Analysis of Function Component Complexity for Hypercube Homotopy Algorithms

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

TR 91-4

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

February 9, 1991

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, briefly reviewing the numerical linear algebra, several communication topologies and problem decomposition strategies, and concentrating on 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. Implications for developing reliable and efficient parallel mathematical software packages for this problem area are also discussed.

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, and to eventually build a production quality math software package which will provide reasonable efficiency and reliability across a range problems and hypercube machines. Previous papers have addressed the computational linear algebra aspects of parallel homotopy algorithms in considerable detail [10],[11],[12]. Here we study another, often more expensive step in the homotopy approach, namely, the function evaluations needed to construct the Jacobian matrix of the homotopy map. Predicting and analyzing the performance of parallel algorithms for the linear algebra computations is relatively straightforward: performance depends "only" on the size of the linear systems, the data distribution, and on 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], [16], [19], [20] and [22]. Large sparse problems also call for different approaches [15], [17].

Section 2 summarizes briefly the mathematical theory behind the homotopy approach, and reviews parallel algorithms for the computational linear algebra aspects of these algorithms, namely orthogonal factorization and triangular system solving. Section 3 discusses various possibilities for parallel Jacobian matrix evaluation, presenting computational results on a 32 node Intel iPSC/1 hypercube and a discussion thereof. Some concluding remarks are made in Section 4.

2. Homotopy theory, algorithms, and linear algebra.

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

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

where $F : E^n \to E^n$ is a C^2 (twice continuously differentiable) function defined on n-dimensional real Euclidean space E^n. The mathematics behind homotopy algorithms [26], [27] 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.

A 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

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

Although (2) has the same form as a standard continuation or embedding mapping, 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 [26], [27]. 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], [22], (an "augmented Jacobian matrix" method). Alternatives 1), 2) and 3) are described in detail in [26], [27] and [6] respectively.

Each of these tracking algorithms requires a unit tangent vector at different points along the zero curve. By the supporting mathematical theory, finding a unit tangent vector amounts to computing the one dimensional kernel of the $n \times (n+1)$ Jacobian matrix $D\rho_a$, which has (theoretical) rank n. Parallel algorithms for this kernel computation have received considerable attention [10], [11], [12]. The best approach depends strongly on the characteristics of the Jacobian matrix. For the applications of interest here (general nonlinear systems with relatively small, dense Jacobian matrices), and in order to achieve the required accuracy, an orthogonal factorization of $D\rho_a$ is required.

In [12], parallel algorithms for both QR and LQ factorizations of the Jacobian matrix are considered, as well as two strategies for mapping the data to the processors of a hypercube. Significant complications in the comparison of QR and LQ include the need to do column pivoting and a triangular system solve for QR, and the need to explicitly form the matrix Q in the LQ factorization. The results and analysis in [12] indicate that the QR factorization is preferred over the LQ. The algorithms in [12] are based on work of Chu and George [13]. Recent work combining pivoting and orthogonal factorizations [7], [8], [14] may be applicable in our context as well, and we are considering these possibilities.

Two strategies for mapping the data to the processors are also considered in [12]. One approach is to organize the processors in a linear ring, and assign rows of the Jacobian matrix to the processors in a wrap-mapping fashion. Thus, processor 0 gets row 1, processor 1 gets row 2, etc.. The second approach is to organize the processors in a $2^{d_1} \times 2^{d_2}$ rectangular grid, where each row and column of processors forms a subcube, and the dimension of the hypercube is $d = d_1 + d_2$. Rows of the Jacobian matrix 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. Of these two data mapping strategies, the rectangular mapping is seen in [12] to be the most efficient for orthogonal factorization and kernel computation, for the types of problems under consideration.

3. Jacobian matrix evaluation.

Despite the significance of the linear algebra summarized above (orthogonal factorization and triangular solution), the function evaluations needed to simply construct the Jacobian matrices are often the most time consuming part of a homotopy algorithm. Fortunately, they are often the most easily parallelized part of the computation as well. Byrd et al. [9] and Schnabel [24] 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 each of the remaining processors computes 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 [9] describes an algorithm that uses a partly computed Hessian matrix, and analyzes the convergence properties of that algorithm.

In the present work we assume that the complete $n \times (n+1)$ Jacobian matrix needs to be formed, and that each component of the Jacobian can be computed independently (although some redundant computation may be required). Since the number of processors is generally less than (n^2+n), each processor must compute more than one component. In this section we consider several strategies for assigning components to processors. Section 3.1 discusses two static and one dynamic assignment strategy, and comments on their expected performance. Section 3.2 presents results from several computational experiments on a 32 node Intel iPSC/1 hypercube. We first consider the basic question of when parallel Jacobian evaluation is even appropriate. Next, we examine the effect of different component complexity distributions, and of Jacobian matrix size, on the different assignment strategies, determining experimentally in what cases one assignment performs better than the others. Finally, we report performance for the entire kernel computation on a test problem from fluid mechanics.

3.1. Component assignment strategies.

Strategies for assigning components to processors may be classified as static or dynamic, depending on whether the assignment is made *a priori* or at runtime. We consider one dynamic and two static assignments in the experiments described below. In the first static assignment the distribution is determined by the requirements of the subsequent linear algebra. That is, each processor computes exactly those components assigned to it during the orthogonal decomposition phase. As mentioned in Section 2, the most efficient mapping for the orthogonal decomposition is the "rectangular" mapping, where processors are organized in a rectangular grid, with each row and column of processors forming a subcube. This means that a single column of the Jacobian matrix is evaluated by more than one processor. The second static assignment strategy is based on what may be most efficient for the Jacobian evaluation itself. Thus, a linear wrap-mapping by columns is used, so that an entire column is located on a single processor. In many cases, considerable redundant computation can be avoided if a single processor has responsibility for an entire column. For example, it is typical in many applications to have many common subexpressions in the components of a given column. Obviously, this cannot be fully exploited if a column is distributed over a subcube. See [19] for a good example of this situation in the context of polynomial systems of equations.

Beyond the issue of redundant computation, a few other points of comparison between the two static strategies should be made before turning to the experimental results. Clearly, a significant potential disadvantage of the linear mapping is the data rearrangement required if the subsequent QR factorization is done using a rectangular mapping. An efficient algorithm for moving the data from the linear column mapping to the rectangular mapping is nontrivial at best; and the straightforward approach of returning all the data to the host for redistribution would clearly be a very serious bottleneck for all but the smallest problems. In [12] parallel QR factorization with a linear mapping was considered, but found to be considerably less efficient than the rectangular mapping. Furthermore, the algorithm in [12] requires a linear mapping by rows and not columns, so significant data movement would still be required (the linear column mapping to linear row mapping is essentially a matrix transposition [18]). With rectangular mapping for Jacobian matrix evaluation however, no data movement is required before the QR factorization begins. In fact, a "pipelined" strategy could be considered, in which the first phase of the QR factorization begins as soon as the required components are computed, but before the entire Jacobian matrix is computed. The potential gain here does not appear to be substantial however, at least with our current formulation of QR. One final important point of comparison between the two static algorithms deals with load balancing. A very rough measure of load balancing is the maximum number of components assigned to a single processor under each scheme. If d_1 and d_2 are the dimensions of the subcubes in the rectangular grid, and the Jacobian matrix is $n \times (n+1)$, then the maximum number of components a processor has to compute under the linear and rectangular mapping is respectively

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

Which strategy is favored by these expressions is not immediately obvious, but it clearly depends on problem size and cube (and subcube) dimension. We address load balancing issues more carefully in Section 3.2 below.

In either of the static assignments, if variation in the evaluation times of the components is high there may be load imbalance. In this situation it may be better to use a dynamic assignment strategy in which the host functions as a master, assigning components to slave processors as they become available. This approach requires a large communication overhead during Jacobian matrix evaluation, in addition to the added cost of redistributing the data for the QR factorization. Also, when there is a large number of components relative to the number of processors, the variation in total evaluation time among the processors is not as likely to be high. Thus the master/slave paradigm is likely to be advantageous only when the Jacobian matrix is relatively small, some components are very expensive to evaluate, and there is a large variation in evaluation times. We make these comments more precise through the experimental results in the next section.

3.2. Computational results.

Serial vs. parallel evaluation.

Since a longterm goal of this research is to produce a robust, reliable, efficient version of the math software package HOMPACK [27], it is important to consider a question that is often overlooked, namely, "Should a (sub)problem be run in parallel at all?". This is a surprisingly difficult question to answer in the context of a large package applied to a wide range of problems

Table 1. Serial and parallel execution time (in seconds) for different costs of component evaluation (in KFLOPS), for a small Jacobian matrix ($n = 11$).

Cost	Serial	Parallel
0.50	9.4	9.8
0.65	9.8	9.9
1.00	15.5	10.2
2.00	27.5	10.3
5.00	55.7	11.8
10.00	103.5	13.5
40.00	372.0	24.1

on a wide range of machines. Obviously, the Jacobian matrix may be large or small, with each component cheap or expensive to compute; and this is the kind of information the software needs in order to determine how best to use parallelism. 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 may be better to do the factorization and triangular system solving on a small subcube, or perhaps even 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. A simple example of Jacobian matrix evaluation for small n is given in Table 2. Here it is assumed that component evaluation is totally independent and the cost of evaluation for each component is the same. The serial evaluation is done on the host; the parallel evaluation is done using the static rectangular mapping, with a 4×8 processor grid. We see that for this case, parallel function evaluation is better than serial evaluation when each component requires more than about 0.7 KFLOPS, and very much better for more expensive components. The crossover point would be higher if the components had significant interdependencies. While the exact numbers are obviously machine dependent, the point is that such a crossover point always exists, and a math software package that claims to be efficient across a range of problems cannot ignore this.

Effect of component cost distribution on assignment strategies.

Experiments were done to study the effects of matrix size, cost of component evaluation, and the distribution of cost on static and dynamic assignment strategies. The cost of evaluating each component is taken as a random variable with various distributions. Tables 2–8 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;
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.

In the tables, R and L refer to a 4×8 rectangular mapping and a linear column mapping respectively; DYNAMIC refers to the master/slave strategy. STATIC(L20) and STATIC(L50) denote algorithms

Table 2. Execution time in seconds (mean and variance) for various assignment strategies and component cost distributions (Uniform, $n = 11$).

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 3. Execution time in seconds (mean and variance) for various assignment strategies and component cost distributions (Normal, $n = 11$).

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 4. Execution time in seconds (mean and variance) for various assignment strategies and component cost distributions (Exponential, $n = 11$).

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

Table 5. Execution time in seconds (mean and variance) for various assignment strategies and component cost distributions (Uniform, $n = 50$).

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

which assume a linear mapping where the computation is 20% and 50% less than for the rectangular mapping, respectively. For example, if the random variable generates a component cost of 10 KFLOPS, the STATIC(L20) case assumes the cost is only 8 KFLOPS due to savings from assigning an entire column to a single processor.

First consider a comparison between the two static assignment schemes. It can be seen from Tables 2, 5, and 8 that as the matrix size increases the linear mapping becomes quite competitive, especially for fairly expensive components and when savings are assumed (L20 and L50). Table 5

Table 6. Execution time in seconds (mean and variance) for various assignment strategies and component cost distributions (Normal, $n = 50$).

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 7. Execution time in seconds (mean and variance) for various assignment strategies and component cost distributions (Exponential, $n = 50$).

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 8. Execution time in seconds (mean and variance) for various assignment strategies and component cost distributions (Uniform, $n = 98$).

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

shows that for $n = 50$ the linear mapping outperforms the rectangular mapping in all cases if there is 50% savings, and in cases with mean component cost at lest 50 KFLOPS if there is 20% savings. Table 8 shows that for a 98×99 Jacobian matrix the linear mapping again does considerably better in all cases with 50% savings, and only 15 KFLOPS are needed before STATIC(L20) is an improvement over STATIC(R).

In fact, the results for the linear mapping on $n = 98$ would be even better if not for an inherent load balancing problem afflicting the linear mapping for moderate n. It is not hard to see that for very small n the linear mapping suffers load imbalance. In fact, for $n < p$, where p is the number of processors, $p - n$ processors will be idle during the evaluation phase. Even for n somewhat larger than p however, the load balancing is much more sensitive to how evenly p happens to divide $n + 1$ for the linear mapping than for the rectangular. Figure 1 illustrates this for a simple case. Here we plot the maximum number of components assigned to a single processor, and an estimate of utilization, vs. n, for $p = 32$ processors. Our estimate of utilization is

$$\text{util} = \frac{1}{p} \sum_{i=0}^{p-1} \frac{k_i}{k_0},$$

where k_i is the number of components assigned to processor i, and k_0 is the maximum number assigned to any processor. It is clear that the rectangular mapping generally has an advantage for n of modest size in that it is much less sensitive to the particular value of n. With $n =$

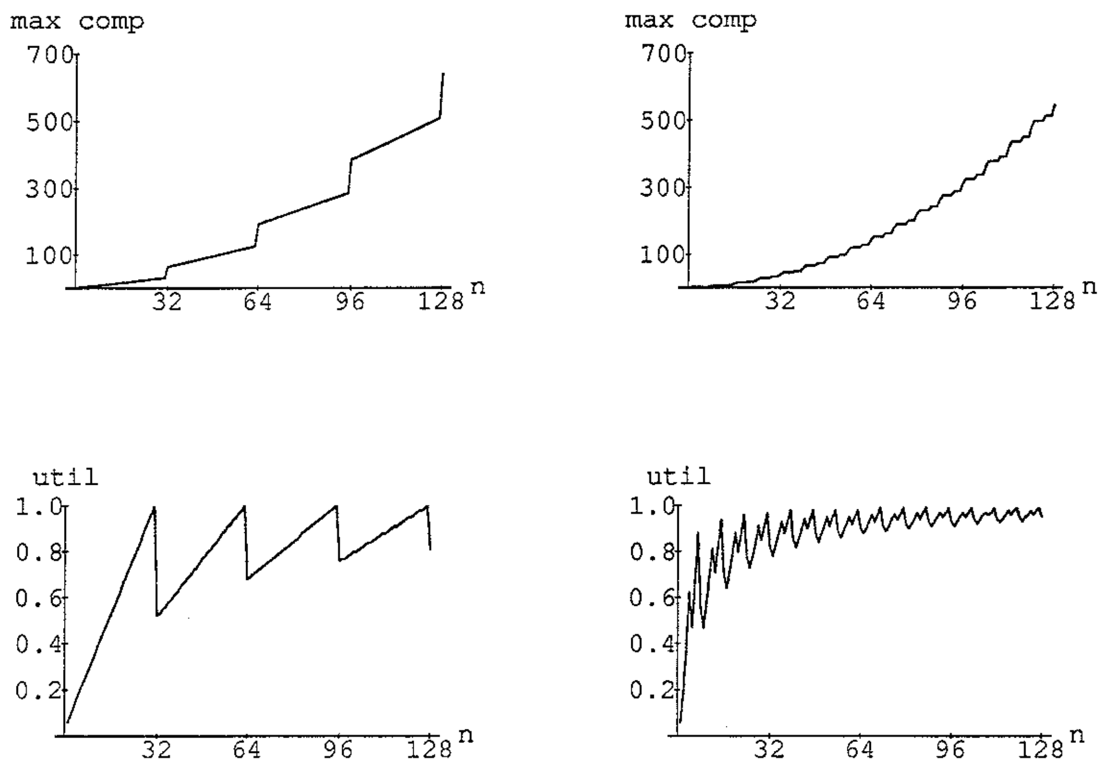

Figure 1. Maximum number of components per processor and utilization vs. matrix size n, for linear (left) and rectangular (right) static mappings.

50 the utilization for linear mapping is 0.80 ($k_0 = 100$) and for rectangular mapping it is 0.88 ($k_0 = 91$); but for $n = 98$, the corresponding figures are 0.77 ($k_0 = 392$) for linear and 0.93 ($k_0 = 325$) for rectangular. Based on these numbers, one would expect the linear mapping to be more competitive for $n = 50$. However, this data takes no account of variation in the costs of the components. As n grows, variation in average component cost for a given processor is less. Hence, although the difference in our estimate of utilization between linear and rectangular mapping is more pronounced for $n = 98$, the larger number of components actually compensates for this and makes load imbalance less of a problem.

We make one final comment regarding static assignment strategies. Careful examination of the data in Table 5 reveals superlinear speedups for several of the cases with large mean component costs (e.g., U(88,112)). There are two contributing factors. One is the lack of sufficient memory on the host for large problems (this problem fits on the host, but it runs more efficiently when the local memories of 32 node processors are all being used). The second reason is that we have assumed no savings for the serial times. Hence, comparing STATIC(L20) and STATIC(L50) directly with SERIAL is misleading. Note that some savings could be realized for the rectangular mapping as well, since a single processor will have responsibility for a large portion of a column (one-fourth of a column in our experiments). We have not assumed this savings in our experiments.

Table 9. Execution time in seconds for Jacobian matrix evaluation and kernel computation for a problem from fluid mechanics.

	Jacobian Evaluation		Kernel Computation		
n	Serial	Parallel	Serial	Par+Ser	Parallel
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

Regarding the comparison between static and dynamic assignment strategies, we see that for small problems (Tables 2, 3, and 4), when the components are not expensive to evaluate (mean is less than 10 KFLOPS), dynamic assignment is not as good as the static assignments. However, when each component takes more than 10 KFLOPS, dynamic assignment performs better than static assignments. This is most evident when component evaluation times are exponentially distributed (Table 4). In this case the variation in computation time is high and thus static assignment results in a poor load balancing among the nodes. Tables 5, 6, and 7 show the results for a 50×51 Jacobian matrix. For low variation cases (Tables 5 and 6), 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 performs better than static assignment even at mean component cost 15 KFLOPS (see Table 7). Finally, in Table 8 we again see that larger n means dynamic assignment is a bad idea, unless the cost of the individual components is very high. Even in the case most favorable to dynamic assignment (small n, large variation in cost, some very expensive), the advantage over STATIC(R) is only about 64% (see column E(100) in Table 4). This may not be enough of an advantage to compensate for the extra complexity and overhead of the master/slave approach, as well as the cost of redistributing the data for subsequent orthogonal factorization. The dynamic approach might be more attractive on a machine with relatively improved communication capabilities (e.g., Intel iPSC/2), or if several masters were used (each controlling a subcube of slaves, for example). But cases where a huge advantage would be realized by using the dynamic approach would still seem to be the exception rather than the rule. A comprehensive, robust parallel math software package should probably have a dynamic assignment capability as an option; but the rectangular static assignment appears to be a reasonable default choice.

Kernel computation on a test problem.

Table 9 shows timing results for Jacobian matrix evaluation in combination with factorization and triangular solving, for systems with small to medium sized Jacobian matrices. The matrices are obtained from a Galerkin approximation to a buoyant rotating disc fluid mechanics problem [25]. Each component evaluation requires approximately 2 KFLOPS, and there is little variation among components. Since the cost and variation is not great enough to justify dynamic assignment, a rectangular assignment is used for the parallel Jacobian matrix evaluations. The columns under

"Kernel Computation" report time for the entire computation: Jacobian matrix evaluation, function evaluations for the right hand side, QR factorization, and triangular system solving. Column "Par+Ser" reports time for parallel Jacobian matrix evaluation combined with serial orthogonal factorization and triangular solution. The Jacobian matrix is computed using central finite difference approximations. Thus $2(n^2 + n)$ function component evaluations are required. Although no communications are involved in the function evaluations (except minor communication at the beginning and at the end), some computations are duplicated. This redundancy explains why a speedup of 32 is not achieved. For this problem the function evaluation time dominates the unit tangent vector computation time and thus parallel function evaluation contributes most to the speedup. We see however that there are cases where the entire computation is best done serially, and other cases where only the Jacobian matrix evaluation should be done in parallel. While the differences in performance are not great, it does suggest that a general parallel software package can not always assume that every step of the computation should be done in parallel.

4. Conclusions.

Machine characteristics, problem dimension, matrix to hypercube mapping, function component complexity, degree of redundancy in component evaluations, 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. However, in an ideal case, a user may be able to supply estimates of some of these factors, enabling a sophisticated math software package to exploit parallelism in the most efficient way. More realistically, in our context we are considering taking advantage of the iterative nature of homotopy algorithms. That is, since the overall solution process requires Jacobian matrices to be evaluated and factored at many points along the curve γ, it is conceivable that the performance of Jacobian evaluation could be monitored, and a strategy adaptively selected which is most appropriate for the problem at hand.

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) 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. 2) Parallel function evaluation is preferred for all but the smallest problems ($n < 10$ for our examples) with cheap component evaluation costs ($<$ 0.7 KFLOPS). 3) If the 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, especially when component cost is high. 4) Performance cannot be predicted by simple measures like the maximum number of components that any processor has to evaluate, although this measure does indicate a slight preference for the rectangular mapping over the static for modest n. 5) In a large heterogeneous calculation like homotopy curve tracking, optimizing the individual parts (e.g., Jacobian matrix evaluation, orthogonal factorization, triangular system solving) may result in a *far from optimal* overall algorithm. 6) Even for apparently inherently sequential homotopy algorithms applied to small ($n \approx 20$ in our examples) nonlinear systems of equations, the hypercube architecture yields a moderate speedup.

5. 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] C. H. Bischof, "QR factorization algorithms for coarse-grained distributed systems", Tech. Report CS-88-939, Computer Science Department, Cornell University, 1988.

[8] C. H. Bischof, "A parallel QR factorization algorithm with controlled local pivoting", *Proceedings Fourth Conference on Hypercube Concurrent Computers and Applications,* J. Gustafson (ed.), ACM, Monterey, CA, 1989, pp. 635–641.

[9] 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.

[10] 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.

[11] A. Chakraborty, D. C. S. Allison, C. J. Ribbens, and L. T. Watson, "Parallel homotopy curve tracking on a hypercube", *Parallel Processing for Scientific Computing,* J. Dongarra, et al. (eds.), SIAM, Philadelphia, 1990, pp. 149-153.

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

[13] E. Chu and A. George, "QR factorization of a dense matrix on a Hypercube multiprocessor", *SIAM J. Sci. Stat. Comput.,* 11 (1990) pp. 990–1028.

[14] T. Coleman and P. Plassmann, "Solution of nonlinear least-squares problems on a multiprocessor", Tech. Report CS-88-923, Computer Science Department, Cornell University, 1988.

[15] C. deSa, K. M. Irani, C. J. Ribbens, L. T. Watson, and H. F. Walker, "Preconditioned iterative methods for homotopy curve tracking", *SIAM J. Sci. Stat. Comput.,* (accepted).

[16] 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.

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

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] C. Y. Wang, "Buoyant rotating disc", manuscript and private communication, 1988.

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

[27] 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.

Address for correspondence and proofs:

L. T. Watson
Department of Computer Science
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061