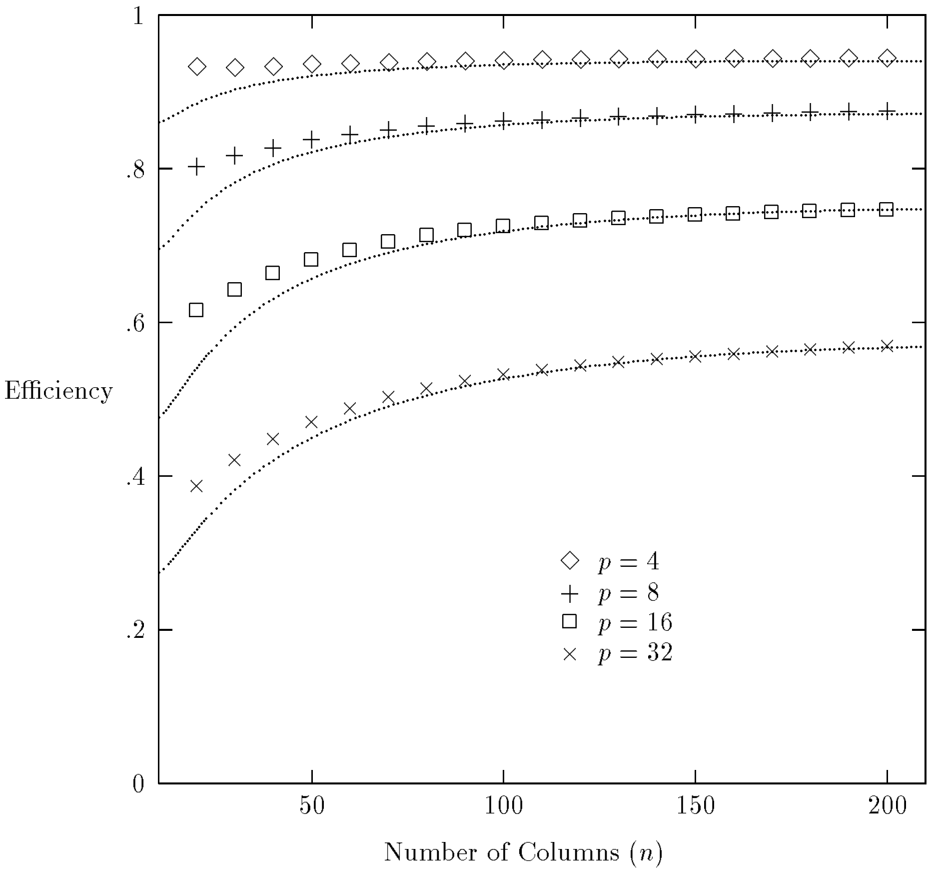

FIG. 3.4. *Efficiencies of Algorithm 3.1 With Pivoting* ($m = 400$)

To compute the communication overhead we define the function $\rho(k)$ to be the length of time for all the processors on an embedded ring to send synchronously a double precision vector of length k to their neighbors. Experimentally, this function is essentially linear; hence, we introduce the approximation

$$\rho(k) \approx \rho_0 + \phi k. \tag{4.3}$$

For the iPSC/2 we determined values of 1105μsec for ρ_0 and 6.85μsec/value for ϕ.

At iteration $n - k$ of Algorithm 4.1 the message length is approximately $k^2/(2p)$; hence, we have that

$$\begin{aligned} t_{\mathrm{comm}}^{(\mathrm{row})} &\approx \sum_{k=1}^{n} \rho(\frac{k^2}{2p}) \\ &\approx \rho_0 n + \phi \frac{n^3}{6p}. \end{aligned} \tag{4.4}$$

Using (3.6), we obtain the following modeling function for the efficiency of Algorithm 4.1

$$\mathrm{efficiency}^{(\mathrm{row})} \approx \frac{1}{1 + \phi/4 + \frac{3}{2}(p/n + \rho_0 p/n^2)}. \tag{4.5}$$

After substituting the necessary coefficients into (4.5), the resulting efficiency functions were plotted in Figure 4.1. Note that for large n the efficiencies do not asymptotically approach 1, but rather approach the constant $1/(1 + \phi/4)$, which is independent of the number of processors used.

A column-oriented approach is also possible and is presented as Algorithm 4.2. Experimental results for this algorithm are compared to those of Algorithm 4.1 in Table 5.1 and also plotted in Figure 4.2. For Algorithm 4.2 the columns, as opposed to the rows, of R and S are wrapped onto

Index Set: I_i {set of row indices assigned to processor i}
Functions: *next* {returns number of next processor in the ring},
prev {returns number of previous processor in the ring}

Proc (i) : {program for processor i}
  For $j = 0, \ldots, n-1$ **do**
    If $(j \neq 0)$ *receive* rows S^T_{k-j}, $k \in I_j$, from processor *prev*(i);
    For $k \in I_j$ **do**
      Compute Givens rotation to zero the bottom
        of the vector $(R_{k,k}, S_{k-j,k})^T$;
      Update rows R^T_k and S^T_{k-j} with above Givens rotation;
      If $(k = j+1)$ Delete $\{k\}$ from I_i;
    enddo
    Send rows S^T_{k-j}, $k \in I_j$, to processor *next*(i);
  enddo

Algorithm 4.1
A Parallel Row-Oriented R-S Reduction

the ring of processors. Rather than communicating rows of S between neighboring processors, the Givens rotations are stored in vectors g that rotate around the ring. Once the algorithm has been running for more than p steps, *i.e.* $j \geq p-1$, then the Givens vector g is completely filled with updates that need to be applied once received. The order in which these rotations are applied in the l loop is important. Since they operate on the same row of S, the rotations must be applied from the oldest to the most recent. Also, by row R^T_k we mean the nonzero components of row R^T_k that are local to processor i. These components are given by the index set K_i.

Even though Algorithm 4.2 is a bit more complicated, the total number of messages that have to be sent is the same as in Algorithm 4.1. However, for large n/p, the total number of values that have to communicated is actually less. For an average step j in Algorithm 4.2 we need only communicate the single Givens vector g of length $O(n)$ between neighboring processors. For the row-oriented version we need to communicate $O(n/p)$ rows of S of length $O(n)$ between processors. In practice, the rows of S are combined into one long message which results in the same number of communication start-ups as appear in Algorithm 4.2. The message start-up cost, measured in equivalent flops, for the Intel iPSC hypercube is very expensive and is normally the dominant factor in the communication cost of an algorithm. For large n/p, however, the average message lengths are extremely different; hence, in comparing Figures 4.1 and 4.2 it is apparent that the column-oriented version is asymptotically superior. By the same argument, for small n/p, the row-oriented is superior. This crossover in the observed efficiencies of the two algorithms can be explained by also modeling the efficiency of Algorithm 4.2.

The computational imbalance of Algorithm 4.2 is the same as that for the row-oriented algorithm. Hence, we need only modify the expression for the communication overhead in the efficiency model. Since at iteration $n-k$ in the column-oriented algorithm, each processor sends k Givens rotations to its ring neighbor, we have that

$$\begin{aligned} t^{(\text{col})}_{\text{comm}} &\approx \sum_{k=1}^{n} \rho(2k) \\ &\approx \rho_0 n + \phi n^2. \end{aligned} \tag{4.6}$$

Combining this expression with the bound for the computational imbalance obtained earlier we obtain an approximation to the efficiency of Algorithm 4.2

$$\text{efficiency}^{(\text{col})} \approx \frac{1}{1 + \frac{3}{2}((1+\phi)p/n + \rho_0 p/n^2)}. \tag{4.7}$$

Comparing (4.5) and (4.7) we note that they are equal for $n^* = 6p$. Experimentally, this crossover in the efficiencies of the two algorithms can be observed in Figure 4.3. In this figure the crossover

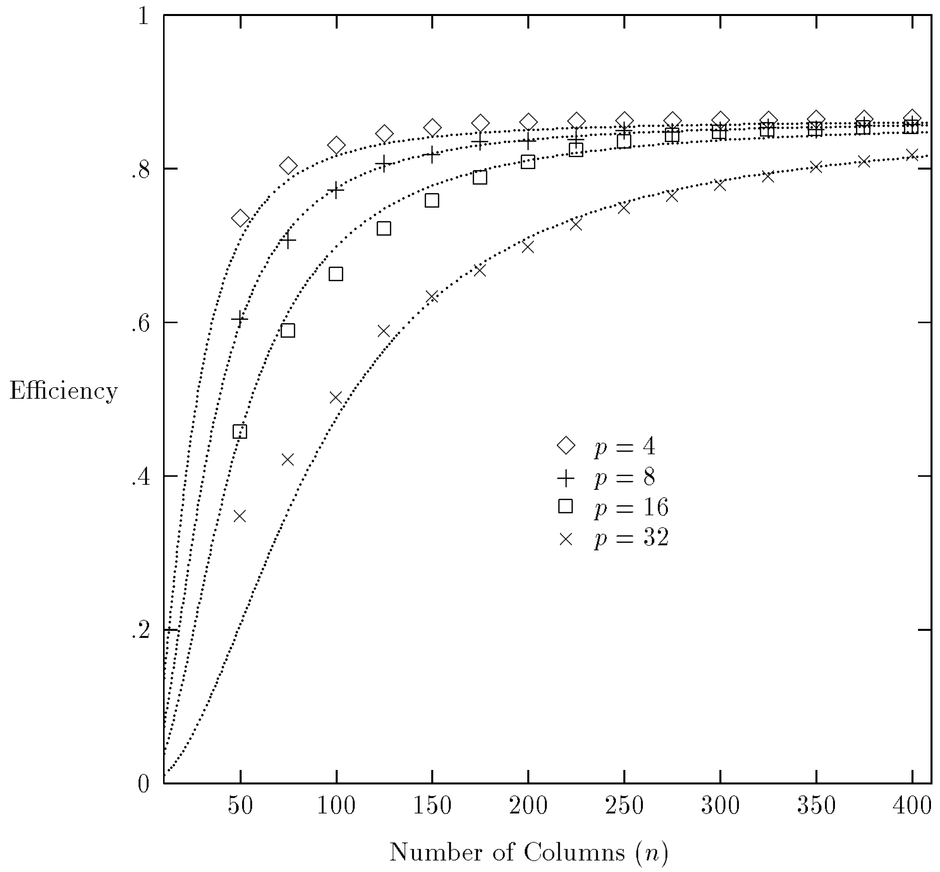

FIG. 4.1. *Efficiencies of Algorithm 4.1 on the iPSC/2*

appears to occur near $n = 150$, close to the value of $n^* = 192$ predicted by the efficiency modeling functions.

Finally, we note that there are two possible ways to improve the asymptotic performance of the row-oriented algorithm. The first would be to wrap blocks of rows, say b rows, onto the processors instead of single rows. This would decrease the length of messages sent at each iteration by a factor of $1/b$. Of course, this approach would also increase the computational imbalance by a factor of b. Following the analysis done above for the row-oriented algorithm we find that the optimum block size is $b^* = \sqrt{\phi n/(6p)}$. For this value of b, the asymptotic efficiency of the algorithm is improved to $1/(1 + \sqrt{3\phi p/(2n)})$. However, note that for the value of ϕ determined above and for $p = 32$, n must be greater than 610 for the efficiency to improve when using block size $b = 2$ instead of block size $b = 1$. A second possible improvement would be to decrease the length of the messages sent in the row-oriented method by postponing the application of the Givens rotations. Unfortunately, both of these algorithms are very complicated and were not implemented.

The reduction of the matrix in (1.4) to upper triangular form is the major task in a parallel algorithm for determining the Levenberg-Marquardt parameter λ_*, and we have shown that there exist effective algorithms to perform this reduction. However, efficient solution of triangular systems is also important in this context. In fact, for each iteration involving a solution of (1.4) there are two associated triangular solutions that are used to bracket the solution λ_* [M78]. Recently, much work has been done on the parallel solution of triangular systems [C86, LC88, LC89, HR88]. We used the triangular solution algorithms developed by Li and Coleman in our implementations, but it should be noted that the efficiencies of these algorithms are not nearly as good as those of Algorithms 4.1 and 4.2. This difference is what accounts for the discrepancies between the efficiencies shown in Figure 4.1 and the efficiencies reported in the next section for solving for the Levenberg-Marquardt parameter. This effect is apparent since even though there is an $O(n)$ difference between the amount of work required for Algorithm 4.1 and the corresponding triangular solutions, their communication costs are comparable. The importance of efficient parallel triangle system solvers has also been

Index Set: K_i {set of column indices assigned to processor i}

Proc (i) : {program for processor i}
  For $j = 0, \ldots, n-1$ **do**
    If $(j \neq 0)$ *Receive* Givens vector g from processor $prev(i)$;
    For $k \in K_j$ **do**
      For $l = min(j, p-1), \ldots, 1$ **do**
        Update rows R^T_{k-l} and S^T_{k-j} with Givens rotation g_{k-l};
      enddo
      Compute Givens rotation to zero the bottom
        of the vector $(R_{k,k}, S_{k-j,k})^T$;
      Update rows R^T_k and S^T_{k-j} with above Givens rotation;
      Update g_k in Givens vector;
      If $(k = j+1)$ Delete $\{k\}$ from I_i;
    enddo
    Send Givens vector g to processor $next(i)$;
  enddo

Algorithm 4.2
A Parallel Column-Oriented R-S Reduction

observed in the parallel solution of systems of nonlinear equations [CL87].

5. Experimental Results and Conclusions. 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 and run under version R3.2 of the iPSC operating system. The efficiencies shown below were calculated by dividing the running time of MINPACK [MGH80] code on a single processor by the number of processors used times the running times of the parallel algorithms. Therefore, a large number corresponds to greater efficiency. The comparison is fair, as both programs generate the same sequence of iterates and consequently do the same number of Jacobian approximations, QR factorizations, and Newton iterations in computing the Levenberg-Marquardt parameter.

The test problems used to obtain the experimental results are described in Table 5.1. Shown is the functional form of the test problems, and the computational complexity of evaluating each function is given in the column labeled "F Eval. Cost." We also make a subjective determination as to whether estimation of the Jacobian is cheap or expensive relative to its QR factorization. To be more specific about the functions used for testing, problem 1 has an $O(m)$ evaluation cost because of a special form for the matrix A: $A_{ij} = -2/m$, $i \neq j$; $A_{ii} = 1 - 2/m$. The matrix A used in problem 2 is of low rank: $A_{ij} = ij + O(\epsilon)$ perturbations, where ϵ is the machine precision. For problem 3 the tensor A has a bandwidth of 2, allowing for function evaluation with $O(m)$ cost. The constants used for problem 4 were $\alpha_j = -j$ and $\tau_i = i/m$. In the final column of Table 5.1 we note whether we consider evaluation of these functions to be separable.

Tables 5.2 through 5.5 summarize the experimental results obtained by comparing our parallel algorithms with the MINPACK code running on a single processor for solving the test problems described in Table 5.1. The efficiencies and the fraction of the total parallel running time spent in each of six sections of the programs are detailed. The six sections refer to: QR, the QR factorization of the Jacobian approximation; L-M, computation of the Levenberg-Marquardt parameter; R-S, the R-S reduction described in Section 5; Tri. S., the solution of triangular systems; and J Appr., the approximation of the Jacobian by forward differences. We include in J Appr. the function evaluations necessary to estimate the Jacobian, but not the function evaluations done to test the step acceptance criteria. These function evaluations are included in the total time but not in one of the six sections. Their efficiency is comparable to that of the Jacobian approximation but their fraction of the total running time is much smaller.

These results were chosen to illustrate an average-case behavior of the parallel algorithms in solving nonlinear least-squares problems. For a particular problem the fraction of time spent in the different routines and the number of iterations required for convergence can vary dramatically for

TABLE 4.1
Execution Times of Algorithms 4.1 and 4.2 on the iPSC/2 Hypercube

			Execution Times (sec)	
n	Single Processor	p	Algorithm 4.1	Algorithm 4.2
50	1.06	8	0.22	0.21
		16	0.15	0.15
		32	0.09	0.12
100	7.78	8	1.26	1.20
		16	0.74	0.72
		32	0.48	0.49
150	25.54	8	3.90	3.59
		16	2.11	2.01
		32	1.25	1.25
200	59.67	8	8.97	8.07
		16	4.63	4.37
		32	2.67	2.59
300	198.28	8	29.43	25.95
		16	14.66	13.68
		32	7.95	7.56
400	466.14	8	67.91	60.27
		16	34.24	31.28
		32	17.79	16.71

various choices of a starting point, initial trust-region size, and termination tolerances. For the above problems the MINPACK tolerances were set to $\sqrt{\epsilon}$, where ϵ is the machine precision. The initial trust-region size was given by $100\|\hat{x}_0\|_2$, where $\hat{x}_0$ is the starting point normalized by the 2-norms of the columns of the initial Jacobian approximation. For problems 1, 2, and 3 an n-vector of ones was used as the starting point, and for problem 4 the starting point $x_j = -(j + 0.1)$ was employed. For the problem sizes shown, problem 1 required 2 to 3 iterations (Jacobian approximations) for convergence, problem 2 used 10 to 12 iterations. Problem 3 required 7 to 8 iterations to converge and more than 10 iterations were required for problem 4 to converge; the results shown in Table 5.5 were taken from the first 10 iterations.

From these results it is apparent that either the QR factorization or the Jacobian approximation is the dominant computational cost for these problems. When the QR factorization cost dominates, the implementation is more efficient as m, the number of rows, increases. The cost of computing the Levenberg-Marquardt parameter can also be significant; this computation could dominate the QR factorization for problems that require a disproportionately large number of R-S reductions and for which the ratio m/n is close to one. This effect would occur in problems where the Jacobian is rank-deficient or the function is very nonlinear (which would require a small trust-region in order to ensure an accurate quadratic model of the function). But again, for a fixed ratio m/n, the efficiency in solving for the Levenberg-Marquardt parameter increases as m increases. As expected, the parallel triangular system solutions are very inefficient when compared to the QR factorization and R-S reductions. However, the time required for these solutions composes only a small fraction of the total computation time of the parallel implementation, and therefore it does not significantly decrease the overall efficiency. However, note that for fixed problem size the fraction of total time spent solving triangular systems does increase significantly as the number of processors is increased.

For functions whose evaluation is very expensive we observe that the computation required for the approximation of the Jacobian by forward differences can equal or exceed the computation required for these other tasks. For the expensive test functions we considered, the function evaluation was separable, and hence the row-oriented Jacobian approximation algorithm yielded efficiencies comparable to the QR factorization and Levenberg-Marquardt parameter solves. If the function evaluation were expensive and not separable, one would have to resort to a column-oriented Jacobian approximation algorithm as described in Section 3. In this case the efficiency of the implementation would depend on the number of columns being sufficiently large. Another possibility would be

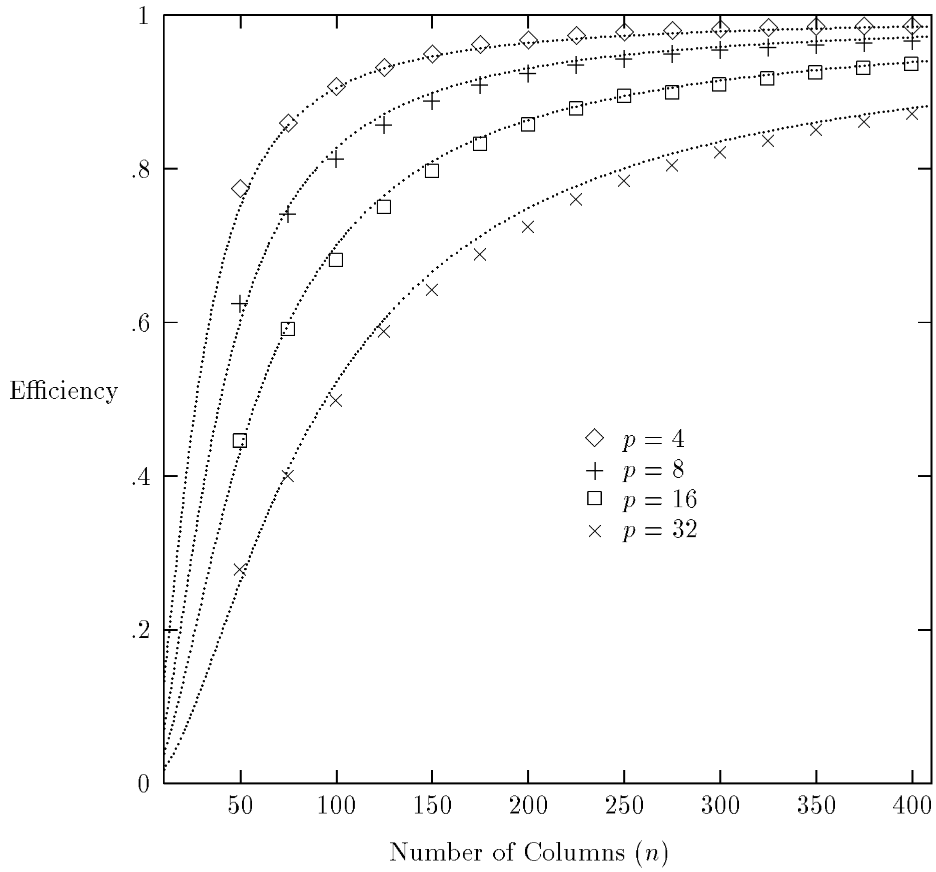

FIG. 4.2. *Efficiencies of Algorithm 4.2 on the iPSC/2*

to obtain an algebraic expression for the Jacobian and evaluate the Jacobian directly rather than using forward differences; the algebraic evaluation of the Jacobian could be separable even when the function itself is not.

When the function evaluation is expensive and not separable, a column-oriented implementation suggests itself. In this context, the improved efficiency of using a pipelined, column-oriented QR factorization is tempting. However, complete pivoting destroys the pipelining aspect of these column-oriented algorithms. A local pivoting strategy is possible [B88], but the effect of a different pivoting strategy on the solution of these nonlinear problems would have to be tested.

In summary, we have observed good efficiencies when solving moderate-sized nonlinear least-squares problems on the Intel hypercube. For the test problems considered we noted that the efficiency of our parallel implementation improved as the ratio m/p increased, where m is the number of rows of the Jacobian and p is the number of processors. We also point out that it is possible to solve much larger problems than those we have described above (which had to be run on one processor for comparison). The efficiencies for such larger problems would be correspondingly better. A related topic is the solution of large sparse problems: Plassmann [P90] has considered the general case, and future consideration should be given to problems that have special structure. For example, the row-oriented approach could also be used if the the Jacobian were banded and would be efficient in solving problems with a block structure.

Acknowledgements. The work reported in this paper was partially completed with the assistance of computing facilities of the Advanced Computing Research Institute at the Cornell Center for Theory and Simulation in Science and Engineering, which is supported by the National Science Foundation and New York State. We would also like to acknowledge discussions at Oak Ridge National Laboratory (as part of their Numerical Linear Algebra Year) and thank the referees for a number of constructive comments.

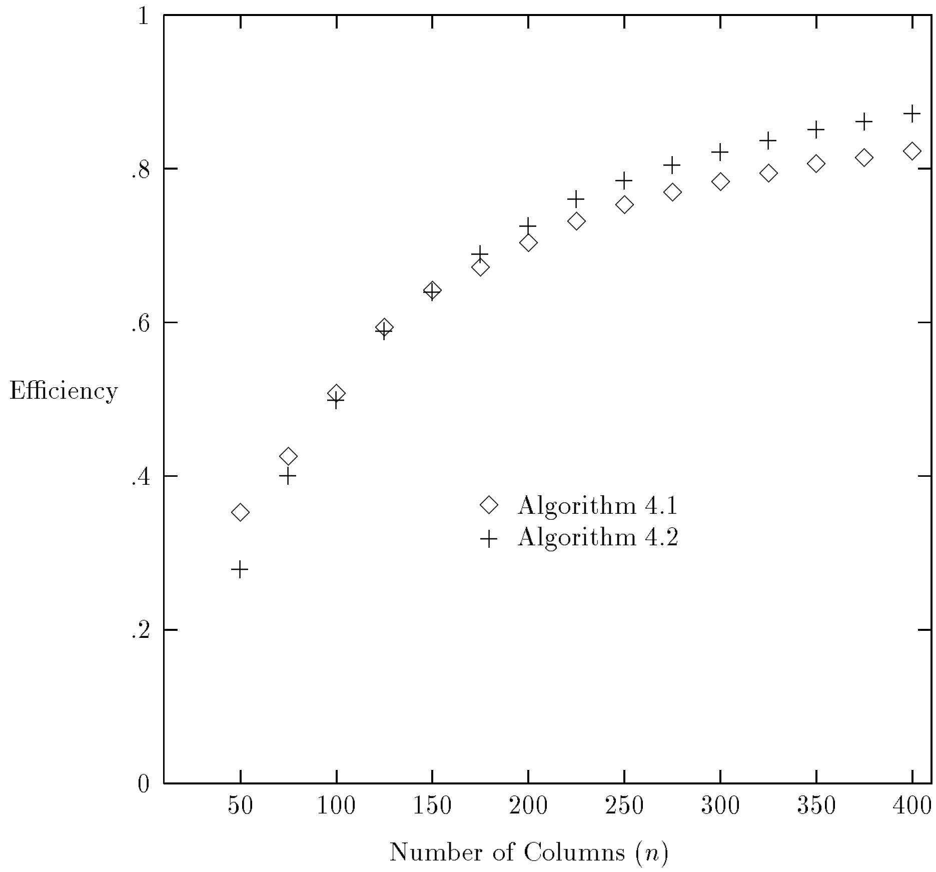

FIG. 4.3. *Efficiencies of Algorithms 4.1 and 4.2 ($p = 32$)*

TABLE 5.1
Description of Test Functions

Problem Number	Function Form	Characteristics	F Eval. Cost	J Approx. Cost	F Eval. Separable?
1	$F = Ax - e$	$A \in \mathbf{R}^{m\times n}$, full rank, Ax easily evaluated, e an m-vector of ones	$O(m)$	cheap	no
2	$F = Ax - e$	$A \in \mathbf{R}^{m\times n}$, low rank, dense	$O(mn)$	expensive	yes
3	$f_i = \sum_{i,k} A_{ijk} x_j x_k - 1$	A sparse	$O(m)$	cheap	yes
4	$f_i = (\exp(x_j \tau_i) - \exp(\alpha_j \tau_i))$	Constrained, $x < 0$ for $\alpha < 0$ and $\tau > 0$	$O(mn)$	expensive	yes

Table 5.2
Experimental Results of Parallel Algorithms Compared with MINPACK

Efficiencies Compared to MINPACK (% time spent in routine)									
Prob	n	m	p	Total	QR	L-M	R-S	Tri-S	J-Appr
1	100	250	8	.732	.757	.182		.202	.498
				(100.0)	(94.1)	(0.8)	(0.0)	(0.7)	(3.0)
			16	.531	.585	.045		.047	.326
				(100.0)	(88.2)	(2.4)	(0.0)	(2.1)	(3.4)
			32	.322	.391	.023		.023	.191
				(100.0)	(80.0)	(2.9)	(0.0)	(2.6)	(3.5)
1	100	500	8	.849	.859	.189		.202	.624
				(100.0)	(96.8)	(0.4)	(0.0)	(0.4)	(2.3)
			16	.709	.745	.045		.047	.451
				(100.0)	(93.2)	(1.5)	(0.0)	(1.4)	(2.7)
			32	.518	.576	.023		.023	.291
				(100.0)	(88.1)	(2.1)	(0.0)	(2.0)	(3.0)
1	100	1000	8	.908	.915	.181		.202	.737
				(100.0)	(97.5)	(0.2)	(0.0)	(0.2)	(1.9)
			16	.822	.850	.046		.047	.591
				(100.0)	(94.9)	(0.8)	(0.0)	(0.8)	(2.1)
			32	.694	.733	.023		.023	.423
				(100.0)	(93.0)	(1.4)	(0.0)	(1.3)	(2.5)
1	200	250	8	.739	.752	.355		.396	.408
				(100.0)	(95.9)	(0.5)	(0.0)	(0.4)	(2.8)
			16	.561	.586	.068		.070	.247
				(100.0)	(93.5)	(2.0)	(0.0)	(1.8)	(3.5)
			32	.361	.396	.041		.044	.137
				(100.0)	(89.0)	(2.1)	(0.0)	(1.9)	(4.0)
1	200	500	8	.851	.860	.354		.396	.518
				(100.0)	(97.6)	(0.2)	(0.0)	(0.2)	(1.7)
			16	.733	.758	.068		.070	.342
				(100.0)	(95.2)	(1.1)	(0.0)	(1.0)	(2.3)
			32	.562	.599	.042		.044	.204
				(100.0)	(92.5)	(1.4)	(0.0)	(1.3)	(2.9)

Table 5.3
Experimental Results of Parallel Algorithms Compared with MINPACK

Efficiencies Compared to MINPACK (% time spent in routine)									
Prob	n	m	p	Total	QR	L-M	R-S	Tri-S	J-Appr
2	100	250	8	.811 (100.0)	.722 (30.9)	.700 (30.8)	.752 (27.7)	.203 (2.8)	.962 (37.1)
			16	.648 (100.0)	.520 (34.3)	.498 (34.6)	.638 (26.1)	.058 (7.9)	.986 (28.9)
			32	.457 (100.0)	.349 (36.0)	.310 (39.2)	.488 (24.1)	.023 (13.8)	.956 (21.0)
2	100	500	8	.873 (100.0)	.833 (37.9)	.657 (11.3)	.740 (9.7)	.201 (1.1)	.972 (48.2)
			16	.780 (100.0)	.686 (41.1)	.477 (14.0)	.631 (10.2)	.058 (3.3)	.983 (42.6)
			32	.623 (100.0)	.531 (42.4)	.288 (18.5)	.481 (10.7)	.023 (6.5)	.956 (35.0)
2	100	1000	8	.930 (100.0)	.902 (41.5)	.616 (2.1)	.733 (1.7)	.202 (0.2)	.979 (54.5)
			16	.867 (100.0)	.797 (43.8)	.387 (3.1)	.625 (1.8)	.058 (0.7)	.998 (49.8)
			32	.795 (100.0)	.695 (46.0)	.284 (3.8)	.487 (2.1)	.023 (1.6)	.956 (47.7)
2	200	250	8	.841 (100.0)	.711 (24.7)	.812 (39.7)	.825 (38.5)	.394 (1.1)	.963 (35.0)
			16	.739 (100.0)	.527 (29.3)	.725 (39.1)	.788 (35.4)	.111 (3.4)	.962 (30.8)
			32	.580 (100.0)	.339 (35.7)	.571 (38.9)	.688 (31.8)	.044 (6.7)	.961 (24.2)
2	200	500	8	.900 (100.0)	.849 (31.6)	.816 (22.6)	.828 (21.9)	.395 (0.6)	.973 (45.2)
			16	.822 (100.0)	.739 (33.2)	.711 (23.7)	.780 (21.2)	.111 (2.1)	.957 (42.0)
			32	.701 (100.0)	.576 (36.4)	.550 (26.1)	.679 (20.8)	.044 (4.5)	.957 (35.8)

TABLE 5.4

Experimental Results of Parallel Algorithms Compared with MINPACK

Efficiencies Compared to MINPACK (% time spent in routine)									
Prob	n	m	p	Total	QR	L-M	R-S	Tri-S	J-Appr
3	100	250	8	.742	.758	.182		.202	.527
				(100.0)	(94.9)	(0.8)	(0.0)	(0.7)	(3.2)
			16	.545	.586	.046		.047	.345
				(100.0)	(90.2)	(2.5)	(0.0)	(2.2)	(3.6)
			32	.344	.391	.023		.023	.207
				(100.0)	(85.3)	(3.1)	(0.0)	(2.8)	(3.8)
3	100	500	8	.847	.862	.627	.751	.198	.641
				(100.0)	(93.9)	(3.2)	(2.5)	(0.6)	(2.3)
			16	.710	.747	.388	.641	.057	.467
				(100.0)	(90.8)	(4.4)	(2.5)	(1.7)	(2.7)
			32	.518	.577	.216	.492	.023	.302
				(100.0)	(85.9)	(5.7)	(2.3)	(3.0)	(3.0)
3	100	1000	8	.911	.920	.633	.754	.202	.744
				(100.0)	(96.1)	(1.7)	(1.3)	(0.3)	(1.9)
			16	.828	.855	.389	.642	.058	.600
				(100.0)	(94.0)	(2.5)	(1.4)	(0.9)	(2.1)
			32	.686	.738	.216	.491	.023	.433
				(100.0)	(90.3)	(3.7)	(1.5)	(1.9)	(2.5)
3	200	250	8	.745	.752	.784	.817	.396	.453
				(100.0)	(86.6)	(9.9)	(9.2)	(0.6)	(2.6)
			16	.577	.586	.652	.789	.111	.279
				(100.0)	(86.0)	(9.2)	(7.3)	(1.7)	(3.3)
			32	.383	.395	.459	.686	.044	.156
				(100.0)	(84.7)	(8.7)	(5.6)	(2.8)	(3.9)
3	200	500	8	.853	.862	.354		.396	.549
				(100.0)	(97.5)	(0.3)	(0.0)	(0.2)	(1.8)
			16	.738	.760	.068		.070	.367
				(100.0)	(95.6)	(1.1)	(0.0)	(1.1)	(2.3)
			32	.571	.601	.042		.044	.222
				(100.0)	(93.6)	(1.4)	(0.0)	(1.3)	(3.0)

Table 5.5
Experimental Results of Parallel Algorithms Compared with MINPACK

Efficiencies Compared to MINPACK (% time spent in routine)									
Prob	n	m	p	Total	QR	L-M	R-S	Tri-S	J-Appr
4	100	250	8	.905 (100.0)	.745 (10.8)	.708 (18.0)	.759 (16.3)	.202 (1.4)	.980 (69.2)
			16	.814 (100.0)	.570 (12.7)	.517 (22.1)	.647 (17.2)	.058 (4.5)	.970 (62.9)
			32	.677 (100.0)	.371 (16.2)	.330 (28.8)	.492 (18.8)	.023 (9.3)	.975 (52.1)
4	100	500	8	.946 (100.0)	.814 (11.9)	.704 (9.5)	.756 (8.6)	.201 (0.8)	.996 (76.4)
			16	.885 (100.0)	.747 (12.2)	.514 (12.2)	.644 (9.5)	.058 (2.5)	.971 (73.3)
			32	.796 (100.0)	.558 (14.6)	.329 (17.1)	.490 (11.2)	.023 (5.6)	.978 (65.5)
4	100	1000	8	.973 (100.0)	.866 (12.5)	.703 (4.4)	.755 (4.0)	.202 (0.4)	0.994 (80.8)
			16	.941 (100.0)	.850 (12.3)	.511 (5.9)	.644 (4.5)	.058 (1.2)	.987 (79.5)
			32	.879 (100.0)	.703 (13.9)	.326 (8.6)	.491 (5.5)	.023 (2.8)	.979 (74.9)
4	200	250	8	.902 (100.0)	.682 (8.1)	.819 (33.2)	.831 (32.3)	.395 (0.8)	.980 (57.8)
			16	.850 (100.0)	.525 (10.0)	.740 (34.6)	.797 (31.7)	.111 (2.7)	.979 (54.5)
			32	.743 (100.0)	.350 (13.0)	.586 (38.2)	.693 (31.9)	.044 (6.0)	.978 (47.6)
4	200	500	8	.945 (100.0)	.805 (10.1)	.821 (17.8)	.833 (17.3)	.393 (0.4)	.996 (71.0)
			16	.904 (100.0)	.717 (10.9)	.739 (19.0)	.797 (17.3)	.111 (1.5)	.979 (69.1)
			32	.837 (100.0)	.572 (12.6)	.585 (22.2)	.694 (18.5)	.044 (3.5)	.978 (64.0)

REFERENCES

[B88] C. Bischof, *QR factorization algorithms for coarse–grained distributed systems,* Tech. Rep. 88-939, Computer Science Department, Cornell University, 1988.

[BSS88] R. Byrd, R. Schnabel, and G. Shultz, *Parallel quasi-Newton methods for unconstrained optimization,* Tech. Rep. CU-CS-396-88, Department of Computer Science, University of Colorado at Boulder, 1988.

[C86] R. M. Chamberlain, *An algorithm for LU factorization with partial pivoting on the hypercube,* Tech. Rep. CCS 86/11, Dept. of Science and Technology, Chr. Michelsen Institute, Bergen, Norway, 1986.

[CP86] R. M. Chamberlain and M. J. D. Powell, *QR factorisation for linear least-squares problems on the hypercube,* Tech. Rep. CCS 86/10, Dept. of Science and Technology, Chr. Michelsen Institute, Bergen, Norway, 1986.

[CG88] E. Chu and A. George, *QR factorization of a dense matrix on a hypercube multiprocessor,* Tech. Rep. ORNL/TM-10691, Mathematical Sciences Section, Oak Ridge National Laboratory, 1988.

[C84] T. F. Coleman, *Large Sparse Numerical Optimization,* Lecture Notes in Computer Science, **165**, G. Goos and J. Hartmanis, eds., Springer-Verlag, New York, 1984.

[CL87] T. F. Coleman and G. Li, *Solving systems of nonlinear equations on a message-passing multiprocessor,* SIAM J. Sci. Stat. Comput., 11, 1116-1135, 1990.

[CP89] T. F. Coleman and P. E. Plassmann, *Solution of nonlinear least squares problems on a multiprocessor* in Parallel Computing 1988, G.A. van Zee and J.G.G. van de Vorst (eds), Lecture Notes in Computer Science, Vol. 384, Springer Verlag, pp. 44-60, 1989.

[HR88] M. T. Heath and C. H. Romine, *Parallel solution of triangular systems on distributed memory multiprocessors,* SIAM J. Sci. Stat. Comput., **9** (1988), pp. 558-588.

[JH88] L. Johnsson and C. T. Ho, *Algorithms for matrix transposition on boolean n-cube configured ensemble architectures,* SIAM J. Matrix Analysis, **9** (1988), pp. 419-454.

[L44] K. Levenberg, *A method for the solution of certain non-linear problems in least squares,* Quart. Appl. Math., **2** (1944), pp. 164–168.

[LC89] G. Li and T. F. Coleman, *A new method for solving triangular systems on distributed memory message-passing multiprocessors,* SIAM J. Sci. Stat. Comput., **10** (1989), pp. 382-396.

[LC88] G. Li and T. F. Coleman, *A parallel triangular solver for a distributed memory multiprocessor,* SIAM J. Sci. Stat. Comput., **9** (1988), pp. 485-502.

[M63] D. W. Marquardt, *An algorithm for least squares estimation of non-linear parameters,* SIAM J. Appl. Math., **11** (1963), pp. 431–441.

[MVV87] O. M. McBryan and E. F. Van de Velde, *Hypercube algorithms and implementations,* SIAM J. Sci. Stat. Comput., **8** (1987), pp. s227-s287.

[M87] C. Moler. *Matrix computations on distributed memory multiprocessors.* Tech. Rep., Intel Scientific Computers, Beaverton, Oregon, 1987.

[M78] J. J. Moré, *The Levenberg-Marquardt algorithm: implementation and theory,* in Lecture Notes in Mathematics, No. 630-Numerical Analysis, G. Watson, ed., Springer-Verlag, New York, 1978, pp. 105-116.

[MGH80] J. J. Moré, Burton Garbow, and Kenneth Hillstrom, *User guide for MINPACK-1,* Argonne National Laboratory Report ANL-80-74, 1980.

[P90] P. E. Plassmann, *Sparse Jacobian estimation and factorization on a multiprocessor* in Large-Scale Numerical Optimization, T.F. Coleman and Y. Li (eds), pp. 152-179, SIAM, 1990.

[PR87] A. Pothen and P. Raghavan, *Distributed orthogonal factorization: Givens and Householder algorithms,* Department of Computer Science, The Pennsylvania State University, 1987.

[SS85] Y. Saad and M. H. Schultz, *Data communication in hypercubes*, Yale University Research Report DCS/RR-428, 1985.