Unit Tangent Vector Computation for Homotopy Curve Tracking on a Hypercube

A. Chakraborty, D.C.S. Allison, C.J. Ribbens
and L.T. Watson

TR 89-39

Unit Tangent Vector Computation for Homotopy Curve Tracking on a Hypercube

A. Chakraborty, D. C. S. Allison, C. J. Ribbens and L. T. Watson

Department of Computer Science
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061

Abstract.

Probability-one homotopy methods are a class of methods for solving nonlinear systems of equations that are globally convergent from an arbitrary starting point. The essence of all such algorithms is the construction of an appropriate homotopy map and subsequent tracking of some smooth curve in the zero set of the homotopy map. Tracking a homotopy curve involves finding the unit tangent vectors at different points along the zero curve. Because of the way a homotopy map is constructed, the unit tangent vector at each point in the zero curve of a homotopy map $\rho_a(\lambda, x)$ is in the kernel of the Jacobian matrix $D\rho_a(\lambda, x)$. Hence, tracking the zero curve of a homotopy map involves finding the kernel of the Jacobian matrix $D\rho_a(\lambda, x)$. The Jacobian matrix $D\rho_a$ is a $n \times (n+1)$ matrix with full rank. Since the accuracy of the unit tangent vector is very important, an orthogonal factorization instead of an LU factorization of the Jacobian matrix is computed. Two related orthogonal factorizations, namely QR and LQ factorization, are considered here. This paper presents computational results showing the performance of several different parallel orthogonal factorization/triangular system solving algorithms on a hypercube. Since the purpose of this study is to find ways to parallelize homotopy algorithms, it is assumed that the matrices are small, dense, and have a special structure such as that of the Jacobian matrix of a homotopy map.

1. Introduction.

Algorithms for solving nonlinear systems of equations can be broadly classified as (1) locally convergent or (2) globally convergent. The former includes Newton's method, various quasi-Newton methods, and inexact Newton methods. The latter includes continuation, simplicial methods, and probability-one homotopy methods. These algorithms are qualitatively significantly different, and their performance on parallel systems may very well be the reverse of their performance on serial processors. The overall purpose of this research is to study how general nonlinear systems of equations might be solved on a hypercube; this paper addresses a part of that topic, namely, the parallel computation of the unit tangent vector of the Jacobian matrix of a homotopy map. Polynomial systems are not considered here since they have a very special structure which leads to different strategies for parallelism. Polynomial systems have been studied in [2], [9], and [11].

Section 2 summarizes the mathematics behind the homotopy algorithm, and how orthogonal factorizations are used in the algorithm. Section 3 describes parallel algorithms for orthogonal factorizations and triangular system solving. Computational results on an Intel iPSC hypercube are discussed in Section 4.

2. Homotopy algorithm.

Let E^n denote n-dimensional real Euclidean space, and let $F : E^n \to E^n$ be a C^2 (twice continuously differentiable) function. The general problem is to solve the nonlinear system of equations

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

The fundamental mathematical result behind the homotopy algorithm [18], [17]) is

Proposition 1. *Let $F : E^n \to E^n$ be a C^2 map and $\rho : E^m \times [0,1) \times E^n \to E^n$ a C^2 map such that*

1) the Jacobian matrix $D\rho$ has full rank on $\rho^{-1}(0)$;
and for fixed $a \in E^m$,

2) $\rho(a,0,x) = 0$ has a unique solution $W \in E^n$,

3) $\rho(a,1,x) = F(x)$,

4) the set of zeros of $\rho_a(\lambda,x) = \rho(a,\lambda,x)$ is bounded.

Then for almost all $a \in E^m$ there is a zero curve γ of $\rho_a(\lambda,x) = \rho(a,\lambda,x)$, along which the Jacobian matrix $D\rho_a(\lambda,x)$ has full rank, emanating from $(0,W)$ and reaching a zero $\bar{x}$ of F at $\lambda = 1$. Furthermore, γ has finite arc length if $DF(\bar{x})$ is nonsingular.

The homotopy algorithm consists of following the zero curve γ of ρ_a emanating from $(0,W)$ until a zero $\bar{x}$ of $F(x)$ is reached (at $\lambda = 1$). It is nontrivial to develop a viable numerical algorithm based on this idea, though, conceptually, the algorithm for solving the nonlinear system of equations $F(x) = 0$ is clear and simple. A typical form for the homotopy map is

$$\rho_W(\lambda,x) = \lambda F(x) + (1-\lambda)(x - W), \tag{2}$$

which has the same form as a standard continuation or embedding mapping. However, there are crucial differences. In standard continuation, the embedding parameter λ increases monotonically from 0 to 1 as the trivial problem $x - W = 0$ is continuously deformed to the problem $F(x) = 0$. In homotopy methods λ need not increase monotonically along γ and thus turning points present no special difficulty. The way the zero curve γ of ρ_a is followed and the full rank of $D\rho_a$ permit λ to both increase and decrease along γ and guarantee that there are never any "singular points" along γ which afflict standard continuation methods. Also, Proposition 1 guarantees that γ cannot just "stop" at an interior point of $[0,1) \times E^n$.

The zero curve γ of the homotopy map $\rho_a(\lambda,x)$ (of which $\rho_W(\lambda,x)$ in (2) is a special case) can be tracked by many different techniques; refer to the excellent survey [1] and recent work [17], [18]. There are three primary algorithmic approaches to tracking γ: 1) an ODE-based algorithm, 2) a predictor-corrector algorithm whose corrector follows the flow normal to the Davidenko flow (a "normal flow" algorithm); 3) a version of Rheinboldt's linear predictor, quasi-Newton corrector algorithm [3], [14], (an "augmented Jacobian matrix" method). Alternatives 1), 2) and 3) are described in detail in [16], [17] and [3] respectively.

All of these tracking algorithms need a unit tangent vector at different points along the zero curve. Because of the way a homotopy map is constructed, finding a unit tangent vector amounts to finding the one dimensional kernel of the $n \times (n+1)$ Jacobian matrix $D\rho_a$, where $D\rho_a$ has (theoretical) rank n. The crucial observation is that the last n columns of $D\rho_a$, corresponding to $D_x\rho_a$, may not have rank n, and even if they do, some other n columns may be better conditioned. The objective is to avoid choosing n "distinguished" columns, rather to treat all columns the

same (not possible for sparse matrices). There are kernel finding algorithms based on Gaussian elimination and n distinguished columns. Choosing and switching these n columns are tricky, and based on *ad hoc* parameters. Also, computational experience has shown that accurate tangent vectors $(d\lambda/ds, dx/ds)$ are essential, and the accuracy of Gaussian elimination may not be good enough. A conceptually elegant, as well as accurate, algorithm is to compute the QR factorization (with column interchanges) $Q\, D\rho_a\, P^t = R$, where Q is a product of Householder reflections, P is a permutation matrix, and R is $n \times (n+1)$ upper triangular. We then obtain a vector $z \in \ker(D\rho_a)$ by solving $R(Pz) = 0$. Setting $(Pz)_{n+1} = 1$ is a convenient choice. This scheme provides high accuracy, numerical stability, and a uniform treatment of all $n+1$ columns.

The kernel of the Jacobian can also be found by computing an LQ factorization of $D\rho_a$. Once an LQ factorization is obtained, an element of the kernel can be found by solving $Lx = 0$ and then solving $Qz = x$. Notice that $e_{n+1} = (0, \ldots, 0, 1)^t$ is a solution of $Lx = 0$, so that $Q^t e_{n+1}$ is a solution to the system $D\rho_a z = 0$. Thus, the last column of Q^t is in the kernel of $D\rho_a$. Since $D\rho_a$ is $n \times (n+1)$ with full rank, row interchanges are not necessary. Also, the method does not require a triangular solver. However, the matrix Q needs to be formed explicitly. This computation can be carried out simultaneously with the factorization of $D\rho_a$ without any extra communications. This will double the arithmetic computation in the factorization phase but will avoid the computation and communication associated with the triangular solver. For a small matrix ($n < 100$) this can be advantageous.

3. Parallel algorithms.

The most time consuming part of homotopy curve tracking is finding the unit tangent vector at different points along the zero curve γ. The three major steps for finding a unit tangent vector at a point (λ, x) on γ are: 1) compute the Jacobian matrix of the homotopy map ρ_a at (λ, x); 2) compute an orthogonal decomposition of the Jacobian matrix $D\rho_a$; 3) solve a triangular system of equations if necessary.

3.1. Parallel Jacobian matrix evaluation.

For many real engineering problems it is not possible to compute the Jacobian matrix analytically. Most often the Jacobian matrix is computed using finite difference approximations. Given that all components of the Jacobian matrix can be computed independently, a good parallel algorithm, with low communication requirements, would be to let each processor compute exactly those components that will be assigned to it during the orthogonal decomposition phase. However, if the variation in the evaluation time of the components is high, there may be uneven loading of the processors and thus inefficiency. In that case it may be better to use the host as the master and let it assign components to the processors as they become available. However, this method has a large communication overhead. Also, when there are a large number of components to evaluate, the variation in the total evaluation time should not be very high. Thus the master/slave paradigm is advantageous only when the Jacobian matrix is relatively small, each component is very expensive to evaluate, and there is a large variation in the evaluation times. Since the algorithm can not determine which method to use, it is up to the user to decide how the Jacobian matrix is to be evaluated (Jacobian matrix evaluation routines are almost always user supplied). However, the user should not have to write a substantial amount of additional code to evaluate a Jacobian matrix in parallel. The user should only have to give some estimate of the cost of component evaluation, specify an appropriate paradigm (e.g., serial, master/slave, uniform distribution across processors, etc.), and supply a routine to evaluate components of the Jacobian matrix.

3.2. Parallel orthogonal decompositions.

Most of the literature on parallel orthogonal decompositions on distributed memory multiprocessors describes how to compute a QR factorization of a matrix (see [4], [13], [12], and [5]). Extending these algorithms to compute an LQ factorization is straight-forward. There are several parallel QR algorithms proposed in the literature. However, most of these algorithms do not consider the case when column switching is necessary. The main objective of this study is to examine existing orthogonal factorization algorithms and incorporate necessary column switching so that they can be used to track a homotopy curve. These methods are then compared with LQ factorizations. The two algorithms described in [5] are the most suitable for our purpose. The algorithms and the necessary modifications are described briefly here. For a detailed description of the original algorithms refer to [5].

In the first algorithm of [5], the processors are mapped into a ring. The rows of the matrix are assigned to the processors in a wrap-mapping fashion. In this, the ith row is assigned to the processor numbered $i - \lfloor (i-1)/p \rfloor p - 1$, where p is the number of processors. Figure 1 shows the wrap-mapping of 9 rows to 4 processors.

$$\begin{bmatrix} P_0 & P_0 & P_0 & P_0 \\ P_1 & P_1 & P_1 & P_1 \\ P_2 & P_2 & P_2 & P_2 \\ P_3 & P_3 & P_3 & P_3 \\ P_0 & P_0 & P_0 & P_0 \\ P_1 & P_1 & P_1 & P_1 \\ P_2 & P_2 & P_2 & P_2 \\ P_3 & P_3 & P_3 & P_3 \\ P_0 & P_0 & P_0 & P_0 \end{bmatrix}$$

Figure 1. The wrap mapping of 9 rows to 4 processors

At each stage (at the kth stage elements below the diagonal of column k are zeroed out), the algorithm performs two steps: 1) each processor independently introduces zeros in the part of the column it holds by Givens rotations or Householder reflections, 2) processors cooperate with each other in a recursive merge fashion to introduce the rest of the zeros. In this case a slight modification of the algorithm permits the switching of columns without any communication. The original algorithm performs some redundant computation in order to use a similar communication algorithm in the beginning of each stage. Because of this redundant computation, each processor has the modified pivot row (i.e., the kth row at the beginning of the $(k+1)$st stage). Each processor also holds an array $sum(j)$ $(j = 1, \ldots, n+1)$, where at the beginning of the kth stage $sum(i)$ $(i = k, \ldots, n+1)$ contains the norm of the ith subcolumn $A(k,i), \ldots, A(n,i)$. (The array A contains the $n \times (n+1)$ Jacobian $D\rho_a$). At the end of the kth stage each processor updates $sum(i)$ by executing $sum(i) := SQRT(sum(i)^2 - A(k,i)^2)$ for $i = k+1, \ldots, n+1$. At the beginning of stage 1, in a hypercube of dimension d, each processor executes the following algorithm to initialize the array sum:

for $i := 1, \ldots, n+1$ **do**
    initialize $sum(i)$ **to the norm of the portion of the** i**th subcolumn it holds**
end
for $j := 1, \ldots, d$ **do**
    send the array sum **to the processor whose id differs only in bit** j

receive array sum **in** t **from the processor whose id differs only in bit** j
for $i := 1, \ldots, n+1$ **do**
$sum(i) := SQRT(sum(i)^2 + t(i)^2)$
end
end

A brief description of the factorization algorithm, consisting of an independent annihilation phase (IAP) and a cooperative merging phase (CMP), follows:

Step 1. (IAP) Among all of the rows with row number $i \geq k$, each processor uses the lowest numbered row as the pivot row to eliminate all of the off-diagonal nonzero elements in the kth column of its remaining rows by a Householder transformation. Since each processor has all the necessary information to update the part of the matrix it holds, no communication is needed here.

Step 2. (CMP) $l := d$, where d is the dimension of the hypercube.

Step 3. (CMP) Every processor sends its current local pivot row to the processor whose id differs in bit b_{l-1}. It also receives a row from the other processor. Let ρ_1 and ρ_2 be the row numbers where $\rho_1 > \rho_2$.

Step 4. (CMP) Each processor computes a Givens rotation to eliminate the element $a_{\rho_2,k}$. If the row ρ_2 is originally assigned to this processor then this row is updated and saved.

Step 5. (CMP) Each processor updates row ρ_1. The updated row ρ_1 becomes the current local pivot row.

Step 6. (CMP) $l := l - 1$.

Step 7. (CMP) If $l \geq 1$ then go to Step 3.

Step 8. (CMP) Retrieve the saved row.

Step 9. (CMP) Update norms of each column.

In the second algorithm of [5], the processors are mapped into a $\lambda_1 \times \lambda_2$ rectangular grid, where $\lambda_1 = 2^{d_1}$, $\lambda_2 = 2^{d_2}$ and $d = d_1 + d_2$. Each row or column forms a subcube. Processors are arranged so that the id's of the processors in the same row differ only in the rightmost d_2 bits. Similarly, the id's of the processors in the same column differ only in the leftmost d_1 bits. Figure 2 shows the embedding of a 4×4 grid in a 16 processor hypercube. Rows are assigned to each subcube in the d_1 direction in a wrap-mapping fashion. Elements of each row, in turn, are assigned to the processors in the corresponding subcube in a wrap-mapping fashion. In this case a single processor does not hold a complete row. Instead, each subcube in the d_1 direction holds a complete row, and each subcube in the d_2 direction holds a complete column. Figure 3 shows the wrap-mapping of a 8×8 matrix to a 4×4 processor grid.

$$\begin{bmatrix} P_0 & P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} \\ P_{12} & P_{13} & P_{14} & P_{15} \end{bmatrix}$$

Figure 2. The embedding of a 4×4 matrix into a 16 processor hypercube.

$$\begin{bmatrix} P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} \\ P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} \\ P_0 & P_1 & P_2 & P_3 & P_0 & P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 & P_7 & P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} & P_8 & P_9 & P_{10} & P_{11} \\ P_{12} & P_{13} & P_{14} & P_{15} & P_{12} & P_{13} & P_{14} & P_{15} \end{bmatrix}$$

Figure 3. The wrap mapping of a 8×8 matrix into a 4×4 grid.

Algorithm 2 is based on the same Householder/Givens sequence employed by the first algorithm. However, a subcube instead of a single processor holds a row, so there is some communication involved in the independent annihilation phase. The processor holding the local pivot element must broadcast all of the computed multipliers to every other processor in its subcube. At the beginning of the kth stage, if the processors in the subcube holding the kth column have at least one more column, they can switch columns without any communication. Otherwise, they cooperate in switching columns with the subcube holding the next column. This requires one communication step. This situation will arise only in the last λ_2 stages, when each subcube holds at most one column that has not yet been treated.

The LQ versions of the above QR algorithms follow nearly the same steps. The difference is that in the case where the processors are mapped into a ring the columns are mapped to the processors in a wrap-mapping fashion. An array $\mathcal{Q}$, which accumulates Q, is initialized to I. Each processor holds exactly the same columns of $\mathcal{Q}$ as the original matrix A. At each stage while the matrix A is being multiplied by an orthogonal matrix Q_i, $\mathcal{Q}$ is also multiplied by Q_i. When two processors need to exchange information during the recursive merging phase, they can exchange some extra information about the matrix $\mathcal{Q}$. This enables the processors to compute $\mathcal{Q}$ without any extra message passing. However, the message length will increase. This also requires each processor to store $((n+1)/nump + 1)(n+1)$ extra numbers for $\mathcal{Q}$, where $nump$ is the number of processors.

3.3. Parallel triangular system solver.

A triangular system solver is needed along with QR factorization to obtain the unit tangent vector. There are two choices here. The first choice is to ship the factorization result back to the host and use a serial triangular system solver. The second choice is to keep the factorization results in the nodes and use a parallel triangular solver. Since triangular system solving is inherently serial, the relative frequency of communication with respect to the amount of computation involved is very high. Most of the literature on parallel triangular system solving on distributed memory multiprocessors assumes the size of the matrix is on the order of 1000. There are many parallel algorithms that are proposed in the literature (e.g., [6], [7], and [8]). Some of them are better than others. However, for small to medium size matrices their performance difference is negligible. Thus, the purpose here is not to find the best parallel triangular solver but to investigate if it is at all advantageous to use a parallel algorithm. If our purpose was only to solve a triangular system it would be better to use an algorithm where each processor has full rows or columns (e.g., the processors are mapped into a ring). However, the results in Section 4 indicate that the factorization algorithms do much better when processors are mapped into a rectangular grid, and the savings in time are much greater than we could possibly get from solving the triangular system with a

ring connected topology. Thus a different triangular solver (which assumes that the processors are mapped into a rectangular grid) is used. A simplified version of this algorithm to solve $Ax = 0$ is given below, where the $n \times (n+1)$ upper triangular matrix A is stored in the array a.

```
if MYCOL(n+1) then
  x(n+1) := 1
  b(.) := -a(.,n+1)
  put (b, LEFT)
endif
for k := n, n-1, ..., 1 do
begin
  if MYCOL(k) then
    if MYROW(k) then
      get (b, RIGHT)
      x(k) := b(k)/a(k,k)
      b(.) := b(.) - x(k) * a(.,k)
      put (x(k), UP)
      put (b, LEFT)
    else
      get (x(k), DOWN)
      put (x(k), UP)
      get (b, RIGHT)
      b(.) := b(.) - x(k) * a(.,k)
      put (b, LEFT)
    endif
  endif
end
```

4. Computational results.

Tables 1–4 show the computational results obtained on a 16 node and a 32 node Intel iPSC/2 machine. All the matrices were $n \times (n+1)$ with full rank. Since the purpose of this study was to find ways to track a homotopy path in parallel, and orthogonal factorizations are appropriate only when the Jacobian matrix is small and dense, the algorithms were tested on relatively small size matrices. The matrices used for Tables 1–3 were generated randomly, while those for Table 4 came from a Galerkin approximation to a buoyant rotating disc fluid mechanics problem [15]. The notation used in the tables is as follows:

SER – serial QR factorization on the host;
QR1 – QR factorization, processors are mapped into a linear array, independent phase is done by Householder reflections, recursive merging phase is done by Givens rotations;
QR2 – same as QR1 except processors are mapped into a $d_1 \times d_2$ rectangular grid;
LQ1 – LQ factorization, processors are mapped into a linear array, independent phase is done by Householder reflections, recursive merging phase is done by Givens rotations;
LQ2 – same as LQ1 except processors are mapped into a $d_1 \times d_2$ rectangular grid;
TSS – triangular system is solved serially;
TSP – triangular system is solved in parallel;
QRS – QR factorization and the triangular system solution are done serially;
QRT – QR factorization (algorithm 2) is done in parallel and the triangular system is solved serially;

Table 1. Execution time in secs (16 nodes).

n	SER	QR1	QR2	LQ1	LQ2
16	1.3	4.2	5.0	4.3	5.0
32	3.1	6.0	6.1	7.3	6.6
50	10.0	9.7	8.3	10.2	9.5
64	21.3	13.1	9.5	18.3	12.3
75	30.0	14.1	11.0	21.4	15.6
100	72.0	20.3	15.0	35.0	20.0
150	200.0	33.5	23.0	85.0	62.2

Table 2. Execution time in secs (32 nodes).

n	SER	QR1	QR2	LQ1	LQ2
16	1.3	5.7	6.4	5.4	6.3
32	3.1	7.6	7.4	8.2	8.2
50	10.0	9.7	8.6	11.9	9.7
64	21.3	10.7	9.0	15.8	9.9
75	30.0	14.4	11.4	19.4	13.7
100	72.0	22.0	14.1	27.5	17.1
150	200.0	32.0	21.9	68.0	34.2

QRP – QR factorization (algorithm 2) and the triangular system solution are done in parallel;
SFER – serial function and Jacobian matrix evaluation;
PFER – parallel function and Jacobian matrix evaluation;
SKER – serial function evaluation, Jacobian matrix evaluation, and unit tangent vector computation;
PFUN – function and Jacobian matrix evaluation are done in parallel but factorization and triangular solving are done serially;
PKER – parallel unit tangent vector computation (parallel function, Jacobian matrix evaluation and QR factorization using algorithm 2).

It can be seen from Tables 1 and 2 that the parallel algorithms are performing better than the serial algorithm for matrices of size larger than 50. The orthogonal decomposition is just one part of the larger homotopy curve tracking algorithm. In this larger context, if the functions are evaluated in parallel, it is possible that this cross-over point will be lower. The reason is that when the functions are evaluated in parallel and the factorization is done serially, the evaluated Jacobian matrix has to be shipped back to the host. This will increase the overall time taken to evaluate the functions and compute an orthogonal factorization. For a parallel orthogonal factorization this initial shipment of data is not necessary. In the case of LQ factorization, the unit tangent vector is already computed and nodes can return it instead of a much larger Jacobian matrix (in the case of a serial factorization algorithm) or resultant triangular matrix (in the case of a QR factorization).

It can also be seen from Tables 1 and 2 that LQ factorizations are doing substantially worse than QR algorithms for matrices of size larger than 64. Despite the savings due to avoiding a triangular solve in the LQ algorithms, the extra computations needed to compute Q make these algorithms less efficient. We also see from Tables 1 and 2 that the algorithms based on a grid

Table 3. Execution time in secs (32 nodes).

n	TSS	TSP	QRS	QRT	QRP
16	–	1.7	1.3	6.4	6.5
32	–	2.5	3.1	7.5	7.4
50	0.3	2.8	10.1	8.8	8.7
64	0.7	2.9	21.6	9.5	9.1
75	1.2	3.1	30.8	12.1	11.6
100	2.0	3.4	73.5	15.5	14.5
150	3.9	3.8	203.0	24.0	22.1

Table 4. Execution time in secs (32 nodes).

n	SFER	PFER	SKER	PFUN	PKER
11	28.7	31.8	29.5	33.0	34.1
23	107.1	37.4	110.3	38.5	39.6
35	236.1	47.0	239.2	49.0	49.8
74	1053.0	76.8	1074.6	93.5	87.1
119	2850.0	130.8	3000.0	278.0	144.0

of processors are more efficient than those that organize the processors in a ring, for all but the smallest problems.

Table 3 shows that the serial triangular solver (TSS) is doing better than the parallel triangular solver (TSP) most of the time. However, as the size of the matrices increases the parallel algorithm becomes comparable. Though the serial triangular solver is doing better than the parallel triangular solver, it is not doing so when used in conjunction with QR factorizations (QRT versus QRP). This occurs because for QRT a large amount of data needs to be shipped back from the nodes to the host. On the other hand, when the parallel triangular solver (QRP) is used it is not necessary to ship the matrix back to the host. Only the solution to the system (which is much smaller than the triangular matrix) needs to be transmitted.

Since the granularity of computation is quite fine for small matrices, using more processors is not helpful, as shown by Tables 1 and 2. For matrices of size smaller than 50, the time taken by 16 processors is less than the time taken by 32 processors. This is because the time saved at each stage (using the additional processors) is less than the extra communication overhead. Though for both LQ and QR factorizations the algorithms using a rectangular grid are performing much better, the other algorithms (using a ring) can still be competitive in the larger curve tracking context since Jacobian matrices may be much easier to compute if a whole column or row is evaluated by each processor.

It is easier to analyze the factorization results than the function evaluation results. The reason is that the speedup of a factorization algorithm can be completely characterized by the size of the matrix. However, function and Jacobian matrix evaluation depend on the nature of the function and how the Jacobian matrix is computed. The Jacobian matrix can be large or small with each component cheap or expensive to compute. If the Jacobian matrix is large enough so that parallel factorization is advantageous, then it is always better to evaluate the components of the Jacobian matrix in parallel. This is true because parallel evaluation has very little overhead in addition to that already incurred by the decomposition algorithm. When the Jacobian matrix is very small but

expensive enough to justify parallel evaluation, it is better to do the factorization and triangular system solving serially. Thus there are three possible cases to consider:

1) system with large ($n \geq 50$) Jacobian matrix – function, Jacobian matrix evaluation, and kernel computation should all be done in parallel;
2) system with small but very expensive Jacobian matrix – function and Jacobian matrix evaluation should be done in parallel but the kernel computation serially;
3) system with small and cheap Jacobian matrix – everything should be done serially.

Table 4 shows some timing results for systems with small to medium Jacobian matrices. Each component was relatively expensive to compute. The Jacobian matrix was computed using finite difference approximations. Thus the function had to be evaluated a total of $n^2 + n$ times. Though there were no communications involved in function evaluations (except minor communication at the beginning and at the end), some computations were duplicated. This redundancy explains why a speedup near 32 was not achieved. For this problem the function evaluation time dominated the unit tangent vector computation time and thus parallel function evaluations contributed most to the speedup.

5. Future Research.

We have discussed the performance of different orthogonal factorizations and triangular system solvers in the context of homotopy curve tracking. There are homotopy algorithms where a new Jacobian matrix is not evaluated at every step; rather a Broyden's rank-one update in QR form is computed and applied. These quasi-Newton update methods have lower convergence rates, but can be advantageous when the Jacobian matrix is very expensive to evaluate. Indeed, there are many problems where quasi-Newton methods are superior to the Newton-like methods that compute a new Jacobian matrix at every step. However, since Broyden's rank-one update (in factored QR form) is not very suitable for parallelization, it is possible that for the same kind of problem the parallel (or serial) update method is inferior to the parallel methods that compute the Jacobian matrix at every step. Future study should resolve these questions, and provide a characterization of these cases similar to what was done here for Newton-type homotopy curve tracking algorithms on a hypercube.

References.

[1] E. Allgower and K. Georg, Simplicial and continuation methods for approximating fixed points, *SIAM Rev.* **22** (1980) 28–85.

[2] D.C.S. Allison, A. Chakraborty, and L. T. Watson, Granularity issues for solving polynomial systems via globally convergent algorithms on a Hypercube, *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* Pasadena, CA (1988) 1463–1472.

[3] S.C. Billups, An augmented Jacobian matrix algorithm for tracking homotopy zero curves, M.S. Thesis, Dept. of Computer Sci., VPI & SU, Blacksburg, VA, 1985.

[4] R.M. Chamberlain and M.J.D. Powell, QR factorization for linear least squares problems on the Hypercube, Tech. Rep. CCS 86/10, Dept. of Science and Technology, Christian Michelson Institute, Bergen, Norway, 1986.

[5] E. Chu and A. George, QR factorizations of a dense matrix on a Hypercube multiprocessor, Tech. Rep. ORNL/TM-10691, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN, 1988.

[6] S.C. Eisenstat, M.T. Heath, C.S. Henkel, and C.H. Romine, Modified cyclic algorithms for solving triangular systems on distributed memory multiprocessors, *SIAM J. Sci. Stat. Comput.* **9** (1987) 589-600.

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

[8] G. Li and T.F. Coleman, A new method for solving triangular systems on distributed memory message-passing multiprocessors, Tech. Rep. TR 87-812, Dept. of Computer Science, Cornell University, Ithaca, NY, 1987.

[9] A.P. Morgan and L.T. Watson, A globally convergent parallel algorithm for zeros of polynomial systems, Tech. Rep. TR-86-25, Dept. of Computer Science, VPI&SU, Blacksburg, VA, 1986.

[10] A.P. Morgan and L.T. Watson, Solving polynomial systems of equations on a hypercube, *Proc. Hypercube Multiprocessors,* M. T. Heath, ed., SIAM, Philadelphia, PA, (1987) 501–511.

[11] W. Pelz and L.T. Watson, Message length effects for solving polynomial systems on a hypercube, *Parallel Computing* **10** (1989) 161-176.

[12] A. Pothen, J. Somesh, and U. Vemulapati, Orthogonal factorizations on a distributed multiprocessor, *Proc. Hypercube Multiprocessors,* M. T. Heath, ed., SIAM, Philadelphia, PA, (1987) 587–596.

[13] A. Pothen and P. Raghaban, Distributed Orthogonal Factorizations, *Proc. Third Conference on Hypercube Concurrent Computers and Applications,* Pasadena, CA, (ACM, 1988) 1610–1620.

[14] W.C. Rheinboldt and J.V. Burkardt, Algorithm 596: A program for a locally parameterized continuation process, *ACM Trans. Math. Software* **9** (1983) 236–241.

[15] C.Y. Wang, Buoyant rotating disc, manuscript and private communication (1988).

[16] L.T. Watson, A globally convergent algorithm for computing fixed points of C^2 maps *Appl. Math. Comput.* **5** (1979) 297–311.

[17] L.T. Watson, Numerical linear algebra aspects of globally convergent homotopy methods, *SIAM Rev.* **28** (1986) 529–545.

[18] L.T. Watson, S.C. Billups and A.P. Morgan, Algorithm 652: HOMPACK: A suite of codes for globally convergent homotopy algorithms *ACM Trans. Math. Software* **13** (1987) 281–310.