High Dimensional Homotopy Curve Tracking on a Shared Memory Multiprocessor

By D. C. S. Allison, K. M. Irani, C. J. Ribbens and L. T. Watson

TR 91-5

High Dimensional Homotopy Curve Tracking on a Shared Memory Multiprocessor

D. C. S. Allison, K. M. Irani, C. J. Ribbens and L. T. Watson

Department of Computer Science
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061-0106

E-mail address: ltw@vtopus.cs.vt.edu

Key words: nonlinear equations, homotopy algorithm, curve tracking, shared memory multiprocessor, globally convergent.

Abstract.

Results are reported for a series of experiments involving numerical curve tracking on a shared memory parallel computer. Several algorithms exist for finding zeros or fixed points of nonlinear systems of equations that are globally convergent for almost all starting points, i.e., with probability one. The essence of all such algorithms is the construction of an appropriate homotopy map and then the tracking of some smooth curve in the zero set of this homotopy map. HOMPACK is a mathematical software package implementing globally convergent homotopy algorithms with three different techniques for tracking a homotopy zero curve, and has separate routines for dense and sparse Jacobian matrices. The HOMPACK algorithms for sparse Jacobian matrices use a preconditioned conjugate gradient algorithm for the computation of the kernel of the homotopy Jacobian matrix, a required linear algebra step for homotopy curve tracking. A parallel version of HOMPACK is implemented on a shared memory parallel computer with various levels and degrees of parallelism (e.g., linear algebra, function and Jacobian matrix evaluation), and a detailed study is presented for each of these levels with respect to the speedup in execution time obtained with the parallelism, the time spent implementing the parallel code and the extra memory allocated by the parallel algorithm.

1. Introduction.

Homotopies are a traditional part of topology, and only recently have begun to be used for practical numerical computation. The (globally convergent probability-one) homotopies considered here are sometimes called "artificial-parameter generic homotopies", in contrast to natural-parameter homotopies, where the homotopy variable is a physically meaningful parameter. In the latter case, which is frequently of interest, the resulting homotopy zero curves must be dealt with as they are, bifurcations, ill-conditioning, and all. The homotopy zero curves for artificial-parameter generic homotopies obey strict smoothness conditions, which generally will not hold if the homotopy parameter represents a physically meaningful quantity, but they can always be obtained via certain generic constructions using an artificial (i.e., nonphysical) homotopy parameter. Not just any random perturbation will suffice to create a globally convergent probability-one (generic) homotopy, e.g., the perturbation implied by discretization is generally not sufficient to produce a probability-one homotopy map.

If the objective is to solve a "parameter-free" system of equations, $F(x) = 0$, then extra attention can be devoted to constructing the homotopy, and the curve-tracking algorithm can be limited

to a well-behaved class of curves. The goal of using these globally convergent probability-one homotopies is to solve fixed-point and zero-finding problems with homotopies whose zero curves do not have bifurcations and other singular and ill-conditioned behavior. The mathematical software package HOMPACK used here for comparative purposes is designed for globally convergent probability-one homotopies.

The theory and algorithms for functions $F(x)$ with small dense Jacobian matrices $DF(x)$ are well developed [1]–[4], which is not the case for large sparse $DF(x)$, the topic of this paper. Solving large sparse nonlinear systems of equations via homotopy methods involves sparse rectangular linear systems of equations and iterative methods for the solution of such sparse systems. Preconditioning techniques are used to make the iterative methods more efficient.

Section 2 summarizes the mathematics behind the homotopy algorithm. Section 3 discusses iterative methods for solving invertible linear systems and some of the linear algebra details of homotopy curve tracking. Section 4 describes the numerical experiments which were carried out and Section 5 discusses the results of the various parallel implementations.

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 fundamental problem is to solve the nonlinear system of equations

$$F(x) = 0.$$

The modern homotopy approach to solving the nonlinear system is to construct a C^2 map $\rho : E^m \times [0,1) \times E^n \to E^n$, such that ρ and $\rho_a(\lambda, x) = \rho(a, \lambda, x)$ have the properties

1) the Jacobian matrix $D\rho$ has full rank on $\rho^{-1}(0)$,

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

3) $\rho_a(1, x) = F(x)$,

4) $\rho_a^{-1}(0)$ is bounded.

Then the supporting theory [5], [16], [17] says that for almost all $a \in E^m$ there is a zero curve γ of $\rho_a(\lambda, x)$, 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 $\lambda = 1$). This homotopy algorithm has two important distinctions from classical continuation: (1) the homotopy parameter λ is not required to increase monotonically along γ, so turning points are permissible, and (2) the use of the random parameter vector a which guarantees the absence of bifurcations and singularities along γ with probability one.

The zero curve γ of the homotopy map $\rho_a(\lambda, x)$ can be tracked by many different techniques. The mathematical software package HOMPACK [18] provides three different algorithmic approaches to tracking γ: 1) an ODE-based algorithm, 2) a predictor-corrector algorithm whose iterates follow a trajectory normal to γ (a "normal flow" algorithm); 3) a simple Newton algorithm on an augmented system (an "augmented Jacobian matrix" method). The parallel experiments reported here were based on the normal flow codes in HOMPACK, so the normal flow algorithm will be sketched here–see [18] for a complete description.

Let $\bar{z} = (\bar{\lambda}, \bar{x})$ be the current point on γ, and let $z^{(0)}$ be a prediction for the next point on γ obtained by extrapolation of some sort. The normal flow iteration is

$$z^{(k+1)} = z^{(k)} - \left[D\rho_a\left(z^{(k)}\right)\right]^+ \rho_a\left(z^{(k)}\right), \quad k = 0, 1, 2, \ldots$$

where $[D\rho_a]^+$ is the Moore-Penrose pseudo-inverse. The iterates $z^{(k)}$ converge to a point z^* on γ along a trajectory normal to γ, hence the name "normal flow". There are of course important details concerning the computation of $z^{(0)}$ and what to do if the iteration fails to converge, but this is the essence of the algorithm.

3. Linear algebra routines.

In the course of homotopy curve tracking, we need to solve nonsquare linear systems of equations for the normal flow iteration calculations. These nonsquare systems are converted to equivalent square linear systems of the form

$$Ay = \begin{pmatrix} B & f \\ c^t & d \end{pmatrix} y = b,$$

where the $n \times n$ matrix B is bordered by the vectors f and c to form a larger system of dimension $(n+1) \times (n+1)$. In the present context $B = D_x\rho_a(\lambda, x)$ is symmetric and sparse, but A is not necessarily symmetric.

Iterative methods, rather than direct methods, are generally used for solving these linear systems. (If B has only a couple nonpositive eigenvalues, direct methods may be a viable alternative.) These methods compute a sequence of approximate solutions $\{y_i\}$ which converge to the exact solution y by some algorithm of the form

$$y_{i+1} = F_i(y_0, y_1, \ldots, y_i),$$

where y_0 is an arbitrary initial guess and F_i may be linear or nonlinear. Iterative methods require the coefficient matrix A in the algorithm, generally only to compute matrix-vector products. Since matrix-vector computations are quite inexpensive for sparse problems, iterative methods have low computational cost per iteration. Iterative methods are also attractive because they have low storage requirements, due to the fact that at each iteration, only a small number of vectors of length $N = n + 1$ need to be computed and stored to calculate the next iterate y_{i+1}, and A itself can be generated or stored compactly.

Iterative methods such as the successive over-relaxation (SOR) algorithm [15] and the alternating direction implicit (ADI) algorithm [19] require the estimation of scalar parameters, which is a drawback. However, the conjugate gradient procedure [11] is an efficient algorithm for solving symmetric positive definite systems which requires no such estimates.

For conjugate gradient methods the rate of convergence depends on the symmetry, inertia, spectrum, and condition number of the coefficient matrix. There are efficient conjugate gradient algorithms for solving linear systems with symmetric positive definite coefficient matrices [7], [12], whereas no comparable theory exists for general systems with nonsymmetric or indefinite A. A recent study [13] advocates the use of Craig's method (a variant of the conjugate gradient algorithm) with preconditioning.

Let Q be a $N \times N$ nonsingular matrix. The solution to $Ax = b$ can also be obtained by solving the system:

$$\tilde{A}x = (Q^{-1}A)x = Q^{-1}b = \tilde{b}.$$

The use of such an auxiliary matrix is known as *preconditioning*. The goal of preconditioning is to decrease the computational effort required to solve linear systems of equations by increasing the rate of convergence of an iterative method. For preconditioning to be effective, the faster convergence must outweigh the costs of applying the preconditioning, so that the total cost of

solving the linear system is lower. The preconditioned coefficient matrix $\tilde{A}$ is usually not explicitly computed or stored. The main reason for this is that although A is sparse, $\tilde{A}$ may not be. The extra work of preconditioning, then, occurs in the preconditioned matrix-vector products involving Q^{-1}. The main storage cost for preconditioning is usually for Q, which typically is stored so that one extra array is required to handle the preconditioning operation.

One iterative method known to converge for general nonsymmetric problems is the conjugate gradient method applied to the normal equations. Given any nonsingular matrix A, the system of linear equations $Ay = b$ can be solved by considering the linear system (normal equations)

$$A^t A y = A^t b,$$

or the similar system

$$AA^t z = b, \qquad y = A^t z.$$

Since the coefficient matrix for the latter system is both symmetric and positive definite, the system can be solved by the conjugate gradient algorithm. Once a solution vector z is obtained, the vector y from the original system can be computed as $y = A^t z$. The drawback of this technique is that, while the coefficient matrix is symmetric and positive definite, the convergence rate depends on $\text{cond}(AA^t) = (\text{cond}(A))^2$ rather than $\text{cond}(A)$; see [8] for a precise statement.

An implementation of the conjugate gradient algorithm in which y is computed directly, without reference to z, any approximations of z, or AA^t is due to Craig [6] and is described in [9] and [10]. (Of course, the convergence rate still depends on $\text{cond}(AA^T) = (\text{cond}(A))^2$ in general.) Craig's preconditioned algorithm is:

choose y_0, Q;
set $r_0 = b - Ay_0$;
set $\tilde{r}_0 = Q^{-1} r_0$;
set $p_0 = A^t Q^{-t} \tilde{r}_0$;
for $i = 0$ **step** 1 until convergence **do**
begin
$$a_i = \frac{(\tilde{r}_i, \tilde{r}_i)}{(p_i, p_i)};$$
$$y_{i+1} = y_i + a_i p_i;$$
$$\tilde{r}_{i+1} = \tilde{r}_i - a_i Q^{-1} A p_i;$$
$$b_i = \frac{(\tilde{r}_{i+1}, \tilde{r}_{i+1})}{(\tilde{r}_i, \tilde{r}_i)};$$
$$p_{i+1} = A^t Q^{-t} \tilde{r}_{i+1} + b_i p_i;$$
end

Here (x, y) denotes the inner product of x and y. For this algorithm, a minimum of $5(n+1)$ storage locations is required (in addition to that for A). The vectors y, $\tilde{r}$, and p all require their own locations; $Q^{-t}\tilde{r}$ can share with Ap; $Q^{-1}Ap$ can share with $A^t Q^{-t}\tilde{r}$. The computational cost per iteration of this algorithm is:

(a) two preconditioning solves;
(b) two matrix-vector products;

(c) $5(n+1)$ multiplications (two inner products and three scalings).

There are several approaches to solving $Ay = b$, but the one we have chosen splits A into the sum of a symmetric matrix M and a low rank correction L. This method also takes advantage of the fact that the leading principal submatrix B is symmetric and can use conjugate gradient algorithms requiring a symmetric coefficient matrix. See [17] for further details.

The preconditioning technique used in conjunction with this algorithm is based on the factorization of Q into the product LU where L is a lower triangular matrix and U is an upper triangular matrix. The heuristic used to insure that the preconditioning is inexpensive to implement is to force the factors to be sparse by allowing nonzeros only within a specified set of locations.

Let Z be a set of indices contained in $\{(i,j) \mid 1 \le i,j \le N,\ i \ne j\}$, typically where A is known to be zero. The incomplete LU factorization is given by $Q = LU$, where L and U are lower triangular and unit upper triangular matrices, respectively, that satisfy

$$\begin{cases} L_{ij} = U_{ij} = 0, & (i,j) \in Z, \\ Q_{ij} = A_{ij}, & (i,j) \notin Z. \end{cases}$$

The incomplete LU factorization algorithm is:

for $i = 1$ **step** 1 **until** N **do**
  for $j = 1$ **step** 1 **until** N **do**
    if $((i,j) \notin Z)$ **then**
      begin

$$s_{ij} = A_{ij} - \sum_{t=1}^{\min\{i,j\}-1} L_{it}U_{tj};$$

      if $(i \ge j)$ **then** $L_{ij} = s_{ij}$ **else** $U_{ij} = s_{ij}/L_{ii}$;
      end

It can happen that L_{ii} is zero in this algorithm. In this case L_{ii} is set to a small positive number, so that $Q_{ii} \ne A_{ii}$.

4. Numerical experiments.

To understand the levels and degrees of parallelism, we first describe briefly the sequential HOMPACK code used as the basis for the parallel implementation. First, we need to compute the function values and the Jacobian matrix for the coefficient matrix A. Then to track the homotopy curve, we need to solve nonsquare linear systems of equations for the tangent vector and the normal flow iteration calculations. Subroutines named PCGDS and PCGNS are called to solve these rectangular systems by first converting them to equivalent linear square systems. The combination of PCGDS and PCGNS involves the solution of four symmetric linear systems which can be solved in parallel because they have no data dependence between them. It is to these symmetric linear systems that Craig's preconditioned method is applied to obtain the solution. Since a preconditioned method is used, the preconditioning matrix Q needs to be computed also. Before calls to PCGDS and PCGNS, a subroutine is invoked to compute the incomplete LU preconditioner. Note that there is only one preconditioning matrix to be computed since both nonsquare systems have the same coefficient matrix A with different right hand sides. With this background, we now describe the levels of parallelism. The parallel programming was carried out on a Sequent Symmetry S81 with ten processors using the system call m_fork and the compiler directive DOACROSS.

The results are reported for three test problems–a shallow arch problem, a shallow dome problem, and a turning point problem, all described below and in more detail in [13].

Shallow arch. The equations of equilibrium of the arch are obtained from the principle of the stationary value of the total potential energy, according to which, of all the kinematically admissible displacement fields, the one that makes the total potential energy of a structure stationary also satisfies its equations of equilibrium. The total potential energy π of a structure is given by the sum of its strain energy and the potential of external loads.

A shallow arch is discretized by an assemblage of straight p-q frame elements. A frame element is a structural component that is initially straight and undergoes axial, bending, and torsional deformation resulting from finite displacements and rotations of its ends (nodes) p and q. The displacements of the end q relative to the end p are

$$\begin{pmatrix} \delta u \\ \delta v \\ \delta w \end{pmatrix} = [T]_p \begin{pmatrix} X_q - X_p \\ Y_q - Y_p \\ Z_q - Z_p \end{pmatrix} - \begin{pmatrix} L \\ 0 \\ 0 \end{pmatrix} + [T]_p \begin{pmatrix} U_q - U_p \\ V_q - V_p \\ W_q - W_p \end{pmatrix},$$

where L is the initial rigid body length, and U_i, V_i, W_i ($i = p$ or q) denote the global displacements of the nodes. The matrix $[T]_p$ can be shown to be $[T]_p = \left[T_1(\phi_x, \phi_y, \phi_z)\right]\left[T_1(\theta_{xp}, \theta_{yp}, \theta_{zp})\right]$ with

$$\left[T_1(\alpha_x, \alpha_y, \alpha_z)\right] = \begin{pmatrix} c_y c_z & c_y s_z & -s_y \\ -c_x s_z + s_x s_y c_z & c_x c_z + s_x s_y s_z & s_x c_y \\ s_x s_z + c_x s_y c_z & -s_x c_z + c_x s_y s_z & c_x c_y \end{pmatrix},$$

$c_i = \cos\alpha_i$ and $s_i = \sin\alpha_i$ for $i = x$, y, and z. Angles ϕ_x, ϕ_y, and ϕ_z are the initial orientation angles and angles θ_{xp}, θ_{yp}, and θ_{zp} are the rigid body rotations of the end p. In the equation for $[T]_p$, Euler angle transformations are implied with the order of the rotations being α_z, α_y, and α_x.

Similarly, with the restriction of small relative rotations within the element, the rotations ψ_x, ψ_y, ψ_z of the end q relative to the end p are

$$\begin{pmatrix} \psi_x \\ \psi_y \\ \psi_z \end{pmatrix} = [T]_p \begin{pmatrix} \theta_{xq} - \theta_{xp} \\ \theta_{yq} - \theta_{yp} \\ \theta_{zq} - \theta_{zp} \end{pmatrix}.$$

With the relative generalized displacements $(\delta u, \delta v, \delta w)$ and (ψ_x, ψ_y, ψ_z) known, the usual deformation patterns of the reference axis of the beam element in the corotational coordinate system are assumed to be

$$u(\xi) = \xi\frac{\delta u}{L}, \qquad v(\xi) = \frac{1}{L}(3\xi^2 - 2\xi^3)(\delta v - z_s\psi_x) + (\xi^3 - \xi^2)\psi_z,$$
$$\beta = \xi\psi_x, \qquad w(\xi) = \frac{1}{L}(3\xi^2 - 2\xi^3)(\delta w + y_s\psi_x) - (\xi^3 - \xi^2)\psi_y,$$

where $\xi = x/L$ and y_s and z_s are the coordinates of the shear center of the cross section of the beam. The strain at any point (y, z) on the cross-section of the frame element can be shown to be

$$\epsilon = \frac{\delta u}{L} - \eta\left[\frac{6}{L}(1 - 2\xi)(\delta v - z_s\psi_x) + 2(3\xi - 1)\psi_z\right] - \zeta\left[\frac{6}{L}(1 - 2\xi)(\delta w + y_s\psi_x) - 2(3\xi - 1)\psi_y\right],$$

with $\eta = y/L$ and $\zeta = z/L$. In these equations it is implicitly assumed that the lateral displacements and twists are referenced to a longitudinal axis through the shear center, while the axial displacements and rotations are referenced to the centroidal axis.

The total potential energy of such a discretized model of the arch can be expressed as

$$\pi = \sum_{e=1}^{m} U^e - q^t Q,$$

where U^e is the strain energy of the eth element, $e = 1, \ldots, m$, q is the vector of nodal displacement degrees of freedom of the entire model and Q is the vector of externally applied loads. The strain energy U^e of the eth frame element is given by

$$U^e = \frac{E}{2}\int_V \epsilon^2 dv = \frac{E}{2}\int_0^{L_e}\int_{A_e} \epsilon^2 dA\, dx,$$

where ϵ is the strain of a point (x, y, z) of the beam, which was derived above. Substituting for ϵ and doing the integration gives

$$U^e = U_{\mathrm{p-q}} = \frac{E}{2L_e}\left\{A_e(\delta u)^2 + \frac{12}{L_e^2} I_z \left[(\delta v)^2 + \frac{1}{3}L_e^2\psi_z^2 - L_e \delta v\, \psi_z\right] + \frac{12}{L_e^2} I_y \left[(\delta w)^2 + \frac{1}{3}L_e^2\psi_y^2 + L_e \delta w\, \psi_y\right]\right\},$$

where A_e is the cross-sectional area, and I_y and I_z are the cross-sectional moments of inertia about the y and z axes respectively. It is evident that the potential energy π of the model is a highly nonlinear function of the nodal displacements. The equations of equilibrium of the model are obtained by setting the variation $\delta\pi$ to zero, or equivalently by

$$\nabla\pi = 0.$$

Closed form analytical expressions for $\nabla\pi$ can be obtained with some difficulty, but obtaining the Jacobian matrix of $\nabla\pi$ analytically seems out of the question. Hence the Jacobian matrix of the equilibrium equations is obtained by finite difference approximations.

By symmetry only half the arch need be modelled, and the results here are for a full arch load of 3000 lbs, which is just below the limit point.

Shallow dome. The shallow dome of Figure 1 is built up from space truss elements with three global displacement degrees of freedom (u_1, u_2, u_3) at each of the two nodes. For an element of original length L between its two nodes p and q, the change in length δL is given by

$$\delta L = \left[\sum_{i=1}^{3}(x_{qi} + u_{qi} - x_{pi} - u_{pi})^2\right]^{1/2} - \left[\sum_{i=1}^{3}(x_{qi} - x_{pi})^2\right]^{1/2},$$

where x_{ij}, u_{ij}, $i = p, q$; $j = 1, 2, 3$ are the global coordinates and displacements of the two nodes. This can be simplified to

$$\delta L = L\left[1 + \sum_{i=1}^{3}\left(\frac{2(\Delta x_i \Delta u_i)}{L^2} + \frac{(\Delta u_i)^2}{L^2}\right)\right]^{1/2} - L,$$

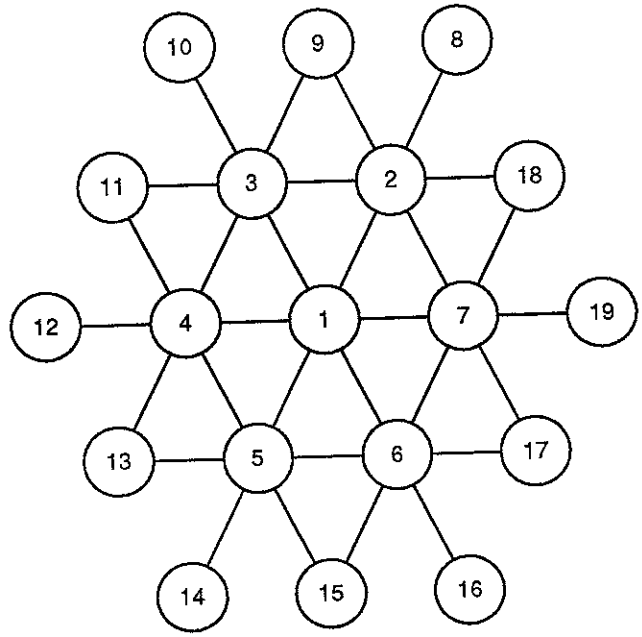

FIGURE 1. *Triangulation for 21 degree of freedom shallow dome.*

where Δ is the difference operator for the q and p values. Accordingly, the axial strain in the e-th element is

$$\epsilon^e = \frac{\delta L}{L} = \left[1 + \sum_{i=1}^{3} \left(\frac{2(\Delta x_i \Delta u_i)}{L^2} + \frac{(\Delta u_i)^2}{L^2} \right) \right]^{1/2} - 1.$$

The strain energy of the e-th element in a purely linearly elastic response is given by

$$U_s^e = \frac{E}{2} \int_V (\epsilon^e)^2 dV = \frac{EA^e L^e}{2} (\epsilon^e)^2,$$

where E and A are the Young's modulus and cross-sectional area, respectively, of the e-th element.

The total potential energy of the dome is then given by

$$\pi = \sum_{e=1}^{m} U_s^e - U^T Q,$$

where U_i, $i = 1, \ldots, 6$ are the six components u_{qk}, u_{pk}, $k = 1, 2, 3$, and Q is the generalized force vector. The equations of equilibrium of the model are then obtained by setting

$$\nabla \pi = \sum_{e=1}^{m} EA^e L^e \nabla \epsilon^e - Q = 0.$$

Both the gradient of π as well as its Hessian can be evaluated explicitly without resorting to finite differencing operations as in the case of the frame element used to model the shallow arch.

The effect of modelling the shallow dome with truss elements in concentric rings is that changing the number of truss elements changes the model and its behavior. Thus the dome

problems with different degrees of freedom reported in the tables are qualitatively different, with different buckling loads and bifurcation points. The results reported here are for shallow domes with base radius 720 and sphere radius 3060, and a point load at the very top.

Artificial turning point problem. The turning point problem is derived from the system of equations

$$F(\mathbf{x}) = \big(F_1(\mathbf{x}), F_2(\mathbf{x}), \ldots, F_n(\mathbf{x})\big)^t = 0$$

where

$$F_i(\mathbf{x}) = \tan^{-1}\big(\sin[x_i(i \bmod 100)]\big) - \frac{(x_{i-1} + x_i + x_{i+1})}{20}, \qquad i = 1, \ldots, n,$$

and $x_0 = x_{n+1} = 0$. The zero curve γ tracked from $\lambda = 0$ to $\lambda = 1$ corresponds to $\rho_a(x, \lambda) = (1 - .8\lambda)(x - a) + .8\lambda\, F(x)$, where a was chosen artificially to produce turning points in γ.

There are five different levels that we considered for the parallel implementation:

1. **Function and Jacobian matrix computations.** Unlike the other levels, the algorithms for this level vary from problem to problem because different problems have different computational structures for the function values and the Jacobian matrix. For the turning point problem, the serial algorithm computes the function values and the Jacobian matrix entries with FORTRAN DO loops. Hence, for the parallel implementation, the DOACROSS directive was used to parallelize the loops and put locks on shared variables. The shallow arch problem has very complex function evaluation computations and, in fact, about 70% of the overall execution time is due to these function evaluations. There are two possible ways of implementing the parallelism at this level for this problem. One way of implementation is to analyze the FORTRAN DO loops and use the DOACROSS directive to implement the parallelism. We refer to this parallel implementation as M8 later in this section. The second way (algorithm M1) is a higher level parallelism in which the columns of the Jacobian matrix are computed in parallel, with the function values still computed as in algorithm M8 above.

2. **Low level linear algebra.** At this level, the lower level functions and linear algebra are implemented in parallel along with LINPACK functions and subroutines. These include copying, scaling, vector norms, inner products and matrix vector products.

3. **Computations with the preconditioner.** There are two subroutines which are candidates for parallelization at this level. The first one computes the incomplete LU preconditioner. The second one computes $Q^{-1}f$ by applying forward and backward substitution to solve $Qx = f$. We have not shown the execution timings for this level in the tables because there was no speedup over the serial execution time. A brief explanation of this is given later in Section 5.

4. **The two linear solves within PCGDS and PCGNS in parallel.** At this level, PCGDS and PCGNS are executed serially one after the other. Within each, as explained earlier, two linear systems of equations need to be solved and these are done in parallel since they are independent of one another.

5. **PCGDS and PCGNS done in parallel.** This is one level higher parallelism than the previous level. Here the subroutines PCGDS and PCGNS are executed in parallel. Note that this means that the two solves within each are still executed serially.

Levels 2–5 described above can be imbedded within each other giving rise to varying degrees of parallelism. For example, if we combine level 4 and level 5, then we are executing PCGDS and PCGNS in parallel as well as the two linear solver algorithms within each of the subroutines in parallel. So actually, all four linear solves are being executed in parallel. This gives a higher degree of parallelism than simply implementing level 4 or 5 individually. For the experiments we wanted to include all possible degrees of parallelism arising from the levels of parallelism, starting from combining levels 2 and 3 and eventually implementing levels 2, 3, 4 and 5 together. Combining levels, in order to obtain the degrees of parallelism, involves implementing a DOACROSS/m_fork within a DOACROSS/m_fork. For example, combining levels 2 and 3 together involves implementing a DOACROSS within a DOACROSS. Unfortunately, all these degrees of parallelism could not be implemented because of the limitation of the Sequent parallel programming directives that within a m_fork or a DOACROSS, we cannot insert another DOACROSS or m_fork. So, due to these constraints, the experiments that could be performed included the following:

1) Levels 4 and 5 together, i.e., all four solves in parallel.

2) Levels 1, 4 and 5 together.

3) Levels 1 and 2 together.

Note that combining level 4 and level 5 involves implementing a m_fork within a m_fork which cannot be done on the Sequent. However, we could get around this problem by using a different strategy. Combining levels 4 and 5 actually means implementing the four solves in parallel as mentioned above. So, since we could not insert a second m_fork at level 4 within the first m_fork at level 5, we modified the code to implement a single m_fork which forks four processes, with each process assigned code for solving a single linear system. Also, note that we could have attempted several different combinations when combining the parallel function and Jacobian computations (level 1) with the linear solver parallelization (levels 2–5). We implemented two of the possible combinations for our experiments, the ones we thought would give the most interesting results.

In the tables which summarize the numerical experiments the following acronyms are used to describe the various levels of parallelism.

M1 – Function and Jacobian matrix evaluations in parallel, with the Jacobian matrix done by columns.

M2 – Lower level linear algebra in parallel.

M3 – PCGDS and PCGNS in parallel.

M4 – Within PCGDS and PCGNS, the two linear solves in parallel.

M5 – Combining M3 and M4, all four linear solves in parallel.

M6 – Combining M1 and M5.

M7 – Combining M1 and M2.

M8 – Function and Jacobian matrix evaluations in parallel (*only* for arch problem, refer to Item 1 above.)

TABLE 1
Execution time in seconds for shallow arch problem with p processors.

p	n	Serial	M1	M2	M3	M4	M5	M6	M7	M8
8	29	440	86	435	432	433	425	69	82	133
8	47	5733	939	5590	5509	5558	5489	750	925	1210
4	47	5733	1496	5623	5509	5558	5489	1467	1495	1818

TABLE 2
Efficiency with p processors for shallow arch problem.

p	n	M1	M2	M3	M4	M5	M6	M7	M8
8	29	0.640	0.126	0.127	0.127	0.129	0.797	0.671	0.414
8	47	0.763	0.128	0.130	0.129	0.131	0.955	0.775	0.592
4	47	0.958	0.255	0.260	0.258	0.261	0.977	0.958	0.788

TABLE 3
Execution time in seconds for shallow dome problem with p processors.

p	n	Serial	M1	M2	M3	M4	M5	M6	M7
8	21	18	16	16	12	13	9	7	14
8	126	95	90	66	61	62	43	41	62
8	252	185	180	125	119	120	83	79	120
8	525	403	394	263	255	255	175	163	254
8	1050	753	743	484	472	478	323	312	474
4	1050	753	763	542	477	478	323	331	532

5. Discussion and conclusions.

As can be observed from the tables, we have not included the timings for the second level, i.e., for the preconditioning computations in parallel. We performed the experiments for this level but did not get any speedup with either four or eight processors. The coefficient matrix for all three test problems is sparse, which means that there are only a few nonzero entries in each row or column of the matrix. These matrices are stored in the packed skyline format. Hence, for all DO loop computations involving the coefficient matrix, the number of computations to be performed per iteration is quite small. This results in each processor not getting enough work to do to overcome the overhead cost of executing a loop in parallel.

Tables 1–6 show the execution time and the parallel efficiency for the three test problems with eight processors for all cases and four processors for the largest case. For the linear solver code only, the most efficient algorithm was M5 amongst algorithms M2, M3, M4 and M5 for all three test problems. This is what one would expect since M5 has the highest degree of parallelism, being a combination of M3 and M4. Note also that the difference in timings for M3 and M4 is very small, since there are only a few computations to be done within each of PCGDS and PCGNS before executing the code for the two linear solves. If there were more code before the two linear solvers' code within each of PCGDS and PCGNS, one would expect the timings for algorithm M3 to be smaller than those for algorithm M4.

Regarding algorithms M1 and M8, note that we have included algorithm M8 only for the shallow arch problem, because unlike the turning point and dome problems, there are two different

TABLE 4
Efficiency with p processors for shallow dome problem.

p	n	M1	M2	M3	M4	M5	M6	M7
8	21	0.141	0.141	0.188	0.173	0.250	0.321	0.161
8	126	0.132	0.180	0.195	0.192	0.276	0.290	0.192
8	252	0.128	0.185	0.194	0.193	0.279	0.293	0.193
8	525	0.128	0.191	0.198	0.198	0.288	0.309	0.198
8	1050	0.127	0.194	0.199	0.197	0.291	0.302	0.199
4	1050	0.247	0.347	0.395	0.394	0.583	0.569	0.354

TABLE 5
Execution time in seconds for turning point problem with p processors.

p	n	Serial	M1	M2	M3	M4	M5	M6	M7
8	20	5	5	5	3	3	2	2	5
8	125	50	46	34	28	29	20	15	29
8	250	87	79	56	49	50	34	26	48
8	500	168	153	105	94	97	65	50	91
8	1000	392	355	238	219	224	151	101	205
4	1000	392	364	265	219	224	151	121	234

TABLE 6
Efficiency with p processors for turning point problem.

p	n	M1	M2	M3	M4	M5	M6	M7
8	20	0.125	0.125	0.208	0.208	0.313	0.313	0.125
8	125	0.136	0.183	0.223	0.216	0.313	0.417	0.216
8	250	0.138	0.194	0.222	0.218	0.320	0.418	0.227
8	500	0.137	0.200	0.223	0.216	0.323	0.420	0.231
8	1000	0.138	0.206	0.224	0.219	0.325	0.485	0.239
4	1000	0.269	0.370	0.447	0.438	0.649	0.810	0.419

ways of parallelizing the code for the function values and the Jacobian matrix evaluations. For the shallow arch problem, M1 is better than M8 because M1 has a higher degree of parallelism than M8. Another interesting point to observe is the efficiency we obtained with algorithm M1 for the shallow arch problem as compared to the turning point problem or the dome problem. This is because about 83% of the serial execution time for the arch problem is spent computing the function values and the Jacobian matrix whereas for the turning point or dome problems, the same number is less than 2%. For the arch problem, each processor has a lot of work to do and is not idle for long, and as the tables reflect, the parallel implementation is very efficient.

Overall, for all three test problems, Algorithm M6 is the best algorithm in terms of timings and the speedup obtained by the parallel implementations. This is because M6 combines the most efficient parallel algorithm for the function values and the Jacobian matrix evaluations with that for the linear solver code. The tables also show the results for the same experiments with four processors, only for the largest dimension n for each of the test problems. In terms of the most efficient algorithm, the same discussion holds as for eight processors. Comparing the efficiencies obtained

TABLE 7
Actual and theoretical speedups for Algorithm M6.

n	Four processors	Eight processors
47	3.91/3.94	7.64/7.76
1050	2.28/2.71	2.41/3.79
1000	3.24/3.40	3.88/5.52

TABLE 8
Programming effort in man-hours.

Cost	M1	M2	M3	M4	M5	M6	M7	M8
Incremental	100,12,3	20	20	22	25	1	1	120
Total						126,38,29	121,33,24	

with four processors to those obtained with eight processors, some very interesting observations can be made. First, for both the turning point and the dome problems, the efficiency obtained with four processors is almost twice as good as that with eight processors. The same holds true for the shallow arch problem, for algorithms M2, M3, M4 and M5, i.e., the linear solver parallel algorithms. However, this is not true for algorithms M1, M6, M7 and M8, i.e., all the algorithms involving the parallel function and Jacobian matrix evaluations for the shallow arch problem. The reason for this can be attributed to the fact that about 83% of the total execution time is spent executing the function and Jacobian matrix evaluation code. Hence the eight processors can be kept busy most of the time.

Amdahl's law provides a useful way of comparing the actual speedup attained by a parallel implementation to the maximum speedup that can be attained taking into consideration the fraction of the total execution time that is spent on sequential code. Amdahl's law states that if a program consists of two parts, one that is inherently sequential and one that is fully parallelizable, and if the inherently sequential part consumes a fraction f of the total computation, then the speedup is limited by

$$S_p(n) \leq \frac{1}{f + (1-f)/p},$$

where p is the number of processors used in the parallel implementation. Table 7 gives the speedup that we obtained with our best parallel implementation (Algorithm M6) along with the maximum theoretical speedup that can be obtained according to Amdahl's law for all three test problems. Note that the fraction f of the sequential part may not be very accurate and is a lower bound on the exact serial execution fraction for the algorithms so that the numbers appearing in Table 7 for the theoretical speedup could be slightly lower than those shown in the table. As can be observed from the table, for all three problems, with four processors, the actual speedup obtained is quite close to the theoretical speedup. This explains why we got an overall poor speedup for the dome and the turning point problems; for these problems, the fraction f of serial execution is high, and so we cannot do any better, being limited by the theoretical speedup as the upper bound. With eight processors, the actual speedup obtained is not close to the theoretical speedup because the algorithm M6 is a combination of algorithms M1 and M5, and for the parallel implementation, algorithm M5 uses only four processors always, even if it is given eight processors. Also as observed from the table, there is only a slight increase in speedup from using eight processors as compared to

TABLE 9
Amount of extra memory allocated for each method.

	M1	M2	M3	M4	M5	M6	M7	M8
$(N+1)$ vectors			5	3	11	11		

four processors. The slight increase in speedup is present only because the M1 algorithm does make use of eight processors. This explains why the timings for algorithm M5 (as well as M3 and M4) are the same for eight processors and four processors, and why there is a significant gap between the theoretical and the actual speedups for eight processors. Note that for the arch problem, the scenario is completely different since unlike the turning point problem, very little time is spent in the linear solver code.

Regarding the amount of extra memory allocated for each method, as observed from Table 9, algorithms M3, M4, M5 and M6 require a few extra $(n+1)$-vectors for the parallel implementation [12]. The algorithms which don't require any extra memory are at the lowest level of parallelization. Algorithms M5 or M6, which require the maximum amount of extra memory, have a greater degree of parallelization than the others. So, the conclusion to be drawn from this table is that the higher the level or the degree of parallelism, the more memory the parallel implementation requires. As already observed, in terms of efficiency, M6 is the best parallel implementation verifying that there is, indeed, a tradeoff between memory and speedup. One generally has to pay for extra memory if speedup is the final goal. However, the amount of extra memory required by the parallel algorithm is usually not very significant in relation to the speedup achieved by the parallel algorithm. For example, algorithm M6, which has the best speedup, requires 11 extra $(n+1)$-vectors for the parallel algorithm implementation, which is just a small fraction more than the overall memory space required by the serial algorithm.

Another interesting issue related to parallel computing is the tradeoff between the speedup and the amount of time spent on the implementation of the parallel algorithm. Table 8 gives the number of man-hours spent for each of the parallel implementations, along with the cumulative man-hours for algorithms M6 and M7, since each of them are combinations of other algorithms. Columns for M1, M6 and M7 have three entries each since these algorithms involve computing function values and Jacobian matrices in parallel, and for all three test problems, varying amounts of programming effort were required. The three entries are for, respectively, the shallow arch problem, the dome problem, and the turning point problem. For algorithm M1, comparing the speedup obtained with the programming effort required for each of the test problems, it is seen that the arch problem, which has the best speedup, also required the maximum number of man-hours to do the parallel implementation. In general, comparing the efficiencies with the programming effort required, one can draw the conclusion that they are directly proportional to each other, i.e., the algorithms which require more time and effort to be implemented in parallel usually have a higher efficiency than those which require fewer man-hours for the parallel implementation. We have already concluded that the higher the level/degree of the parallel implementation, the better the efficiency. Hence, we can also conclude that the higher the degree desired of the parallel algorithm, the greater the amount of time needed to implement it; as one moves up in degree or level, it becomes more and more time consuming to parallelize the serial code.

Another question which arises in the same context is whether it was worthwhile spending many hours for the parallel implementation considering the efficiency that was obtained. For example, for the arch problem, we spent 120 hours implementing Algorithm M8 obtaining an efficiency of

0.788. It was probably not worth the time attempting the parallel implementation. Similarly for the arch problem, implementing parallel algorithms M2, M3, M4 and M5 was not worth the effort put in because the amount of time spent within the linear solver code is very small, around only 10%, and each of these algorithms attempts to parallelize sections of the linear solver code. In general, it is not worth the effort to parallelize some part of the program if just a small fraction of the total execution time is spent within that part of the program.

Special mention needs to be made of the shallow arch problem with regard to algorithms involving the function values and Jacobian matrix evaluations, i.e., algorithms M1, M6, M7 and M8. As observed from the tables, all these algorithms required many more man-hours for the arch problem than the other test problems. The arch problem could have taken a lot less programming effort had it not been for the fact that the serial code for the function values and Jacobian matrix evaluations was extremely difficult to parallelize. To a certain extent, the programming effort also depends on how the serial code has been written and how easily the serial code can be modified for the parallel implementation, and not simply on the amount of code to be implemented in parallel or what degree/level of parallelism one is attempting for the implementation.

Most of the conclusions drawn above reaffirm existing parallel computing theory and are very general. Regarding specific conclusions to be drawn for parallel HOMPACK, some levels are simply not worth the time and effort required for the implementation, considering the speedup obtained. Algorithm M2 (lower linear algebra in parallel) was not worth the effort. It took 20 man-hours to obtain a maximum speedup of 1.65 using eight processors. Similarly, attempting to implement computations relating to the preconditioner in parallel is not worthwhile since we get no speedup at all. Regarding the other levels/degrees, the test problem used determines whether the level or degree implementation is worth the effort put in. For the dome and turning point problems, the implementation of the function values and the Jacobian matrices was not worth the effort whereas the degrees/levels relating to the linear solver code did give a good speedup considering the effort we put into the parallel implementation. For the arch problem it was exactly the opposite, although it could be debated that spending 100 hours to obtain a speedup of 6.1 with eight processors is just not worth the effort. Regarding extra memory allocation for the parallel implementation, the parallel algorithms required only just a few extra $(n+1)$-vectors and hence memory is not an important issue for parallel HOMPACK. Hence a general purpose parallel HOMPACK, applicable to any problem, should implement the four linear solves in parallel. The parallelization of the function and Jacobian matrix evaluation subroutines will depend on the problem being solved.

References.

[1] E. L. Allgower and K. Georg, *Introduction to Numerical Continuation Methods,* Springer Verlag, Berlin, 1990.

[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] 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.

[5] S. N. Chow, J. Mallet-Paret, and J. A. Yorke, Finding zeros of maps: Homotopy methods that are constructive with probability one, *Math. Comput.* **32** (1978) 887–899.

[6] E. J. Craig, *Iteration procedures for simultaneous equations,* Ph.D. thesis, MIT, Cambridge, 1954.

[7] 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.*, to appear.

[8] H. C. Elman, *Iterative methods for large, sparse, nonsymmetric systems of linear equations,* Ph.D. thesis, Computer Sci. Dept., Yale Univ., 1982.

[9] D. K. Fadeev and V. N. Fadeeva, *Computational Methods of Linear Algebra,* Freeman, London, 1963.

[10] M. R. Hestenes, The conjugate gradient method for solving linear equations, *Proc. Symp. Appl. Math.* **6** Numer. Anal., AMS, New York, 1956, pp. 83–102.

[11] M. R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. National Bureau of Standards* **49** (1952) 409–435.

[12] K. M. Irani, Preconditioned sequential and parallel conjugate gradient algorithms for homotopy curve tracking, M.S. thesis, Dept. of Computer Sci., VPI & SU, Blacksburg, 1990.

[13] 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.*, to appear.

[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] R. S. Varga, *Matrix iterative analysis,* Prentice-Hall, New York, 1962.

[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.

[19] D. M. Young, *Iterative solution of large linear systems,* Academic Press, New York, 1971.