# Solution of Nonlinear Least Squares Problems on a Multiprocessor

*Thomas F. Coleman*
*Paul E. Plassmann*

Department of Computer Science &
Center for Applied Mathematics
Cornell University
Ithaca, New York 14853

*ABSTRACT*

In this paper we describe algorithms for solving nonlinear least squares problems on a message-passing multiprocessor. We demonstrate new parallel algorithms, including an efficient parallel algorithm for determining the Levenberg-Marquardt parameter and a new row-oriented QR factorization algorithm. Experimental results obtained on an Intel iPSC hypercube are presented and compared with sequential MINPACK code executed on a single processor. These experimental results show that essentially full efficiency is obtained for problems where the column size is sufficiently larger than the number of processors. These algorithms have the advantage of involving only simple data movements and consequently are not constrained to the hypercube architecture.

## 1. Introduction

A common computational problem is the minimization of the function $\psi : \mathbf{R}^n \to \mathbf{R}$ where $\psi$ is the sum of squares of nonlinear functions. That is, $\psi$ can be expressed in terms of a function $F : \mathbf{R}^n \to \mathbf{R}^m$, $m \geq n$, by the equation:

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

where $f_i$ is the $i$-th component of $F$. In this paper we will describe a parallel implementation of the Levenberg-Marquardt algorithm for solving these nonlinear least squares problems. The experimental results presented in this paper were obtained on a hypercube multiprocessor but the algorithms themselves are more general. In fact, all that is required of multiprocessor interconnection topology is support of a ring embedding and means for efficient gather and broadcast operations.

Unlike in the parallel solution of systems of nonlinear equations [CL87], low rank updates to the Jacobian approximation are usually not used in the small residual nonlinear least squares setting [GMW81]. Consequently, the dominant costs in these problems are the approximation of the Jacobian $J(x_k)$, where $x_k$ is the $k$-th iterative approximation to the solution, and the computation of the QR factorization of this approximation to $J(x_k)$.

The development of parallel algorithms for these two aspects of the problem, the approximation and QR factorization of $J$, represents a dilemma when considering how to distribute the elements of the Jacobian onto the processors. This dilemma arises because the approximation of $J$ by finite differences is most naturally approached by a column-oriented method and the QR factorization stage suggests a row-oriented solution. In a column-oriented algorithm for approximating the Jacobian the columns of $J$ are partitioned into sets that assign computation of columns to processors by these sets. Assuming that $F(x_k)$ is globally available, the $j$-th column of $J$ can be approximated by computing $F(x_k+\tau e_j)$, where $e_j$ is the $j$-th column of the identity matrix and $\tau$ a constant, for each column $j$ assigned to a processor. These computations are entirely local and if $n/p \gg 1$, where $p$ is the number of processors, then the computational load can be well-balanced by an even distribution of work to the processors. On the other hand, because we have that $m \geq n$, we expect a row-oriented approach, in which the rows of $J$ are assigned to processors, to outperform a column-oriented method in the QR factorization stage. Using a row-oriented method in this case results in an algorithm whose efficiency depends on the ratio $m/p$ rather than $n/p$.

We have chosen to pursue a row-oriented algorithm because experience has shown that computational costs involved in the QR factorization stage often dominate the Jacobian approximation stage. In addition, it is often the case that evaluation of the function $F$ is *separable*. That is, if $I_i$ is the set of row indices assigned to processor $i$ then the evaluation of $F(x)$ can be effectively broken up into blocks, $F_{I_i}(x) = \{ f_j(x) \mid j \in I_i \}$, where evaluation of each block is sufficiently independent to allow for parallel evaluation.

The distribution of the rows of the Jacobian onto the processors determines the basic communication structure of the algorithms used in an implementation. We have chosen to use a ring embedding (e.g. a Gray code ordering of the nodes on a hypercube) and wrap the rows of the Jacobian onto this ring. Specifically, if the processors on the ring are numbered

0,1,2,...,p-1 then row $k$ of the Jacobian would be assigned to processor $(k-1) mod(p)$. Note, however, that components of $F$ may be reordered to facilitate separability. This observation is especially applicable in the sparse setting where grouping rows with similar nonzero structure on the same processor would help ensure separable evaluation. Such an assignment would also make sense for a sparse QR factorization since much progress in the triangular reduction of the system could be made by locally applied Givens rotations.

The remainder of this paper is organized as follows. In section 2 we briefly review the relevant aspects of the Levenberg-Marquardt algorithm necessary to explain details of our algorithms. In section 3 we consider the problem of approximating the Jacobian and in section 4 we present a new row-oriented parallel QR factorization. Section 5 describes our parallel algorithm for determining the Levenberg-Marquardt parameter. Finally, we present experimental results and conclusions in section 6.

## 2. Basics of the Levenberg-Marquardt Algorithm

In this section we consider the essential aspects of the Levenberg-Marquardt algorithm relevant to a parallel perspective. For a more detailed description of the algorithm we refer the interested reader to the excellent article by Moré [M78]. As stated previously, the nonlinear least squares problem is to minimize $\psi(x)$ as given in equation (1.1). Assuming that each $f_i \in C^2$, then the gradient and Hessian of $\psi$ are given by the expressions:

$$\nabla\psi = J^T F \tag{2.1}$$

and

$$\nabla^2\psi = J^T J + \sum_{i=1}^{m} f_i \, \nabla^2 f_i \quad , \tag{2.2}$$

where $J$ is the Jacobian of $F$. When one expects the value of the function to be small at the solution† the Hessian can be approximated by $J^T J$ in a neighborhood of the solution. This approximation, which ignores the second term in the expression for the Hessian, represents a significant computational savings.

† known as "small residual" problems [MGW81]

Based on a quadratic model to the function $\psi$ using the above Hessian approximation the Levenberg-Marquardt algorithm [M77,C84] solves a sequence of trust region problems of the form

$$\min \; \{ \; \| Jp + F \|_2 \quad s.t. \quad \| p \|_2 \le \Delta \; \} \quad . \tag{2.3}$$

A solution to this problem is obtained in two steps. First, the Gauss-Newton step, $p_{GN}$ is determined by solving the problem

$$\min \; \{ \; \| Jp + F \|_2 \; \} \quad , \tag{2.4}$$

by computing a QR factorization of $J$ and solving the upper triangular least squares system

$$\begin{bmatrix} R \\ 0 \end{bmatrix} p \cong Q^T F \quad . \tag{2.5}$$

If $\| p_{GN} \| \le \Delta$, then the step is accepted‡, otherwise we determine the Lagrange multiplier $\lambda_*$ and vector $p_*$, $\| p_* \| = \Delta$, such that

$$(J^T J + \lambda_* I) p_* = -J^T F \quad . \tag{2.6}$$

Using the previously computed QR factorization of $J$ and given a value of $\lambda$, the solution of the least squares system

$$\begin{bmatrix} R \\ 0 \\ \lambda^{1/2} I \end{bmatrix} p(\lambda) \cong \begin{bmatrix} Q^T F \\ 0 \end{bmatrix} \tag{2.7}$$

yields the value of the function $\phi(\lambda) = \| p(\lambda) \| - \Delta$. Since $\phi$ is monotonically decreasing on $[0, \infty)$ from the positive value of $\| p_{GN} \| - \Delta$ at $\lambda = 0$ to the negative number $-\Delta$ as $\lambda \to \infty$, an appropriate implementation of Newtons method is guaranteed to find the zero of $\phi$ at $\lambda_*$, the Lagrange multiplier [M78].

From this discussion we note that there are three main computational tasks that need to be addressed in a parallel implementation of the Levenberg-Marquardt algorithm. These features are: (1) the approximation of the Jacobian $J(x)$, (2) the QR factorization of $J(x)$ necessary to solve equation (2.5), and (3) computation of the Levenberg-Marquardt parameter that solves equation (2.6). In the following sections we present our implementation of the first task and new parallel algorithms for the last two problems.

‡ contingent, of course, upon sufficient decrease in $\psi$

## 3. Parallel Approximation of the Jacobian

The approach used in the parallel approximation of the Jacobian by forward differences depends on two criteria: (1) whether the function $F(x)$ is sufficiently separable, and (2) if the function is not separable, whether evaluation of the function is computationally expensive. These criteria are somewhat subjective; whether they apply is a symptom of the specific problem considered. In this section we describe parallel Jacobian approximation schemes which deal with the cases delineated by these criteria.

As before, let $I_i$ be the set of row indices assigned to processor $i$ and let $J_{I_i}^T(x)$ be this set of the rows of the Jacobian evaluated at the point $x$. When evaluation of the function is separable, then the Jacobian can be evaluated in parallel by having each processor compute its components of the $j$-th column according to the formula

$$J_{I_i}^T(x)\, e_j \cong \frac{F_{I_i}(x+\tau e_j) - F_{I_i}(x)}{\tau} \quad . \tag{3.1}$$

It is often the case that 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. In this approach a set of column indices, $K_i$, is assigned to each processor $i$. For each $k \in K_i$ the $k$-th column of $J$ is approximated at processor $i$ by the usual forward differences formula,

$$J(x)\, e_k \cong \frac{F(x+\tau e_k) - F(x)}{\tau} \quad , \tag{3.2}$$

where in this case the function $F$ can be thought of as a "black box." Assuming that the computation required to evaluate $F(x)$ is independent of $x$, the computational load can be balanced by making the index sets $K_i$ as close to the same size as possible. Using a wrap mapping will keep the cardinality of these sets within one, therefore any disparity in workload becomes relatively better as $n/p$ increases.

The resulting problem is how to get this column-oriented distribution of the data converted to a row-oriented distribution for the QR factorization stage. Fortunately, there exist efficient algorithms for transposing a matrix on the hypercube [MVV87]. Of course, this whole problem can be avoided by using column-oriented algorithms in the first place. Experimentally, we did not take such an approach, but good column-oriented QR factorization algorithms exist [M87]. In addition, in section 5 we describe an efficient column-oriented version of the Levenberg-Marquardt algorithms.

A more subtle problem occurs when the evaluation of $F(x)$ is not separable and the this evaluation is computationally expensive relative to the QR factorization. Suppose a step $p_k$ is to be considered at the $k$-th iteration of the algorithm then $F(x_k+p_k)$ 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, then the Jacobian at $x_k+p_k$ 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 for cases of difficult functions.

## 4. A New Parallel Row-Oriented Householder QR Algorithm

The efficiency of the parallel QR factorization used to solve equation (2.4) 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. As we will see in Section 6 the QR factorization represents the dominant computational cost for the test problems we considered. In this section we present a new parallel row-oriented Householder QR factorization that was found to be more efficient than previous hybrid (Householder/Givens) factorization algorithms. In addition, this algorithm has the advantage that it produces the same Householder vectors that would be produced by a standard sequential Householder QR algorithm (unlike the hybrid scheme) which can be 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 determine matrix rank.

Most of the research on QR algorithms for the hypercube has been directed toward column-oriented methods, however two row-oriented algorithms have been considered previously [CP86,PR87]. These two algorithms are very similar in that to reduce each column of the matrix first a reduction involving only data local to each processor is performed and then this stage is followed by a global reduction requiring communication between the processors. The reduction of rows local to a processor results in one row per processor with a nonzero in the column being reduced to upper triangular form. The advantage of this approach is that all these reductions and matrix updates will be local to the processors and with the wrap mapping of rows the computational load will be well-balanced. Following this local stage is a global stage where a minimum 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 because the full Householder vector is calculated and the intermediate Givens reductions are avoided. The difficulty is obtaining the same communication complexity as the hybrid approach. The trick is to notice that computation of the Householder vector and the rank one update to the matrix can be combined to half the number of messages that seem to be required at first inspection.

To illustrate the algorithm 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 and we need only consider the $(m-j+1)\times(n-j+1)$ lower right submatrix of $A$, which we will denote by $A^{(j)}$, with columns $a_k^{(j)}$, $k = j,..., n$. The Householder transformation, $P^{(j)}$, to reduce the first column of $A^{(j)}$, $a_j^{(j)}$, is given by

$$P^{(j)} = [ I - 2 \frac{v_{(j)} v_{(j)}^T}{v_{(j)}^T v_{(j)}} ] \quad , \tag{4.1}$$

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 $A^{(j)}$ given by:

$$a_k^{(j+1)} = a_k^{(j)} - \frac{2}{v_{(j)}^T v_{(j)}} v_{(j)}^T a_k^{(j)} \, v_{(j)} \tag{4.2}$$

$$= a_k^{(j)} - \alpha_k^{(j)} \, v_{(j)} \quad , \tag{4.3}$$

with $\alpha_k^{(j)}$ defined as shown. Suppose row $j$ is assigned to a processor that we will designate *leader*. Note that $v_{(j)}$ agrees with $a^{(j)}$ except in the first component, hence 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 to 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 below.

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

**Algorithm 4.1**: A Parallel Row-Oriented Householder QR Algorithm

Here 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 $\alpha_k^{(j)}$, $k = j+1,...,n$. In Figure 4.1 we show 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. For this plot the number of columns is fixed at 100.

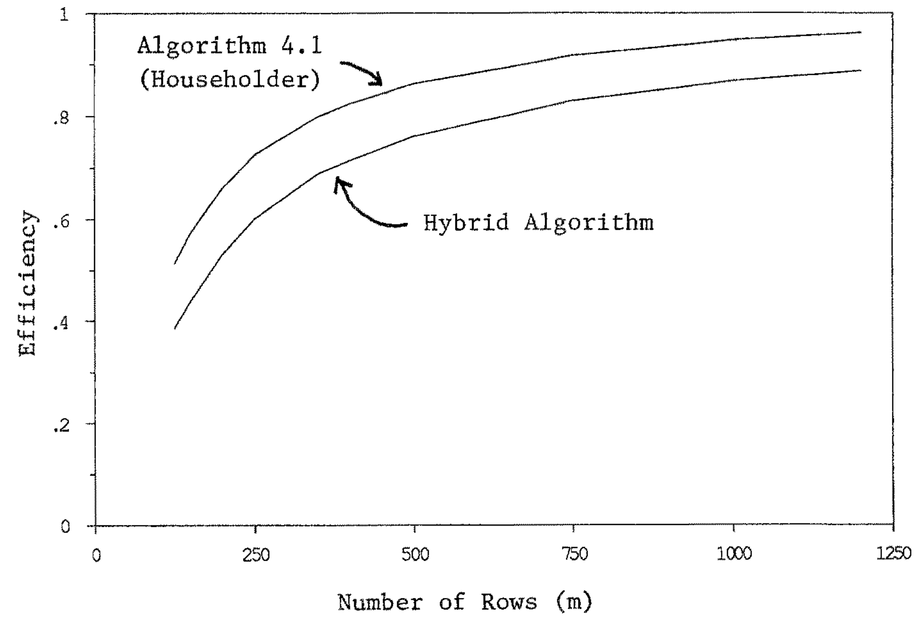

**Figure 4.1:** Efficiency of Algorithm 4.1 and Hybrid on a 16 Node Hypercube (n=100)

| Execution Times (sec) | | | | | Efficiencies | |
|---|---|---|---|---|---|---|
| $n$ | $m$ | Single processor | Hybrid | Householder | Hybrid | Householder |
| 100 | 150 | 109.5 | 15.5 | 11.95 | .44 | .57 |
| 100 | 500 | 434.2 | 35.7 | 31.5 | .76 | .86 |
| 100 | 1000 | 897.8 | 64.7 | 59.3 | .87 | .95 |
| 200 | 350 | 1040.1 | 93.9 | 81.0 | .69 | .80 |

**Table 4.1:** Execution Times and Efficiencies of Algorithm 4.1 on a 16 Node Hypercube

In Table 4.1 we show some representative execution times of our implementations of the hybrid algorithm (Hybrid) and Algorithm 4.1 (Householder) as compared to a sequential QR factorization program executed on a single processor. These results were all obtained on a 16 node iPSC hypercube and the comparison was made with the MINPACK QR factorization subroutine QRFAC executed on a single processor of the hypercube.

A subtle point in solving equation (2.5) is that 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 aim 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 4.1 in the following manner. The column norms of the matrix $A$ are initialized at the beginning of the algorithm and are then updated after each stage of the computation to obtain the column norms of $A^{(j)}$. For example, suppose at stage $j$ in the QR factorization the column norms $\|a_k^{(j)}\|_2$, $k=1,...,n$ are known by *leader*. The column of maximum norm, $k_{\max}$, is determined by *leader*, the result is broadcast, and columns $j$ and $k_{\max}$ are interchanged by all processors. After stage $j$ of the algorithm the updated norms can be obtained by the formula:

$$\|a_k^{(j+1)}\|_2 = \|a_k^{(j)}\|_2 \left[1 - \left[\frac{[a_k^{(j+1)}]_j}{\|a_k^{(j)}\|_2}\right]^2\right]^{1/2} , \tag{4.4}$$

for $k=j+1, \cdots, n$, and the results sent to the next processor on the ring for stage $j+1$ of the QR algorithm. Note that numerical cancellation can be a problem in computing these norm updates. However, circumstances that would result in this problem can be monitored for and in such cases the suspect column norm can be recomputed. In our implementation this 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 effect the efficiency of the algorithm.

One last concern for numerical stability might be the possibility of overflow from the way the $\tilde{\alpha}_k$ are computed in Algorithm 4.1. We note that these partial sums can be scaled by the most recent approximation to the column norms available to all the processors. However, we did not find it necessary to include this scaling in our implementation.

A graph comparing the efficiencies of Algorithm 4.1 with and without pivoting is included as Figure 4.2. The efficiency is again computed by comparing the running times of parallel algorithms to running times of the MINPACK QR subroutine QRFAC on a single processor. These results were obtained on a 16 node hypercube with the number columns fixed at 100. In table 4.2 we include some representative times from these experiments.

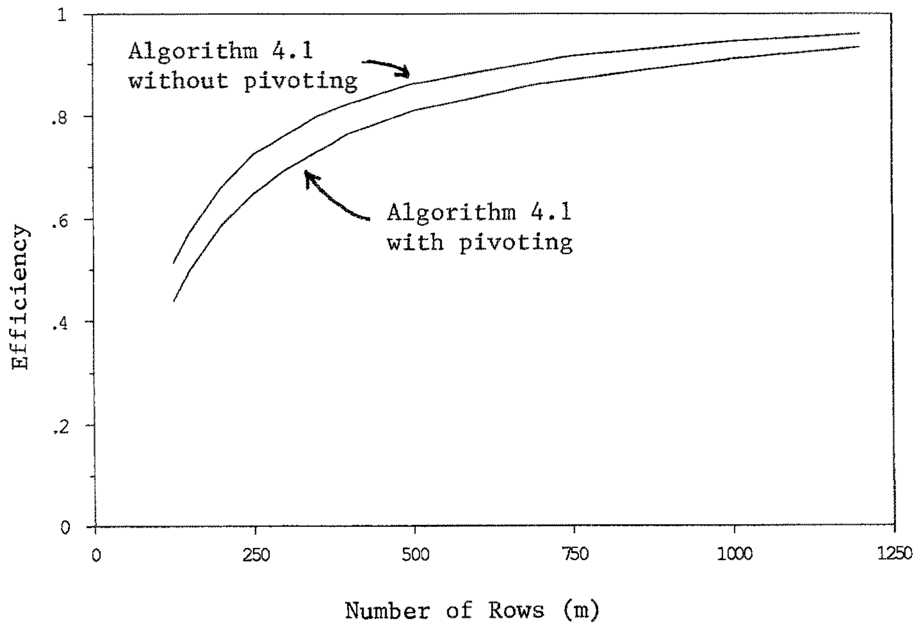

**Figure 4.2:** Efficiency of Algorithm 4.1 With and Without Column Pivoting

| Execution Times (sec) | | | | Efficiencies |
|---|---|---|---|---|
| *n* | *m* | Single processor | Householder | Householder |
| 100 | 150 | 112.6 | 14.2 | .50 |
| 100 | 500 | 439.0 | 34.1 | .80 |
| 100 | 1000 | 906.5 | 62.1 | .91 |

**Table 4.2:** Execution Times and Efficiencies of Algorithm 4.1 with Pivoting

## 5. A Parallel Implementation of the Levenberg-Marquardt Algorithm

To determine the Levenberg-Marquardt parameter Algorithm 5.1 was used to reduce equation (2.7) to upper triangular form. This reduction requires a surprising amount of work: $n(n+1)/2$ 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 5.1 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 $\lambda^{1/2} I$ in equation (2.7). Remember that the rows of $R$ and $S$ are wrapped onto an embedded ring of processors as described in Section 1.

```
Functions: next {returns number of next processor in the ring},
           prev {returns number of previous processor in the ring}
Index Set: $I_i$ {set of rows assigned to processor $i$}
**Proc** ($i$) : {program for processor $i$}
    For $j = 0, n-1$ **do**
        For $k \in I_j$ **do**
            **If** ($j \neq 0$) **receive** row $S_{k-j}^T$ from processor $prev(i)$
            Compute Givens rotation to zero the bottom of the vector $(R_{k,k}, S_{k-j,k})^T$
            Update rows $R_k^T$ and $S_{k-j}^T$ with above Givens rotation
            **Send** row $S_{k-j}^T$ to processor $next(i)$
            **If** ($k=j+1$) Delete {$k$} from $I_i$
        **enddo**
    **enddo**
```

**Algorithm 5.1**: A Parallel Row-Oriented R-S Reduction

Algorithm 5.1 proceeds in $n$ stages which have been indexed by $j=0,\ldots,n-1$ in the description. At stage $j$ of algorithm 5.1 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 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. Note, that 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 number of columns are shown in Figure 5.1.

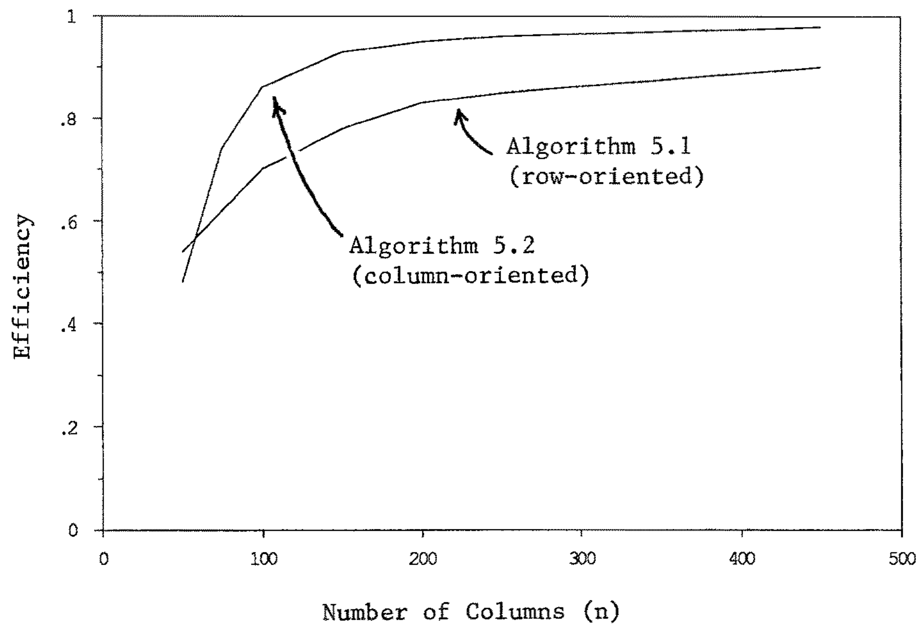

**Figure 5.1:** Efficiencies of Algorithms 5.1 and 5.2 on a 16 Node Hypercube

A column-oriented approach is also possible and is presented as Algorithm 5.2. From Figure 5.1 it is apparent that this version is superior experimentally. Even though the algorithm is a bit more complicated, the total number of messages that have to be sent is the same as Algorithm 5.1 and the total number of values that have to communicated is actually less. For an average step $j$ in Algorithm 5.2 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 the same number of communication start-ups as appear in Algorithm 5.2. The message start-up cost, measured in "flops," for the Intel iPSC hypercube is very expensive and is normally the dominant factor in the communication cost of an algorithm. However, for these two algorithms the average message lengths are extremely different, hence one sees a difference in efficiency between these two algorithms experimentally.

```
Index Set: $K_i$ {set of columns assigned to processor $i$}
Proc ($i$) : {program for processor $i$}
    For $j = 0, n-1$ do
        If ($j \neq 0$) Receive Givens vector $g$ from processor $prev(i)$
        For $k \in K_j$ do
            For $l = min(j, p-1),..,1$ do
                Update rows $R_{k-j+l}^T$ and $S_{k-j}^T$ with Givens rotation $g_{k-j+l}$
            enddo
            Compute Givens rotation to zero the bottom of the vector $(R_{k,k}, S_{k-j,k})^T$
            Update rows $R_k^T$ and $S_{k-j}^T$ 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 5.2:** A Parallel Column-Oriented R-S Reduction

For Algorithm 5.2 the columns, as opposed to the rows, of $R$ and $S$ are wrapped onto the ring of processors. Rather than communicating rows of $S$ between neighboring processors, the Givens rotations are stored in vectors $g$ which 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 that 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 oldest to newest. Also, note by row $R_k^T$ we mean the nonzero components of row $R_k^T$ that are local to processor $i$. These components are given by the index set $K_i$. Note that the difference in the average length of the messages sent between Algorithms 5.1 and 5.2 translates into a much more efficient algorithm for the column-oriented approach. It is possible to decrease the length of the messages sent in the row-oriented method by postponing the application of the Givens rotations. Unfortunately, this algorithm is very complicated and was not implemented.

The reduction of equation (2.7) to upper triangular form is the major task to be rendered parallel in an algorithm for determining the Levenberg-Marquardt parameter $\lambda_*$ 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 equation (2.7) there are two associated triangle solves that are used to bracket the solution $\lambda_*$ [M78]. Experimentally we used the triangle solve algorithms developed by Li and Coleman [LC86], but it should be noted that the efficiencies of these algorithms are not nearly as good as those of Algorithms 5.1 and 5.2. This difference is what accounts for the discrepancies between the efficiencies shown in Figure 5.1 and the efficiences reported in the next section for solving for the Levenberg-Marquardt parameter. This effect is apparent even though there is an $O(n)$ difference between the amount of work required for Algorithm 5.1 and the corresponding triangular solves. The importance of efficient parallel triangle solvers has also been observed in the solution of systems of nonlinear equations [CL87].

## 6. Experimental Results and Conclusions

These algorithms were implemented on a 16-node Intel iPSC hypercube in RM/Fortran and run under version R3.1.1 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 16 times the running times of the algorithms described above on the hypercube. The comparison is fair as both programs generate the same sequence of iterates and consequently do the same number of Jacobian approximations (J Appr), QR factorizations (QR Fact), and calculations of the Levenberg-Marquardt parameter (L-M). Results from this comparison for representative test problems of moderate size are summerized below in Table 6.1.

Shown in Table 6.1 are the efficiences and the fraction of the total running time spent in each of the three previously described sections of the program. For these problems the total time of computation is dominated by the QR factorization, hence the implementation tends to be more efficient as $m$, the number of rows, increases. We feel that the relative time spent in function evaluation, and consequently approximation of the Jacobian, is too short and not representative of real world problems. For these problems the efficiency of the Jacobian approximation would play a more dominant role in the performance of the program. Then the sequential nature of the function evaluation while testing for adequate decrease in the objective function at the candidate iterate would become more of a problem. However, as discussed earlier, this problem can be alleviated by simultaneous function evaluation and computation of the

Jacobian at the new point [CL87].

| Efficiencies Compared to MINPACK | | | | | | | % Time Spent in Routine | | |
|---|---|---|---|---|---|---|---|---|---|
| Prob | $n$ | $m$ | Total | QR Fact | L-M | J Appr | QR Fact | L-M | J Appr |
| 1 | 100 | 200 | .86 | .91 | .51 | .52 | .85 | .10 | .02 |
| 2 | 50 | 200 | .65 | .78 | .25 | .35 | .75 | .19 | .02 |
| 2 | 100 | 200 | .72 | .80 | .46 | .37 | .72 | .26 | .01 |
| 3 | 100 | 200 | .77 | .86 | .57 | .58 | .92 | .03 | .03 |

**Table 6.1:** Experimental Results of Parallel Algorithms Compared with MINPACK

In sum, we have observed good efficiencies when solving moderately sized nonlinear least squares problems on the Intel hypercube. 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. In addition to comparisons with larger dense problems one can consider solving large sparse problems that have special structure. For example, we note that a simple generalization of Algorithm 5.1 would work well if the Jacobian were banded and that the row-oriented approach to the QR factorization would be efficient in solving problems with block structure.

**Acknowledgements**

The work that produced 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.

## References

[BSS88] R. Byrd, R. Schnabel, & G. Shultz. *Parallel Quasi-Newton Methods for Unconstrained Optimization.* Manuscript, Department of Computer Science, University of Colorado at Boulder, 1988.

[CP86] Richard Chamberlain and M. J. D. Powell. *QR Factorisation for Linear Least Squares Problems on the Hypercube.* Chr. Michelsen Institute, 1986.

[C84] Thomas F. Coleman. *Large Sparse Numerical Optimization.* Springer-Verlag, 1984.

[CL87] Thomas F. Coleman and Guangye Li. *Solving Systems of Nonlinear Equations on a Message-Passing Multiprocessor.* Tech. Rep. 87-887, Computer Science Department, Cornell University, 1987.

[LC86] Guangye Li and Thomas F. Coleman. *A Parallel Triangular Solver for a Distributed Memory Multiprocessor.* Tech. Rep. CS-86-787, Computer Science Department, Cornell University, 1986.

[M87] C. Moler. *Matrix Computations on Distributed Memory Multiprocessors.* Tech. Rep., Intel Scientific Computers, 1987.

[MVV87] O. M. McBryan and E. F. Van de Velde. *Hypercube Algorithms and Implementations.* SIAM J. Sci. Comput., 1987.

[GMW81] Phillip Gill, Walter Murray, and Margaret Wright. *Practical Optimization.* Academic Press, 1981.

[M77] Jorge J. Moré. *The Levenberg-Marquardt Algorithm: Implementation and Theory.* Springer-Verlag, 1977.

[MGH80] Jorge J. Moré, Burton Garbow, and Kenneth Hillstrom. *User Guide for MINPACK-1.* Argonne National Laboratory, 1980.

[PR87] Alex Pothen and Padma Raghavan. *Distributed Orthogonal Factorization: Givens and Householder Algorithms.* Pennsylvania State University, 1987.