Analysis of Function Component Complexity for Hypercube Homotopy Algorithms

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

TR 90-27

Analysis of Function Component Complexity for Hypercube Homotopy Algorithms

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

Index Terms—Function component complexity, globally convergent homotopy, hypercube, nonlinear equations, parallel algorithm.

Analysis of Function Component Complexity for Hypercube Homotopy Algorithms

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

May 23, 1990

Abstract.

Probability-one homotopy algorithms are a class of methods for solving nonlinear systems of equations that are globally convergent from an arbitrary starting point with probability one. The essence of these homotopy algorithms is the construction of a homotopy map ρ_a and the subsequent tracking of a smooth curve γ in the zero set $\rho_a^{-1}(0)$ of ρ_a. Tracking the zero curve γ requires repeated evaluation of the map ρ_a, its $n \times (n+1)$ Jacobian matrix $D\rho_a$, and numerical linear algebra for calculating the kernel of $D\rho_a$. This paper analyzes parallel homotopy algorithms on a hypercube, considering the numerical linear algebra, several communication topologies and problem decomposition strategies, function component complexity, problem size, and the effect of different component complexity distributions. These parameters interact in complicated ways, but some general principles can be inferred based on empirical results.

1. Introduction.

Algorithms for solving nonlinear systems of equations may be broadly classified as locally or globally convergent. Locally convergent methods include Newton's method and various modifications of Newton's method. Globally convergent methods include continuation, simplicial methods, and probability-one homotopy methods. The overall purpose of this research is to design efficient algorithms for solving systems of nonlinear equations using probability-one homotopy methods on a hypercube. Previous papers have addressed the computational linear algebra aspects of parallel homotopy algorithms in considerable detail [8],[9],[10]. Here we study another, often more expensive step in the homotopy approach, namely, evaluation of the Jacobian matrix of the homotopy map. Predicting and analyzing the performance of parallel algorithms for the linear algebra computations is relatively straightforward: the performance depends "only" on the size of the linear systems, the data distribution, and the communication and synchronization requirements. In evaluating the Jacobian matrix, however, one has to deal with additional complicating factors. For example, it is possible that the computational complexity of evaluating the various components of the Jacobian matrix differs significantly from row to row, or even from component to component. There may also be varying degrees of dependence among the components (e.g., the components of a column may depend on several common subexpressions). Furthermore, these characteristics are usually not accurately known *a priori*. In this paper we study efficient parallel Jacobian matrix evaluation by considering several hypothetical but realistic distributions for component complexity.

Both static and dynamic assignment of work to processors are studied, for various matrix sizes and cost distributions. We also consider briefly a realistic problem from fluid mechanics.

Our focus here is on general nonlinear systems of equations with small and dense Jacobian matrices. Polynomial systems are not considered here since they have a very special structure which leads to different strategies for parallelism [2], [3], [4], [5], [18], [19], and [21]. Large sparse problems also call for different approaches [16].

Section 2 summarizes briefly the mathematical theory behind the homotopy approach. In Section 3 we review parallel algorithms for the computational linear algebra steps of these algorithms, namely orthogonal factorization and triangular system solving. Section 4 discusses various possibilities for parallel Jacobian matrix evaluation, and presents our computational results and a discussion thereof. Some concluding remarks are included in Section 5.

2. Theory of globally convergent homotopy algorithms.

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 fundamental problem under consideration is to solve the nonlinear system of equations

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

The mathematics behind homotopy algorithms [28], [29] for solving (1) is summarized in

Theorem. *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. If $G(x) = 0$ is a simple problem with a unique, easily obtainable solution W, then a typical form for the homotopy map is

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

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 $G(x) = 0$ 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, the theorem 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 the simple convex combination of $F(x)$ and $G(x)$ in (2) is a special case) can be tracked by many different techniques; refer to the excellent survey [1] and recent work [28], [29]. 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 [6], [24], (an "augmented Jacobian matrix" method). Alternatives 1), 2) and 3) are described in detail in [28], [29] and [6] respectively.

All of these tracking algorithms need a unit tangent vector at different points along the zero curve. Because of the supporting mathematical theory, finding a unit tangent vector amounts to finding the one dimensional kernel of the $n \times (n+1)$ Jacobian matrix $D\rho_a$, which 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 (this is not possible for sparse matrices, which are not being considered here). There are kernel finding algorithms based on Gaussian elimination and n distinguished columns, but computational experience has shown that 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. A vector $z \in \ker(D\rho_a)$ is then obtained by solving $R(Pz) = 0$ for Pz using back substitution with $(Pz)_{n+1} = 1$. This scheme provides high accuracy, numerical stability, and a uniform treatment of all $n+1$ columns.

The kernel of the Jacobian matrix $D\rho_a$ can also be found by computing an LQ factorization of $D\rho_a$, where L is $n \times (n+1)$ lower triangular, and Q is again a product of Householder reflections. 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 kernel computation.

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. Chakraborty et al. [10] discussed the parallel computation of steps 2) and 3). Since the discussion here of Jacobian matrix evaluation refers to some of the results previously reported in [10], we repeat a portion of the results discussed in [10].

3.1. 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 [11], [12], [22], and [23]). 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. For the analyses here it was necessary to examine existing orthogonal factorization algorithms and incorporate necessary column switching so that they could be used to track a homotopy curve. An interesting side issue, touched on briefly here and discussed more fully in [10], is a comparison of QR and LQ factorizations. The two algorithms described in [12] 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 [12].

In the first algorithm of [12], 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.

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

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 matrix $D\rho_a$). 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 initializes the array sum using a multinode accumulation.

Description of the factorization algorithm, consisting of an independent annihilation phase (IAP) and a cooperative merging phase (CMP), can be found in [10].

In the second algorithm of [12], 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. Matrix rows are assigned to subcubes corresponding to grid rows in a wrap-mapping fashion. Elements of each matrix row, in turn, are assigned to the processors in the corresponding row subcube in a wrap-mapping fashion. Thus a single processor does not hold a complete row or column of the matrix. Instead, each row subcube holds a complete matrix row, and each column subcube holds a complete matrix column. Figure 3 shows the wrap-mapping of a 8×9 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.

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

Figure 3. The wrap mapping of a 8×9 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 $((n+1)/p+1)(n+1)$ extra numbers for $\mathcal{Q}$, where p is the number of processors.

Table 1 shows the computational results obtained on a 32 node Intel hypercube. All the matrices were $n \times (n+1)$ with full rank. The notation used in the tables is as follows:

Table 1. 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

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 $\lambda_1 \times \lambda_2$ 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 $\lambda_1 \times \lambda_2$ rectangular grid;

It can be seen from Table 1 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 Table 1 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 Table 1 that the algorithms based on a grid of processors are more efficient than those that organize the processors in a ring, for all but the smallest problems.

4. Jacobian matrix evaluation.

The Jacobian matrix and function evaluations are often the most time consuming part of a homotopy algorithm. However, they are also the most easily parallelized part of the computation. Byrd et al. [7] and Schnabel [25] discuss parallel function evaluation in the context of unconstrained optimization. Their approach is to compute the function and gradient completely but only part of the Hessian matrix. Each component of the gradient is computed by a single processor and the rest of the processors compute a single component of the Hessian matrix, assuming that the number of processors is greater than $(n+1)$ but less than $(n^2+3n+2)/2$. They do not let any processor compute more than one component. Byrd [7] describes an algorithm that uses a partly computed Hessian matrix and analyzes the convergence properties of that algorithm. The

present work assumes that the complete Jacobian matrix needs to be formed. Since the number of processors is generally less than (n^2+n), each processor must compute more than one component. This section examines the effect of different component complexity distributions and the size of the Jacobian matrix on the different assignments of components to the processors, and determines in what context one assignment would perform better than others.

4.1. Parallel Jacobian matrix evaluation.

For many engineering problems it is not feasible to compute the Jacobian matrix analytically. Most often the Jacobian matrix is estimated 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 (static assignment). Since the orthogonal decomposition is most efficient when a rectangular mapping is used, this assignment will mean that a single column is evaluated by more than one processor. In many cases, this will result in redundant computation and extra communication overhead. If d_1 and d_2 are the dimensions of the subcubes in the rectangular grid, then the maximum number of components that a processor has to compute is

$$\left\lceil \frac{n}{2^{d_1}} \right\rceil \left\lceil \frac{n+1}{2^{d_2}} \right\rceil .$$

Another approach would be to use a linear mapping by column in the evaluation phase and a rectangular mapping in the decomposition phase. In this method, the host will collect the complete matrix from the nodes and then redistribute the matrix according to a rectangular mapping. If the linear mapping by column in the evaluation phase is used, then the maximum number of components that a processor has to evaluate is

$$\left\lceil \frac{n+1}{2^{d_1+d_2}} \right\rceil n.$$

The maximum number of elements that a processor has to compute in the linear mapping case can be greater than or less than the maximum number of components that a processor has to compute in the rectangular mapping case, so there is no clear preference on these grounds. If the Jacobian matrix is not large enough to justify parallel decomposition, both the linear and rectangular mapping algorithms have to return the evaluated Jacobian matrix to the host before serial decomposition can proceed. However, because in this case there there may be fewer columns than processors, processor loads could be unbalanced. If p is the number of processors and n is the column dimension of the Jacobian matrix then $p-n$ processors will be idle in the evaluation phase. Because $n(n+1)$ components can be evaluated in parallel, a rectangular mapping scheme would utilize more processors. When $n >> p$, the load distribution for a linear mapping is not as bad as when $n \approx p$. Because the host has to collect all the columns and redistribute them to the processors, there is some communication involved at the end of the evaluation phase. Also, all the processors have to wait for the Jacobian matrix evaluation to finish. In the case of a rectangular mapping, a significant portion of the IAP phase of the first stage could be completed before the whole Jacobian matrix is evaluated. This may or may not save any time, as all the processors have to be synchronized for the CMP phase. In any event the savings would not be enough to justify

the rectangular mapping if loads are evenly balanced with the linear mapping. This is especially true because redundancy of computation may be high with the rectangular mapping case.

However, in either static assignment, if the variation in the evaluation time of the components is high, there may be uneven loading of the processors and thus inefficiency. In this situation it may be better to use the host as the master and let it assign components to the processors as they become available (dynamic assignment). However, this method has a large communication overhead. Also, when there is a large number of components to evaluate, the variation in the total evaluation time among the processors 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.

4.2. Computational results.

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 little overhead in addition to that already incurred by the decomposition algorithm. When the Jacobian matrix is very small but component evaluation expensive enough to justify parallel evaluation, it is better to do the factorization and triangular system solving serially. Tables 2–10 show timings for different situations. In the tables, R and L refers to a 4×8 rectangular and linear mapping respectively, and DYNAMIC refers to the master/slave model where the host dynamically assigns component evaluations to the nodes. The experiments were done on a 32 node Intel iPSC/1 hypercube. The notation used in the tables is as follows:

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).

If the system has a small and cheap to compute Jacobian matrix then everything should be done serially. The crossover point at which it is better to use a parallel evaluation algorithm depends not only on the complexity of component evaluation but also on the interdependency among the components. Table 2 shows the results when component evaluation is totally independent and the cost of evaluation for each component is the same. Here parallel function evaluation is better than serial evaluation when each component requires at least 0.7 KFLOPS, and very much better for larger values of KFLOPS.

Experiments were done to study the effects of matrix size, cost of component evaluation, and the distribution of cost on static and dynamic assignment algorithms. The cost of component evaluation, ρ, is a random variable whose distribution is varied in these experiments. Tables 3–9 show the mean and variance of timings taken from three runs with different seeds. The cost of each component was generated randomly from the following probability distributions:

1) $U(a,b)$ - uniform distribution with lower bound a KFLOPS and upper bound b KFLOPS;

Table 2. Execution time in seconds for different costs of component evaluation (32 nodes, $n = 11$).

KFLOPS	SFER	PFER
.5	9.4	9.8
.65	9.8	9.9
1	15.5	10.2
2	27.5	10.3
5	55.7	11.8
10	103.5	13.5
40	372.0	24.1

Table 3. Execution time in seconds for different costs of component evaluation (Uniform, 32 nodes, $n = 11$).

Methods	U(0,2)	U(0,4)	U(0,10)	U(0,20)	U(3,27)	U(38,62)	U(88,112)
SERIAL	12.8, .01	21.5, .03	46.6, .06	91.0, 0.3	133, 1.6	443, 3.0	874, 9.0
STATIC(R)	12.4, 0.4	12.8, .03	14.1, .07	16.8, .4	18.1, .60	31.0, .70	49.2, 1.5
STATIC(L)	13.7, .03	15.0, .15	17.4, .22	22.1, 1.33	25.5, 2.5	48.7, 3.0	80.5, 5.5
STATIC(L20)	13.5, .03	14.8, .03	16.1, .04	20.5, .06	23.8, .66	42.3, 1.1	68.0, 1.4
STATIC(L50)	13.2, .03	13.7, .03	14.8, .05	17.4, .22	20.3, .49	33.2, 2.3	48.7, 3.0
DYNAMIC	14.5, .03	14.6, .04	14.8, .04	15.3, .04	16.4, .06	25.9, 1.56	39.1, 2.0

Table 4. Execution time in seconds for different costs of component evaluation (Normal, 32 nodes, $n = 11$).

Methods	N(1,1)	N(2,2)	N(5,5)	N(10,7)	N(15,7)	N(50,7)	N(100,7)
SERIAL	13.1, .04	22.4, .33	52.1, 6.0	96.3, 8.0	141.5, 13.0	451, 25	879, 44
STATIC(R)	12.1, .03	12.8, .13	14.5, 0.8	16.8, 1.7	18.5, 1.8	31.2, 4.7	51.1, 6.0
DYNAMIC	13.2, .06	13.2, .14	14.8, .22	15.2, .66	17.8, 1.1	28.1, 4.7	44.5, 6.3

Table 5. Execution time in seconds for different costs of component evaluation (Exponential, 32 nodes, $n = 11$).

Methods	E(1)	E(2)	E(5)	E(10)	E(15)	E(50)	E(100)
SERIAL	12.8, 1.8	22.5, 1.8	51.8, 6.0	100.3, 12.0	148, 43	502, 104	963, 1400
STATIC(R)	12.3, .06	13.1, .22	15.2, .36	19.1, .94	22.9, 1.6	49.1, 12.0	86.1, 101
DYNAMIC	14.0, 0.7	14.2, 0.7	14.9, 0.7	15.5, 1.1	17.7, 2.2	31.1, 4.3	52.5, 6.0

2) $N(a, b)$ - normal distribution truncated on the left at 0 with mean a KFLOPS and standard deviation b;

3) $E(a)$ - exponential distribution with mean a KFLOPS.

STATIC L20 and STATIC L50 denote algorithms which assume a linear mapping scheme where the computation is 20% and 50% less than for the rectangular mapping. It can be seen from Tables 3, 4, and 5 that when the components are not expensive (mean is less than 10 KFLOPS) to evaluate, dynamic assignments are not as good as static assignments. However, when each component takes

Table 6. Execution time in seconds for different costs of component evaluation (Uniform, 32 nodes, $n = 50$).

Methods	U(0,2)	U(0,4)	U(0,10)	U(0,20)	U(3,27)	U(38,62)	U(88,112)
SERIAL	196.5, 1.3	377.9, 2.2	925.3, 4.3	1838, 8.0	2742, 12.0	9014, 18	18008, 70
STATIC(R)	27.7, .03	32.0, .04	47.3, .06	78.7, 1.1	104.5, 1.1	294, 2.0	563, 3.0
STATIC(L)	29.4, .04	39.1, .06	61.0, .66	94.0, 2.0	125.0, 4.7	334, 8.0	632, 13.0
STATIC(L20)	31.3, .04	36.7, .04	53.0, .06	92.6, .22	107.4, 1.3	274, 4.7	511, 6.0
STATIC(L50)	29.3, .03	29.4, .04	42.7, .06	60.0, .66	76.4, .73	195, 1.3	365, 8.0
DYNAMIC	95, .04	97.0, .04	99.4, .06	107.5, .66	109, 1.3	256, 2.2	491, 6.7

Table 7. Execution time in seconds for different costs of component evaluation (Normal, 32 nodes, $n = 50$).

Methods	N(1,1)	N(2,2)	N(5,5)	N(10,7)	N(15,7)	N(50,7)	N(100,7)
STATIC(R)	23.5, .06	29.1, .06	44.9, .22	72.2, 1.1	97.8, 1.3	286.3, 1.7	570.7, 2.2
DYNAMIC	94, .04	96.8, .04	98.7, .06	99.8, .50	106.5, 1.1	256.1, 1.7	490.3, 4.0

Table 8. Execution time in seconds for different costs of component evaluation (Exponential, 32 nodes, $n = 50$).

Methods	E(1)	E(2)	E(5)	E(10)	E(15)	E(50)	E(100)
STATIC(R)	26.6, .06	32.5, .22	48.9, .66	77.2, 1.3	109.6, 2.2	314, 8.0	607, 13.3
DYNAMIC	94.2, 1.1	96.4, 1.1	98.7, 1.3	100.2, 2.2	108, 6.0	269, 12.0	518, 50.0

Table 9. Execution time in seconds for different costs of component evaluation (Uniform, 32 nodes, $n = 98$).

Methods	U(0,2)	U(0,4)	U(0,10)	U(0,20)	U(3,27)	U(38,62)	U(88,112)
STATIC(R)	60.7, .66	82.5, 1.1	142, 1.3	240, 2.2	339, 6.6	1015, 8.0	1982, 16.0
STATIC(L)	74.0, 1.1	99.4, 1.1	167.0, 1.6	286.0, 3.0	422.0, 8.0	1204, 12.0	2397, 51.0
STATIC(L20)	70.5, .70	89.5, 1.1	149.0, 1.4	241.5, 2.2	335.5, 6.6	991, 8.0	1925, 18.0
STATIC(L50)	63.5, .66	75.5, 1.1	113.0, 1.1	170.0, 1.6	230.0, 1.7	639, 8.0	1221, 12.0
DYNAMIC	434.0, 2.2	434.6, 2.2	435.5, 2.2	438.0, 2.2	441, 3.1	803, 6.6	1836, 16.0

more than 10 KFLOPS, dynamic assignments are performing better than static assignments. This is more apparent when component evaluation times are exponentially distributed (Table 5). In this case the variation in computation time is high and thus static assignment results in a poor load balancing among the nodes. Tables 6, 7, and 8 show the results for a 50×51 Jacobian matrix. For low variation situations (Tables 6 and 7), the dynamic assignment is performing better than static assignments only when component evaluation takes at least 50 KFLOPS. This is true because there are more assigned components per processor for the 50×51 matrix than for the 11×12 Jacobian matrix. Thus the load balancing among the nodes is not as uneven. However, when component evaluation time is exponentially distributed (high variation), the dynamic assignment (Table 8) is performing better than static assignment even at the mean component cost of 15 KFLOPS.

Table 10. 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

Tables 3, 6, and 9 show the relative performances of the 4×8 rectangular mapping versus the 32-node ring linear mapping. It can be seen that as the matrix size increases the linear mapping becomes more competitive. Table 6 shows that the linear mapping will outperform the rectangular mapping if the computation can be reduced by half. Table 9 shows that for a 98×99 matrix the linear mapping will do better if the computations can be reduced by 20% and mean component complexity is more than 15 KFLOPS. This is in contrast to the 50×51 (Table 6)case where the component complexity has to be at least 50 KFLOPS for STATIC(L20) to do better than STATIC(R). These results are not exactly predictable if we consider the amount of computation that processor 0 has to perform (in either mapping processor 0 will have maximum load). When $n = 11$, processor 0 has to compute 6 components in the 4×8 rectangular mapping case whereas it has to compute as many as 11 components in the linear mapping by column case. When $n = 50$ and $n = 98$ these numbers are $(91, 100)$ and $(325, 392)$ respectively. This is because the computation per component is more even among the nodes when $n = 98$.

Table 10 shows some timing results for Jacobian matrix evaluation in combination with factorization and triangular solving for systems with small to medium Jacobian matrices. The matrices were obtained from a Galerkin approximation to a buoyant rotating disc fluid mechanics problem [26]. Each component was approximately 2 KFLOPS. However, they were not expensive enough to justify dynamic assignment, so a rectangular assignment was used. The Jacobian matrix was computed using central finite difference approximations. Thus there were $2(n^2+n)$ function component evaluations. 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 close to 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.

When the system has a large enough ($n \geq 50$) Jacobian matrix, Jacobian matrix evaluation and kernel computation should all be done in parallel. If the system has a small but very expensive Jacobian matrix, then function and Jacobian matrix evaluation should be done in parallel but the kernel computation may be done serially. This can be observed from Table 10, where for $n \leq 35$ PFUN is slightly better than PKER.

5. Conclusions.

The problem dimension, matrix to hypercube mapping, function component complexity, and the statistical distribution of the function component complexities all interact with each other in complicated ways. Short of conducting expensive experiments or analysis to measure these

quantities, it may be impossible to predict *a priori* the optimal choices. The detailed discussion of the computational results in the previous section cannot be succinctly summarized with complete precision, but some general principles can be inferred. 1) The parallel QR factorization is better than the parallel LQ factorization. 2) A master/slave paradigm will be advantageous for only a restricted class of problems – small dimension, and very expensive components from a heavy-tailed distribution. 3) Parallel function evaluation is preferred for all but the smallest ($n < 10$) problems with cheap component evaluation costs (< 0.7 KFLOPS). 4) If the function and Jacobian matrix component evaluations are mostly independent (i.e., there are few computations common to several components), the rectangular hypercube mapping is better than the linear wrap-mapping. If the level of redundancy (shared expressions or function values) between the components is high, the reverse is true. 5) Performance cannot be predicted by simple measures like the maximum number of components that any processor has to evaluate. 6) In a large heterogeneous calculation (like homotopy curve tracking), optimizing the individual parts (such as function and Jacobian matrix evaluation, matrix factorization, triangular system solving) may result in a *far from optimal* overall algorithm. 7) Even for apparently inherently sequential homotopy algorithms applied to small ($n \approx 20$) nonlinear systems of equations, the hypercube architecture yields a moderate speedup.

6. Acknowledgement.

The work of Chakraborty, Ribbens, and Watson was supported in part by Department of Energy grant DE-FG05-88ER25068, NASA grant NAG-1-1079, National Science Foundation grant CTS-8913198, and Air Force Office of Scientific Research grant 89-0497.

References.

[1] E. Allgower and K. Georg, "Simplicial and continuation methods for approximating fixed points", *SIAM Rev.*, **22** (1980) pp. 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) pp. 1463–1472.

[3] D. C. S. Allison, S. Harimoto, and L. T. Watson, "The granularity of parallel homotopy algorithms for polynomial systems of equations", *Internat. J. Comput. Math.*, 29 (1989) pp. 21–37.

[4] D. C. S. Allison, S. Harimoto, and L. T. Watson, "The granularity of parallel homotopy algorithms for polynomial systems of equations", *Proc. 1988 Internat. Conf. on Parallel Processing*, Vol. III, D. H. Bailey (ed.), St. Charles, IL, 1988, pp. 165–168.

[5] D. C. S. Allison, A. Chakraborty, and L. T. Watson, "Granularity issues for solving polynomial systems via globally convergent algorithms on a hypercube", *J. of Supercomputing*, **3** (1989) pp. 5–20.

[6] 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.

[7] R. H. Byrd, R. B. Schnabel, and G. A. Shultz, "Using parallel function evaluation to improve Hessian approximation for unconstrained optimization", Tech. Rep. CS-CU-361-87, Dept. of Computer Science, University of Colorado, Boulder, Colorado 80309, 1987.

[8] A. Chakraborty, D. C. S. Allison, C. J. Ribbens, and L. T. Watson, "Parallel orthogonal decompositions of rectangular matrices for curve tracking on a hypercube", *Proc. Fourth Conf. on Hypercube Concurrent Computers and Applications,* J. Gustafson (ed.), ACM, Monterey, CA, 1989, pp. 651–654.

[9] A. Chakraborty, D. C. S. Allison, C. J. Ribbens, and L. T. Watson, "Parallel Homotopy curve tracking on a hypercube", *Proc. 1989 Fourth SIAM Conference on Parallel Processing for Scientific Computing,* Chicago, 1989.

[10] A. Chakraborty, D. C. S. Allison, C. J. Ribbens, and L. T. Watson, "Unit tangent vector computation for homotopy curve tracking on a hypercube", *Parallel Comput.,* submitted.

[11] 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.

[12] 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.

[13] 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) pp. 589–600.

[14] S. Harimoto and L. T. Watson, "The granularity of homotopy algorithms for polynomial systems of equations", *Parallel Processing for Scientific Computing,* G. Rodrigue (ed.), SIAM, Philadelphia, PA, 1989, pp. 115–120.

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

[16] K. M. Irani, C. J. Ribbens, H. F. Walker, L. T. Watson and M. P. Kamat, "Preconditioned conjugate gradient algorithms for homotopy curve tracking", *SIAM J. Optim.,* (submitted.)

[17] 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.

[18] A. P. Morgan and L. T. Watson, "A globally convergent parallel algorithm for zeros of polynomial systems", *Nonlinear Anal.,* **13** (1989) pp. 1339–1350.

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

[20] A. P. Morgan and L. T. Watson, "Solving nonlinear equations on a hypercube, *Super and Parallel Computers and Their Impact on Civil Engineering",* M. P. Kamat (ed.), ASCE Structures Congress '86, New Orleans, LA, 1986, pp. 1–15.

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

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

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

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

[25] R. B. Schnabel, "Concurrent function evaluations in local and global optimization", Tech. Rep. CS-CU-345-86, Dept. of Computer Science, Univ. of Colorado, Boulder, Colorado 80309, 1986.

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

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

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

[29] 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) pp. 281–310.