SIAM J. SCI. STAT. COMPUT.
Vol. 11, No. 6, pp. 1116–1135, November 1990

006

SOLVING SYSTEMS OF NONLINEAR EQUATIONS ON A MESSAGE-PASSING MULTIPROCESSOR*

THOMAS F. COLEMAN† AND GUANGYE LI‡

Abstract. Parallel algorithms for the solution of dense systems of nonlinear equations on a message-passing multiprocessor computer are developed. Specifically, a distributed finite-difference Newton method, a multiple secant method, and a rank-1 secant method are proposed. Experimental results, obtained on an Intel hypercube, indicate that these methods exhibit good parallelism.

Key words. systems of nonlinear equations, hypercube computer, message-passing multiprocessor, secant method, finite-difference Newton method, parallel algorithms

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

1. Introduction. In this paper we investigate parallel algorithms, tailored to the hypercube multiprocessor context, for the solution of systems of nonlinear equations

$$\text{solve } F(x) = 0 \tag{1}$$

where $F: R^n \to R^n$. Component i of F is denoted by f_i. We assume that F is differentiable; let $J(x)$ denote the Jacobian matrix evaluated at point x.

Our implementations are specific to a hypercube multiprocessor; however, the algorithmic ideas are applicable more generally. In particular, the parallel algorithms presented here can be tailored to any multiprocessor computer provided that the communication topology allows for efficient "fan-in" and "fan-out" operations and the processors themselves have significant local memory. Furthermore, some of our proposed algorithms—multisecant update, triangular solve—are most meaningful when a "ring" communication pattern is used; hence, the topology of the multiprocessor should allow for a ring embedding. Finally, we remark that we always assume that the dimension of the problem n is greater than the number of processors p; indeed, the algorithms uniformly become more efficient as n/p increases.

Our ultimate interest is in large sparse problems; however, in this paper we restrict our attention to the case in which the Jacobian matrix is assumed to be dense.

In a nutshell, this paper represents our attempt to parallelize the popular globalized Newton-like approaches to (1), such as secant and finite-difference Newton methods with a dogleg step or linesearch procedure. Consequently, the algorithms under consideration actually solve the structured minimization problem

$$\text{minimize } \{f(x): f(x) = \tfrac{1}{2}F(x)^T F(x)\}. \tag{2}$$

Obviously a solution to (1) is also a solution to (2); unfortunately, the converse is not always true. Nevertheless, such algorithms often are used successfully to obtain solutions to (1).

* Received by the editors January 1, 1988; accepted for publication (in revised form) October 17, 1989.

† Computer Science Department and Center for Applied Mathematics, Cornell University, Ithaca, New York, 14853. This research was 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.

‡ Computer Center, Jilin University, People's Republic of China. This research was partially supported by the U.S. Army Research Office through the Mathematical Sciences Institute, Cornell University, Ithaca, New York 14853.

We divide the world into two classes of functions: functions F that are most conveniently evaluated as a single entity (i.e., a single subroutine evaluates the entire vector function F, sequentially), and functions F that can be evaluated in a distributed, parallel fashion. In this paper, to be concrete, we restrict the latter category to functions that conveniently separate into component sequential subroutines for each f_i, $i = 1:n$. We call such functions *row-separable.*[1]

If F is to be evaluated as a single entity, then our approach is to distribute copies of the F-evaluation subroutine to all the processors. We then assume that any node (processor) can evaluate $F(x)$, given x, with no other communication necessary. Note that the usual (rank-1) secant method cannot be parallelized in any obvious way since the evaluation of $F(x)$ is not distributed. If the evaluation of F is cheap relative to the other computations (e.g., matrix updating, triangular solves, ...) then this poses no problem—let one node evaluate F while the others remain idle; however, if F is relatively expensive, then it is not clear how to effectively parallelize the rank-1 secant method. For this reason we have developed the *multisecant method* in which each processor evaluates F at a different point (or perhaps several different points). The result is a rank-q update, where q is a multiple of the number of processors available. This approach is discussed in § 5. In the extreme case, when each node is evaluating F at many different points, the multiple secant method resembles a parallel finite-difference Newton method. The latter approach is explored in § 4.

When F is row-separable the evaluation of $F(x)$ can be done in parallel. This allows for an efficient parallel version of the (globalized) rank-1 secant method. The crucial problem here is the design of the effective parallel QR-updating scheme. We discuss this in § 3.

Next we briefly describe the salient features of a hypercube computer. See Wiley [15], for example, for more information.

A message-passing multiprocessor consists of several independent processors connected by communication links. Each processor has significant local memory. (For example, the Intel iPSC with extra memory boards has approximately four megabytes of available memory, per node.) There is also a host computer, connected to one or several of the nodes (processors), whose purpose is to load programs and data onto the nodes of the cube, as well as collect the answer; we take the view that the host does not participate in intermediate computations.

Each processor supports two message-passing primitives: *send* and *receive.* When a node *sends* an array, it is transported through a sequence of nodes and communication links—the sequence is usually determined by the operating system—until the target node is reached. Upon executing a *receive*, a node checks to see if a new message is in its buffer. If so, the message is read and execution continues; if not, execution is suspended (on the receiving node only) until a message arrives.

A hypercube computer is a particular kind of message-passing multiprocessor. Specifically, the name refers to the topology defined by the communication links. A zero-dimensional hypercube, or 0-cube, is a single processor. To construct a 1-cube (i.e., 2 nodes), join two 0-cubes with a single communication link. In general, construct an m-cube (i.e., 2^m nodes) from two $m-1$ cubes: find a one-to-one correspondence between the nodes in each cube and add a communication link between each pair.

[1] We restrict our attention to row-separable functions to provide specific explicit algorithms, and for purposes of implementation and experimentation. However, the ideas and algorithms developed for row-separable functions are easily adapted to the general situation: i.e., functions that can be evaluated in a distributed, parallel manner.

The hypercube topology allows for a number of interesting properties (e.g., [14]). A particularly important one, for our purposes, is that a spanning tree of depth m can be embedded in an m-cube, *rooted at any node.* This allows for the efficient implementation of a number of global operations. For example, a node can send information to every other node in m "timesteps." This is usually called a *broadcast* or *fan-out* operation. Alternatively, a vector distributed over the cube can be collected onto any single node (the target node) in m "timesteps" using a spanning tree rooted at the target node. This is often called a *fan-in* operation.

Each node has a unique name *myid*, which is a number in the range $[0, p-1]$ where p is the number of processors; each node is aware of its own name. We use two labelings, or assignment of node names. The first is the *natural* ordering, which is an assignment of integers in $[0, p-1]$ such that each neighbor of node i (a neighbor of node i is a node connected to i by a single communication link) differs by a single bit in its binary representation of its name. For example, the neighbors of node 5, in a 4-cube, are nodes 4, 7, 1, and 13. A *ring* ordering is also used. In this case node i is connected by single communication links to node $(i-1)$ mod p and node $(i+1)$ mod p. An m-cube always allows for an embedding of a ring on 2^m nodes.

Experimental results reported in this paper were obtained using the Cornell Theory Center 16-node Intel iPSC hypercube under Xenix 286 release 3.4 of the host operating system and iPSC release 3.0 of the node operating system. The nodes were equipped with extra memory boards yielding approximately four megabytes of available memory per node. All our programs were written in Fortran.

2. The sequential secant algorithm. We begin by summarizing a simplified version of the Minpack [11] sequential secant algorithm. This algorithm, in turn, is based on the work of Powell [12].

Suppose x_c is the current approximation to a solution of (2) and define $F_c \overset{\text{def}}{=} F(x_c)$. Let B_c be the current Jacobian approximation and let $B_c = Q_c R_c$ be the QR-factorization of B_c. Assume B_c to be nonsingular.

A *trial step* s_c is computed by approximately solving the trust region problem

$$\text{minimize } \{\|F_c + B_c s\|_2^2 : \|s\|_2 \leqq \Delta_c\} \tag{3}$$

where Δ_c is the current radius of the trust region. The approximate solution s_c is obtained by further restricting (3): specifically, s_c solves the problem

$$\text{minimize } \{\|F_c B_c s\|_2^2 : \|s\|_2 \leqq \Delta_c,\ s \in P_c\} \tag{4}$$

where P_c is a piecewise linear path defined as follows: First, connect x_c to the Cauchy point, $x_c + s_c^{\text{Cauchy}}$, where

$$s_c^{\text{Cauchy}} \overset{\text{def}}{=} -\frac{\|B_c^T F_c\|_2^2}{\|B_c B_c^T F_c\|_2^2} B_c^T F_c, \tag{5}$$

and then connect $x_c + s_c^{\text{Cauchy}}$ to the Newton point, $x_c + s_c^{\text{Newton}}$, where

$$s_c^{\text{Newton}} \overset{\text{def}}{=} -B_c^{-1} F_c. \tag{6}$$

The computation determining the trial step s_c boils down to the algorithm in Fig. 1. (Note. *newx* is a logical input parameter. If *newx* = *false* then all quantities in step 1 have not changed since the previous call; otherwise, *newx* = *true* and all quantities have changed values.)

The possible correction s_c determined by algorithm *Dogleg* is accepted (i.e., $x_+ \leftarrow x_c + s_c$) only if $\|F(x_c + s_c)\|_2 < \|F(x_c)\|_2$. If s_c is not accepted then Δ_c is reduced and step 2 of algorithm *Dogleg* is repeated.

{1: Compute Cauchy and Newton Steps}
If {*newx*} **then**
{Compute s_c^{Cauchy} (given Q_c, R_c)}
$u \leftarrow Q_c^T F_c$;
$g \leftarrow R_c^T u$;
$w \leftarrow Q_c R_c g$;
$$s_c^{\text{Cauchy}} \leftarrow -\frac{\|g\|_2^2}{\|w\|_2^2} g;$$
{Compute s_c^{Newton}}
Solve $R_c s_c^{\text{Newton}} = -u$;
Endif

{2: Solve (4)}
If $\{\|s_c^{\text{Newton}}\|_2^2 \leqq \Delta_c\}$ **then** $s_c \leftarrow s_c^{\text{Newton}}$

Elseif $\{\|s_c^{\text{Cauchy}}\| \geqq \Delta_c\}$ **then** $s_c \leftarrow \left(\frac{\Delta_c}{\|s_c^{\text{Cauchy}}\|_2}\right) s_c^{\text{Cauchy}}$

Else $s_c \leftarrow s_c^{\text{Cauchy}} + \alpha(s_c^{\text{Newton}} - s_c^{\text{Cauchy}})$
where α is the positive root of the quadratic equation
$\|s_c^{\text{Cauchy}} + \alpha(s_c^{\text{Newton}} - s_c^{\text{Cauchy}})\|_2^2 = \Delta_c^2$
Endif

FIG. 1. *Algorithm Dogleg.*

There must also be a mechanism for increasing Δ_c so that progress is not impeded by unnecessarily small steps. This is accomplished by comparing the predicted reduction to the actual reduction. If this ratio, *ratio*, is sufficiently large then Δ_c is increased.

Besides updating x and Δ, it is necessary to evaluate $F(x_c + s_c)$ and update the QR-factorization of B to reflect the rank-1 secant update (due to Broyden [2]). We will not go into the QR-updating details here; however, orthogonal rotations can be used to stably perform this update using a total of $26n^2$ floating point operations (e.g., [5]). The algorithm in Fig. 2 is a formalization of the ideas expressed above.

The algorithm described in Fig. 2 represents a simplified version of the Minpack implementation. For example, Minpack will refresh B by finite-differences when it appears that convergence is not proceeding rapidly enough. In addition, Minpack will modify R if singularity is detected. Furthermore, the Minpack "ratio test" and subsequent adjustment of Δ is somewhat more complicated. We will not spell out these details in this description since they do not bear significantly on questions of parallelization. Subscript "c" denotes current: e.g., x_c refers to the current point. Subscript "+" denotes the updated (new) quantity: e.g., x_+ is the new value of x, $x_+ \leftarrow x_c + s_c$.

3. Parallel secant method for row-separable functions.

3.1. The algorithm. As mentioned in § 1, a row-separable function F is defined to be one in which it is convenient to have available a separate subroutine to evaluate each $f_i(x)$, $i = 1:n$. Assuming row-separability (see footnote 1), we now develop a parallel *Secant/Dogleg* secant method for (1) based on the sequential secant method described in § 2.

Our general approach is to distribute data and functions around the cube so that the work in the computationally intensive steps in algorithm *Secant/Dogleg* is well distributed; we are averse to redistributing information if it can be avoided. We make no attempt to parallelize steps that involve relatively insignificant computational work.

{0: Initialize}
Choose x_c, evaluate $F(x_c)$, determine $B(x_c)$ by finite differences;
Compute $B(x_c) = Q_c R_c$
newx $\leftarrow$ *true*;

{1: Attempt to find a zero of $F(x)$}
Repeat
Determine s_c by *Algorithm Dogleg*(*newx*);
Evaluate $F(x_c + s_c)$;

{Compute *ratio*}

$$actred \leftarrow 1 - \frac{\|F(x_c + s_c)\|_2^2}{\|F_c\|_2^2};$$

$$prered \leftarrow 1 - \frac{\|F_c + Q_c R_c s_c\|_2^2}{\|F_c\|_2^2};$$

$$ratio \leftarrow \frac{actred}{prered};$$

{Update x}
If $\{ratio \leqq .0001\}$ **then** $x_+ \leftarrow x_c$
Else $x_+ \leftarrow x_c + s_c$ **Endif**

{Update Δ}
If $\{ratio \leqq \frac{1}{4}\}$ **then** $\Delta_c \leftarrow \frac{1}{2}\Delta_c$
Elseif $\{ratio \leqq \frac{3}{4}\}$ **then** $\Delta_+ \leftarrow \Delta_c$
Else $\Delta_+ \leftarrow 2\Delta_c$ **Endif**

{Update *B*, *newx*}
If $\{x_+ \neq x_c\}$ **then**
newx $\leftarrow$ *true*;
Update $Q_c R_c \rightarrow Q_+ R_+$ to reflect the rank-1 change:

$$B_+ \leftarrow B_c + \frac{([F_+ - F_c] - B_c s_c) s_c^T}{s_c^T s_c};$$

Else *newx* $\leftarrow$ *false*
Endif
Until {convergence}

FIG. 2. *Algorithm Secant.*

From our perspective the significant steps in algorithm *Dogleg* are the matrix-vector multiplies and the upper triangular solve in Step 1; Step 2 is relatively insignificant and can be performed on a single node. Beyond the call to *Dogleg*, the significant steps in algorithm *Hybrid* are: the initial finite-difference approximation *B*, the initial QR-factorization, the evaluation of $F(x_c + s_c)$, matrix-vector multiplies in the computation of *prered*, and the update of the QR-factorization.

We distribute F as follows: For $i = 0: p-1$, node i is assigned component subroutine f_j for each $j = i + 1 (\text{mod } p)$, $1 \leqq j \leqq n$. Therefore, to evaluate $F(y)$ each node must just evaluate its resident component functions: the simple node program is illustrated in Fig. 3.

```
k ← myid + 1;
While {k ≦ n} do
    Evaluate f^k(y)
    k ← k + p
Endo
```

FIG. 3. *Algorithm* F-*evaluate* (*node program*).

The initial finite-difference determination of B can be accomplished in parallel using algorithm F-*Evaluate* as a subroutine. However, it turns out to be convenient to have the initial B distributed by columns; therefore, given our distribution of F, some communication is required within the parallel finite-difference method. The basic idea is to use algorithm F-*Evaluate* to parallelize the usual column-oriented finite-difference scheme. This is based, in turn, on the approximation to the jth column of the Jacobian matrix,

$$J(x)e_j \cong \frac{F(x+\tau e_j)-F(x)}{\tau} \tag{7}$$

where e_j is the jth column of the identity matrix and τ is an appropriate positive scalar. The node program for the parallel finite-difference approximation is given in Fig. 4. It is assumed that every node has a copy of x, the current point, and τ, the differencing scalar.

Note that each node determines some of the components of column j; the "Fan-in" step collects column j onto node $(j-1) \bmod p$ where it is stored.

The initial QR-factorization of B can be accomplished by a parallel column-oriented algorithm based on orthogonal Householder transformations. Moler [9] has described the framework for such an algorithm in which the column-distributed matrix B is overwritten with R and the Householder vectors that define Q. However, our rank-1 updates require an explicit representation of Q; therefore, we have modified Moler's Algorithm to produce a *row-distributed* Q-matrix, while overwriting B with the *column-distributed* matrix R. This modification is rather straightforward and we will not describe it here; however, in Table 1 of § 3.2 we do provide results of numerical experiments designed to measure parallel efficiency.

There are numerous matrix-vector multiplies in algorithm *Secant/Dogleg* involving matrices Q, R, Q^T, and R^T: we have built our (straightforward) routines based on the communication utility routines provided by Intel [10]. Design of an efficient parallel triangular solver turns out to be a difficult problem. Nevertheless, there has been significant recent progress (e.g., [6]–[8], [13]); we use the algorithm of Li and Coleman [8] in our implementation.

It remains to consider the QR-updating step. Indeed, our decision to crossthread Q and R—i.e., distribute Q by rows and R by columns—was arrived at with this step in mind. Hence we assume that if $j-1=i \bmod p$, $1 \leqq j \leqq n$, then node i houses row j

```
For j = 1: n do
    {Estimate column j}
    y ← x + τe_j;
    F-Evaluate (y) → w;
    Participate in Fan-in of w → node (j − 1) mod p;
Endo
```

FIG. 4. *Algorithm* J-*evaluate* (*node program*).

of Q and column j of R. Let

$$B_+ = QR + rs^T = Q(R + \bar{r}s^T) \tag{8}$$

where $\bar{r} = Q^T r$. Assume that $\bar{r}$ is stored on a single node, node 0, say; let s^T be distributed, by column, to conform to the distribution of R.

Recall that the usual sequential algorithm [4] introduces zeros in $\bar{r}$ by applying orthogonal rotations to its rows, from bottom to top. Each rotation is applied, in turn, to the two corresponding rows of R and columns of Q. But the distribution we have chosen for Q and R is ideal for parallelization: each node contains a segment of the rows of R (and columns of Q) being rotated. Hence, the work involved in the rotation is well distributed.

Upon completion of the step described above we have $B_+ = \hat{Q}\hat{R}$ where $\hat{Q}$ is orthogonal and $\hat{R}$ is upper-Hessenberg. Next we must reduce $\hat{R}$ to upper triangular form. This is done using orthogonal rotations applied from top to bottom. Rotation G_i is applied to rows i, $i+1$ of $\hat{R}$ as well as columns i, $i+1$ of Q (for $i = 1{:}\ n-1$). Rotation G_i involves computations that can be done concurrently because each node has a segment of rows (columns) i, $i+1$ of $\hat{R}(\hat{Q})$.

Figure 5 provides the detailed algorithm. For each node k, let

$$I(k) = \{1 \leqq i \leqq n\colon i - 1 = k \bmod p\}. \tag{9}$$

Remark. As we have demonstrated, the parallelization of the *Secant/Dogleg* Algorithm is fairly straightforward under a row-separability assumption. The two most

```
{Reduce to Upper Hessenberg form}
For i = n-1: 1(-1) do
    If {myid = 0} then
        Determine G_i; {Givens rotation defined by r̄_{i+1}, r̄_i}
        Apply G_i to r̄;
        Broadcast G_i;
    Endif

    For each k ∈ I(myid) do
        Rotate elements in rows i, i+1 of column k of R, using G_i;
        Rotate elements in columns i, i+1 of row k of Q, using G_i;
    Endo
Endo
If {myid = 0} then broadcast α ≝ r̄_1 Endif

Add α × s to first row of R;

{Reduce R back to upper triangular form}
For i = 1: n-1 do
    If {myid = (i-1) mod p} then
        Determine G_i, based on R_{i+1,i}, R_{i,i}; {Givens rotation}
        Broadcast G_i;
    Endif

    For each k ∈ I(myid) do
        Rotate elements in rows i, i+1 of column k of R, using G_i;
        Rotate elements in columns i, i+1 of row k of Q, using G_i;
    Endo
Endo
```

FIG. 5. *Algorithm* QR-*update* (*node program*).

challenging steps are the triangular solve and QR-update. Indeed, it is possible to avoid these two steps altogether if we recur B^{-1} instead of B. This is quite possible (e.g., [3]) though from a numerical point of view it is probably preferable to update B. There are two major reasons we do not pursue this possibility here. First, one of our goals is to determine if the Minpack Algorithm, which includes updating QR-factors, can be efficiently parallelized. Second, we maintain an eye toward the sparse case in this development: in general, J^{-1} is dense when J is sparse and therefore recurring an approximation to J^{-1} is unreasonable in the large sparse situation.

3.2. Numerical experiments. In Table 1 we present timing results reflecting the performance of the QR-updating Algorithm described in Fig. 5, and, for comparison, the QR-factorization routine (which we have not described in detail here) and the triangular solve. Note that even though both the QR-update and the triangular solve are $O(n^2)$ operations, the efficiency of the rank-1 update is much better than the triangular solve efficiency. This is due to the constant factors involved: i.e., the QR-update requires $26n^2$ arithmetic operations, whereas the triangular solve involves $1n^2$ arithmetic operations. Moreover, the efficiency of the update is not dramatically worse than for the full factorization that, in turn, exhibits close to optimal megaflop rate. The near-maximum efficiency of the QR-factorization is due to the intensive computational work required (recall that the orthogonal matrix Q is being explicitly formed).

In Table 2 we present the running times for our parallel implementation of the secant algorithm described in § 2. Our stopping criterion was $\|F\| \leqq 10^{-8}$; the "update Δ" step was modified to conform with the Minpack code. Problem 15 is the well-known extended Rosenbrock function; the other problems were chosen from the Minpack collection of nonlinear equation problems.

To provide a measure of speedup, in Table 3 we have divided the numbers in Table 2 by the sequential running times of a *modified* Minpack code running on a

TABLE 1
Timing results for the parallel secant update, $p = 16$.

n	QR time	QR mflps	Update time	Update mflps	Solve time	Solve mflps
100	7.2	.458	.995	.261	.70	.028
200	51.3	.52	2.6	.39	1.3	.06
300	166.0	.54	5.6	.46	2.2	.08

TABLE 2
Running times for the parallel secant algorithm.

Problem	p	$n = 50$	$n = 100$	$n = 300$
9	1	20.5	128.27	2494.7
9	16	5.54	15.6	162.0
10	1	60.14	418.6	10264.3
10	16	9.4	36.7	643.9
14	1	92.6	397.0	5210.3
14	16	40.19	89.2	468.7
15	1	193.14	945.1	13121.2
15	16	85.7	227.9	1445.5

TABLE 3
Speedup for the parallel secant method, $p = 16$.

Problem	$n = 50$	$n = 100$	$n = 300$
9	3.8	8.2	15.4
10	6.4	11.4	15.9
14	2.3	4.5	11.1
15	2.25	4.1	9.1

single processor. The Minpack code was modified to force secant updates (after the initial finite-difference Jacobian estimation). Moreover, the Minpack stopping criteria were replaced by the rule mentioned above. Hence the two codes produce exactly the same sequence of x-iterates (we verified that this claim held on the four test problems in question).

Remarks. (1) Obviously speedups improve as n increases. This is due to the increase in distributed work to be performed.

(2) Problems 9 and 10 require only unit steps each iteration; on the other hand, problems 14 and 15 require many nonunit steps—many trial steps are rejected. It is this fact, *in combination* with the fact that function evaluations are *extremely cheap* that accounts for the relatively poor parallel efficiency demonstrated on problems 14 and 15.

To support this claim numerically, we have artificially increased the expense of each function evaluation in problem 14 by a factor of 100. The cost of a function evaluation is then about the same order of magnitude as the cost of a function evaluation in problem 10. For $p = 16$ and $n = 100$ the running time is 177.4 compared to 1562.3 when $p = 1$; the resultant speedup is 8.9, which compares favorably to the speedups obtained on problems 9 and 10.

The purpose of Table 4 is to provide some indication of how the computing time is distributed amongst the various tasks. The table entries represent the normalized time to do each task—for each row, each task time is divided by the time required by the most expensive task. Column "Int. J/QR" represents the time to do the initial estimation of J by finite-differences plus the time required to do the initial QR-factorization. The "F-evaluation" column represents the total time spent on evaluations of F excluding the initial estimation of J. The column labeled "Other" reflects the time spent on all remaining tasks. This includes several parallel matrix multiplies (used in the determination of Cauchy and Newton steps) as well as some sequential work (such as the adjustment of Δ).

Problem 14+ is the Minpack problem 14 with the cost of a function evaluation increased by a factor of 100.

TABLE 4
Secant Algorithm breakdown for $p = 16$, $n = 100$.

Problem	Init. J/QR	QR-update	F-evaluation	Tri. solve	Other
9	1.0	.26	.07	.22	.18
10	1.0	.14	.06	.10	.10
14+	1.0	.28	.42	.28	.06
15	.10	1.0	.27	.56	.40

It is now easy to see why problem 15 experiences relatively poor speedup: too little (relative) time is spent on the highly parallel tasks such as the initialization step and "F-evaluation." Instead, the QR-update and the triangular solve tasks consume the most time: unfortunately, neither task is as parallel-efficient as the QR-factorization or the distributed evaluation of F.

Our numerical experiments indicate to us that the proposed parallel secant method is acceptably efficient (for both cheap and expensive functions) except when F is cheap *and* many iterations are required. Note that many iterations may be required due to a poor Jacobian approximation; a hybrid routine such as the Minpack Algorithm would tend to refresh the approximation (e.g., by finite differences) under such circumstances and this, in turn, would tend to decrease the number of iterations. In particular, we note that the parallel finite-difference method actually outperforms the parallel secant method on problems 14 and 15: this is due to many fewer iterations and (ridiculously) cheap function evaluations.

4. Parallel finite-difference Newton method.

4.1. The algorithm. In the remainder of this paper we assume that the evaluation of $F(x)$ is not a distributed computation; every node has a copy of the F-evaluation subroutine. In addition, we discontinue the use of the subscript c; e.g., x and s refer to vectors x_c and s_c, respectively. The finite-difference approximation of the Jacobian matrix can obviously be done in parallel, given $F(x)$, with each node computing its resident columns independently. Communication between nodes is not required. Since efficient parallel routines for the LU and QR factorizations exist, and since the triangular solve problem has been extensively researched, with acceptable results, the only remaining difficulty is the evaluation of $F(x+s)$ when determining if $x+s$ is an acceptable point. If F is relatively expensive to compute, then it is not reasonable to designate one distinguished node to evaluate $F(x+s)$ while the others idle.

Our solution breaks into two parts. First, if in the course of a run previous experience suggests that the initial trial step s is likely to be accepted, then we take a chance and overlap the computation of $F(x+s)$ with the estimation of $J(x+s)$. This is at some risk because s might be determined to be unacceptable—due to the value of $\|F(x+s)\|$—and then any work expended on the computation of $J(x_c+s)$ has been wasted. But, as indicated in Fig. 6, each node will waste at most one evaluation of F before the suitability of s is determined.

Assign the task of evaluating $F(x+s)$ to node $n \bmod p$ (think of F as column $n+1$ of the Jacobian): $n+1 \in I(n \bmod p)$. Figure 6 describes the algorithm.

```
If {myid = n mod p} then {i.e., if I am the node that evaluates F(x+s)}
    Evaluate F(x+s);
    Broadcast F(x+s);
Else
    Choose j ∈ I(myid);
    Evaluate F(x+s+τe_j);
Endif

If x+s is not acceptable then exit
Else
    Evaluate the remainder of the Jacobian columns by finite differences;
Endif
```

FIG. 6. *Algorithm* J-*and*-F-*evaluation* (*node program*).

On the other hand, if previous experience suggests that there is a good chance that the initial step s will not be adequate, then our strategy is quite different. Specifically, each step of the parallel linesearch procedure involves p function evaluations, $F(w_{p-1}), \cdots, F(w_0)$, done in parallel, where $w_{p-1}, \cdots, w_0$ are points along the dogleg step P_c (see § 2). If the first step is unsuccessful, then a second parallel search is performed along a smaller segment of P_c. This process is repeated until a suitable point is found. The linesearch procedure is judged successful if the following "alpha condition" is satisfied for some $w_* \in \{w_{p-1}, \cdots, w_0\}$:

$$f(w_*) \leqq f(x) + \alpha \nabla f(x)^T [w_* - x], \tag{10}$$

where $0 < \alpha < \frac{1}{2}$, and $f(x) \overset{\text{def}}{=} \frac{1}{2}\|F(x)\|_2^2$. See Dennis and Schnabel [3] for a discussion of the "alpha condition."

The procedure can be considered a parallel *generalized* bisection algorithm or perhaps a generalized Armijo rule [1]. In each step we begin with a stepsize bound of Δ. Specifically, in the first step

$$\Delta \leftarrow \min \{\|s^{\text{Newton}}\|, \text{BOUND} \times \text{XNORM}\} \tag{11}$$

where $\text{XNORM} \overset{\text{def}}{=} \max \{\|x\|, \text{TYPX}\}$ and TYPX is a positive user-supplied constant representing the norm of a "typical" x-iterate; BOUND is a positive user-supplied constant.

In subsequent steps (if needed) Δ is defined by the (unsuccessful) evaluation point nearest to x (used in the previous step).

In each step the evaluation points are defined as follows:

$$w_i \in P_c, \quad \|w_i - x\| = \frac{\Delta}{\gamma^i}, \quad i = 0{:}\, p-1 \tag{12}$$

where γ is a positive number strictly greater than unity. The choice of γ is guided by the following two concerns.

First, γ should be chosen so that the point nearest to x, w_{p-1}, is not too close to x: i.e., we require

$$\frac{\Delta}{\gamma^{p-1}} \geqq \mu \times \text{XNORM} \tag{13}$$

where μ is small positive number (usually unit roundoff). We assume $\text{BOUND} \gg \mu$. Obviously expression (13) yields the upper bound on γ,

$$\gamma \leqq \left(\frac{\Delta}{\mu \times \text{XNORM}}\right)^{1/(p-1)} \tag{14}$$

Second, it is usually advantageous to spread the evaluation points so that at least one point (i.e., w_{p-1}) is on the Cauchy segment $(x, x + s^{\text{Cauchy}}]$. This leads to the condition, if $\|s^{\text{Cauchy}}\| < \Delta$, we require

$$\frac{\Delta}{\gamma^{p-1}} \leqq \|s^{\text{Cauchy}}\|. \tag{15}$$

However, this condition may lead to a γ so large that a very large segment of $s^{\text{Newton}} - s^{\text{Cauchy}}$ is devoid of function evaluations. Hence we compromise (15) and require only

$$\gamma \geqq \min \left\{ \left(\frac{\Delta}{\|s^{\text{Cauchy}}\|}\right)^{1/(p-1)}, 2 \right\}. \tag{16}$$

Note that if we assume that

$$\|s^{\text{Cauchy}}\| > \mu \times \text{XNORM}, \tag{17}$$

then (14) and (16) are consistent and $\gamma > 1$. If condition (17) does not hold, then it is reasonable to stop, claiming optimality.

Figure 7 exhibits the algorithm. The "Decrease Δ" step is implemented as follows. Let $p^+ = p+1$; determine $\gamma^+ > 1$ satisfying (14) and (16), replacing p with p^+. Finally, assign

$$\Delta \leftarrow \frac{\Delta}{[\gamma^+]^{(p^+-1)}}. \tag{18}$$

The "Fan-in" step determines $w^* \in W = \{w_{p-1}, \cdots, w_0\}$ such that $\|w^* - x\|$ is maximum and w^* satisfies (10). If no such point exists we define $w^* = x$. Determining if (10) is satisfied at point w_i is a simple computation. Specifically, we can write

$$w_i - x = \lambda_i^C s^{\text{Cauchy}} + \lambda_i^N s^{\text{Newton}}. \tag{19}$$

But, $\nabla f^T s^{\text{Newton}} = -\|F(x)\|_2^2 \stackrel{\text{def}}{=} \omega^N$ and $\nabla f^T s^{\text{Cauchy}} = -\beta \|J^T F\| \stackrel{\text{def}}{=} \omega^C$. (The constant β is computed when P_c is determined.) Therefore,

$$\nabla f^T [\omega_i - x] = \lambda_i^C \omega^C + \lambda_i^N \omega^N, \tag{20}$$

which is a trivial expression to compute on a single node.

The overall procedure is sketched in Fig. 8. Note that we have provided a high-level global view, as opposed to a node program. The parallelization of each computationally significant step has been discussed above.

4.2. Numerical results. We performed computational experiments using problems 9, 10, 14, and 15 referred to previously. The standard starting point (Factor = 1) was used in all cases. The stopping criterion was $\|F\| \leqq 10^{-8}$.

Problem 10 is distinguished from the others: F is relatively expensive to compute. In this sense problem 10 probably represents a more realistic test function. However, problems 14 and 15 are useful for testing because nonunit steplengths are required, whereas Newton iterations converge quickly, with unit steps, for problems 9 and 10.

In Table 5 we have recorded the iteration counts and running times obtained for the finite-difference Newton method described above (e.g., y/z indicates y iterations taking a total of z seconds); in Table 6 we divide the $p = 1$ running times by the $p = 4$ and $p = 16$ times to obtain a measure of speedup. In this case we do *not* compare our algorithm to a finite-difference version of the Minpack code because we feel such a

```
Choose Δ as in 11
Repeat
    If {myid = 0} then
        Determine γ > 1 satisfying (14) and (16);
        Broadcast γ;
    Endif
    i ← myid;
    Determine w_i ∈ P_c: ‖w_i − x‖ = Δ/γ^i;
    Evaluate z_i ≝ F(w_i);
    Participate in fan-in: z → z*, w → w* on node 0;
    Decrease Δ;
Until f(w_*) ≦ f(x) + α∇f(x)^T[w_* − x]
```

FIG. 7. *Algorithm Dogleg-Linesearch (node program).*

```
Guess an initial x;
Evaluate F(x), J(x);
Assign prev-step ← "not-Newton";
Repeat
    Factor J = LU;
    Determine P_c:
        Compute s^Cauchy; {see Fig. 1}
        Compute s^Dogleg; {see Fig. 1}
        Determine Δ; {use (11)}
    If {prev-step = "Newton"} then
        Try algorithm J-and-F;
        If {J-and-F is unsuccessful} then
            prev-step ← "not-Newton"
        Endif
    Endif
    If {prev-step = "not-Newton"} then
        Perform Dogleg-Linesearch;
        Evaluate J(x^+);
    Endif
Until {convergence}
```

FIG. 8. *Algorithm Newton with Dogleg-Linesearch.*

TABLE 5
Results for the Newton Algorithm with Dogleg-Linesearch.

Problem	p	$n = 50$	$n = 100$	$n = 300$
9	1	3/7.6	3/43.5	3/911.0
9	4	3/3.3	3/13.6	3/240.1
9	16	3/3.0	3/7.9	3/79.8
10	1	4/107.9	4/826.8	4/21701.4
10	4	4/30.4	4/220.0	4/5528.2
10	16	4/13.1	4/69.2	4/1447.4
14	1	9/41.1	9/218.2	9/4050.2
14	4	9/15.5	9/65.7	9/1062.6
14	16	9/12.3	9/33.4	9/344.0
15	1	22/65.0	27/496.3	31/12871.5
15	4	22/30.1	27/158.2	31/3398.8
15	16	11/13.7	12/40.6	14/496.1

comparison would be unfair. Specifically, the Minpack code is QR-based and ours is LU-based: such a comparison would give meaningless advantage to our method.

Our implementation struggles with problem 15 for small p (relative to its performance with $p = 16$). This is because our linesearch parameter γ was chosen with a moderately large p in mind. In particular, in a first iteration of the Linesearch Algorithm, we choose γ so that condition (16) is an equality. If a second iteration of the linesearch is needed, then γ is chosen to satisfy condition (14) exactly. This strategy appears to work quite well for $p = 16$ but can lead to small steps if p is small.

In Table 6 we have normalized the execution times reported in Table 5: for each problem divide the execution times in the subcolumn by the $p = 1$ execution time. Hence the entries in Table 6 reflect the speedup factor over the running time on a single node.

TABLE 6
Speedups for the Newton Algorithm with Dogleg-Linesearch.

Problem	p	$n=50$	$n=100$	$n=300$
9	4	2.3	3.2	3.8
9	16	2.5	5.5	11.4
10	4	3.5	3.8	3.9
10	16	8.2	11.9	15.0
14	4	2.7	3.3	3.8
14	16	2.5	5.5	11.4
15	4	2.2	3.1	3.8
15	16	4.7	12.2	25.9

Remarks on Table 6. (1) For fixed $p>1$, the speedup improves as n increases. The primary reason for this is that as n increases the parallel factorization becomes increasingly efficient (e.g., [9]).

As Table 7 indicates, the factorization accounts for a significant percentage of the total computational expense in the test problems.

(2) In general, the 4-processor speeds are closer to optimality (optimal speedup = 4) than the 16-processor speedups (optimal speedup = 16). Again, the primary factor here is the increased efficiency of the parallel LU-factorization as n/p increases. There are two exceptions to this trend in Table 6.

First, a nearly optimal speedup is obtained on problem 10, $n=300$. Table 7 explains this: the Jacobian estimation time dominates the factorization time—parallel finite-difference is a highly parallel task.

The other exception occurs on problem 15, $n=300$: a speedup of 25.9 is attained (considerably better than the "optimal" speedup of 16!). This is possible because the sequence of points generated is a function of the number of processors used (due to the linesearch). This dependence contrasts with the Parallel Secant Algorithm described in the previous section.

In Table 7 we break the total execution time down into the times required by the different substeps. Each row of Table 7 is normalized so that the maximum entry in each row is unity.

Except for problem 10, the factorization represents the dominant cost. We believe this is an anomaly: in practice function evaluations are often quite expensive. However, in either case, the linesearch and triangular solve times are relatively insignificant.

We are satisfied with the performance of this parallel finite-difference algorithm that combines a dogleg step with a generalized bisection algorithm. Our experience on our test collection indicates that the required number of iterations is almost always fewer than for a dogleg/trust region strategy (i.e., no linesearch). However, we admit that from an aesthetic point of view the linesearch procedure is unattractive: moreover,

TABLE 7
The Newton Algorithm breakdown for $p=16$, $n=300$.

Problem	Jac. est.	Factor	Linesearch	Tri. solve
9	.08	1.0	.01	.08
10	1.0	.11	.02	.01
14	.17	1.0	.00	.08
15	.04	1.0	.01	.08

it is somewhat heuristic in nature and is unrelated to the quadratic model philosophy. Considering these remarks, it might be preferable, overall, to stick with the usual dogleg/trust-region philosophy and always employ the algorithm in Fig. 6 to obtain parallelism. Our experiments indicate this would be slightly less efficient.

5. Parallel multiple secant method.

5.1. The algorithm. The obvious disadvantage to the finite-difference Newton method discussed in § 4 is that the estimation of the Jacobian matrix can be extremely time-consuming. This is less true in the parallel context since finite-differencing is a highly parallel task; nevertheless, it is often unnecessary to obtain such accuracy. The success of the sequential rank-1 secant method attests to this claim.

Section 3 presented a parallel secant method under the row-separability assumption; however, if F is to be treated as a single entity we do not know how to implement an efficient rank-1 secant method (when the evaluation of F is expensive). But, it is possible to fill the gap between rank-1 and rank-n (i.e., finite-difference approximation) with an efficient parallel rank-q secant method where q is a multiple of p, the number of nodes.

To introduce the *multisecant method* let us consider the case when $q = p$. For the moment we also assume that our algorithm is purely local: a unit step $x^+ \leftarrow x + s$ is always taken (we consider the general situation later).

Assume that the function value $F(x)$ is known to every node. The first step is to re-label the nodes so that a *ring* is induced (i.e., node i is a neighbor of both node $(i+1) \bmod p$ and node $(i-1) \bmod p$, for $i = 0: p-1$). A gray code mapping can be used for this purpose (e.g., [10]). Assume that B_c is distributed in the usual fashion, using this labeling. Hence, node j is assigned column k provided $k - 1 = j \bmod p$. Let s be the correction to x: i.e., $x_+ \leftarrow x + s$. (We assume for the moment that s will be accepted.)

The next step is key. Each node evaluates F at a different point. Node 0 evaluates $F(x + s^0)$, where $s^0 = s$; node j, $1 \leqq j \leqq p-1$, evaluates $F(x + s^j)$ where s^j is a sparse projection of s. That is, component i of s^j will be either s_i or zero. In particular,

$$\text{if } \{i - 1 = j \bmod p\} \text{ or } \{s_i^k = 0,\ 0 \leqq k < j\} \text{ then } s_i^j = 0, \tag{21}$$

$$\text{otherwise } s_i^j = s_i. \tag{22}$$

For example, if $p = q = 4$, $n = 8$,

$$s^0 = (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8), \tag{23}$$

$$s^1 = (s_1, 0, s_3, s_4, s_5, 0, s_7, s_8), \tag{24}$$

$$s^2 = (s_1, 0, 0, s_4, s_5, 0, 0, s_8), \tag{25}$$

$$s^3 = (s_1, 0, 0, 0, s_5, 0, 0, 0). \tag{26}$$

After evaluation, each node sends a copy of its newly computed function value to its higher numbered neighbor on the ring. Hence, after this shift, node j will have the vectors $F(x)$, $F(x + s^j)$, and $F(x + s^{(j-1) \bmod p})$. Therefore, if $p = q = 4$ then in addition to $F(x)$, each node has the pair of function values listed below:

$$\text{node 0: } F(x+s),\ F(x+s^3), \tag{27}$$

$$\text{node 1: } F(x+s^1),\ F(x+s), \tag{28}$$

$$\text{node 2: } F(x+s^2),\ F(x+s^1), \tag{29}$$

$$\text{node 3: } F(x+s^3),\ F(x+s^2). \tag{30}$$

We now demand that each node satisfy its own local secant equation.

Notation. For a matrix M let $M_{I(j)}$ denote the matrix of the same dimensions that matches M in columns $I(j)$ and whose other columns are zero columns. For $j=1:p-1$ the secant equation for node j is

$$B^+_{I(j)}[(x+s^{j-1})-(x+s^j)]=y^j \tag{31}$$

where $y^j \stackrel{\text{def}}{=} F(x+s^{j-1})-F(x+s^j)$. Equation (31) is reasonable because $(s^{j-1}-s^j)_i \neq 0 \Rightarrow i \in I(j)$ and

$$\left\{\int_0^1 J_{I(j)}([x+s^j]+\tau[s^{j-1}-s^j])\,d\tau\right\}(s^{j-1}-s^j)=y^j. \tag{32}$$

On node 0 we demand satisfaction of the secant equation

$$B^+_{I(0)}[s^{p-1}]=y^0 \tag{33}$$

where $y^0 \stackrel{\text{def}}{=} F(x+s^{p-1})-F(x)$.

This requirement is also reasonable because $s_i^{p-1} \neq 0 \Rightarrow i \in I(0)$, and

$$\left\{\int_0^1 J_{I(0)}(x+\tau s^{p-1})\,d\tau\right\}s^{p-1}=y^0. \tag{34}$$

Of course "reasonableness" does not establish that the method possesses desirable local convergence properties. We will consider this theoretical question elsewhere [16].

An important property of this parallel multisecant update is that there is very little communication required: each node sends (receives) exactly one vector to (from) an *adjacent* node. Moreover, beyond satisfying p local secant equations, the updated matrix B^+ also satisfies the *global* secant equation

$$B^+s=y \tag{35}$$

where $y \stackrel{\text{def}}{=} F(x+s)-F(x)$. To see this note that

$$B^+_{I(0)}s^{p-1}+B^+_{I(1)}(s^0-s^1)+\cdots+B^+_{I(p-1)}(s^{p-2}-s^{p-1})=B^+s \tag{36}$$

and

$$y^0+\cdots+y^{p-1}=F(x+s)-F(x)=y. \tag{37}$$

Figure 9 summarizes the node program representing the algorithm sketched above.

```
j ← myid;
Evaluate F(x+s^j);
Send a copy of the vector F(x+s^j) to node (j+1) mod p;
Receive a copy of the vector F(x+s^((j-1) mod p)) from node (j-1) mod p;

If {myid = 0} then
    y^0 ← F(x+s^(p-1)) - F(x);
    d^0 ← s^(p-1);
Else
    y^j ← F(x+s^(j-1)) - F(x+s^j);
    d^j ← s^(j-1) - s^j;
Endif

{Update resident columns}
If {d^j ≠ 0} then

    B+_I(j) ← B_I(j) + (y^j - B d^j) d^jT / (d^jT d^j);

Endif
```

FIG. 9. *Multiple secant update (node program).*

Note that the product Bd^j can be computed entirely locally on node j: the vector d^j has nonzero components only in locations corresponding to columns residing on node j (i.e., the nonzero columns in $B_{I(j)}$).

Two subtopics remain to be discussed: the generalization of the multisecant method to the case where q is a multiple of p, and the globalization strategy.

The generalization of the multisecant method to the case where q is a multiple of p is quite straightforward. Divide the columns on each node into q/p groups (typically with as many equal-sized groups as possible). Each group accounts for one evaluation of F; a local secant equation is defined with respect to each group. Otherwise, the method is the same as the $p = q$ case except now the ring is viewed as having q conceptual nodes. Note that the number of transferred messages between physical nodes remains at p (neighbor-to-neighbor, each message of size n).

Globalization can be achieved in a manner similar to the parallel finite-difference algorithm. Indeed, the only change is to replace the finite-difference Jacobian calculation with the local secant update. With this exception, the algorithm described in Fig. 8 can be used unaltered.[2] We note that if $p = q$ and Newton steps are being successfully used, then each node is involved in exactly one F-evaluation per iteration (in each Newton iteration there are p F-evaluations). However, when the secant update follows a linesearch there is a slight redundancy. Specifically, after the linesearch procedure is completed the value $F(x^+)$ is known. But the local secant update requires only $p-1$ additional F-evaluations beyond $F(x^+)$. Therefore, under these circumstances either one node remains idle during the parallel evaluation of F for the multiple secant update, or $F(x^+)$ is computed twice.

5.2. Numerical results. Experimentally we compared the multisecant method to the parallel finite-difference procedure discussed in § 4 using problems 9, 10, and 14 of the Minpack collection and the extended Rosenbrock function (problem 15). Table 8 provides the results. (Table entry y/z indicates y iterations taking a total of z seconds.)

TABLE 8
The multisecant method versus the finite-difference Newton method, $p = 16$.

Problem	Method	$n = 50$	$n = 100$	$n = 300$
9	MS	3/4.6	3/12.3	3/140.9
9	FD	3/3.0	3/7.9	3/79.8
10	MS	4/12.4	4/46.6	4/733.6
10	FD	4/13.1	4/69.2	4/1447.4
14	MS	20/32.4	23/99.3	23/1175.8
14	FD	9/12.3	9/33.4	9/344.0
15	MS	21/35.2	25/101.9	41/1912.6
15	FD	11/13.7	12/40.6	14/496.1

In general, the secant method requires more iterations. Therefore, when the factorization is the dominant cost—as it is in problems 9, 14, and 15—then the parallel finite-difference Newton method is faster. However, when F is expensive—as it is in problem 10—the multisecant method is probably preferable. Indeed, we have tried

[2] The remarks concerning linesearch versus trust region, made at the end of the previous section, are applicable to the globalized multisecant method as well.

problem 14+ (i.e., increase the expense of a function evaluation in problem 14 by a factor of 100) with $n = 100$: the multisecant then requires 405.5 seconds compared to 551.1 required by the parallel finite-difference method.

To determine where the algorithm spends most of its time, on this test collection, Table 9 provides a breakdown of the total execution time.

Table 9 is fairly similar to Table 7; however, the factorization cost for problem 10 is relatively more expensive (.38 versus .11). This is because the dominant cost—Jacobian estimation—has decreased considerably. In a similar vein, the relative Jacobian costs have almost become insignificant in problems 9, 10, and 15.

As mentioned above, the multisecant method can allow for the independent estimation of several groups per node (as opposed to just one group per node). In Table 10 we provide results indicating the effect of varying the number of groups, q, per node.

The results are as expected. For problem 10 the computational expense increases as q increases. This is because of the expensive nature of F (and the number of iterations stays constant). Problem 15 exhibits exactly the opposite behavior: as q increases the execution time decreases. The reason for this is that the number of iterations decreases as q increases: this is reasonable since the Jacobian approximations become increasingly accurate as q increases. In general then, the optimal q will depend on the particular problem: the relative costs of evaluating F, factoring the matrix, and the convergence dependence on q, play a role.

6. Conclusions. We have proposed parallel algorithms for the solution of systems of nonlinear equations $F(x) = 0$. The algorithms are applicable on local-memory multiprocessors (such as a hypercube computer) provided that each processor has significant memory and computational power and the problem dimension is greater than the number of processors.

TABLE 9
The Multisecant Algorithm breakdown for $p = 16$, $n = 300$.

Problem	Jac. est.	Factor	Linesearch	Tri. solve
9	.04	1.0	.00	.07
10	1.0	.38	.04	.02
14	.05	1.0	.00	.06
15	.02	1.0	.01	.07

TABLE 10
Multisecant: Vary # groups-per-node, $p = 16$, $n = 300$.

Problem	# groups-per-node	# iterations	Total time
10	1	4	733.6
10	2	4	804.7
10	4	4	932.4
10	8	4	1087.2
15	1	41	1912.6
15	2	40	1679.8
15	4	29	1003.6
15	8	21	853.6

When F is efficiently computable in a distributed parallel manner, the globalized rank-1 secant method can be efficiently parallelized. Specifically, it is possible to parallelize the Minpack implementation, updating the QR factorization of an approximate Jacobian matrix every step. On balance we are satisfied with the parallel performance of our global parallel secant implementation. However, it should be noted that we have efficiently distributed and parallelized the F-subroutines in our experiments. In general such a task falls to the user and speedup will be strongly affected by the user's success in this task. Note that distributing F by rows, even if feasible, does not always lead to the most efficient distribution of work since it does not exploit the presence of common expensive subexpressions.

Since it is not always possible to efficiently parallelize the computation of F, we have developed parallel finite-difference and multisecant methods. In general it is difficult to achieve good speedup, relative to the sequential rank-1 secant method, for this class of functions; however, the finite-difference algorithm is efficiently parallelized and the multisecant method will generally improve on this (especially for large n/p). It is perhaps possible to further improve on the efficiency of the multisecant method by incorporating parallel multirank updates to the current distributed QR factorization. We have not yet investigated this possibility.

Finally, we note that sparse systems are partially separable (i.e., each component function depends only on a few variables); therefore, in theory, it is usually possible to effectively evaluate $F(x)$ in a distributed parallel manner. Hence, it may be possible to efficiently solve large sparse systems of nonlinear equations using a parallel sparse secant method.

Acknowledgments. Our numerical experiments were performed with the assistance 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. Specifically, we are grateful to Ashley Doershug for her help with the Intel iPSC.

REFERENCES

[1] L. ARMIJO, *Minimization of functions having Lipschitz continuous first partial derivatives*, Pacific J. Math., 16 (1966), pp. 1-3.

[2] C. BROYDEN, *A class of methods for solving nonlinear simultaneous equations*, Math. Comp., 19 (1965), pp. 577-593.

[3] J. E. DENNIS AND R. B. SCHNABEL, *Numerical Methods for Unconstrained Optimization*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[4] P. GILL AND W. MURRAY, *Quasi-Newton methods for unconstrained optimization*, J. Inst. Maths. Appl., 9 (1972), pp. 91-108.

[5] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1983.

[6] M. HEATH AND C. ROMINE, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 558-588.

[7] G. LI AND T. F. COLEMAN, *A parallel triangular solver for a distributed memory multiprocessor*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 485-502.

[8] ———, *A new method for solving triangular systems on a distributed memory message-passing multiprocessor*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 382-396.

[9] C. MOLER, *Matrix computations on distributed memory multiprocessors*, Tech. Report, Intel Scientific Computers, Beaverton, OR, 1987.

[10] C. MOLER AND D. S. SCOTT, *Communications utilities for the ipsc*, Tech. Report, Intel Scientific Computers, Beaverton, OR, 1986.

[11] J. J. MORÉ, B. S. GARBOW, AND K. E. HILLSTROM, *User guide for minpack-1*, Tech. Report ANL-80-74, Argonne National Laboratory, Argonne, IL, 1980.

[12] M. J. D. POWELL, *A hybrid method for nonlinear equations*, in Numerical Methods for Nonlinear Algebraic Equations, P. Rabinowitz, ed., Academic Press, New York, 1970, pp. 87–114.

[13] C. ROMINE AND J. ORTEGA, *Parallel solution of triangular systems of equations*, Tech. Report RM-86-05, Department of Applied Mathematics, University of Virginia, Charlottesville, VA, 1986.

[14] Y. SAAD AND M. H. SCHULTZ, *Topological properties of hypercubes*, Tech. Report CS-RR-389, Computer Science Department, Yale University, New Haven, CT, 1985.

[15] P. WILEY, *A parallel architecture comes of age at last*, IEEE Spectrum (1987), pp. 46–50.

[16] T. F. COLEMAN AND G. LI, *Local convergence of the multi-secant method for the parallel solution of systems of nonlinear equations*, Appl. Math. Lett., 1 (1988), pp. 141–145.