PARALLEL SPARSE LINEAR ALGEBRA FOR HOMOTOPY METHODS

by

Maria Sosonkina Driver

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:

Layne T. Watson

Donald C. S. Allison | Christopher A. Beattie

Lenwood S. Heath | Mark T. Jones

September, 1997

Blacksburg, Virginia

Parallel Sparse Linear Algebra for Homotopy Methods

Maria Sosonkina Driver

Abstract

Globally convergent homotopy methods are used to solve difficult nonlinear systems of equations by tracking the zero curve of a homotopy map. Homotopy curve tracking involves solving a sequence of linear systems, which often vary greatly in difficulty. In this research, a popular iterative solution tool, GMRES(k), is adapted to deal with the sequence of such systems. The proposed adaptive strategy of GMRES(k) allows tuning of the restart parameter k based on the GMRES convergence rate for the given problem. Adaptive GMRES(k) is shown to be superior to several other iterative techniques on analog circuit simulation problems and on postbuckling structural analysis problems.

Developing parallel techniques for robust but expensive sequential computations, such as globally convergent homotopy methods, is important. The design of these techniques encompasses the functionality of the iterative method (adaptive GMRES(k)) implemented sequentially and is based on the results of a parallel performance analysis of several implementations. An implementation of adaptive GMRES(k) with Householder reflections in its orthogonalization phase is developed. It is shown that the efficiency of linear system solution by the adaptive GMRES(k) algorithm depends on the change in problem difficulty when the problem is scaled. In contrast, a standard GMRES(k) implementation using Householder reflections maintains a constant efficiency with increase in problem size and number of processors, as concluded analytically and experimentally. The supporting numerical results are obtained on three distributed memory homogeneous parallel architectures: CRAY T3E, Intel Paragon, and IBM SP2.

ACKNOWLEDGEMENTS.

I am greatly indebted to my advisor, Professor Layne Watson, for his inexhaustible attention, help, and support in the pursuing of my degree, and for setting an inspiring example of a researcher and a teacher.

I thank Professors Donald Allison, Christopher Beattie, Lenwood Heath, and Mark Jones for taking time to serve on my Ph.D. committee and for providing me with valuable insights and comments during the research process.

Professor Homer Walker introduced me to various implementations of the GMRES iterative solution method. His helpful explanations are gratefully acknowledged. I have also benefited from discussions with Professor Calvin Ribbens and I am very grateful to him for administering the Intel Paragon at Virginia Tech, so that the experimental part of my work proceeded smoothly.

I would like to acknowledge the staff of Virginia Tech Computing Center, NASA Langley Research Center, Center for Computational Sciences at Oak Ridge National Laboratory, and National Energy Research Scientific Computing Center for providing state of the art research facilities and making this research possible.

TABLE OF CONTENTS.

1. INTRODUCTION 1

1.1. Related Work 2

1.2. Organization 3

2. PROBABILITY-ONE HOMOTOPY METHODS FOR LARGE SCALE APPLICATIONS 4

2.1. Global convergence of homotopy methods 4

2.1.1. Theoretical foundations 5

2.2. Homotopy algorithms for large scale applications 7

2.2.1. Homotopy zero curve tracking 7

2.2.2. Sparse normal flow algorithm 9

3. ITERATIVE SOLUTION TECHNIQUES AND PRECONDITIONING 13

3.1. Iterative methods 13

3.1.1. Arnoldi's procedure 14

3.1.2. The GMRES method 14

3.1.3. The QMR algorithm 15

3.2. Preconditioning 16

4. ADAPTATION OF GMRES(k) FOR HOMOTOPY METHODS 19

4.1. An implementation of the adaptive GMRES(k) algorithm 20

4.2. Numerical experiments 24

4.2.1. Circuit design and simulation 25

4.2.2. Space truss stability analysis 26

4.2.3. Shallow shell stability analysis 29

4.3. Results 32

4.4. Conclusions 40

5. PARALLEL IMPLEMENTATION OF ADAPTIVE GMRES(k) ... 42
5.1. Rationale for Developing Parallel Implementation ... 42
5.2. Partitioning of a General Sparse Matrix ... 43
5.3. Distributed Matrix-Vector Multiplication ... 47
5.4. Approaches to Distributed Orthogonalization ... 48
5.5. Householder Reflections for a Special Case of Partitioning ... 49
5.6. Householder Reflections for an Arbitrary Partitioning ... 51
5.7. Experimental Results ... 52
5.8. Conclusions ... 54

6. SCALABILITY ANALYSIS OF GMRES(k) ... 56
6.1. Terminology for the Scalability Analysis ... 56
6.2. Operation Count for the Useful Work in GMRES(k) ... 58
6.3. Description of the Test Problem ... 58
6.4. Architectures of the IBM SP2, Intel Paragon, and Cray T3E ... 59
6.5. Derivation of the Isoefficiency Function ... 61
6.5.1. Assumptions ... 61
6.5.2. Overhead due to matrix-vector multiplication ... 61
6.5.3. Overhead due to Householder reflection generation and application ... 62
6.5.4. Overhead due to the residual norm update ... 63
6.5.5. Total parallel overhead ... 63
6.5.6. Relation between the useful work and the parallel overhead ... 64
6.6. Machine-Dependent Constants ... 65
6.7. Numerical Experiments ... 69
6.7.1. Isoefficiency of GMRES(k) on the IBM SP2 ... 70
6.7.2. Isoefficiency of GMRES(k) on the Intel Paragon ... 73
6.7.3. Isoefficiency of GMRES(k) on the Cray T3E ... 75
6.7.4. Speed-up as an indication of resource saturation ... 76
6.8. Conclusions ... 78

7. SCALABILITY ANALYSIS OF ADAPTIVE GMRES(k) 79
7.1. Efficiency of Adaptive GMRES(k) 79
7.2. Conclusions 80

8. RELATION BETWEEN PARALLEL SPARSE LINEAR ALGEBRA, STEPPING ALGORITHM, AND USER-DEFINED FUNCTIONS IN PARALLEL HOMPACK90 81
8.1. Internal Data Format in ||HOMPACK90 81
8.2. User-Defined Function Evaluations in ||HOMPACK90 84
8.3. Global View of ||HOMPACK90 85

9. CONCLUSION 87

REFERENCES 90

VITA 96

LIST OF FIGURES.

Fig. 1. The inverse image $\rho_a^1(0)$ 6

Fig. 2. Davidenko flow, normal flow iterates •, and
augmented Jacobian matrix iterates ∘ 10

Fig. 3. Pseudocode for adaptive GMRES(k) 22

Fig. 4. Circuit diagram for the circuit "vref" 24

Fig. 5. Jacobian matrix sparsity pattern corresponding to the circuit "bgatt" 25

Fig. 6. Space truss element 26

Fig. 7. Lamella dome (21-degree-of-freedom) 27

Fig. 8. Cylindrical shell geometry; load applied at A 28

Fig. 9. Thin-shell element coordinate system 29

Fig. 10. Load factor λ vs. displacement u^3 at the point A for the shell in Fig. 8 32

Fig. 11. Effect of adaptive strategy on residual norm 36

Fig. 12. Block-chessboard partitioning of 4×4 grid into 4 subdomains 43

Fig. 13. Partitioning of 12×12 matrix into 4 subdomains 44

Fig. 14. Contraction of $G(A)$ to a multigraph using the procedure CONTRM 45

Fig. 15. Local matrix A_d and vector components after the preprocessing stage 48

Fig. 16. Parallel Householder reflection generation 50

Fig. 17. Parallel Householder reflection application 50

Fig. 18. Modified parallel Householder reflection generation 52

Fig. 19. Modified Householder reflection application 53

Fig. 20. Scaling of problem size 59

Fig. 22. Observed on the IBM SP2 and predicted by linear regression CPU and
communication times. 66

Fig. 23. Observed on the Paragon and predicted by linear regression CPU and
communication times. 67

Fig. 24. Observed on the Cray T3E and predicted by linear regression CPU and
communication times. 67

Fig. 25. Isoefficiency curves for HG_S with k = 15 71
Fig. 26. Isoefficiency curves for MHG_S with k = 15 72
Fig. 27. Isoefficiency curves for MHG_S with k = 25 72
Fig. 28. Isoefficiency curves for HG_P (k = 15, top) and MHG_P (k = 15, bottom) 74
Fig. 29. Isoefficiency curves for HG_P with k = 35 74
Fig. 30. Isoefficiency curves for HG_T (k = 15, top) and MHG_T (k = 15, bottom) 75
Fig. 31. Speed-up curves with k = 15 for HG_S, HG_P, and HG_T 77
Fig. 32. Efficiency on 21 processors, GMRES($kmax$) (solid lines), GMRES(k) (dashed lines) 79
Fig. 33. Decision tree for implementing numerical linear algebra in parallel 86

LIST OF TABLES.

Table 1. Average, maximum, and minimum number of iterative solver iterations per linear system along homotopy zero curve for circuit design problems (ILU preconditioner) 35

Table 2. Average, maximum, and minimum number of iterative solver iterations per linear system along homotopy curve for thin shell problem (Gill-Murray preconditioner) 35

Table 3. Average, maximum, and minimum number of iterative solver iterations per linear system along homotopy zero curve for lamella dome problem (Gill-Murray preconditioner) 36

Table 4. Iterative solver execution time in seconds for circuit problems (ILU preconditioner) 36

Table 5. Iterative solver execution time in seconds for thin shell problem (Gill-Murray preconditioner) 36

Table 6. Iterative solver execution time in seconds for lamella dome problem (Gill-Murray preconditioner) 36

Table 7. Convergence trace for a typical system from thin shell problem (n = 55) comparing AGS with AGH 39

Table 8. Total and matrix-vector multiplication parallel times for GMRES(15) with MHG and HG orthogonalizations 53

1. INTRODUCTION.

Challenging large scale scientific applications require sophisticated solution techniques as well as significant hardware resources. This research addresses both these requirements by studying computationally intensive parts of globally convergent homotopy methods implemented on parallel architectures. Globally convergent homotopy methods are usually applied to solve difficult nonlinear analysis problems, such as structural mechanics postbuckling and analog circuit simulation problems, where a good initial estimate of the solution is hard to obtain. Although these methods are expensive in general, they become an affordable choice when their numerical linear algebra routines take advantage of the memory and computational resources of parallel architectures.

The overall purpose of this work is to design scalable parallel implementations of sparse linear algebra, which have been adapted to the requirements of homotopy methods used for solving large scale applications. The adaptation process is crucial to support the inherent robustness and accuracy of homotopy methods. Since homotopy methods are used for a wide range of scientific applications, the nature of the linear systems generated vary widely. Thus their solution method requires the ability to find an approximate solution with sufficient accuracy and to adapt itself to the characteristics of a given problem. An ad-hoc method may drastically increase the solution time and memory requirements and ultimately cause failure of the homotopy algorithm.

In parallel environments, linear system solution techniques have to preserve adaptivity and achieve high accuracy (required by homotopy methods), and simultaneously utilize processors efficiently thereby lowering the cost of processor idling and nonessential work. Numerous factors, such as the degree of concurrency, data locality, and synchronization time, affect processor utilization and, thus, the parallel algorithm performance on a given architecture. Furthermore, the same algorithm may perform very differently on various architectures. Hence, it is desirable to consider performance of an algorithm in conjunction with performance of a given architecture. In this work, the performances of several combinations of distributed memory architectures and sparse linear algebra routines are analyzed as the number of processors and problem size increase.

1.1. Related work.

Parallel homotopy curve tracking for problems with dense Jacobian matrices was considered by Chakraborty [12] on a hypercube architecture. In [12], several parallel implementations of linear system solution by direct factorization are presented in the context of homotopy algorithms for dense Jacobian matrices. The choice of an implementation depends on the complexity of user-defined function evaluations. For large scale applications solved by homotopy algorithms with sparse Jacobian matrices Allison et al. [2] propose a parallel shared memory implementation of an iterative linear system iterative solver. In particular, Craig's iterative method for nonsymmetric linear systems is considered. Adapting an iterative method to a given problem has been investigated, for example, by De Sturler [17] and Saad et al. [56] in the form of various truncated strategies for the Krylov subspace dimension of the GMRES algorithm.

With the development of parallel supercomputing, a variety of mathematical software packages deals with the solution of large scale linear systems of equations. Among them are Aztec [35], BlockSolve95 [38], \\ELLPACK [33], ISIS++ [5], PETSc [4], and PSPARSLIB [54]. Each of these packages incorporates three main components: (a) efficient distributed matrix operations based on a particular representation of a matrix, (b) parallel implementations of standard iterative methods, and (c) sophisticated parallel preconditioning techniques. BlockSolve95 contains parallel ILU(0) based on scalable parallel graph coloring heuristics [37], [39] that are used to reorder matrix entries. In ISIS++, a novel implementation of preconditioning without matrix factorization (called sparse approximate inverse preconditioning [6], [34], and [14]) is provided. Aztec, \\ELLPACK, PETSc, and PSPARSLIB feature a suite of preconditionings based on domain decomposition, such as block Jacobi and block Gauss-Seidel.

Although iterative methods for general linear systems possess a high degree of concurrency, some of their phases, such as computation of an orthogonal basis in GMRES by the modified Gram-Schmidt method, incur a substantial communication overhead. Much research has been done to reduce this overhead. Bai et al. [3], Calvetti et al. [11], and

Erhel [22], construct a nonorthogonal Krylov subspace basis with its post-orthogonalization before solving a GMRES least squares problem. This post-orthogonalization relies on efficient parallel implementations of block QR factorization [16] and [58]. De Sturler and van der Vorst [18] implemented restarted GMRES using a modified Gram-Schmidt variant that alternates between two parts of local vectors to achieve overlapping of communication and computation.

A scalability analysis of Householder reflections underlying QR factorization has been carried out by Sun et al. [64] on the KSR parallel architecture. Gupta et al. [29] derive the isoefficiency function for a preconditioned conjugate gradient method on parallel architectures with mesh, hypercube, and that of CM-5 interconnection networks. They show the dependence of the isoefficiency function on block diagonal and incomplete Cholesky preconditionings. In general, scalability analysis of an algorithm-architecture combination is based on the concept of scaled speed-up introduced by Gustafson et al. [30].

1.2. Organization.

Chapter 2 contains a description and the supporting theoretical explanation of homotopy algorithms for large scale applications. In Chapter 3, key concepts of sparse matrix computations are stated followed by a detailed description of a linear system solution process in the context of homotopy methods (Chapter 4). Several parallel implementations of a chosen solution method, adaptive GMRES(k), are discussed in Chapter 5. Performance analyses of standard and adaptive GMRES(k) with Householder reflection orthogonalization are given in Chapters 6 and 7, respectively. Chapter 8 establishes a relation between parallel sparse linear algebra routines and other important parts of a homotopy method, such as the stepping procedure and user-defined function evaluations. Concluding remarks constitute Chapter 9.

2. PROBABILITY-ONE HOMOTOPY METHODS FOR LARGE SCALE APPLICATIONS.

2.1. Global Convergence of Homotopy Methods.

For numerous highly nonlinear realistic applications, probability-one homotopy methods are a primary solution choice, and are robust and accurate. Other nonlinear analysis methods, such as quasi-Newton methods, often fail whenever a good initial estimate of the solution to a given problem is hard to obtain. On the other hand, probability-one homotopy methods converge to a solution from an *arbitrary* starting point (outside of a set of measure zero). Thus they are also termed *globally convergent* homotopy methods. Globally convergent probability-one homotopies were proposed in 1976 by Chow, Mallet-Paret, and Yorke [15]. Since then these methods have been successfully applied to solve Brouwer fixed point problems, zero finding problems, polynomial systems of equations, optimization problems, discretizations of nonlinear two-point boundary value problems based on shooting, finite differences, collocation, and finite elements [69]. However, the global convergence together with robustness and accuracy of homotopy methods is the result of a rather costly computational process similar to continuation methods. The idea of a continuation method is to continuously deform an easy (simple) problem into a given (hard) one, while solving a family of deformed problems. The solutions to the deformed problems are related and can be tracked as the deformation proceeds. A function describing the deformation is called a *homotopy map*. For example, a homotopy map in its traditional sense for a zero finding problem $F(x) = 0$, which has a simple version $s(x) = 0$ with a known unique solution x_0, is

$$\rho(x, \lambda) = \lambda F(x) + (1 - \lambda)s(x), \quad 0 \leq \lambda \leq 1.$$

The family of problems is $\rho(x, \lambda) = 0$, $0 \leq \lambda \leq 1$. Continuation means tracking the solutions of $\rho(x, \lambda) = 0$, starting from $(x, \lambda) = (x_0, 0)$, as λ increases monotonically from 0 to 1. The tracking procedure stops at $\lambda = 1$, since $F(\bar{x}) = \rho(\bar{x}, 1) = 0$. At each step, starting from a point (x_i, λ_i) with $\rho(x_i, \lambda_i) = 0$, the problem $\rho(x, \lambda_i + \Delta\lambda) = 0$ is solved for x, with $\Delta\lambda$

being a sufficiently small fixed positive number. Possible shortcomings of this approach follow.

(1) The points (x_i, λ_i) may diverge to infinity as $\lambda \to 1$.

(2) There may be no solution of $\rho(x, \lambda_i + \Delta\lambda) = 0$ near (x_i, λ_i).

(3) The curve of zeros of ρ may have turning points.

(4) The problem $\rho(x, \lambda_i + \Delta\lambda) = 0$ may be singular at its solution, causing numerical instability.

For globally convergent homotopy methods, once a homotopy map describing the deformation process is constructed, tracking of some smooth curve in the zero set of this map is performed. Furthermore, the homotopy parameter λ need not increase monotonically along the curve, therefore turning points are tracked successfully by a homotopy method. All four deficiences of traditional continuation methods are circumvented by an appropriate homotopy map construction, which involves an additional "artificial" parameter. To apply a continuation method, the zero curve must obey strict smoothness conditions, which may not hold if the homotopy parameter represents a physically meaningful quantity. However these conditions are always obtained via certain generic constructions using an artificial parameter. Furthermore, "artificial-parameter generic homotopies" attempt to solve a *single* system of nonlinear equations rather than a parameterized family of systems as that (natural) parameter is varied. Thus drastic changes in the solution behavior with respect to that (natural) parameter pose no difficulty for a probability-one homotopy algorithm, contrary to traditional continuation. In other words, the purpose of using an artificial parameter is to construct homotopy maps that lead to a point $(1, \bar{x})$ for almost all choices of the starting point, and that avoid bifurcations and other singular behavior.

2.1.1. Theoretical foundations.

The following definition and theorem from differential geometry stated in [69] give the mathematical underpinning of all probability-one globally convergent homotopy methods.

Definition. Let $U \subset \mathbf{R}^m$ and $V \subset \mathbf{R}^n$ be open sets, and let $\rho : U \times V \times [0, 1) \to \mathbf{R}^n$ be a C^2 map. ρ is said to be transversal to zero if the Jacobian matrix $D\rho$ has full rank on $\rho^{-1}(0)$.

Transversality Theorem. If $\rho(a, x, \lambda)$ is transversal to zero, then for almost all $a \in U$ the map

$$\rho_a(x, \lambda) = \rho(a, x, \lambda)$$

is also transversal to zero; i.e., with probability one the Jacobian matrix $D\rho_a(x, \lambda)$ has full rank on $\rho_a^{-1}(0)$.

The following example illustrates the construction of a globally convergent homotopy map. Consider the nonlinear system of equations

$$F(x) = 0,$$

where $F : \mathbf{R}^n \to \mathbf{R}^n$ is a C^2 map. To solve this problem one need only to construct a C^2 homotopy map $\rho : U \times \mathbf{R}^n \times [0, 1) \to \mathbf{R}^n$ such that

1) $\rho(a, x, \lambda)$ is transversal to zero,

for each fixed $a \in U$,

2) $\rho_a(x, 0) = \rho(a, x, 0) = 0$ is trivial to solve and has a unique solution x_0,

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

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

Then (from the Transversality Theorem) for almost all $a \in U$ there exists a zero curve γ of ρ_a, along which the Jacobian matrix $D\rho_a$ has rank n, emanating from $(x_0, 0)$ and reaching a zero $\bar{x}$ of F at $\lambda = 1$ (cf. Figure 1). This zero curve γ does not intersect itself, is disjoint from any other zeros of ρ_a, and has finite arc length in every compact subset of $\mathbf{R}^n \times [0, 1)$. Furthermore, if $DF(\bar{x})$ is nonsingular, then γ has finite arc length. ρ_a is called a *probability-one* homotopy because the properties of a zero curve of ρ_a hold almost surely with respect to a, i.e., with probability one. The framework for a homotopy algorithm is now clear: construct the homotopy map ρ_a and then track its zero curve γ from the known point $(x_0, 0)$ to a solution $\bar{x}$ at $\lambda = 1$.

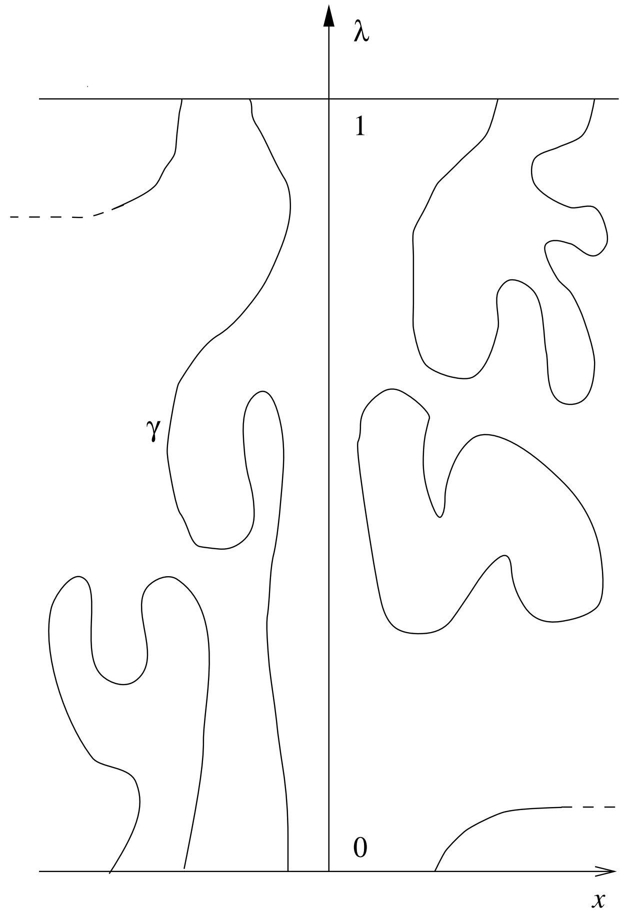

FIG. 1. *The inverse image* $\rho_a^{-1}(0)$.

2.2. Homotopy Algorithms for Large Scale Applications.

2.2.1. Homotopy zero curve tracking.

Three general curve tracking methods, implemented in the homotopy software package HOMPACK90 [73], are: ordinary differential equation (ODE) based, normal flow, and augmented Jacobian matrix.

The details of homotopy zero curve tracking are similar for the fixed point ($x = f(x)$), zero finding ($F(x) = 0$), and general homotopy map ($\rho(a, x, \lambda) = 0$) cases. Thus the zero finding problem (a typical case) will serve as an example for describing homotopy algorithms. The special case of the zero finding problem where the homotopy map ρ_a is a simple convex combination is addressed by the following theorem [70]:

Theorem. Let $F : \mathbf{R}^n \to \mathbf{R}^n$ be a C^2 map such that for some $r > 0$ and $\tilde{r} > 0$, $F(x)$ and $x - a$ do not point in opposite directions for $\|x\| = r$, $\|a\| < \tilde{r}$. Then F has a zero in $\{x \in \mathbf{R}^n \mid \|x\| \le r\}$, and for almost all $a \in \mathbf{R}^n$, $\|a\| < \tilde{r}$, there is a zero curve γ of

$$\rho_a(x, \lambda) = \lambda F(x) + (1 - \lambda)(x - a),$$

along which the Jacobian matrix $D\rho_a(x, \lambda)$ has full rank, emanating from $(0, a)$ and reaching a zero $\bar{x}$ of F at $\lambda = 1$. Furthermore, γ has finite arc length if $DF(\bar{x})$ is nonsingular.

Assuming that $F(x)$ is C^2, a is such that the Jacobian matrix $D\rho_a(x, \lambda)$ has full rank along γ, and γ is bounded, the zero curve γ is C^1 and can be parametrized by the arc length s. In the ODE-based algorithm, an initial value problem with implicitly defined derivative is solved. To use the available sophisticated ordinary differential equation techniques [57], the derivative must be given explicitly. Watson, Billups, and Morgan [71] describe a robust numerical linear algebra procedure, which involves a linear system solution, to obtain the derivative. The final stage (at $\lambda = 1$) of the ODE-based algorithm is the correction of the computed solution that, possibly, drifted away from γ.

Both the normal flow and augmented Jacobian matrix algorithms are predictor-corrector type algorithms. They have four phases: prediction, correction, step size estimation, and computation of the solution at $\lambda = 1$ (called the "end-game"). The prediction is accomplished (after the first step) by Hermite cubic interpolation. In the normal flow algorithm, the correction phase is essentially Newton's method with a rectangular $n \times (n + 1)$ Jacobian matrix, which must be handled with care due to its rectangular shape [70]. In the augmented Jacobian matrix algorithm, the correction phase is performed by updated QR factorizations and quasi-Newton updates. Each of the two algorithms has its own refined step size estimation procedure, which attempts to balance progress along γ with the amount of work spent in the correction phase (in the normal flow algorithm) and takes into account curvature estimates and empirical convergence (in the augmented Jacobian matrix algorithm). The fourth phase, the end game, is similar in both algorithms. This phase is a combination of the chord and the secant methods followed by Newton correction iterations in the normal flow algorithm (quasi-Newton correction iterations in the augmented Jacobian matrix algorithm)

[61]. In practice, the augmented Jacobian matrix algorithm is preferred when the evaluation of the Jacobian matrix is expensive. However, the normal flow algorithm is more suitable in general since it does not require fine tuning of algorithm parameters to exhibit good performance (in terms of the number of matrix evaluations).

The ability of homotopy algorithms to solve large real-word problems is of great importance. Such problems, which are often based on the discretization of a physical domain, usually have sparse Jacobian matrices. For storage and efficiency considerations, sparse matrix computations require techniques that are different from the ones used for dense matrices. This distinction gives rise to the dichotomy of the algorithms in HOMPACK90 into dense and sparse Jacobian matrix algorithms. Employing the same stepping procedure, sparse and dense versions of a homotopy algorithm differ mainly at the level of the underlying numerical linear algebra. The properties of sparse linear systems will be described next in the context of the normal flow algorithm.

2.2.2. Sparse normal flow algorithm.

For the prediction phase, assume that two points $P^{(1)} = (x(s_1),\lambda(s_1))$ and $P^{(2)} = (x(s_2),\lambda(s_2))$ on γ with corresponding tangent vectors $(dx/ds(s_1),\ d\lambda/ds(s_1))$, and $(dx/ds(s_2),\ d\lambda/ds(s_2))$ have been found, and h is an estimate of the optimal step (in arc length) to take along γ. The prediction of the next point on γ is

$$Z^{(0)} = p(s_2 + h),$$

where $p(s)$ is the Hermite cubic interpolating $(x(s),\lambda(s))$ at s_1 and s_2. Precisely,

$$p(s_1) = (x(s_1), \lambda(s_1)), \quad p'(s_1) = (dx/ds(s_1), d\lambda/ds(s_1)),$$
$$p(s_2) = (x(s_2), \lambda(s_2)), \quad p'(s_2) = (dx/ds(s_2), d\lambda/ds(s_2)),$$

and each component of $p(s)$ is a polynomial in s of degree less than or equal to 3.

Starting at the predicted point $Z^{(0)}$, the corrector iteration is

$$Z^{(k+1)} = Z^{(k)} - \left[D\rho_a\big(Z^{(k)}\big)\right]^{\dagger} \rho_a\big(Z^{(k)}\big), \qquad k = 0, 1, \ldots$$

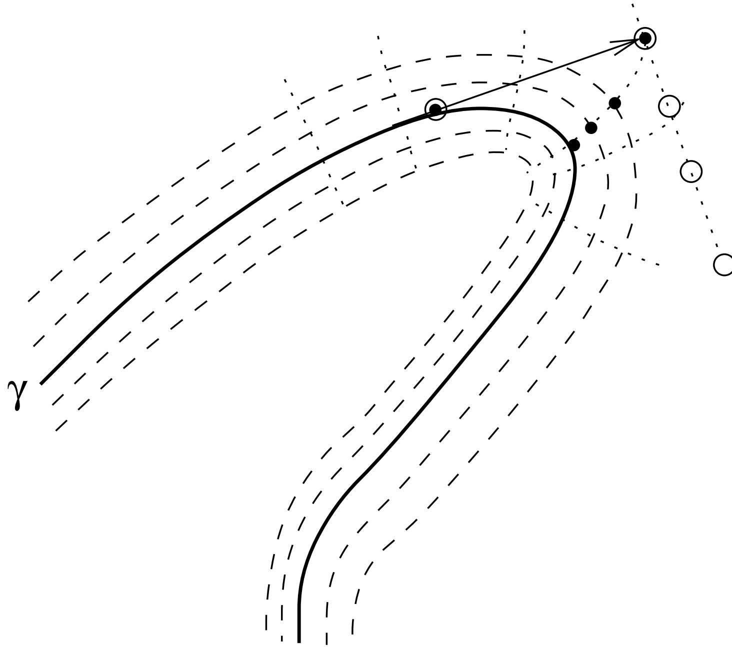

FIG. 2. *Davidenko flow, normal flow iterates* $\bullet$, *and augmented Jacobian matrix iterates* $\circ$.

where $\left[D\rho_a(Z^{(k)})\right]^\dagger$ is the Moore-Penrose pseudoinverse of the $n \times (n+1)$ Jacobian matrix $D\rho_a$. Small perturbations of a produce small changes in the trajectory γ, and the family of trajectories γ for varying a is known as the "Davidenko flow". Geometrically, the corrector iterates return to the zero curve along the flow normal to the Davidenko flow (the family of trajectories γ for varying a as in Figure 2), hence the name "normal flow algorithm". A tangent vector v to γ at Z is a unit vector in the one-dimensional kernel of $[D\rho_a(Z)]$ [69].

A corrector step ΔZ is the unique minimum norm solution of the equation

$$\left[D\rho_a\right]\Delta Z = -\rho_a.$$

Fortunately ΔZ can be calculated at the same time as the kernel of $\left[D\rho_a\right]$, and with just a little more work. In most applications where $D_x\rho_a$ is symmetric, it is also a low rank modification of a positive definite matrix. These properties, exactly as by Watson, Billups, and Morgan [71], are exploited by matrix splitting in the linear system solution process. Let $(\bar{x}, \bar{\lambda})$ be a point on the zero curve γ, and $\bar{y}$ the unit tangent vector to γ at $(\bar{x}, \bar{\lambda})$ in the direction of increasing arc length s. Let $|\bar{y}_k| = \max_i |\bar{y}_i|$. Then the matrix

$$A = \begin{bmatrix} D\rho_a(x,\lambda) \\ e_k^t \end{bmatrix},$$

where e_k is a vector with 1 in the kth component and zeros elsewhere, is invertible at $(\bar{x}, \bar{\lambda})$ and in a neighborhood of $(\bar{x}, \bar{\lambda})$ by continuity. Thus the kernel of $D\rho_a(x, \lambda)$ can be found by solving the linear system of equations

$$Ay = \bar{y}_k e_{n+1} = b.$$

The coefficient matrix A in the system $Ay = b$, whose solution y yields the kernel of $D\rho_a(x, \lambda)$, has a very special structure which can be exploited as follows. Note that the leading $n \times n$ submatrix of A is $D_x \rho_a$, which is symmetric and sparse, but possibly indefinite. Write

$$A = M + L$$

where

$$M = \begin{bmatrix} D_x \rho_a(x, \lambda) & c \\ c^t & d \end{bmatrix},$$

$$L = u e_{n+1}^t, \quad u = \begin{pmatrix} D_\lambda \rho_a(x, \lambda) - c \\ 0 \end{pmatrix}.$$

The choice of e_k^t as the last row of A to make A invertible is somewhat arbitrary, and in fact any vector (c^t, d) outside a set of measure zero (a hyperplane) could have been chosen. Other choices for the last row of A have been thoroughly studied [36]. For almost all vectors c the first n columns of M are independent, and similarly almost all $(n+1)$-vectors are independent of the first n columns of M. Therefore for almost all vectors (c^t, d) both A and M are invertible. Assume that (c^t, d) is so chosen.

Using the Sherman-Morrison formula (L is rank one), the solution y to the original system $Ay = b$ can be obtained from

$$y = \left[I - \frac{M^{-1} u e_{n+1}^t}{(M^{-1} u)^t e_{n+1} + 1} \right] M^{-1} b,$$

which requires the solution of two linear systems with the sparse, symmetric, invertible matrix M.

If the matrix $D_x\rho_a(x,\lambda)$ has an arbitrary sparsity pattern, i.e. unstructured, there is no advantage to splitting. The kernel of $D\rho_a$ is computed by solving the system with

$$A = \begin{bmatrix} D_x\rho_a(x,\lambda) & D_\lambda\rho_a(x,\lambda) \\ c^t & d \end{bmatrix},$$

(c^t, d) is chosen to guarantee that A is invertible [36]. The kernel and Newton step ΔZ are calculated together by solving

$$\begin{bmatrix} D_x\rho_a(x,\lambda) & D_\lambda\rho_a(x,\lambda) \\ c^t & d \end{bmatrix} [v\ w] = \begin{bmatrix} 0 & -\rho_a(x,\lambda) \\ (c^t, d)\bar{y} & 0 \end{bmatrix}$$

at points (x,λ) near $(\bar{x},\bar{\lambda})$ on γ, using the unit tangent vector $\bar{y}$ to γ at $(\bar{x},\bar{\lambda})$ in the direction of increasing arc length s. The Newton step ΔZ is recovered from v and w in the usual way [71].

The process of linear system solution is a vital part of any homotopy method. Without high accuracy of the linear system solution, a homotopy method often fails by losing the curve. An overview of sparse linear system solution techniques (called iterative methods) will be given in the next chapter followed by the description of a new solution method adopted in the sparse version of homotopy methods.

When the corrector iteration converges, the final iterate $Z^{(k+1)}$ is accepted as the next point on γ, and the tangent vector to the integral curve through $Z^{(k)}$ is used for the tangent—this saves a Jacobian matrix evaluation and factorization at $Z^{(k+1)}$. The step size estimation described in [71] attempts to balance progress along γ with the effort expended on the corrector iteration. The final phase of the normal flow algorithm computation of the solution at $\lambda = 1$ begins when a point $P^{(2)}$ is generated such that $P^{(2)}_{n+1} \geq 1$. This phase is crucial for the convergence of the algorithm because the predictor-corrector algorithm alone does not guarantee convergence to a zero of $\rho_a(x,\lambda)$ at which $\lambda = 1$.

3. ITERATIVE SOLUTION TECHNIQUES AND PRECONDITIONING.

3.1. Iterative Methods.

Computational challenges of the contemporary world make iterative methods significant "team players" in a sophisticated problem solving process, the overall success of which largely depends on sparse linear system solutions. Iterative solution techniques are prominent due to their moderate computational and storage demands: they do not require matrix factorization and preserve the sparsity of matrices, which is crucial for large scale applications. Among the most effective and frequently used iterative methods are Krylov subspace methods.

A *Krylov subspace* method is a projection method that solves the linear system

$$Ax = b,$$

where A is an $n \times n$ coefficient matrix, by finding an approximate solution x_m from an affine subspace $x_0 + \mathcal{K}_m(A, r_0)$ such that

$$b - Ax_m \perp \mathcal{L}_m,$$

where $\mathcal{L}_m$ and $\mathcal{K}_m(A, r_0)$ are subspaces of dimension m. The subspace

$$\mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, A^2 r_0, \cdots, A^{(m-1)} r_0\},$$

where $r_0 = b - Ax_0$, is called the Krylov subspace.

Different choices of the subspace $\mathcal{L}_m$ lead to variations in Krylov subspace methods. If $\mathcal{L}_m = \mathcal{K}_m(A, r_0)$, the resulting Krylov subspace method is known as *Arnoldi's* method or the *full orthogonalization* method (see, e.g., [53]). If $\mathcal{L}_m = A\mathcal{K}_m(A, r_0)$, then the Krylov subspace method is characterized by the residual minimization property over all vectors in $x_0 + \mathcal{K}_m(A, r_0)$ and is called the *generalized minimum residual* method GMRES [55]. If $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0)$, the associated Krylov subspace methods for solving nonsymmetric linear systems are termed nonsymmetric Lanczos-type methods. Such methods include the *biconjugate gradient* method BiCG [44], the *quasi-minimal residual* method QMR [24], and their transpose-free counterparts: the *biconjugate gradient stabilized* method BiCGSTAB [65] (a version of the *conjugate gradient squared* algorithm) and the *transpose-free quasi-minimal residual* method TQMR [24].

3.1.1. Arnoldi's procedure.

At the core of all Krylov subspace methods lies the construction of an orthogonal basis for the Krylov subspace. This basis is often called an Arnoldi basis, since it is implemented by the Arnoldi procedure, described as follows.

choose a vector v_1 of unit norm

for $j = 1$ **step** 1 **until** m **do**

begin

  for $i = 1$ **step** 1 **until** j **do** $h_{i,j} := (Av_j)^T v_i$;

  $\tilde{v}_{j+1} := Av_j - \sum_{i=1}^{j} h_{i,j} v_i$;

  $h_{j+1,j} := \|\tilde{v}_{j+1}\|$;

  if $h_{j+1,j} = 0$ **then** exit**;**

  $v_{j+1} := \tilde{v}_{j+1} \,/\, h_{j+1,j}$;

end

3.1.2. The GMRES method.

GMRES approximates the linear system solution with the unique vector $x_m \in x_0 + \mathcal{K}_m(A, r_0)$ that minimizes $\|b - Ax\|_2$. The work and memory required by GMRES grow proportionately to the subspace $\mathcal{K}_m(A, r_0)$ dimension since GMRES needs all m vectors to construct an orthogonal (Arnoldi) basis of $K_m(A, r_0)$. In practice, the restarted version, GMRES(k), is used, where the algorithm is restarted every k iterations, taking x_k as the initial guess for the next cycle of k iterations, until the residual norm is small enough. Pseudocode for GMRES(k) is

choose x, tol

$r := b - Ax$;

while $\|r\| > tol$ **do**

begin

do Arnoldi's procedure with $m = k$ and $v_1 := r/\|r\|$;

$e_1 := (1, 0, \cdots, 0)^T$;

solve $\min_y \| \, \|r\| e_1 - \bar{H}_j y \|$ for y_j where $\bar{H}_j$ is described in [53];

$V_j := [v_1, \cdots, v_j]; \qquad x := x + V_j y_j; \qquad r := b - Ax$

end

3.1.3. The QMR algorithm.

The quasi-minimal residual method is based on a modification of the classical nonsymmetric Lanczos algorithm. Given two starting vectors v_1 and w_1, the Lanczos algorithm uses three-term recurrences to generate sequences $\{v_i\}_{i=1}^L$ and $\{w_i\}_{i=1}^L$ of vectors satisfying

$$\mathrm{span}\{v_1, \cdots, v_m\} = \mathrm{span}\{v_1, Av_1, \cdots, A^{m-1} v_1\},$$
$$\mathrm{span}\{w_1, \cdots, w_m\} = \mathrm{span}\{w_1, A^T w_1, \cdots .(A^T)^{m-1} w_1\},$$

for $m = 1, \cdots, L$, and $w_i^T v_j = d_i \delta_{ij}$, with $d_i \neq 0$, for all $i, j = 1, \cdots, L$, where δ_{ij} is the Kronecker delta.

An implementation of QMR based on the look-ahead Lanczos process avoids most breakdowns associated with Lanczos-type algorithms [23]. At the kth iteration, QMR computes an iterate x_k as

$$x_k = x_0 + V^{(k)} z,$$

where the columns of the matrix $V^{(k)}$ are the right look-ahead Lanczos vectors $v_1, \cdots, v_k$, and the vector z is the unique minimizer of

$$\left\| ||r_0|| e_1 - H_e^{(k)} z \right\|,$$

where $H_e^{(k)}$ is a $k \times k$ block tridiagonal matrix augmented with a row of the form ρe_k^T.

3.2. Preconditioning.

Using iterative methods in lieu of direct methods poses the question impertinent to solving a linear system with a direct method: how fast can a solution of the desired accuracy be obtained? The answer often depends on the algebraic properties of the matrix, such as the condition number and the matrix spectrum. For ill conditioned matrices, iterative methods are notorious for their slow convergence and lack of robustness. To make iterative methods more reliable, the technique called *preconditioning* is used, which is essentially a technique for substituting another linear system for the original one, such that the solutions of both linear systems are the same and that an iterative method is likely to solve the system-substitute more easily.

Let M be any nonsingular $n \times n$ matrix. The solution to the system $Ax = b$ can be calculated by solving the system

$$Bx = (M^{-1}A)x = M^{-1}b = g.$$

The use of such a matrix is known as left preconditioning. Since the goal of using preconditioning is to decrease the computational effort needed to solve the original system, M should be inexpensive to construct and apply and should approximate A in some way. Then $M^{-1}A$ would be close to the identity matrix and iterative methods would likely converge more rapidly when applied to the preconditioned rather than the original system. In practice, B is never explicitly formed. Right preconditioning has the form

$$Bz = (AM^{-1})z = AM^{-1}Mx = Ax = b,$$

where the final step is to compute $x = M^{-1}z$. The spectra of the left and right preconditioning are the same, and the rates of convergence of an iterative method with either left or right preconditioning are practically the same, as shown in [53], unless the preconditioning matrix M is itself poorly conditioned.

Right preconditioning, however, allows more flexibility in choosing a particular preconditioning technique. Several variations of iterative methods are designed based on

this property of right preconditioning. One such variation is the *flexible* GMRES method FGMRES [52]. Flexible GMRES(k) differs from the standard preconditioned GMRES(k) implementation by allowing preconditioning to change at each iteration; that is, the matrix B in the preconditioned linear system $Bz = b$ changes at each iteration. In particular, let M_i be the computed preconditioner at the ith iteration. Then $B_i = AM_i^{-1}$ at the ith iteration. At the end of a restart cycle, an iterate x_k is given by

$$x_k = x_0 + \left[M_1^{-1}v_1, \cdots, M_k^{-1}v_k\right] y_k,$$

where y_k is the solution to the GMRES least-squares problem and the v_i are the same as in the standard GMRES(k). As an example of preconditioning in flexible GMRES(k), consider GMRES-ILU(0) preconditioning, where the application of M_i corresponds to an "inner" method consisting of GMRES($\tilde{k}$) preconditioned with the ILU(0) factorization (described in the next paragraph). In GMRES($\tilde{k}$), $\tilde{k} \ll k$ and the total number of iterations allowed is also much smaller than in flexible GMRES(k), the "outer" method.

One way to construct the preconditioning matrix M is by means of *incomplete LU factorization* (ILU factorization) of the original matrix A , such that

$$A = LU + E \quad \text{and} \quad M = LU,$$

where L and U are lower triangular and unit upper triangular matrices, respectively, and E is the error of the factorization. The number and pattern of nonzero entries in L and U distinguish different types of ILU factorization of A. For example, an obvious choice for the nonzero structure of the L and U factors is to take it identical to the sparsity pattern of A. The corresponding incomplete factorization of A is termed ILU factorization with zero fill-in (ILU(0)) and defined as follows. Let $Z \subset \{(i,j)|1 \leqq i,j \leqq n,\ i \neq j\}$ be a subset of the set of indices where the matrix A has zeros. Then the ILU(0) factorization of A, given by $M = LU$, satisfies

$$\begin{aligned} L_{ij} = U_{ij} = 0, &\qquad \text{for } (i,j) \in Z; \\ M_{ij} = A_{ij}, &\qquad \text{for } (i,j) \notin Z, i \neq j; \\ M_{ii} = A_{ii}, &\qquad \text{whenever possible.} \end{aligned}$$

The Gill-Murray preconditioner [25] for the matrix A with a symmetric nonzero structure is a positive definite approximation $M = GG^T$ to A. The lower triangular matrix G has the same nonzero structure as the lower (upper) triangular submatrix of A, and is chosen such that G is always well conditioned. ILU(0) preconditioning may lead to a crude approximation of the original matrix A. In this case, some other ILU factorization is more appropriate; that is, a factorization with L and U having more nonzeros than in the original matrix. Different types of ILU factorization are discussed extensively in [53]. The next chapter considers practical applications of the iterative methods and preconditioning techniques defined here in the context of homotopy methods.

4. ADAPTATION OF GMRES(k) FOR HOMOTOPY METHODS.

Robustness and versatility are the two main criteria for sparse linear algebra algorithms for homotopy methods so that they can deal successfully with a wide range of difficult problems. In the context of homotopy zero curve tracking algorithms to locate the difficult parts of the curve (sharp turning points) a linear system solution in the correction phase of the stepping algorithm along the curve has to be very accurate. In this chapter, the ability of the generalized minimal residual algorithm (GMRES) [55] to deal successfully with matrices of varying difficulty and to give high accuracy solutions is investigated.

High accuracy computation is known to be especially important for some circuit design and simulation problems, which involve nonsymmetric unstructured matrices [48]. These problems have proven to be very difficult for any iterative method, and presently, only specially tailored direct methods are used to solve them in a production environment. The impact of numerical error is shown here to be significant in the postbuckling stability analysis of structures which exhibit snap-back and snap-through phenomena. A postbuckling analysis amounts to tracking what is called an equilibrium curve, involving the solution of linear systems with (tangent stiffness) matrices that vary along the equilibrium curve. Mathematically, the snap-back and snap-through behavior of structures corresponds to symmetric, structured, and strongly indefinite tangent stiffness matrices, which can be difficult for Krylov subspace iterative methods. Along some portions of the curve the tangent stiffness matrices may be well conditioned and positive definite—easy for iterative methods. This wide variation in linear system difficulty clearly suggests an adaptive strategy. With good preconditioning, the details of an iterative solver become less important, and for many real problems good preconditioning strategies (e.g., multigrid) are available. For some classes of problems, like analog DC circuit design and structural postbuckling stability analysis, good preconditioning strategies are not known, and subtle implementation details of iterative algorithms become paramount.

Section 4.1. develops the proposed subspace enlarging strategy for GMRES(k) and addresses some technical issues of the GMRES implementation. Section 4.2. describes

realistic test problems for numerical experiments. The numerical results are presented and discussed in Section 4.3. followed by conclusions in Section 4.4.

4.1. An Implementation of the Adaptive GMRES(k) Algorithm.

A fairily robust iterative method, GMRES(k) [55], described in the previous chapter, has a disadvantage: It may stagnate and never reach the solution if the Krylov subspace dimension used in a restart cycle is not large enough for a given problem. The essence of the adaptive GMRES strategy proposed here is to adapt the restart cycle parameter k to the problem, similar in spirit to how a variable order ODE algorithm tunes the order k. With FORTRAN 90, which provides pointers and dynamic memory management, dealing with the variable storage requirements implied by varying k is not too difficult. The parameter k can be both increased and decreased—an increase-only strategy is described next followed by pseudocode in Figure 3.

Walker [68] discusses a stagnation test for singular systems, applied only before a new restart cycle, but does not propose an adaptive strategy with the subspace dimension. The details of the adaptive strategy with multiple stagnation tests proposed here are substantially different from earlier work. If k in GMRES(k) is not sufficiently large, GMRES(k) can stagnate. A test of stagnation developed in [10], [68] detects an insufficient residual norm reduction in the restart number k of steps by estimating the GMRES behavior on a particular linear system. Precisely, GMRES(k) is declared to have stagnated and the iteration is aborted if at the rate of progress over the last restart cycle of steps, the residual norm tolerance cannot be met in some large multiple (*bgv*) of the remaining number of steps allowed (*itmax* is a bound on the number of steps permitted). Slow progress of GMRES(k) which indicates that an increase in the restart value k may be beneficial [66] can be detected with a similar test. The near-stagnation test uses a different, smaller multiple (*smv*) of the remaining allowed number of steps. If near-stagnation occurs, the restart value k is incremented by some value m and the *same* restart cycle continues. Restarting would mean repeating the nonproductive iterations that previously resulted in stagnation, at least in the case of complete stagnation (no residual reduction at all). Such incrementing is used whenever

needed if the restart value k is less than some maximum value $kmax$. When the maximum value for k is reached, adaptive GMRES(k) proceeds as GMRES($kmax$). The values of the parameters smv, bgv, and m are established experimentally and can remain unchanged for most problems.

The convergence of GMRES may also be seriously affected by roundoff error, which is especially noticeable when a high accuracy solution is required. The orthogonalization phase (performed by the Arnoldi's procedure) of GMRES is susceptible to numerical instability, even though a more stable version of it (the modified Gram-Schmidt process) is used in practice.

Let Q be a matrix whose columns are obtained by orthogonalizing the columns of a matrix M, and define the error matrix $E = Q^T Q - I$. The matrices E_{MGS} and E_{HR} are the error matrices for modified Gram-Schmidt and Householder reflection methods, respectively. When Q is constructed from M, they satisfy [7]

$$\|E_{MGS}\|_2 \sim \mathbf{u}\,\mathrm{cond}(M), \qquad \|E_{HR}\|_2 \sim \mathbf{u},$$

where $\mathbf{u}$ is the machine unit roundoff. Clearly the orthogonalization with Householder reflections is more robust. An implementation of GMRES(k) using Householder reflections and its block version are given in [67]. (The convergence properties of the block version on the test problems here are no better than those of the point version, and so block versions are not considered further here.) In theory, the implementation of GMRES using Householder reflections is about twice as expensive as when modified Gram-Schmidt is used [68]. However, the Householder reflection method produces a more accurate orthogonalization of the Arnoldi basis when the basis vectors are nearly linearly dependent and the modified Gram-Schmidt method fails to orthogonalize the basis vectors; this can result in fewer GMRES iterations compensating for the higher cost per iteration using Householder reflections. Let e_j be the jth standard basis vector and all norms the 2-norm. Pseudocode for an adaptive version of GMRES(k) with orthogonalization via Householder reflections implemented as in [67] and [68] follows. Call this algorithm AGMRES(k).

choose $x, itmax, kmax, m$;

$r := b - Ax$; $itno := 0$; $cnmax := 1/(50\mathbf{u})$;

$xtol := \max\{100.0, 1.01 avnz\}\mathbf{u}$; $tol := \max\{\|r\|, \|b\|\} xtol$;

while $\|r\| > tol$ **do**

begin

  $r^{old} := r$;

  determine $P_1 r = \pm\|r\| e_1$

   where the Householder transformation matrix P_1 is defined in [26];

  $k_1 = 1$; $k_2 = k$;

L1: **for** $j := k_1$ **step** 1 **until** k_2 **do**

  begin

   $itno := itno + 1$;

   $v := P_j \cdots P_1 A P_1 \cdots P_j e_j$;

   determine P_{j+1} such that $P_{j+1} v$ has zero components

    after the $(j+1)$st;

   update $\|r\|$ as described in [55];

   estimate condition number $\text{cond}(AV_j)$ of GMRES least squares

    problem via the incremental condition number ICN as in [9];

   if $ICN > cnmax$ **then** abort;

   if $\|r\| \leqq tol$ **then goto** L2

  end

  $test := k_2 \times \log\left[tol/\|r\|\right] / \log\left[\|r\|/\left((1.0 + 10\mathbf{u})\,\|r^{old}\|\right)\right]$;

  if $k_2 \leqq kmax - m$ **and** $test \geqq smv \times (itmax - itno)$ **then**

   $k_1 := k_2 + 1$; $k_2 := k_2 + m$;

goto L1

end if

L2: $e_1 := (1, 0, \cdots, 0)^T$; $k := k_2$;

solve $\min_y \big\| \|r\| e_1 - \bar{H}_j y \big\|$ for y_j where $\bar{H}_j$ is described in [55];

$q := \begin{pmatrix} y_j \\ 0 \end{pmatrix}$; $x := x + P_1 \cdots P_j q$; $r := b - Ax$;

if $\|r\| \leqq tol$ **then** exit;

if $\|r^{old}\| < \|r\|$ **then**

if $\|r\| < tol^{2/3}$ **then**

exit

else

abort

end if

end if

$test := k \times \log\left[tol/\|r\|\right] / \log\left[\|r\| / \left((1.0 + 10\mathbf{u})\, \|r^{old}\|\right)\right]$;

if $test \geqq bgv \times (itmax - itno)$ **then**

abort

end if

end

FIG. 3. *Pseudocode for adaptive GMRES(k).*

The rounding error of a sparse matrix-vector multiplication depends on only the nonzero entries in each row of the sparse matrix, so the error tolerance $xtol$ is proportional to the average number of nonzeros per row $avnz =$ (number of nonzeros in A)$/n$. The code from [68] assumes that the initial estimate $x_0 = 0$, which is certainly not the case for homotopy zero curve tracking, where the previous tangent vector provides a good estimate x_0. Since

GMRES convergence is normally measured by reduction in the initial residual norm, the convergence tolerance is $tol = \max\{\|r_0\|, \|b\|\} xtol$.

A possible symptom of AGMRES(k) going astray is an increase in the residual norm between restarts (the residual norm is computed by direct evaluation at each restart). If the residual norm on the previous restart is actually smaller than the current residual norm, then AGMRES(k) terminates. The solution is considered acceptable if $\|r\| < tol^{2/3}$, although this loss of accuracy may cause the client algorithm (the "outer" algorithm requiring solutions to linear systems) to work harder or fail. A robust client algorithm can deal gracefully with a loss of accuracy in the linear system solutions. If $\|r\| \geqq tol^{2/3}$, AGMRES(k) is deemed to have failed. In this latter case, the continuation of GMRES(k) would typically result in reaching a limit on the number of iterations allowed and a possible repetition of $\|r^{old}\| < \|r\|$ in later restarts. AGMRES(k) may exceed an iteration limit when it is affected by roundoff errors in the case of a (nearly) singular GMRES least-squares problem. The condition number of the GMRES least-squares problem is monitored by the incremental condition estimate [9] as in [10]. AGMRES(k) aborts when the estimated condition number is greater than $1/(50\mathbf{u})$.

4.2. Numerical Experiments.

A comparative study of the proposed adaptive version of GMRES(k) considers some other versions of GMRES(k): the standard implementation of GMRES(k) [55], the adaptive GMRES(k) with modified Gram-Schmidt orthogonalization, GMRES(k) with GMRES-ILU(0) preconditioning [52], and the popular iterative method QMR [24].

For future reference the considered algorithms will be called:

AGH — adaptive restarted GMRES with Householder reflections in the orthogonalization phase;

AGS — adaptive restarted GMRES with modified Gram-Schmidt in the orthogonalization phase;

FGS — flexible restarted GMRES with $\tilde{k} = k/3$ and the number of iterations equal to $itmax/20$ in an "inner" method;

GS — standard implementation of restarted GMRES;

Q — three-term recursion QMR.

4.2.1. Circuit design and simulation.

The circuit simulation test problems considered here are from [46] and [48], and are representative of full-scale commercial design at AT&T Bell Laboratories. The DC operating point problem is typically formulated as a system of n equations in n unknowns, i.e., $F(x) = 0$, where $F : E^n \to E^n$, and E^n is n-dimensional real Euclidean space. The unknowns are voltages and branch currents and the equations represent the application of Kirchoff's current and voltage laws at various points in the circuit. Different circuit components (resistor, transistor, diode, etc.) impose linear and nonlinear relations among the unknowns. In present fabrication technologies, circuit components are arranged in the form of a graph that is almost planar, which limits the density of interconnections between circuit elements, leading to a sparse Jacobian matrix. Figure 4 shows one of the circuits (Brokaw voltage reference [48], "vref") from which the test problems were derived.

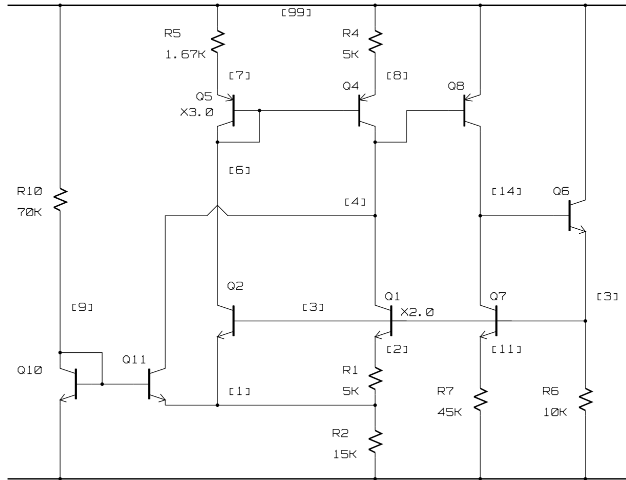

FIG. 4. *Circuit diagram for the circuit "vref".*

Except for rather special cases, circuit equations lead to large nonsymmetric unstructured matrix problems. In some cases, a circuit element like a transistor is replicated as a small "subcircuit", which is installed in place of every transistor in the network. This policy results in a replication of structurally identical submatrices throughout the overall system Jacobian matrix. The sparsity pattern for the Jacobian matrix corresponding to one of the

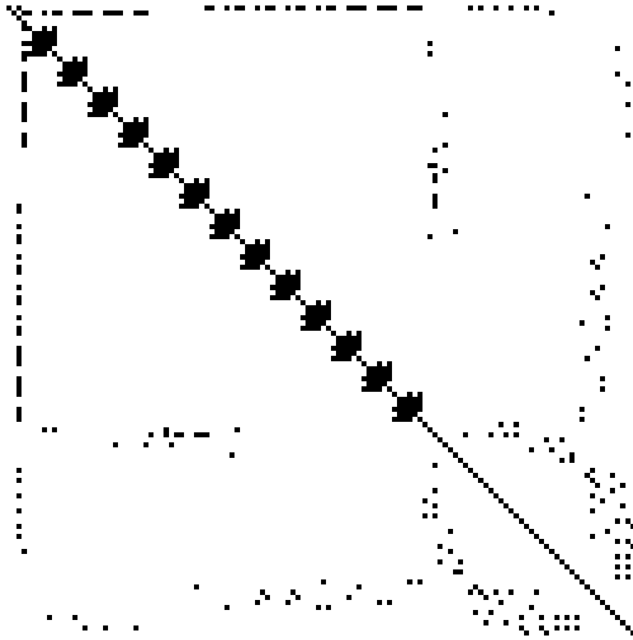

FIG. 5. *Jacobian matrix sparsity pattern corresponding to the circuit "bgatt".*

test circuits (bgatt) with the diagonal blocks corresponding to the transistor subcircuits is shown in Figure 5.

For each test problem, the various iterative methods were applied to a sequence of Jacobian matrices obtained along a homotopy zero curve for that problem. For the rest of the test problems described below, the iterative methods were actually implemented as subroutines within the homotopy curve tracking software HOMPACK [71]. Specifically a FORTRAN 90 version of the normal flow subroutine FIXPNS was used, resulting in the $(n+1) \times (n+1)$ linear systems described in [36]—note that these are slightly different from the linear systems for the sparse normal flow algorithm given in [71].

4.2.2. Space truss stability analysis.

The stability characteristics of a space truss under multiple independent loads are described in [32]. A brief derivation of the nonlinear equilibrium equations, whose solution by a homotopy method generates the linear systems to be solved, is given here.

Finite element model. The axial deformation of the truss element in Figure 6 is $e = L - L_0$, where L_0 and L are the element lengths in the initial and deformed states,

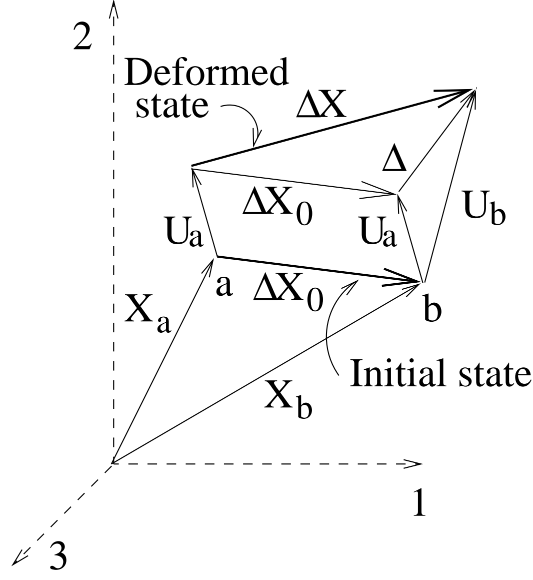

FIG. 6. *Space truss element.*

respectively. It follows from Figure 6 that

$$L = \|\boldsymbol{\Delta} X\| = \left(\sum_{j=1}^{3} L_j^2 \right)^{1/2}, \qquad L_0 = \|\boldsymbol{\Delta} X_0\| = \left(\sum_{j=1}^{3} L_{0j}^2 \right)^{1/2},$$

$$L_j = L_{0j} + \Delta_j, \quad \boldsymbol{\Delta} X_0 = X_b - X_a, \quad \Delta = U_b - U_a, \quad \boldsymbol{\Delta} X = \boldsymbol{\Delta} X_0 + \Delta,$$

where X_a, X_b are the initial coordinate vectors at the a, b-end of the element; U_a, U_b are the global displacement vectors at the a, b-end of the element; and the vectors $\boldsymbol{\Delta} X_0$, $\boldsymbol{\Delta} X$ coincide with the initial, deformed state of the element. The strain energy of the element is

$$\pi = \frac{1}{2} \gamma_0 e^2,$$

where $\gamma_0 = EA/L_0$ is the extensional stiffness of the element, E is Young's modulus of elasticity, and A is the cross sectional area of the truss element. The principle of virtual work yields the equation of equilibrium

$$F(q, \lambda, \bar{Q}) = -\lambda \bar{Q} + \sum_{i=1}^{m} R^i = 0,$$

where q is the generalized displacement vector, λ is the scalar load parameter, $\bar{Q}$ is the load distribution vector, the vector function F is the force imbalance, and

$$R_k^i = \frac{\partial \pi^i}{\partial q_k}, \quad \text{for } k = 1, 2, \cdots, n.$$

The superscript i identifies the m elements of the assemblage. The $n \times (n+1)$ Jacobian matrix becomes

$$DF = \left[K, \; -\bar{Q}\right], \quad \text{where } K = \sum_{i=1}^{m} K^i, \quad K^i = \left[\frac{\partial^2 \pi^i}{\partial q_k \partial q_l}\right].$$

K^i is the tangent stiffness matrix of the element i expressed relative to the generalized displacements of the assemblage. It is computed by the code number method [72] from the global stiffness matrix of element i.

Structure 1. Figure 7 shows a 21-degree-of-freedom lamella dome. The joints lie on the surface of a spherical cap with a radius of 157.25 inches. The support ring of the cap has a radius of 50 inches. All joints are located so that the structure is symmetric about the global x_1-axis which extends down vertically through the apex. The six support joints are fixed in all three directions. Each member of the structure has a cross-sectional area of 0.18 square inches and an elastic modulus of 2900 ksi. The structure is subjected to a vertical load at the apex.

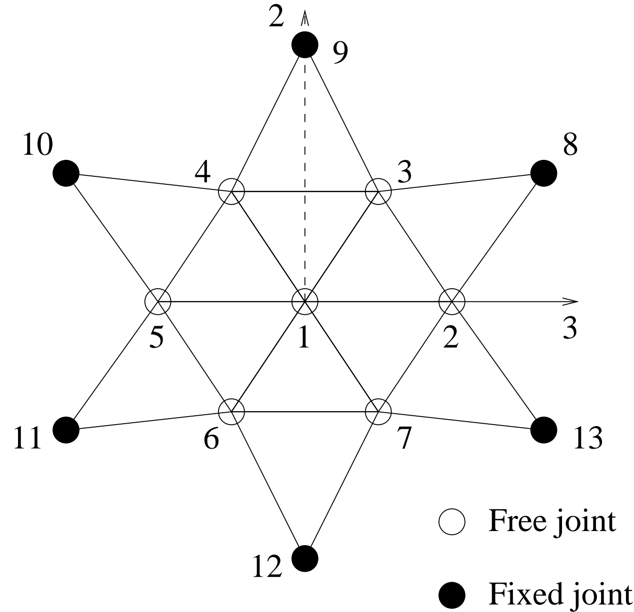

FIG. 7. *Lamella dome (21-degree-of-freedom).*

Structure 2. This is another lamella dome with a tension ring at the base having 69 degrees of freedom, similar to Structure 1 but with a different geometrical arrangement of the free joints, different joint constraints, and an unsymmetric loading pattern [32]. The effect of modeling a space truss 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 considered here are qualitatively different, with different buckling loads.

4.2.3. Shallow shell stability analysis.

Another test problem is the finite element analysis of a shallow cylindrical shell (Figure 8) under a concentrated load [76]. The shell shown in Figure 8 is hinged at the longitudinal edges and free along the curved boundaries.

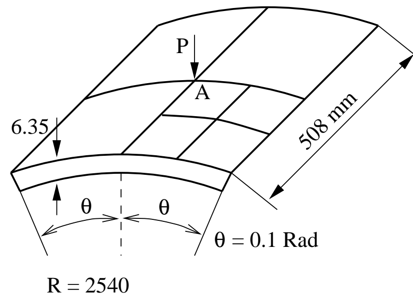

FIG. 8. *Cylindrical shell geometry; load applied at A.*

The structure exhibits snap-through as well as snap-back phenomena, with horizontal and vertical tangents. A finite element model based on a curved quadrilateral 48-degree-of-freedom thin-shell element (Figure 9) is used to form the equations of equilibrium [42]. The element has four corner nodes and each node has 12 degrees of freedom. Figure 9 shows the undeformed middle surface of the shell element embedded in a fixed Cartesian coordinate system x^i ($i = 1, 2, 3$). The Cartesian coordinates x^i ($i = 1, 2, 3$) of the middle surface are modeled by polynomial functions of the curvilinear coordinates ξ and η with a total of N terms:

$$x^i(\xi, \eta) = \sum_{j=1}^{N} C_j^i \xi^{m_j} \eta^{n_j},$$

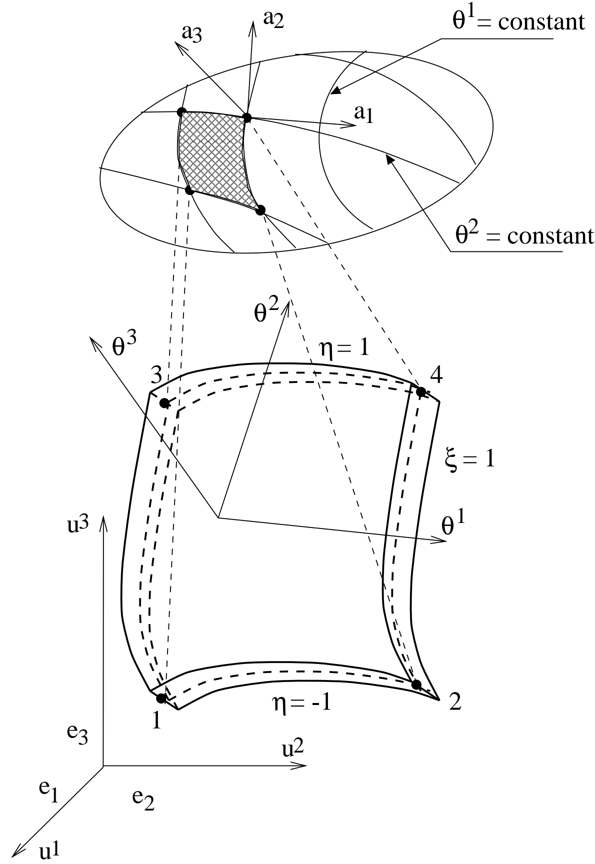

FIG. 9. *Thin-shell element coordinate system.*

where the constants m_j and n_j define the powers of ξ and η, respectively, for the jth term. The constants C_j^i are solved for based on the coordinates x_j at N selected points. Precisely, bicubic Hermite polynomials in the curvilinear coordinates ξ and η are used as basis functions for the interpolating polynomial.

The Cartesian coordinates of a given point are described by a system of parametric equations

$$x^i = f^i(\theta^1, \theta^2),$$

where the parameters θ^α ($\alpha = 1, 2$) serve as coordinates on the surface and can be regarded as an arbitrary selected system.

The two base vectors a_1 and a_2 are obtained as derivatives of f with respect to each component θ^α:

$$a_\alpha = \frac{\partial f^i}{\partial \theta^\alpha} e_i = f^i_\alpha e_i,$$

where the e_i are the standard basis vectors, and tensor notation is being employed. The unit normal a_3 to the surface is given as

$$a_3 = \frac{a_1 \times a_2}{|a_1 \times a_2|} = n^i e_i,$$

where $n^i = c e_{ijk} f_1^j f_2^k$, e_{ijk} is the alternating symbol, $c = 1/|a_1 \times a_2| = 1/\sqrt{a}$,

$$a = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}, \qquad a_{\alpha\beta} = a_\alpha \cdot a_\beta = f_\alpha^i f_\beta^i.$$

The metric tensor $a_{\alpha\beta}$ yields the first fundamental form of a surface $ds^2 = a_{\alpha\beta} d\theta^\alpha d\theta^\beta$, where ds is the distance between two neighboring points at θ^α and $\theta^\alpha + d\theta^\alpha$ $(\alpha = 1,\ 2)$. The curvature tensor $b_{\alpha\beta}$ is given as

$$b_{\alpha\beta} = n^i \frac{\partial^2 f^i}{\partial\theta^\alpha \partial\theta^\beta} = n^i f_{\alpha\beta}^i.$$

After deformation, the same point has Cartesian coordinates $\bar{f}^i = f^i + u^i$, where u^i $(i = 1,\ 2,\ 3)$ are the Cartesian components of the displacement vector. The 12 degrees of freedom for each node are u^i, u_1^i, u_2^i, u_{12}^i $(i = 1,\ 2,\ 3)$. In the deformed state, the metric and curvature tensors take the form

$$\bar{a}_{\alpha\beta} = (f^i + u^i)_\alpha (f^i + u^i)_\beta = \bar{f}_\alpha^i \bar{f}_\beta^i, \qquad \bar{b}_{\alpha\beta} = \bar{n}^i \bar{f}_{\alpha\beta}^i,$$

where

$$\bar{n}^i = (\bar{a})^{-1/2} e_{ijk} \bar{f}_1^j \bar{f}_2^k, \qquad (\bar{a})^{-1/2} \approx (a)^{-1/2}(1 - A/a),$$

$$A = (\varepsilon_{11} a_{22} + \varepsilon_{22} a_{11} - 2\varepsilon_{12} a_{12}).$$

The tangential strain measure $\varepsilon_{\alpha\beta}$ is given as

$$\varepsilon_{\alpha\beta} = \frac{1}{2}(\bar{a}_{\alpha\beta} - a_{\alpha\beta}) = \frac{1}{2}\left(f_\alpha^i u_\beta^i + f_\beta^i u_\alpha^i + u_\alpha^i u_\beta^i\right), \qquad i = 1, 2, 3.$$

The curvature strain measure $\kappa_{\alpha\beta}$ is given as

$$\kappa_{\alpha\beta} = -\left(\bar{b}_{\alpha\beta} - b_{\alpha\beta}\right) + \frac{1}{2}\left(b_\alpha^\delta \varepsilon_{\beta\delta} + b_\beta^\delta \varepsilon_{\alpha\delta}\right),$$

where b_α^δ is defined in terms of the contravariant tensor $a^{\beta\delta}$ by $b_\alpha^\delta = b_{\alpha\beta} a^{\beta\delta}$.

The total potential energy of the model can be written as

$$\Pi_p(u) = \int\!\!\int_A \left[W(\varepsilon_{\alpha\beta}(u), \kappa_{\alpha\beta}(u)) - p^i u^i \right] \sqrt{a}\, d\theta^1 d\theta^2,$$

where W is the strain energy density per undeformed middle surface unit of the shell, p^i ($i = 1$, 2, 3) are the Cartesian components of the externally applied load, and u^i ($i = 1$, 2, 3) are the Cartesian components of the displacement vector. The strain energy W is given by

$$W = \frac{1}{2} H^{\alpha\beta\lambda\mu} \left(\varepsilon_{\alpha\beta}\varepsilon_{\lambda\mu} + \frac{h^2}{12} \kappa_{\alpha\beta}\kappa_{\lambda\mu} \right),$$

where h is the thickness of the shell. The tensor of elastic moduli $H^{\alpha\beta\lambda\mu}$ is defined as

$$H^{\alpha\beta\lambda\mu} = \frac{Eh}{2(1+\nu)} \left[a^{\alpha\lambda}a^{\beta\mu} + a^{\alpha\mu}a^{\beta\lambda} + \frac{2\nu}{1-\nu} a^{\alpha\beta}a^{\lambda\mu} \right],$$

where E is Young's modulus, ν is Poisson's ratio, $a^{\alpha\beta} a_{\lambda\beta} = \delta_{\alpha\lambda}$ defines the contravariant tensor $a^{\alpha\beta}$, and $\delta_{\alpha\lambda}$ is the Kronecker symbol.

The equilibrium equations of the model are obtained by setting the variation $\delta\Pi_p(u)$ to zero, or equivalently

$$\nabla\, \Pi = 0.$$

The Jacobian matrix of the equilibrium equations is obtained by finite difference approximations.

By symmetry, only a quarter of the shell is modeled using a 2×2 mesh. The material properties are $\nu = 0.3$ and $E = 3.101$ kN/mm^2. An initial external concentrated load of 0.2 kN is applied. The equilibrium curve (load factor λ vs. displacement at the point A) of the cylindrical shell is shown in Figure 10, where the actual load is 0.2λ kN.

4.3. Results.

Tables 1–6 show the iteration counts and timing results for adaptive GMRES(k) with modified Gram-Schmidt and Householder reflection orthogonalization procedures (AGS, AGH respectively) and different subspace increment values m, standard GMRES(k) (GS),

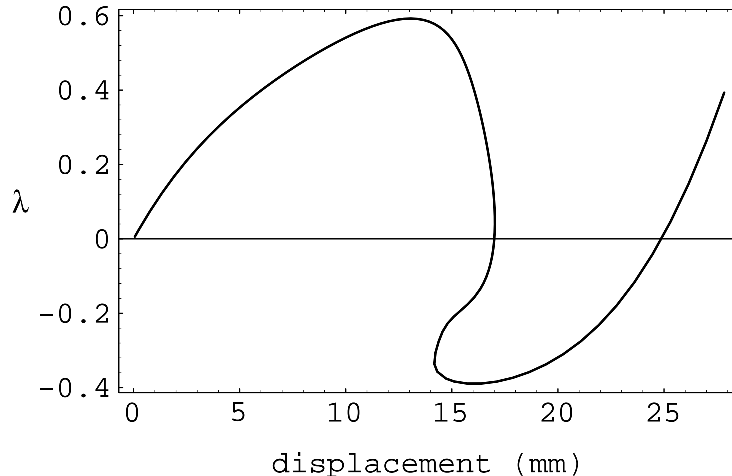

FIG. 10. *Load factor λ vs. displacement u^3 at the point A for the shell in Fig. 8.*

flexible GMRES(k) (FGS), and QMR (Q). The notation AGH:2, for example, means AGH with $m = 2$. Each of the iterative methods was applied to all of the test problems. An iterative solver is deemed to have converged when the relative residual norm reduction is less than $\max\{100, avnz\}\mathbf{u}$. The limit on the number of iterations is $30n$. This number is quite reasonable for realistic problems to be solved on modern computer hardware. This iteration limit requires explanation. First, ideally convergence should occur in far fewer iterations than n, and for well preconditioned PDE problems this usually occurs. However, for other classes of problems and for mediocre preconditioners, more than n iterations may be required, and permitting $30n$ iterations is preferable to permitting very large subspace dimensions k or permitting failure of the linear system solution server to return an accurate solution to the client homotopy algorithm. Second, low accuracy might be acceptable if only Newton corrections were being computed (although inaccurate Newton steps will wreak havoc on step size control algorithms used in homotopy curve tracking), but inaccurate tangents may result in losing the path or in very small, inefficient steps near extremely sharp turns of the homotopy path. Clearly high accuracy is not required *everywhere* along the homotopy path, and effectively adjusting the linear system solution accuracy along the homotopy path, while preserving reliable path step size control, is an open research question.

All reported CPU times are for 64-bit IEEE arithmetic on a DEC Alpha 3000/600 workstation. The average, maximum, and minimum number of iterations per linear system solution along the homotopy zero curve are shown in Tables 1-3. Note that in the FGS column the numbers for the x_k iterations of the "outer method", flexible GMRES(k), are given. Tables 5, 6 report the total CPU time in seconds required for an iterative method to obtain all the linear system solutions needed for HOMPACK to track a homotopy zero curve for a certain arc length specified for each problem. For the circuit problems (Table 4), the reported CPU time is the time necessary to solve a certain number of linear systems arising along a portion of the homotopy zero curve.

An asterisk denotes convergence failure, meaning any of the following occurred: (1) desired error tolerance not met after $30n$ iterations, (2) dangerous near singularity detected for the GMRES least squares problem, (3) residual norm increased between restart cycles and was not small, (4) stagnation occurred for GMRES-like methods. In Tables 1–3, the numbers in parentheses (k) show the restart values k whenever applicable. For AGS and AGH, the notation ($a \rightarrow b$) shows the initial restart value a and the maximum restart value b reached by the adaptive strategy anywhere along the homotopy zero curve.

The initial k values for the problems are chosen to compare GMRES-like methods when: (1) GMRES(k) does not exhibit near stagnation behavior at all ($n = 468$ circuit problem, $n = 119$ shell problem), (2) near stagnation causes an increase in k for *all* the matrices along the curve (circuit problem $n = 125$), and (3) near stagnation is detected for *some* matrices along the curve. In the first case, adaptive and nonadaptive GMRES(k) perform the same. In the second and third cases, GS with k the same as the initial k in AGMRES(k) reaches no final solution, which has a disastrous effect on the outer (client) algorithm calling GS. An adaptive strategy is especially useful when the degree of difficulty of linear systems in a sequence varies, and it is hard to predict what value of k is needed for convergence of GMRES(k) on all the linear systems. Even for a small shell problem ($n = 55$), k can vary between 8 and 20, and the number of iterations can vary between a low of 1 and a high of 1239. The AGH:4 adaptive strategy on one of the matrices from the "vref" circuit is shown in Figure 11, where a circle indicates an increase in the subspace dimension ($k = 8$

Table 1
Average, maximum, and minimum number of iterative solver iterations per linear system along homotopy zero curve for circuit design problems (ILU preconditioner).

n	AGH:2	AGH:4	AGH:6	AGS:2	AGS:4	AGS:6
31	23,112,1 ($2 \to 6$)	19,108,1 ($2 \to 6$)	19,108,1 ($2 \to 6$)	24,112,1 ($2 \to 6$)	19,108,1 ($2 \to 6$)	19,108,1 ($2 \to 6$)
59	52,231,14 ($4 \to 6$)	48,231,14 ($4 \to 8$)	49,231,14 ($4 \to 10$)	53,231,14 ($4 \to 6$)	53,231,14 ($4 \to 8$)	*
67	587,1150,344 ($5 \to 15$)	557,1050,339 ($5 \to 15$)	471,734,286 ($5 \to 15$)	576,1578,291 ($5 \to 15$)	613,1520,330 ($5 \to 15$)	531,1295,312 ($5 \to 15$)
125	143,193,84 ($4 \to 6$)	121,169,84 ($4 \to 8$)	102,131,82 ($4 \to 10$)	146,230,84 ($4 \to 6$)	122,156,84 ($4 \to 8$)	110,140,73 ($4 \to 10$)
468	623,1842,252 ($15 \to 15$)	623,1842,252 ($15 \to 15$)	623,1842,252 ($15 \to 15$)	603,1740,252 ($15 \to 15$)	603,1740,252 ($15 \to 15$)	603,1740,252 ($15 \to 15$)
1854	2719,4340,1173 ($35 \to 47$)	2768,4234,956 ($35 \to 47$)	2419,3953,874 ($35 \to 47$)	2710,4165,1347 ($35 \to 47$)	2725,3990,1096 ($35 \to 47$)	2523,3947,1060 ($35 \to 47$)

n	GS	FGS	Q
31	(2) * (6)12,48,1	(2) * (3)8,196,1	*
59	(4) * (10)9,10,7	*	*
67	(5) * (15)288,359,217	(3) * (8)20,107,9	*
125	(4) * (10)83,110,49	(4) * (6)62,142,9	*
468	(15)603,1740,252	*	*
1854	(35) * (47)611,838,539	*	*

Table 2
Average, maximum, and minimum number of iterative solver iterations per linear system along homotopy curve for thin shell problem (Gill-Murray preconditioner).

n	AGH:2	AGH:4	AGH:6	AGS:2	AGS:4	AGS:6	GS	FGS	Q
55	80,1239,1 ($8 \to 20$)	57,888,1 ($8 \to 20$)	44,748,1 ($8 \to 24$)	*	*	*	*	*	*
119	6,134,1 ($5 \to 5$)	6,134,1 ($5 \to 5$)	6,134,1 ($5 \to 5$)	6,140,1 ($5 \to 5$)	6,140,1 ($5 \to 5$)	6,140,1 ($5 \to 5$)	6,140,1 (5)	*	*
1239	3,14,1 ($12 \to 20$)	3,14,1 ($12 \to 20$)	3,14,1 ($12 \to 20$)	*	*	*	*	*	*

initially). The adaptive strategy is invoked for the first time rather early—to jump off the near-stagnation "plateau" after the 24th iteration. The subspace dimension is increased once more (after 172 iterations), which suggests that AGH:4 has enough vectors to solve the

TABLE 3

Average, maximum, and minimum number of iterative solver iterations per linear system along homotopy zero curve for lamella dome problem (Gill-Murray preconditioner).

n	AGH:2	AGH:4	AGH:6	AGS:2	AGS:4	AGS:6	GS	FGS	Q
21	22,452,1 $(4 \to 10)$	19,536,1 $(4 \to 12)$	16,528,1 $(4 \to 10)$	23,508,1 $(4 \to 10)$	18,387,1 $(4 \to 12)$	*	*	*	*
69	38,1374,1 $(8 \to 18)$	34,1347,1 $(8 \to 18)$	35,1576,1 $(8 \to 18)$	40,1331,1 $(8 \to 18)$	39,1442,1 $(8 \to 18)$	*	*	(4) * (9)3,75,1	*

TABLE 4

Iterative solver execution time in seconds for circuit problems (ILU preconditioner).

n	AGH:2	AGH:4	AGH:6	AGS:2	AGS:4	AGS:6	GS	FGS
31	0.66	0.55	0.55	0.40	0.36	0.36	0.18	1.53
59	0.50	0.45	0.46	0.33	0.31	*	0.08	*
67	3.70	3.53	2.96	1.97	2.09	1.82	0.55	11.04
125	1.51	1.34	1.18	0.98	0.81	0.74	0.56	14.56
468	11.45	11.46	11.45	5.39	5.39	5.38	5.32	*
1854	383.60	386.29	338.76	133.17	133.04	123.31	32.98	*

TABLE 5

Iterative solver execution time in seconds for thin shell problem (Gill-Murray preconditioner).

n	AGH:2	AGH:4	AGH:6	AGS:2	AGS:4	AGS:6	GS
55	25.82	19.68	14.53	*	*	*	*
119	2.70	2.71	2.69	2.43	2.42	2.41	2.44
1239	111.40	107.12	100.90	*	*	*	*

TABLE 6

Iterative solver execution time in seconds for lamella dome problem (Gill-Murray preconditioner).

n	AGH:2	AGH:4	AGH:6	AGS:2	AGS:4	AGS:6	FGS
21	2.24	1.89	1.66	1.74	1.39	*	*
69	29.45	25.93	27.27	24.26	23.64	*	38.86

system within the iteration limit. Note that a smaller iteration limit would cause AGH:4 to increase the subspace dimension quicker.

For every circuit problem, GS with k the maximum restart value produced by the adaptive strategy converges faster than AGS or AGH. However, neither AGS nor GS always converges for the shell problems (Table 2) with $n = 55$ and $n = 1239$, due to the failure of the modified Gram-Schmidt process to accurately orthogonalize the Krylov subspace basis.

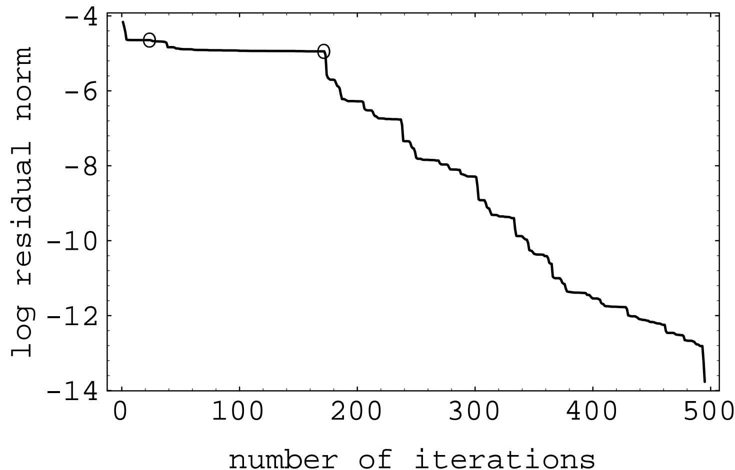

FIG. 11. *Effect of adaptive strategy on residual norm.*

When the vectors encountered in the orthogonalization process become increasingly nearly linearly dependent, the orthogonalization error of the modified Gram-Schmidt process grows with the number of vectors orthogonalized. For example, AGS:6 fails on all lamella dome problems whereas AGS:2 and AGS:4 succeed (Table 3). The same situation is encountered for the circuit problem with $n = 59$. All these failures just mentioned ensue from the estimated condition number of the GMRES least squares problem indicating numerical singularity. AGH successfully passes the condition number test on all the problems, producing a more accurate orthogonal basis for the Krylov subspace.

Table 7 shows the AGS and AGH convergence traces for a typical linear system solution along the equilibrium curve for a thin shell problem. The (preconditioned) matrix A of this particular linear system has condition number $1.15 \cdot 10^{10}$, inertia $(4, 0, 52)$, and smallest eigenvalue -8.014. Table 7 reports the recursively maintained (computed) GMRES residual, the exact (calculated in 128-bit arithmetic) residual, and the estimated condition number of the GMRES least-squares problem at the end of each iteration for AGS and AGH. The first iteration in each restart cycle is shown in bold face type, and the adaptive strategy was not invoked on this system. There are several observations from Table 7: (1) the performances of AGS and AGH are almost indistinguishable until the exact residual norm becomes smaller than $\text{cond}(A)\mathbf{u} \approx 2.5 \cdot 10^{-6}$, which is consistent with theoretical results in

[28]. (2) However, much higher accuracy is consistently achievable by AGH on a wide range of qualitatively different problems (the linear systems along the homotopy zero curves for the problems in Sections 4.2.1–4.2.3 are fundamentally different), as proven by the results in Tables 1-6. The right hand sides, coming from nonlinear normal flow iteration corrections, are not "special" in any way whatsoever. (3) When the computed residual norm for AGS reaches $\mathcal{O}(\mathrm{cond}(A)\ \mathbf{u})$, there is a pronounced oscillation of the true residual norm, and stagnation of the computed residual norm. At this point, AGH is behaving differently, producing a computed residual norm small enough to terminate the current restart cycle before the GMRES least-squares problem becomes ill-conditioned enough to trigger the condition number test. This behaviour of AGH relative to AGS, shown in Table 7, occurred consistently on all the test problems where AGS failed. (4) With the condition number test turned off, AGH always reached a higher accuracy solution, but AGS frequently completely stagnated and did not. For this particular linear system, both AGS and AGH converge to a higher accuracy solution, but AGS does so at a cost of more iterations, which is caused by stagnation of its computed residual norm. For larger subspaces, such a stagnation severely impedes the convergence of AGS.

AGH never fails, but at a cost of more CPU time. For example, good candidates for a CPU time comparison (Tables 1, 4) are AGH:4 and AGS:4 for the $n = 31$ circuit problem and AGH:2 and AGS:2 for the $n = 59$ circuit problem, since they exhibit similar adaptive behavior in terms of iteration counts. In both cases, AGS outperforms AGH only by a factor of 1.5. This factor varies slightly depending on the performance of the adaptive strategy and the orthogonalization accuracy for a particular problem. If the maximum number of AGS iterations is large relative to n (e.g., dome problem with $n = 69$), then AGH is only a little slower than AGS (Table 6), but for large n ($n = 1854$ circuit problem), AGH is slower than AGS by a factor of 2.8 and also takes more iterations (Tables 1, 4). When a high accuracy solution is required, it is hard to predict which orthogonalization procedure will lead to the smallest number of iterations. For example, from Table 1, AGH produces a lower maximum number of iterations than AGS for some cases but not all cases.

TABLE 7

Convergence trace for a typical system from thin shell problem ($n = 55$) comparing AGS with AGH.

	AGS:2(25)			AGH:2(25)		
j	computed residual	exact residual	cond(AV_j)	computed residual	exact residual	cond(AV_j)
1	**9.88233703+1**	**9.88233703+1**	**1.00000000+0**	**9.88+1**	**9.88+1**	**1.00+0**
2	8.41201033+0	8.41201033+0	2.71517580+3	8.41+0	8.41+0	2.72+3
3	5.30062772+0	5.30062772+0	3.60300441+4	5.30+0	5.30+0	3.60+4
4	6.00276476−1	6.00276476−1	2.36085135+5	6.00−1	6.00−1	2.36+5
5	2.09306671−1	2.09306671−1	8.30979753+6	2.09−1	2.09−1	8.31+6
6	1.66690256−1	1.66690256−1	8.53997763+6	1.67−1	1.67−1	8.54+6
7	9.14942978−2	9.14942978−2	5.08538248+7	9.15−2	9.15−2	5.09+7
8	8.73673945−2	8.73673947−2	1.38782061+9	8.73−2	8.73−2	1.39+9
9	8.71466153−2	8.71466144−2	1.41042577+9	8.73−2	8.73−2	1.40+9
10	3.93514895−3	3.93514905−3	3.81499136+9	3.94−3	3.94−3	3.82+9
11	7.93411548−8	3.28110407−7	1.40706496+10	1.50−7	1.82−6	1.41+10
12	3.28744409−9	2.94899177−7	1.40738642+10	4.98−18	2.48−7	1.41+10
13	3.28744385−9	2.94772120−7	3.21019087+15	**2.03−7**	**2.03−7**	**1.00+0**
14	2.84586612−10	3.62878483−7	8.32790167+18	1.85−7	1.85−7	6.66+4
15	2.84586612−10	3.62900605−7	8.32790210+18	1.78−7	1.78−7	3.25+5
16	3.76425255−11	1.67507226−5	9.84930800+21	3.24−8	3.24−8	4.29+5
17	2.73041662−11	1.67328247−5	8.91283895+21	7.02−9	7.02−9	1.30+7
18	2.73041608−11	1.67581289−5	8.91305834+21	6.93−9	6.93−9	3.09+7
19	6.82351657−13	6.90349476−7	9.53171000+22	4.46−9	4.46−9	1.50+8
20	6.82351657−13	6.92385648−7	9.53179594+22	4.46−9	4.46−9	6.07+8
21	6.72452102−13	6.54577897−7	9.83975191+22	4.38−9	4.38−9	9.04+8
22	6.70811951−13	6.48248321−7	9.88983042+22	8.20−10	8.20−10	5.39+9
23	6.70811951−13	6.50677103−7	9.89048681+22	2.24−14	7.38−13	2.58+10
24	6.58769415−13	4.69784705−7	1.25798140+23	**7.37−13**	**1.01−12**	**1.00+0**
25	6.58769415−13	4.73672603−7	1.25802892+23	3.61−13	3.30−13	1.17+4
26	**4.70128524−7**	**4.70128328−7**	**1.00000000+0**	3.07−13	6.31−13	1.01+5
27	1.87736873−7	1.87736870−7	9.59415345+3	3.28−14	7.34−13	2.56+5
28	1.11797857−7	1.11797808−7	6.15064900+4	**7.33−13**	**6.52−13**	**1.00+0**
29	1.62744063−8	1.62744158−8	3.12694899+5	2.19−13	3.52−13	6.67+3
30	5.34499291−9	5.34499087−9	9.40826259+6		converged	
31	5.29674513−9	5.29674296−9	5.47342000+7			
32	3.54672002−9	3.54672030−9	2.69515192+8			
33	1.02373805−9	1.02370991−9	4.03740084+9			
34	9.69052509−10	9.69016629−10	1.23940329+10			
35	5.77211087−10	5.77211328−10	1.30139651+10			
36	8.28643039−14	6.05887230−13	2.72026861+10			
37	**6.05884655−13**	**6.04524129−13**	**1.00000000+0**			
38	4.30512161−14	2.45523317−13	2.43627507+3			

The number of restart cycles in AGMRES(k) depends on the increment value m.

Tables 1-3 indicate that even a small increment in the restart value leads to convergence. However, if the increment value is too small, often extra restart cycles are executed, which increases the average number of iterations in all cases, comparing $m = 2$ and $m = 6$. As m grows, the execution time of smaller test problems decreases since the cost of an extra iteration in a restart cycle is essentially the same as the cost of earlier ones and the later iterations sometimes contribute the most towards convergence as noted in [66]. For large problems ($n = 1854$), the cost of the later added iterations becomes significant and increases CPU time (compare AGH:2 and AGH:4 for $n = 1854$ in Table 4). Thus, some moderate value for m seems best for the proposed adaptive strategy.

It is clear from the data presented that AGMRES(k) outperforms both flexible GMRES($\hat{k}$) and QMR. Since restarted flexible GMRES($\hat{k}$) requires twice as much memory as restarted AGMRES(k), the restart value for FGS is taken as $\hat{k} = k/2$. Whenever FGS converges, it requires more work per iteration than AGH because a new preconditioner is computed at each FGS iteration. The QMR algorithm, subroutine DUQMR from QMRPACK [23], fails to find a solution with the required accuracy for any of the problems, but can solve some linear systems along the homotopy zero curve if the error tolerance is relaxed.

4.4. Conclusions.

For applications requiring high accuracy linear system solutions, the adaptive GMRES(k) algorithm proposed here, which is based on Householder transformations and monitoring the GMRES convergence rate, outperforms several standard GMRES variants and QMR. The superiority is not in CPU time or memory requirements, but in robustness as a linear system solution "server" for a much larger nonlinear analysis "client" computation. The extra cost of Householder transformations is well justified by the improved reliability, and the adaptive strategy is well suited to the need to solve a sequence of linear systems of widely varying difficulty. In the context of large scale, multidisciplinary, nonlinear analysis it is hard to imagine not wanting the various components to be adaptive.

The high accuracy requirement causes types of failures in GMRES not usually seen in moderate accuracy uses of GMRES, and monitoring condition number estimates and the onset of stagnation within GMRES becomes crucial. Furthermore, "sanity" checks (e.g., residual norm should be nonincreasing) within a GMRES algorithm must be modified for high accuracy. For instance, the limit on the number of iterations must be increased, and depending on other factors, an increasing residual norm may or may not dictate an abort (such details are in the pseudocode in Section 4.1). Finally, the three test problems are rather different, so the details of AGMRES(k) have not been tuned to one particular class of problems.

5. PARALLEL IMPLEMENTATION OF ADAPTIVE GMRES(k).

5.1. Rationale for Developing Parallel Implementation.

Growing computer capacity is a major reason for homotopy algorithms becoming an affordable alternative to other methods. The benefits of parallel architectures for homotopy methods have been shown in [1]. Despite the serial nature of curve tracking, the burden of computation in homotopy methods is in the numerical linear algebra and user-defined function evaluations [13], which can be made parallel. For large scale problems, the cost of this serial work may be negligible in comparison with linear system solving (and user-written function evaluation). As concluded in [2], a high degree of parallelism in linear system solutions contributes much to the overall parallel performance of homotopy methods. In evaluating the performance of a complex algorithm, parallel efficiency is a large concern. Efficiency is usually defined as the ratio of speedup to the number of processors.

In the context of homotopy methods, an efficient parallel implementation of a linear system solver has to satisfy the requirements set for a sequential linear solver — robustness and high accuracy in the solution. Under these requirements, the proposed adaptive GMRES(k) algorithm with Householder reflection orthogonalization exhibits better performance than a number of other iterative methods, as shown in the previous chapter. Therefore, a parallel version of adaptive GMRES(k) is chosen as a parallel linear solver for homotopy methods. In this version, such important properties of adaptive GMRES(k) as the adaptive strategy and the incremental condition number estimation are implemented similarly to their sequential counterparts. However, the selection of a distributed sparse matrix-vector multiplication and a distributed orthogonalization with Householder reflections is nontrivial and allows several options. In the rest of this chapter, the design choices are explained and two parallel implementations of GMRES(k) using Householder reflections are studied.

5.2. Partitioning of a General Sparse Matrix.

A parallel procedure for multiplying a sparse matrix by a dense vector depends on how the problem is split into subproblems and assigned to processors. Often, the knowledge of a problem leads to a natural way of splitting into subproblems. For example, in the finite element method, it may be advantageous to confine a whole element within a subdomain together with all information related to that element. If a sparse matrix has a special nonzero pattern, then a particular type of partitioning (and thus a particular version of matrix-vector multiplication) yields the most efficient implementation. Consider, for example, a block-tridiagonal matrix arising from solving partial differential equations with the finite difference method. When the grid points produced by this method are ordered and partitioned into subdomains in the block-chessboard fashion (Figure 12), the corresponding parallel matrix-vector multiplication is more efficient than implementations for other partitionings of the grid points [43].

S_1		S_2	
1	2	3	4
5	6	7	8
S_3		**S_4**	
9	10	11	12
13	14	15	16

FIG. 12. *Block-chessboard partitioning of 4 × 4 grid into 4 subdomains.*

However, in the context of homotopy methods, which are used for a wide range of different problems, an "all-time winner" partitioning scheme for sparse matrices cannot be predetermined. In this case, one choice is a straightforward partitioning (call it *block-striped*) that defines a subdomain as a block of consecutive rows (or columns). A better alternative, however, is to use graph theory results to obtain matrix row partitioning that achieves good load balancing and a small communication to computation ratio.

Let A be an $n \times n$ matrix with a symmetric nonzero structure ($a_{ij} \neq 0 \iff a_{ji} \neq 0$). Let $G(A)$ be an undirected graph with n vertices such that there is an edge between the ith and jth vertices if and only if $a_{ij} \neq 0$. Each vertex of $G(A)$ corresponds to a row of A. The vertices of $G(A)$ are partitioned into p disjoint (nonempty) subsets (subdomains) $V_1, \dots, V_p$, such that each subset belongs to a processor. The edges of $G(A)$ describe the connectivity of vertices within each subdomain and between a pair of subdomains. Figure 13 shows a 12×12 matrix A and its graph $G(A)$ partitioned into four subdomains.

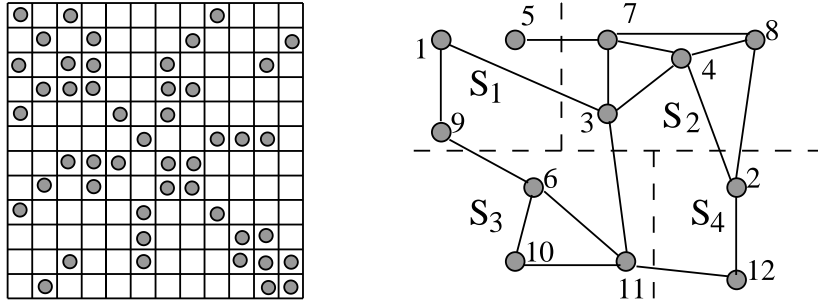

FIG. 13. *Partitioning of 12 × 12 matrix into 4 subdomains.*

Given a partition $\mathcal{V} = \{V_1, \dots, V_p\}$, call an *internal edge* an edge of $G(A)$ that has both vertices incident with it in the same V_i. Conversely, an *external edge* is an edge incident with two vertices from two different subsets. Following the definition in [74], let *contracting* an edge e of $G(A)$ mean replacing both endpoints of e by a single vertex with edges to those vertices that were adjacent to either of the endpoints of e.

Consider a procedure (call it CONTRM) that produces a multigraph corresponding to the communication pattern among processors given a partitioning of a matrix into subdomains.

Step 1. Perform contractions on all internal edges of $G(A)$.

Step 2. Let W_i be the set of vertices left from V_i after step 1. Replace all the vertices in W_i by a single vertex u_i such that the edges incident with it are all the external edges that have one endpoint in V_i,

This procedure results in a multigraph $G'(U, E)$, where $U = \{u_1, \dots u_p\}$ is a set of vertices and $E = \{(l_{ij}, u_i, u_j) \mid l_{ij} = 1, \dots, m_{ij},\ i \neq j, \text{ and } i, j = 1, \dots, p\}$ is a set of labeled

edges, where m_{ij} is the number of edges between u_i and u_j. Figure 14 shows a multigraph $G'(U, E)$ obtained from the graph in Figure 13 by the procedure CONTRM.

When minimizing the communication to computation ratio, a criterion for measuring the quality of the partition $\mathcal{V}$ can be taken as the sum (over all subdomains) of costs of computation and communication (see, e.g., [51]). The communication mechanism together with the interconnection network and the type of messages affect the communication cost greatly. Communication start-up requires much more time than transmission of data on homogeneous distributed-memory architectures. For example, on the Intel Paragon, the start-up time becomes negligible compared with the transmission time only when the size of a message nears a hundred megabytes. Furthermore, in sparse numerical linear algebra, say in sparse matrix-vector multiplication, an array of values of the same type (floating point numbers) is exchanged by a single message between subdomains. Thus, under some reasonable restrictions on the number of data values sent between any two subdomains, consider the cost function $f : U \times U \to \{0, 1\}$ such that $f(u_i, u_j) = 1$ if *at least one* edge connects u_i and u_j, and $f(u_i, u_j) = 0$ otherwise. Then an edge in E incident with both u_i and u_j ($i \neq j$ and $i, j = 1, \ldots, p$) has a cost $1/m_{ij}$, where m_{ij} is the number of edges incident with u_j and u_j. For the multigraph shown in Figure 14, the total cost is equal to 5.

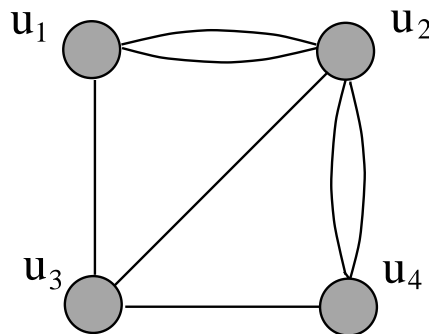

FIG. 14. *Contraction of $G(A)$ to a multigraph using the procedure CONTRM.*

Since establishing communication overwhelms the other phases of communication and since a single message can contain all the data to be exchanged between adjacent vertices in $G'(U, E)$, minimizing the sum of edge costs would minimize of the communication cost for the partition $\mathcal{V}$. Note that this cost function may not be valid for other application domains, when each edge in E corresponds to some unique piece of information, and adopting some other cost function may be appropriate (see, e.g, [50]). A number of graph-theoretical heuristics exist to produce partitioning (call it *vertex-based*) into subdomains with the load balanced and a minimum number of external edges (minimum number of edges that connect different V_i in $\mathcal{V}$). For parallel implementations considered here, a recently developed graph partitioning algorithm, Multilevel Recursive Bisection [41], is used. This algorithm follows a multilevel graph partitioning scheme that consists of three phases: reducing the size of a graph (coarsening), partitioning a smaller graph, restoring the graph to its original size (uncoarsening) [31], [40]. In general, this algorithm produces a partition in which all external edges are treated uniformly; that is, all edges in $G'(U, E)$ have the same cost (say, 1). Thus, the total communication cost of this partition is the total number of edges. For the multigraph shown in Figure 14 this cost equals 7, which is clearly not minimum, for the type of communication considered here.

However, for very large problems, when communication start-up time is small compared with the data transmission time, minimizing the number of external edges also achieves a minimum of the communication cost. Since the graph partitioning algorithm is designed to balance the load among subdomains, the communication to computation ratio can be smaller compared with a partitioning that has less communication cost but also a less balanced load.

Subdomain-to-processor assignment is generally architecture-dependent. However, an arbitrary one-to-one mapping is acceptable for the parallel computers that emulate a fully connected network by minimizing the difference in performance between various mappings. Since the developed parallel implementation of adaptive GMRES(k) targets this very class of parallel architectures, such as the IBM SP2 and Intel Paragon, the task of subdomain-to-processor mapping is regarded here to be coupled with the partitioning task. Thus,

without loss of generality, assume that the subset of vertices V_d of the graph $G(A)$ (where $V_1 \cup \ldots \cup V_p$ is a partition of the vertices) is assigned to the processor d. Then, assume also that the processor d holds those vector components the indices of which match the matrix rows mapped to this processor. In other words, matrix row partitioning is defined by vector component partitioning.

5.3. Distributed Matrix-Vector Multiplication.

Since matrix rows are partitioned according to some partition of vector components and the matrix is structurally symmetric (corresponding to an undirected graph), a distributed matrix-vector multiplication can be efficiently implemented, as shown in [53], by introducing a preprocessing stage performed concurrently on each processor. The purpose of this stage is twofold. Firstly, preprocessing creates the lists of the local vector components to be communicated repeatedly at each matrix-vector multiplication. Secondly, preprocessing reorders vector components and nonzeros of a local submatrix to facilitate overlapping of communication and computation while performing global matrix-vector multiplication. As a result of the preprocessing stage, local vector components are decoupled into *internal* vector components that do not participate in communication, and *local interface* vector components that are requested by neighboring processors, such that internal components are ordered before local interface components. The indices of *external interface* vector components that are received from neighboring processors are also identified. The reordered local submatrix A_d (Figure 15) is represented by two matrices: the square matrix B_d of dimensions $n_d \times n_d$ corresponding to the number of vector components in the processor d, and the rectangular $d_1 \times d_2$ matrix X_d, where d_1 is the number of local interface components and d_2 is the number of external interface components. To perform a matrix-vector multiplication, the submatrix B_d is first multiplied by the local vector components $x_d(1 : n_d)$, then the X_d part of A_d is multiplied by the received external interface components $x'_d(1 : d_2)$ and the results of these two multiplications are added. This representation by B_d and X_d is valid because of the assumption of structural symmetry of A. For each processor, the following pseudocode shows the distributed matrix-vector multiplication together with the exchange of interface components.

send $x_d(n_d - d_1 + 1 : n_d)$ to neighbors;

$y_d := B_d x_d$;

receive $x'_d(1 : d_2)$ from neighbors;

$y'_d := X_d x'_d$;

$y_d := y_d + (0, \ldots, 0, y'_d)^t$;

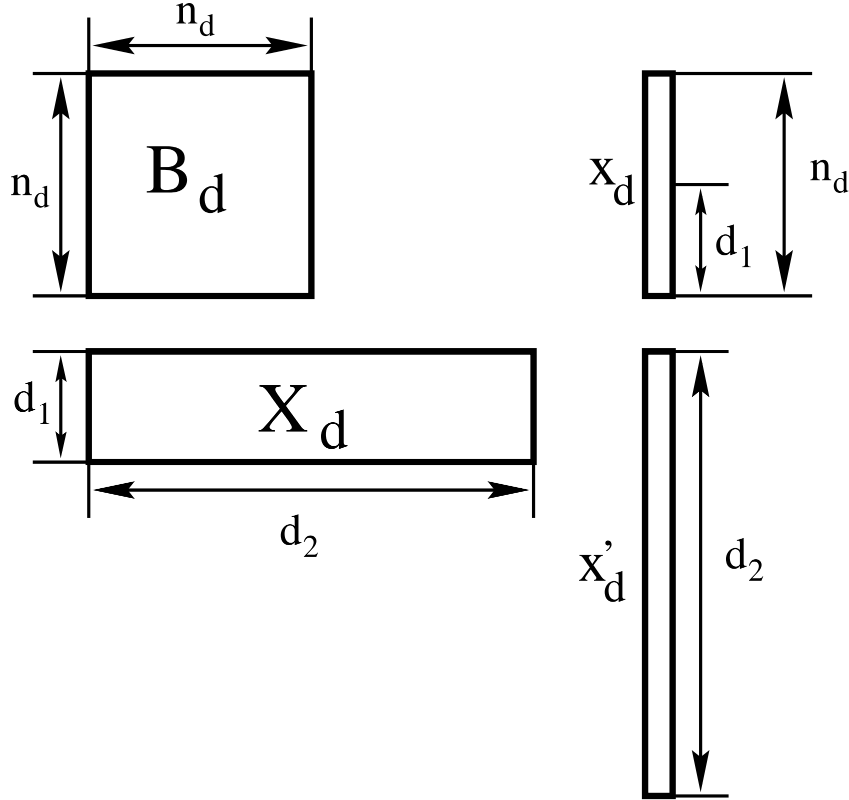

FIG. 15. *Local matrix A_d and vector components after the preprocessing stage.*

5.4. Approaches to Distributed Orthogonalization.

In parallel environments, the choice of the orthogonalization process for the Krylov subspace basis vectors depends not only on the accuracy of the process but also on the amount and type of global communication it incurs. For some orthogonalization procedures, only one of the two requirements is satisfied. For example, in serial implementations of the GMRES method, the modified version of the Gram-Schmidt process is often used as being sufficiently accurate for a number of problems. However, in parallel GMRES implementations,

other orthogonalization procedures are preferable since the modified Gram-Schmidt process exhibits a large communication overhead. Better efficiency is achieved with the classical Gram-Schmidt, which is unstable. A compromise between the accuracy and communication overhead resulted in the development of parallel GMRES(k) variations with a non-Arnoldi basis that undergoes orthogonalization only at the end of a restart cycle (see, e.g., [3], [22]). At this point, an equivalent to the Arnoldi basis (the matrix V_{m+1}) is recovered as the matrix Q in the QR factorization of the non-Arnoldi basis. The QR factorization is performed in parallel by a point or block version of Householder reflections [58] that have a high degree of parallelism and avoid all-to-all communications. Under the requirement of high accuracy, the errors associated with performing GMRES(k) iterations in a nonorthogonal basis are not acceptable. However, an efficient parallel implementation of Householder reflections can be employed successfully in a parallel version of the GMRES(k) algorithm, in which Householder reflections are used to compute the Arnoldi basis.

5.5. Householder Reflections for a Special Case of Partitioning.

The degree of parallelism equal to the number of processors in parallel Householder reflections generation is achieved by letting each processor create its local portion of the Householder reflections vector independently. After that, local Householder reflections vectors are assembled into a global Householder reflections vector. To construct a global Householder reflections for a given vector x, the (first) single nonzero local component of x is modified: it is changed to zero for all the processors but the first one in accordance with some sequential ranking of processors. Finally, the first processor updates its nonzero component based on the information obtained from the other processors. In the outlined modification, to preserve the orthogonality of Householder reflections, the orthogonal procedure, such as Givens rotation, updates the component of x received from a neighbor and zeros out the local component. Then the updated value is transmitted to the other neighbor. Thus, to zero out all $p-1$ (where p is the number of processors) components of x and update the value located on the first processor, the ring topology of processors is the most suitable for the communication between processors. However, as it will be shown, different approaches

if $(proc = 1)$ **then** $s := j$ **else** $s := 0$ **end if**

determine H_{s+1} such that $H_{s+1} v_d \equiv w_d$ has zeros

after the $(s+1)$st component;

if $(proc = 1)$ **then**

send $w_d(s+1)$ to $right$;

else

receive w from $left$;

determine G_1 such that $w_d(1) = 0$; update w;

if $(proc \neq p)$ send w to $right$;

end if

FIG. 16. *Parallel Householder reflection generation.*

in organizing communications on a ring affect partitioning of a problem. The approach taken in [58] assumes a *fixed* ring of processors, in which the communication always starts from a fixed (first) processor. The fixed ring topology restricts problem partitioning to the block-striped case, in which the first component of x belongs to that fixed processor.

Pseudocode for the algorithm generating Householder reflections at the jth GMRES(k) iteration on the processor *proc* is given in Figure 16. It is followed in Figure 17 by the pseudocode for the application of Householder reflections for the same iteration and processor. Before applying a Householder reflections, a sequence of Givens rotations has to be applied. Thus, Householder reflections application can be viewed as the Householder reflections generation process reversed with respect to the order of using Givens rotations and Householder reflections. Denote by HG the algorithm in Figure 16 and Figure 17 for the block-striped partitioning case.

In Figures 16 and 17, H and G denote the Householder transformation matrix and the Givens rotation matrix (as given in [26]), respectively; v_d denotes a portion of the Krylov subspace vector $A^j r_0$ located on a processor; p, $left$, $right$ are the processors with the highest rank, with the $proc - 1$ rank, and the $proc + 1$ rank, respectively. It is also assumed that the first processor has the jth row of the input matrix.

```
for i := j downto 1 step -1
    if (proc = 1) then
        send w_d(i) to p;
        receive w_d(i) from right;
        sc := i;
    else
        receive w from right;
        apply G_1 to (w, w_d(1));
        send w to left;
        sc := 1;
    end if
    apply H_sc to w_d(sc :);
end for
```

FIG. 17. *Parallel Householder reflection application.*

5.6. Householder Reflections for an Arbitrary Partitioning.

To use the algorithms in Figures 16 and 17, the redistribution of a vector requires $\mathcal{O}(p^2)$ communications at each GMRES(k) iteration, which is highly impractical and reduces the efficiency gained by the distributed matrix-vector product. Thus, it is beneficial to develop an extension of the algorithms in Figures 16 and 17 which accepts an arbitrary row distribution among processors. The key idea of this extension is to allow a *flexible* ring in performing Givens rotations. The flexible ring is the ring of processors in which the communication may start from an arbitrary processor that (in a given vertex-based partitioning) holds the first component of the vector x. Denote by MHG the algorithm in Figure 18 and Figure 19 for the vertex-based partitioning case. In practice, each processor in the flexible ring determines the current *ring_start* by consulting the global mapping array *mpart*, each entry of which is a processor number indexed by a matrix row number (where the row numbering refers to the original matrix before partitioning) located on that processor.

if $(j = 1)$ **then**

$s := 1;$

else

if (*proc* has $(j-1)$st row) **then** $s := s + 1;$

end if

determine H_s such that $H_s v_d \equiv w_d$ has zeros

after the sth component;

$ring_start := mpart(j);$

if $(proc = ring_start\ )$ **then**

send $w_d(s+1)$ to $right$;

else

receive w from $left$;

determine G_s such that $w_d(s) = 0$; update w;

if $(proc \neq ring_start)$ **then** send w to $right$;

end if

FIG. 18. *Modified parallel Householder reflection generation.*

Usually, the subspace dimension is much smaller than the matrix dimension and the vertex-based partitioning produces a balanced workload by assigning an almost equal number of rows to each processor. Thus, the case when the index s within v_d becomes equal to the size of a local partition (size of v_d) occurs rarely for large matrices, unless the number of processors is very large.

5.7. Experimental Results.

To compare the Householder reflections restricted to the block-striped partitioning with the proposed extension for the vertex-based partitioning, GMRES was instrumented to collect the timing information relevant only to the parallel Householder reflections generation and application. The time spent in HG and MHG for generating and applying a global Householder reflections vector is independent of the type of partitioning and thus presents

```
sc := s;
for i := j downto 1 step -1
    if (proc has ith row) then
        if (sc ≠ 1) then sc := sc − 1;
        send w_d(sc) to left;
        receive w_d(sc) from right;
    else
        receive w from right;
        if (G_sc exists) then
            apply G_sc to (w, w_d(sc));
        end if
        send w to left;
    end if
    apply H_sc to w_d(sc :);
end for
```

FIG. 19. *Modified Householder reflection application.*

a good measure of the relative computational complexity of these two implementations. As a test problem, the matrix BCSSTK13 (2003×2003) was taken from the Harwell-Boeing collection [21]. This matrix is the stiffness matrix for a fluid flow eigenvalue problem from a dynamic analysis in structural engineering. BCSSTK13 has 83883 nonzeros and a symmetric sparsity pattern. MHG and HG generated the first Householder vector in 0.26s and 0.25s, respectively, whereas MHG was slightly faster (0.23s and 0.25s, respectively) than HG in applying a Householder reflections. (The times quoted here are the averaged parallel times over 3 experiments performed on 8 processors of an Intel Paragon.) Thus, the proposed MHG implementation exhibits performance comparable to HG and outperforms HG in some instances.

For the given test problem, two restart cycles of GMRES(15) were executed to determine the dependence of the overall parallel time on the type of partitioning (and thus on HG

TABLE 8

Total and matrix-vector multiplication parallel times for GMRES(15) with MHG and HG orthogonalizations.

		4	8	16	32	64	100
MHG	T_p	1.52	2.21	3.23	5.30	8.84	12.49
	T^{MV}	0.36	0.29	0.34	0.42	0.59	0.76
	np	3	7	12	17	24	25
	$Comm$	6	22	74	177	457	808
HG	T_p	1.66	1.91	3.37	5.25	8.85	12.62
	T^{MV}	0.34	0.23	0.29	0.42	0.76	1.09
	np	3	6	13	13	22	30
	$Comm$	6	17	55	135	388	699

and MHG) with an increase in the number of processors. Table 8 shows the averaged results over three experiments for 4, 8, 16, 32, 64, and 100 processors, where T_p denotes the overall parallel time for GMRES(15), T^{MV} denotes the parallel time for matrix-vector multiplication, np is the maximum number of neighboring processors, and $Comm$ is the total number of communication channels among processors. For the times observed in three experiments, the largest standard deviation was 0.02. Comparison of the results in the last two columns of Table 8 shows that with increasing numbers of processors the performance of GMRES with the MHG orthogonalization deteriorates slower than that of GMRES with the HG orthogonalization. On larger numbers of processors (cf. the last three columns), the relative difference in the number of communication channels for GMRES with MHG and HG decreases. On 100 processors, GMRES with MHG organizes communications among processors more uniformly by having a smaller maximum number of neighbors. Thus, with a rapid growth in the number of neighbors HG performs worse than MHG for large numbers of processors.

5.8. Conclusions.

The developed parallel implementation of adaptive GMRES(k) takes advantage of an efficient distributed matrix-vector multiplication and the Householder reflections orthogonalization that avoids all-to-all communications and has a high degree of parallelism. Generating and applying Householder reflections requires only pairwise communications on a ring of processors. The modification proposed here extends the parallel Householder

reflections orthogonalization from accepting a block-striped partitioning of rows to a general vertex-based partitioning. As a result, matrix-vector multiplication and the Householder reflections orthogonalization operate on the data partitioned in the same way. Thus, no parallel overhead for data redistribution is incurred. For a large number of processors, using the extended parallel implementation of Householder reflections preserves the advantage gained from applying graph-theoretical heuristics to partition a problem into subdomains.

6. SCALABILITY ANALYSIS OF GMRES(k).

Scalability of an algorithm-architecture combination is an important concept of parallel performance analysis that provides insights into the behavior of an algorithm in solving large scale problems on large numbers of processors This chapter studies how the efficiency of two parallel GMRES(k) implementations can be maintained constant with an increase in the number of processors.

Organizational details of this chapter are as follows. Section 6.1 presents the terminology for the scalability analysis. In Section 6.2, an operation count for the sequential algorithm is calculated for comparison with the amount of work conducted by a parallel algorithm. Section 6.3 describes a test problem and its properties, such as size, scaling factor, and nonzero structure of the coefficient matrix, which influence the scaled behavior of a parallel GMRES(k) implementation. For three parallel architectures, characterized in Section 6.4, and their machine-specific parameters, estimated in Section 6.6, this behavior is analyzed theoretically under the constraint of a fixed parallel efficiency in Section 6.5. The numerical experiment section (Section 6.7) compares the theoretical behaviors of scaled algorithm-architecture combinations with their behaviors observed experimentally. Section 6.8 contains conclusions.

6.1. Terminology for the Scalability Analysis.

An algorithm-architecture scalability analysis estimates in a single expression characteristics of the algorithm as well as parameters of the architecture on which the algorithm is implemented. Thus, testing an algorithm on a few processors allows one to predict its performance on a larger number of processors. The following terminology is needed to support the proposed scalability analysis. A *parallel system* is a parallel algorithm and machine combination. The *useful work* W is the total number of basic floating point operations required to solve the problem. W depends on the *problem size* $\bar{N}$, which is a vector of problem-specific parameters such as problem dimensions and the number of nonzero entries in a matrix. For numerical linear algebra problems solved by iterative methods, $\bar{N}$ may

also include an indication of problem difficulty, such as the Krylov subspace dimension k used in GMRES(k) or the condition number. In general, choosing $\bar{N}$ requires a detailed investigation of a given problem and of the way scaling of problem dimensions affects the increase in W [59]. The *sequential execution time* T_1 characterizes the time interval needed to solve a given problem on a uniprocessor. If the time of executing an integer operation is negligible compared with the time t_c of performing a floating point operation and if T_1 is spent in useful work only, then

$$T_1 = t_c W.$$

The *parallel execution time* T_p is the time taken by p processors to solve the problem. The *total parallel overhead* V is the sum of all overheads incurred by all the processors during parallel execution of the algorithm. V includes the communication cost, nonessential work, and the cost of idling. For a given parallel system, V is a function of the useful work W and number of processors p:

$$V = pT_p(W, p) - T_1(W).$$

The *efficiency* E is the ratio of the speedup $S(p)$ to p, where the speedup $S(p) = T_1/T_p$. Hence,

$$E = \frac{T_1}{pT_p} = \frac{1}{(1 + V/T_1)}.$$

A parallel system is called *scalable* if $V = \mathcal{O}(W)$ and unscalable otherwise. For scalable systems, it is possible to keep the efficiency fixed and to monitor the rate of increase in W and p with the *isoefficiency function* $f_E(p)$, as proposed by Kumar [27]. The isoefficiency function is defined *implicitly* by the following relation between W and the parallel overhead V:

$$W(\bar{N}) = eV(W(\bar{N}), p, \bar{h}), \tag{1}$$

where $e = \frac{E}{t_c(1-E)}$ is a constant and $\bar{h}$ is a vector of machine-dependent parameters affecting the amount of the parallel overhead. Usually, the communication cost of the total parallel overhead incorporates these parameters, which are defined in accordance with a communication model supported by a given architecture. For example, if the communication

model is a fully connected network, then the time T_{comm} for sending a message between any two processors depends only on the time t_s required to start the communication, the unit (e.g., double precision constant) transmission time t_w, and the number of units L to be transmitted. In other words, $T_{comm} = t_s + t_w L$.

If the expression of V is complicated, it is often sufficient to find the fastest growing term in V. For a given parallel algorithm-architecture combination, this term reflects asymptotically the increase in the components of the problem size $\bar{N}$ as the number of processors increases, because efficiency is guaranteed not to decrease if none of the terms grows faster than the useful work W.

6.2. Operation Count for the Useful Work in GMRES(k).

Let N be the scaled matrix dimension and N_z be the number of nonzeros in the scaled matrix. To obtain an operation count for the useful work in GMRES(k) at the jth iteration, the following parts of the pseudocode for AGMRES(k) can be distinguished:

1. matrix-vector multiplication, which takes N_z operations;
2. Householder reflection generation, which takes $2(N - j + 1)$ operations;
3. Householder reflection application, which requires $4\sum_{i=1}^{j}(N - i + 1)$ operations.

The remaining work is accomplished in $\mathcal{O}(k)$ operations. For a single restart cycle, the useful work is

$$
\begin{aligned}
W &\approx (k+1)N_z + 2\sum_{j=1}^{k+1}(N - j + 1) + N + \sum_{j=1}^{k}\left(4\sum_{i=1}^{j}(N - i + 1) + (N - j + 1)\right), \qquad (2)\\
&\approx (k+1)N_z + N(2k^2 + 5k + 3) + C_k,
\end{aligned}
$$

where C_k is a term depending only on k.

6.3. Description of the Test Problem.

The scalability analysis is performed on a real-world problem representing a commercial circuit design at AT&T Bell Laboratories. The dimension of this problem is $n = 125$ and the number of nonzeros $n_z = 782$. Here scaling the problem K times means assembling K replicas of the $n \times n$ matrix of coefficients in an $N \times N$ matrix, where $N = K \times n$ and

the number of nonzeros $N_z = K \times n_z$, as shown in Figure 20 for $K = 5$. Both the initial solution vector and right-hand side are $e_N^t = (0, \ldots, 0, 1)^t$. A K-block diagonal structure of the matrix was purposely avoided to distinguish between block-striped assignment of rows to processors (used in HG) and vertex-based assignment (used in MHG). However, since the matrix does possess a specific (block) structure, the effect of using graph partitioning algorithms to distribute rows may not be pronounced compared with the block-striped partitioning.

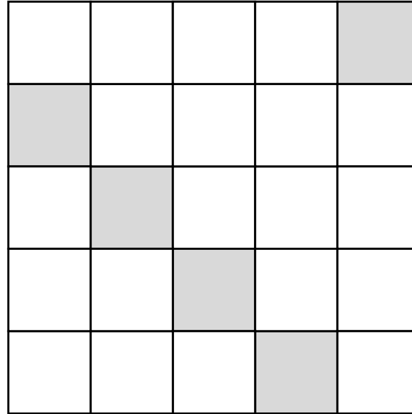

FIG. 20. *Scaling of problem size.*

6.4. Architectures of the IBM SP2, Intel Paragon, and Cray T3E.

Numerical experiments were performed on three parallel computing platforms: an IBM SP2 (at Virginia Tech and NASA Langley Research Center), an Intel Paragon (at Virginia Tech and the Center for Computational Sciences at Oak Ridge National Laboratory), and a Cray T3E (at the National Energy Research Scientific Computing Center). These computers have distributed-memory architectures, each of which is characterized by a large number of processing elements of a particular type and by the main memory coupled with each processing element; thus, data sharing among processors is performed by message passing. The physical interconnection of processors emulates a fully connected network.

The theoretical uniprocessor peak computing rates on all three architectures are quite high: 266 Mflop/s, 75 Mflop/s, and 600 Mflop/s for an IBM SP2, an Intel Paragon, and

Cray T3E, respectively. However, a realistic application achieves only a fraction of the peak performance, depending on such factors as the nature of the application and the level of code optimization used. Given some characteristics of an application (e.g. application domain, precision and type of operations), the actual uniprocessor rate is much more closely reflected by the performance of a benchmark application with similar characteristics. Thus, as a benchmark, the uniprocessor rate achieved in solving a linear system of equations by a LINPACK [20] algorithm (in double precision) will be specified (as the "Towards Peak Performance" benchmark in [19]) in the following description of the IBM SP2, Intel Paragon, and Cray T3E.

Each node of the IBM Scalable POWER2 Parallel System (SP2) is functionally equivalent to a RISC System/6000 390 workstation (67 MHz thin node) or a RISC System/6000 590 workstation (67MHz wide node). The LINPACK benchmark performance is 181 Mflop/s and 236 Mflop/s, respectively. The IBM SP2 at Virginia Tech has two wide nodes with 512MB of memory and twelve thin nodes. Eight of the thin nodes have 128MB and four thin nodes have 256MB of memory. Message passing is accomplished by the High Performance Switch that provides a minimum of four paths between any pair of nodes in the system.

The Paragon XP/S-7 at Virginia Tech has 100 General Purpose (GP) compute nodes. Each GP node has two Intel i860XP processors (50 MHz) — one dedicated to applications and the other to message passing — and 32MB of memory. The Paragon XP/S-5 at Oak Ridge National Laboratory provides 128 GP compute nodes, 64 of which have 16MB of memory and the other 64 have 32MB of memory. Also available are four 128MB MP compute nodes. For an Intel Paragon processor, the LINPACK benchmark performance is 34 Mflop/s. The communication between compute nodes is handled by a high speed network which connects the nodes in a two-dimensional mesh.

The Cray T3E at the National Energy Research Scientific Computing Center has 152 Application (APP) nodes, each equipped with a DEC Alpha processor (300 MHz) and 256MB of memory. The LINPACK benchmark performance of a DEC Alpha processor is 420 Mflop/s. Nodes are connected by a low-latency, high-bandwidth interconnection network arranged in a three-dimensional torus.

6.5. Derivation of the Isoefficiency Function.

Equation (1) takes the following form when GMRES(k) is used for solving the test problem:

$$W(Kn, k) = eV(W(Kn, k), p, \bar{h}), \tag{3}$$

where $\bar{h} = (t_s, t_w)^t$. Parallel GMRES(k) with Householder reflection orthogonalization incurs an overhead in matrix-vector multiplication, Householder reflection generation and application, and in the residual norm update, performed on a single processor.

6.5.1. Assumptions.

The isoefficiency function is derived under the following assumptions for the serial time calculation, graph partitioning of a matrix, and communication handling: (a) all the work in the sequential algorithm is considered useful; (b) graph partitioning algorithm produces balanced partitions; (c) for large scale problems, partitions produced by graph partitioning keep the computation to communication ratio no smaller than block-striped partitioning; (d) each processor holds Kn/p rows of the matrix (Figure 20), where $p \leq K$; (e) if $p \leq K$, then each $n \times n$ matrix block is partitioned between no more than two processors; (f) in matrix-vector multiplication, communication is performed by asynchronous sends and synchronous receives; (g) the time complexity of the MPI [49] broadcast operation implemented on the Intel Paragon, IBM SP2, and Cray T3E does not grow substantially as the number of processors increases. Note that since for the majority of realistic applications the amount of computation grows superlinearly in the number of processors assumption (d) is not very constraining.

6.5.2. Overhead due to matrix-vector multiplication.

The parallel overhead due to a matrix-vector multiplication can be predicted by considering the nonzero structure of a given coefficient matrix. For the type of matrices shown in Figure 20, a processor receives at most n vector components from no more than $\lceil p/K \rceil$ processors and sends its Kn/p vector components to at most $\lceil p/K \rceil$ processors. Let $V_{r_j}^{MV}$ and $V_{s_j}^{MV}$ be the total overheads incurred by the processors at data receiving and sending

stages of a matrix-vector multiplication, respectively. Then the total parallel overhead of the matrix-vector multiplication at the jth iteration is $V_j^{MV} = V_{r_j}^{MV} + V_{s_j}^{MV}$ with

$$V_{r_j}^{MV} \approx \left(\left\lceil \frac{p}{K} \right\rceil t_s + nt_w + C_0\right) \times p \quad \text{and} \quad V_{s_j}^{MV} \approx \left(\left\lceil \frac{p}{K} \right\rceil t_s + \frac{Kn}{p} t_w - C_1\right) \times p,$$

where the constants C_0 and C_1 describe the waiting and communication-computation overlapping times for asynchronous communications, respectively. For the problem sizes considered here, $Kn/p \geq n$, and thus $\lceil p/K \rceil t_s \leq t_s$. Combining $V_{r_j}^{MV}$ and $V_{s_j}^{MV}$ yields

$$V_j^{MV} \approx \left(2t_s + \left(n + \frac{Kn}{p}\right) t_w + C_0 - C_1\right) \times p. \tag{4}$$

Observe that $C_0 - C_1 \approx 0$ since the waiting time of each processor during asynchronous matrix vector multiplication is compensated by the time gain in communication-computation overlapping while sending the information. Thus, $C_0 - C_1$ will be dropped from the expression for V_j^{MV}.

6.5.3. Overhead due to Householder reflection generation and application.

Householder reflection generation and application cause a noticeable communication and nonessential work overhead. At the jth GMRES(k) iteration, the overhead V_j^H due to Householder reflection generation and application comprises the overheads $V_{a_j}^H$ and $V_{c_j}^H$ caused by applying and generating Householder reflections, respectively, such that $V_j^H = 2V_{a_j}^H + V_{c_j}^H$ with

$$V_{a_j}^H \approx \Big[jp\big(2(t_s + t_w) + g_a\big)\Big] \times p \quad \text{and} \quad V_{c_j}^H \approx \Big[p\big(2(t_s + t_w) + g_c\big)\Big] \times p, \tag{5}$$

where g_c is the number of operations needed to create a Givens rotation and g_a is the number of operations needed to perform a Givens rotation to zero out one vector component. Since Householder reflections are applied twice per GMRES iteration, $V_{a_j}^H$ has a coefficient of two.

6.5.4. Overhead due to the residual norm update.

Another source of the parallel overhead appears in estimating the condition number of the GMRES least squares problem, which is done on a single processor. For a typical case, when the Krylov subspace dimension is $j, j = 1, \ldots, k$, gathering $\mathcal{O}(j)$ vector components on a processor, estimating the condition number incrementally, and updating the residual norm are relatively inexpensive, since the subspace dimension is much smaller than the matrix dimension N for large scale problems. Global all-to-all communication would be required to perform the condition number estimation in parallel. At the jth GMRES(k) iteration, the parallel overhead V_j^{IC} is caused by the time spent to exchange $\mathcal{O}(j)$ values, to update the residual norm by application of j previous Givens rotations and by generation of a new Givens rotation, and to perform approximately $(j-1)$ operations of the incremental condition number estimation. Thus,

$$V_j^{IC} \approx \Big[2j(t_s + t_w) + jg_a + g_c + (j-1)\Big] \times p. \tag{6}$$

6.5.5. Total parallel overhead.

The total parallel overhead V_j ($V_j = V_j^{MV} + V_j^{H} + V_j^{IC}$) incurred at the jth iteration of the GMRES(k) algorithm with Householder reflections in its orthogonalization stage is

$$\begin{aligned} V_j \approx & \left[2t_s + \left(n + \frac{Kn}{p}\right) t_w\right] \times p \\ & + \Big[2jp\Big(2(t_s + t_w) + g_a\Big) + p\Big(2(t_s + t_w) + g_c\Big)\Big] \times p \\ & + \Big[2j(t_s + t_w) + jg_a + g_c + (j-1)\Big] \times p. \end{aligned}$$

When a restart cycle is finished, i.e., $j = k$, the GMRES(k) algorithm has performed k matrix-vector multiplications, $k+1$ Householder reflection generations, k residual norm updates along with k incremental condition number estimates, and $2k$ Householder reflection applications. At the end of a restart, GMRES(k) calculates the true residual norm using one more matrix-vector multiplication and one more Householder reflection application to

correct the current solution. Combining all the overhead terms incurred during k iterations, the expression for the total overhead is

$$V = \sum_{j=1}^{k+1} \left(V_j^{MV} + V_{c_j}^{H} \right) + \sum_{j=1}^{k} \left(2V_{a_j}^{H} + V_j^{IC} \right) + V_{a_k}^{H}. \tag{7}$$

Substituting equations (4), (5), and (6) into this equation results in

$$\begin{aligned} V \approx\ & (k+1)\, p \left(2t_s + \left(n + \frac{Kn}{p} \right) t_w \right) \\ & + k(k+1)p^2 \Big(2(t_s + t_w) + g_a \Big) + (k+1)p^2 \Big(2(t_s + t_w) + g_c \Big) \\ & + kp^2 \Big(2(t_s + t_w) + g_a \Big) + kp \left((k+1)(t_s + t_w) + \frac{(k+1)}{2} g_a + g_c + \frac{(k-1)}{2} \right) \\ & = C_g p^2 + (\bar{h}^t \bar{c} + C_k')p + Kn(k+1)t_w, \end{aligned}$$

where

$$C_g = k(k+2)\Big(2(t_s + t_w) + g_a \Big) + (k+1)\Big(2(t_s + t_w) + g_c \Big),$$

$$C_k' = k \left(\frac{(k+1)}{2} g_a + \frac{(k-1)}{2} + g_c \right),$$

and $\bar{h} = (t_s, t_w)^t$, $\bar{c} = (c_1, c_2)^t$ with $c_1 = k^2 + 3k + 2$ and $c_2 = k^2 + (n+1)k + n$.

The expression for V is a quadratic polynomial in p. Thus for a fixed k, the fastest growing term is the leading term. The leading term in V comes from creating and applying Givens rotations on processors logically connected in a ring.

6.5.6. Relation between the useful work and the parallel overhead.

In the test problem considered here, $N_z \approx 6Kn$. Thus $W \approx Kn(2k^2 + 11k + 9) + C_k$. Substituting the expressions for V and W into equation (3) gives

$$Kn\Big[(2k^2 + 11k + 9) - e(k+1)t_w \Big] + C_k \approx\ e \times \Big[C_g p^2 + (\bar{h}^t \bar{c} + C_k')p \Big]. \tag{8}$$

The following relation can be derived for K:

$$K \approx\ e \times \Big[\frac{C_g}{n\,(2k^2 + 11k + 9 - e(k+1)t_w)} p^2 + \frac{\bar{h}^t \bar{c} + C_k'}{n\,(2k^2 + 11k + 9 - e(k+1)t_w)} p \Big]. \tag{9}$$

Note that since the term containing C_k is small and has no dependence on either K or p, it does not appear in equation (9).

If one considers the GMRES(k) solution process as consisting of l restart cycles and including the computation of the initial residual r_0 (which requires one matrix-vector multiplication and one subtraction), then equation (4) can be rewritten as

$$l\times Kn\Big[(2k^2+11k+9)+C_k-e(k+1)t_w\Big]+Kn(7-et_w)\approx\ l\times e\times\Big[C_gp^2+(\bar{h}^t\bar{c}+C'_k)p\Big].\quad(10)$$

Comparison of equations (8) and (10) suggests that the isoefficiency function value is smaller if the initial residual computation is taken into consideration, since the useful work becomes larger and makes the leading term coefficient of equation (9) smaller.

6.6. Machine-Dependent Constants.

To calibrate the expression of the isoefficiency function, the constants t_c, t_s, and t_w are determined for the target parallel computers. Their numerical values are obtained using simple appropriate models, for which a sufficient amount of empirical data can be collected to estimate accurately these values. The time t_c needed to perform a floating point operation is obtained as the parameter of a linear regression model $T_1 = t_cW'$, where W' includes an operation count for W and additional terms capturing the effects of memory hierarchy operations when solving large scale problems by a sequential algorithm. Clearly, the value of W' differs from one parallel architecture to another depending on such factors as the memory size and the interface mechanism among memory layers and processor. The amount of time that accounts for the memory hierarchy operations can often be modeled only by studying particular cases of a problem solved on a given architecture. On supercomputers with small cache size and a centralized I/O mechanism, such as the Intel Paragon (16KB of level-one on-board data cache), slow memory (paging) operations affect the overall computation time considerably. In a given sequential algorithm, this effect is noticeable already for medium-size ($N\approx 5000$) problems. Thus the cost of memory accesses (the cost of load/store operations) to compute a floating point value is added to the overall number of operations performed by the sequential algorithm. To perform a floating point operation, two loads from slow memory and one store operation are needed in the worst case.

Observation of the performance of the sequential algorithm on the IBM SP2 also suggests that a portion of its execution time is spent on fetching data from a slower memory. Although an IBM SP2 processor has a significant amount of data cache (64KB and 256KB for Models 390 and 590, respectively), this cache is not placed on chip, thus the interface with the processor presents a bandwidth bottleneck. Also, neither Model 390 nor Model 590 has level-two cache, which plays an important role in floating point computations. Solution of large scientific problems on the IBM SP2 causes a high cache miss rate and, subsequently, frequent references to main memory. Similarly to the Intel Paragon, the useful work W on the IBM SP2 was augmented by the cost of the load and store operations. The total work of a sequential algorithm W' is approximately equal to $3\Big[(k+1)N_z + N(2k^2+5k+3) + C_k\Big]$ on the Intel Paragon and IBM SP2.

On the other hand, each processor of the Cray T3E is coupled with 8KB of data cache and a 96KB of the level-two data cache. In addition, there are 4MB of on-board data cache. This is enough cache capacity to hold all the floating point data of the problem sizes considered here with a high cache hit rate. Thus, $W' \approx W$ is acceptable in this case. Measurement of T_1 was repeated three times for each value of W'. Figures 22(a), 23(a), and 24(a) show the time T_1 averaged over three repetitions for the IBM SP2, Intel Paragon, and Cray T3E, respectively.

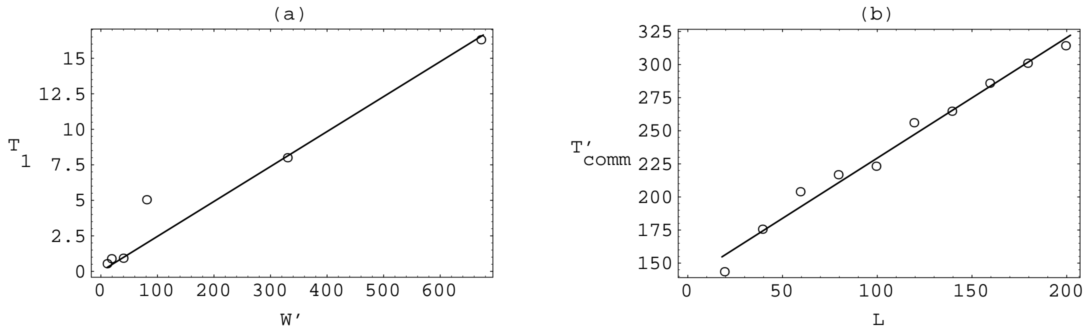

FIG. 22. *Observed on the IBM SP2 and predicted by linear regression CPU and communication times. (a) Uniprocessor CPU time T_1 (in seconds) for W' (in Mflops). (b) Communication time T'_{comm} (in microseconds) to exchange L double precision constants between two processors.*

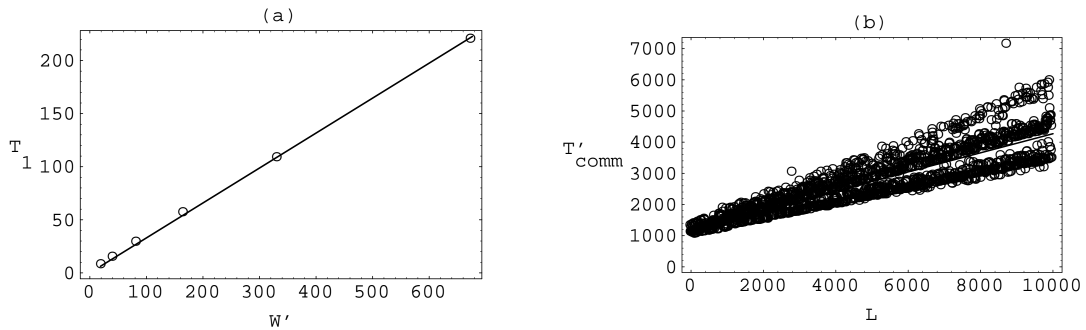

FIG. 23. *Observed on the Paragon and predicted by linear regression CPU and communication times. (a) Uniprocessor CPU time T_1 (in seconds) vs. total work W' (in Mflops). (b) Communication time T'_{comm} (in microseconds) to exchange L double precision constants between two processors.*

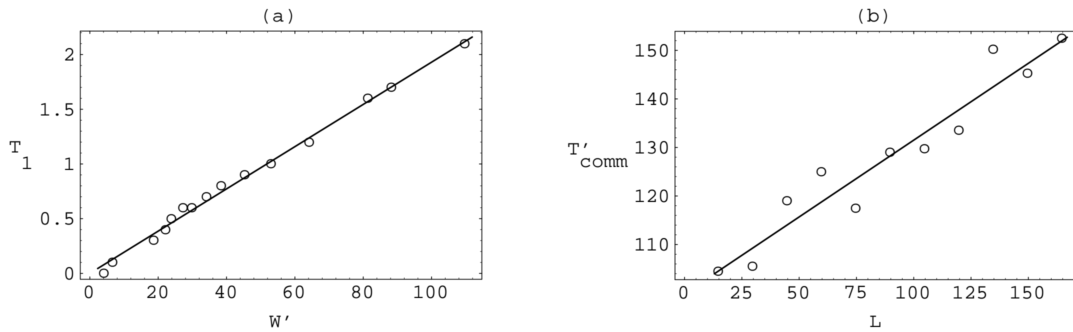

FIG. 24. *Observed on the Cray T3E and predicted by linear regression CPU and communication times. (a) Uniprocessor CPU time T_1 (in seconds) for W' (in Mflops). (b) Communication time T'_{comm} (in microseconds) to exchange L double precision constants between two processors.*

The communication start-up time and the transmission time of a double precision number are determined from a communication cost model $T_{comm} = t_s + t_w L$, where T_{comm} is the response variable and L is the predictor variable. This model estimates the communication time T_{comm} between two processors under the assumption that the time required to send a message from one processor to another is independent of processor location and the number of other processors that might be communicating at the same time. The experimental data were gathered from measuring the time T'_{comm} needed to exchange a message between two

processors ($T'_{comm} = 2T_{comm}$). Measurements were repeated two times for each processor. For the IBM SP2 and Cray T3E, Figures 22(b) and 24(b), respectively, show the averages over four observations of T'_{comm}. For the Intel Paragon, Figure 23(b) depicts all observed values of T'_{comm}.

For the IBM SP2, the linear regression $T_1 = 0.024W'$ models the uniprocessor CPU time with the standard deviation of errors equal to 1.84. The regression explains 81% of the variation in T_1, because the coefficient of determination of this model is 0.81. For the linear regression $T_{comm} = 69 + 0.45L$, the standard deviation of errors is 9.6 and the coefficient of determination is 0.97.

For the Intel Paragon, the linear model is $T_1 = 0.33W'$ with the standard deviation of errors equal to 2.25. The regression explains 99% of the variation in T_1. The standard deviation of values T_{comm} observed in each repetition of the experiment was very large. It is obvious from Figure 23(b), showing the observations from the experiments, that the variance of the linear model is large, which diminishes the usefulness of the linear model. However, if the message length is known or lies within a certain range, some constant value of T_{comm} can be estimated by the linear regression within that specific range. In particular, if a small array of double precision constants is transmitted, the communication latency can be approximated by the start-up time, i.e., $T_{comm} = 605\mu$s.

For the Cray T3E, the linear model is $T_1 = 0.019W'$ with the standard deviation of errors equal to 0.07. The regression explains 99% of the variation in T_1. To estimate the communication time T_{comm}, the linear regression $T_{comm} = 50 + 0.16L$ is obtained. For this regression, the standard deviation of errors is 6.3 and the coefficient of determination is 0.91.

As a result, the times spent for computing a million floating point operations are 0.024s, 0.33s, and 0.019s for the IBM SP2, Intel Paragon, and Cray T3E, respectively. The reciprocal of t_c defines uniprocessor computing rates of 41.6 Mflop/s, 3.0 Mflop/s, and 53 Mflop/s, correspondingly. Start-up–transmission time pairs (in microseconds) are (69, 0.45) and (50, 0.16) for the IBM SP2 and Cray T3E, respectively.

6.7. Numerical Experiments.

The behavior of six different parallel systems has been studied. An algorithm component of each parallel system is either GMRES(k) with HG orthogonalization or GMRES(k) with MHG orthogonalization and an architecture component is one of three parallel computers: IBM SP2, Cray T3E, or Intel Paragon. For convenient reference, the following abbreviations denote these parallel systems:

MHG_P — GMRES(k) with MHG executed on the Intel Paragon;
MHG_S — GMRES(k) with MHG executed on the IBM SP2;
MHG_T — GMRES(k) with MHG executed on the Cray T3E;
HG_P — GMRES(k) with HG orthogonalization on the Intel Paragon;
HG_S — GMRES(k) with HG orthogonalization on the IBM SP2;
HG_T — GMRES(k) with HG orthogonalization on the Cray T3E.

Both the algorithm and the architecture were subject to scaling in the experiments, where scaling an algorithm means increasing the problem size and scaling an architecture is a mapping of an application to increasing numbers of processors. To examine the isoefficiency scalability of these parallel systems, the (processor number, problem size) pairs with the same efficiency were selected. For the efficiency computation, the parallel time needed to perform two GMRES(k) restart cycles and the initial residual r_0 computation by a parallel system was recorded as well as the time for executing the same algorithm on a uniprocessor. Three repetitions of each experiment were performed (the plots show the average of these three values). For each parallel architecture, the maximum coefficient of variation for the measured data (parallel time) in three repetitions is determined as 0.09, 0.14, and 0.08, for the IBM SP2, Intel Paragon, and Cray T3E, respectively.

In the remainder of this chapter, the predicted and computed isoefficiencies are compared for each parallel system and across different parallel systems, and the chapter concludes with a discussion of the effects of resource saturation with scaling and the range of applicability of the isoefficiency function.

Figures 25–30 show the (processor number, problem size) pairs with a fixed efficiency for HG_S, MHG_S, HG_P, MHG_P, HG_T, and MHG_T applied to solve a given test problem,

respectively. The curve shown as a dashed line represents the predicted isoefficiency function f_E for the corresponding target parallel architecture. The accuracy of the prediction is compared with the least squares fit $\tilde{f}_E$ of a quadratic polynomial to experimental data, where only the averages of three runs are shown in figures. $\tilde{f}_E$ was obtained using Mathematica [75] and plotted as a solid line.

6.7.1. Isoefficiency of GMRES(k) on the IBM SP2.

Least squares fits to HG_S data in Figure 25 for efficiencies .14, .22, .34, and .42 with $k = 15$, respectively, are

$$0.17p^2 - 1.11p, \qquad 0.31p^2 - 4.15p, \qquad 0.48p^2 - 6.06p, \qquad 0.74p^2 - 7.42p.$$

Least squares fits to MHG_S data in Figure 26 for efficiencies .14, .22, .34, and .42 with $k = 15$, respectively, are

$$0.10p^2 + 5.15p, \qquad 0.17p^2 + 3.14p, \qquad 0.40p^2 - 1.17p, \qquad 0.56p^2 - 4.31p.$$

Predicted isoefficiency functions for the same efficiences and the same value of k, respectively, are

$$0.13p^2 + 0.07p, \qquad 0.23p^2 + 0.13p, \qquad 0.42p^2 + 0.23p, \qquad 0.58p^2 + 0.33p.$$

Least squares fits to MHG_S data in Figure 27 for efficiencies .22 and .42 and $k = 25$, respectively, are

$$0.24p^2 + 2.03p \qquad \text{and} \qquad 0.75p^2 - 5.06p.$$

Predicted isoefficiency functions for efficiences .22 and .42 with $k = 25$, respectively, are

$$0.26p^2 + 0.14p \qquad \text{and} \qquad 0.66p^2 + 0.36p.$$

Each least squares approximation grows in accordance with the corresponding predicted isoefficiency. Thus, the same function f_E can be used to estimate the isoefficiency scalabilities of MHG_S and HG_S. However, the leading term coefficients for MHG_S are slightly smaller than the leading term coefficients for HG_S, which implies that MHG_S may be more

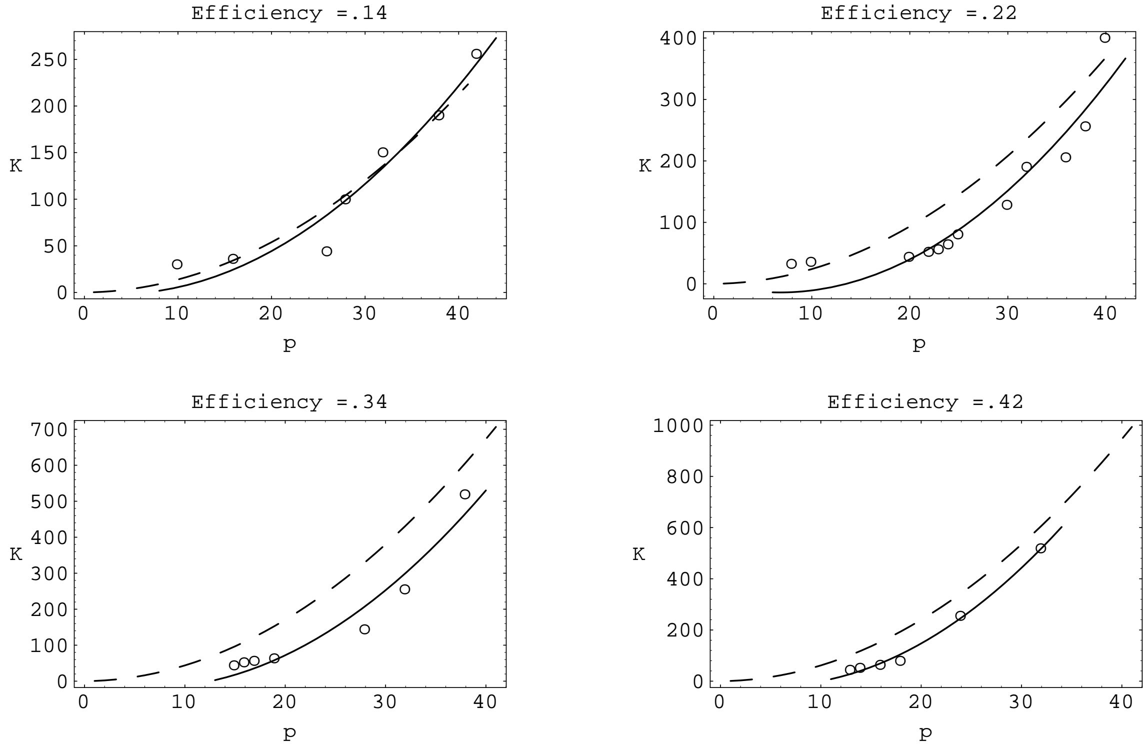

FIG. 25. *Isoefficiency curves for HG_S with $k = 15$. Dashed line — theoretical; solid line — fit to data.*

scalable. As reflected in the form of the isoefficiency function, the larger the efficiency to be maintained, the larger the increase in the problem size that is required with scaling of an architecture component. In particular, the leading term coefficient in f_E grows with the fixed efficiency parameter. Consider, for example, f_E with $E = .14$ and $E = .42$, where the coefficient of the quadratic term increases by a factor of 4.5.

When the restart parameter k increases, the leading term coefficient also increases. This variation in the isoefficiency function value is in agreement with consideration of the whole restart cycle (k iterations) in the isoefficiency analysis of GMRES(k), which differs in this sense from the analysis of the preconditioned conjugate gradient method conducted by Gupta and Kumar [29] for a single iteration of the method.

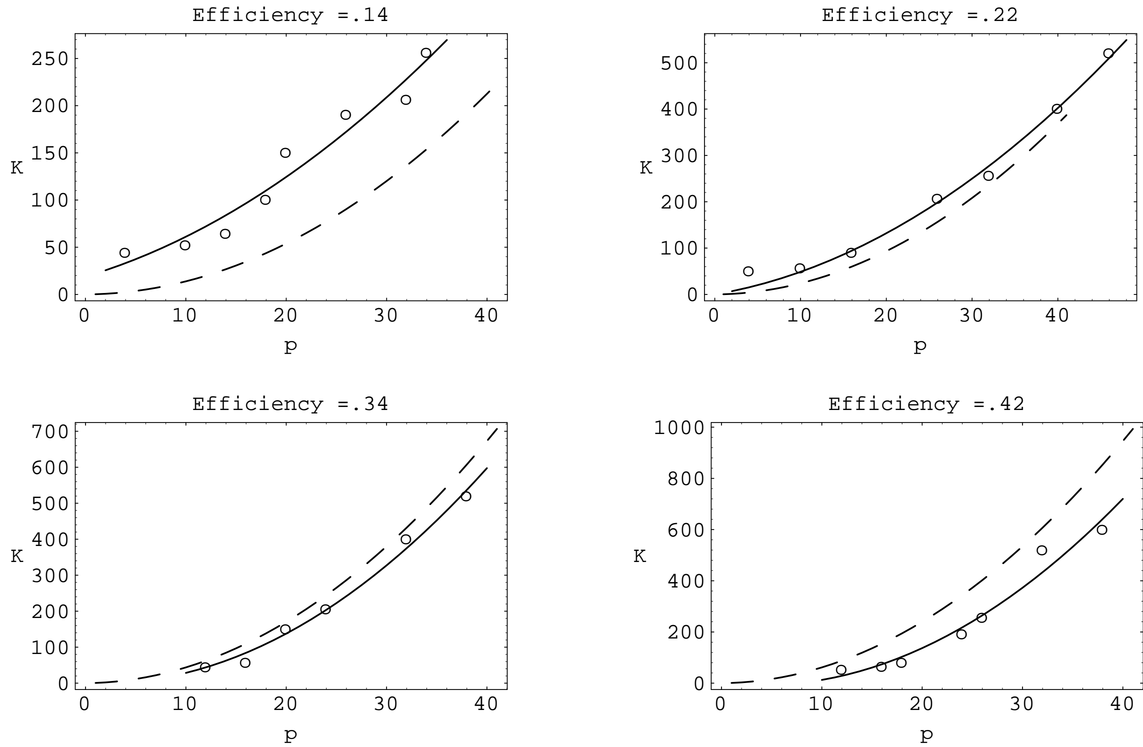

FIG. 26. *Isoefficiency curves for MHG_S with k = 15. Dashed line — theoretical; solid line — fit to data.*

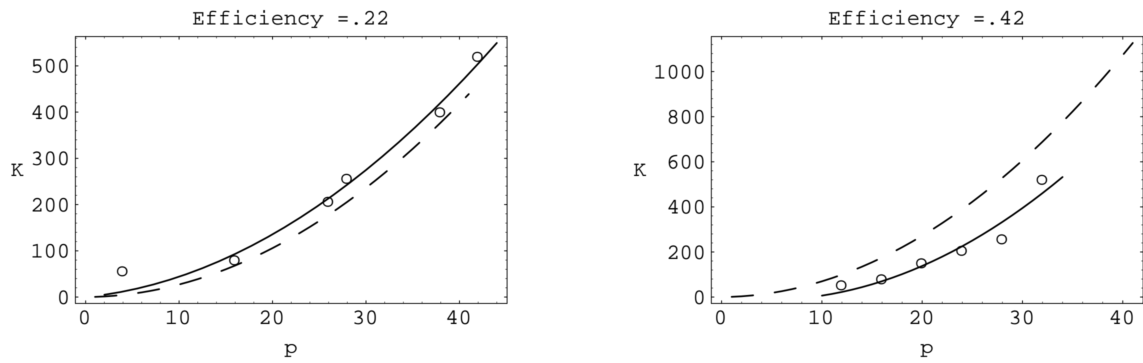

FIG. 27. *Isoefficiency curves for MHG_S with k = 25. Dashed line — theoretical; solid line — fit to data.*

6.7.2. Isoefficiency of GMRES(k) on the Intel Paragon.

Least squares fits to HG_P data in Figure 28 (top) for efficiencies .28, and .46 with $k = 15$, respectively, are

$$0.03p^2 + 1.98p \quad \text{and} \quad 0.13p^2 + 0.05p.$$

Least squares fits to MHG_P data in Figure 28 (bottom) for efficiencies .28, and .46 with $k = 15$, respectively, are

$$0.05p^2 + 1.58p \quad \text{and} \quad 0.13p^2 + 0.54p.$$

Predicted isoefficiency functions for the same efficiences and value of k, respectively, are

$$0.07p^2 + 0.04p \quad \text{and} \quad 0.15p^2 + 0.08p.$$

Least squares fits to HG_P data in Figure 29 for efficiencies .28, and .46 with $k = 35$, respectively, are

$$0.08p^2 + 0.28p \quad \text{and} \quad 0.18p^2 - 1.14p.$$

Predicted isoefficiency functions for the same efficiences and $k = 35$, respectively, are

$$0.08p^2 + 0.04p \quad \text{and} \quad 0.18p^2 + 0.09p.$$

For the Intel Paragon parallel systems, each least squares approximation grows similarly to the corresponding predicted isoefficiency.

6.7.3. Isoefficiency of GMRES(k) on the Cray T3E.

Least squares fits to HG_T data in Figure 30 (top) for efficiencies .24 and .36, respectively, are

$$0.45p^2 - 1.87p \quad \text{and} \quad 0.70p^2 - 5.36p.$$

Least squares fits to MHG_T data in Figure 30 (bottom) for efficiencies .24 and .36, respectively, are

$$0.32p^2 + 1.87p \quad \text{and} \quad 0.58p^2 - 2.71p.$$

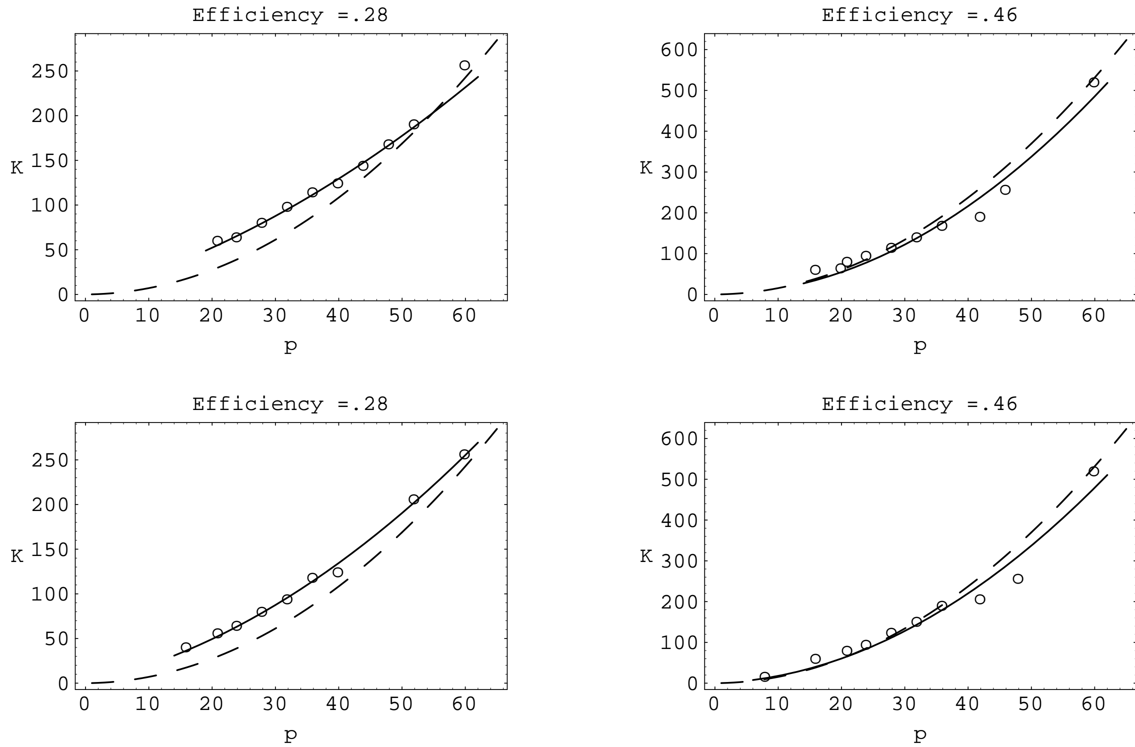

FIG. 28. *Isoefficiency curves for HG_P (k = 15, top) and MHG_P (k = 15, bottom). Dashed line — theoretical; solid line — fit to data.*

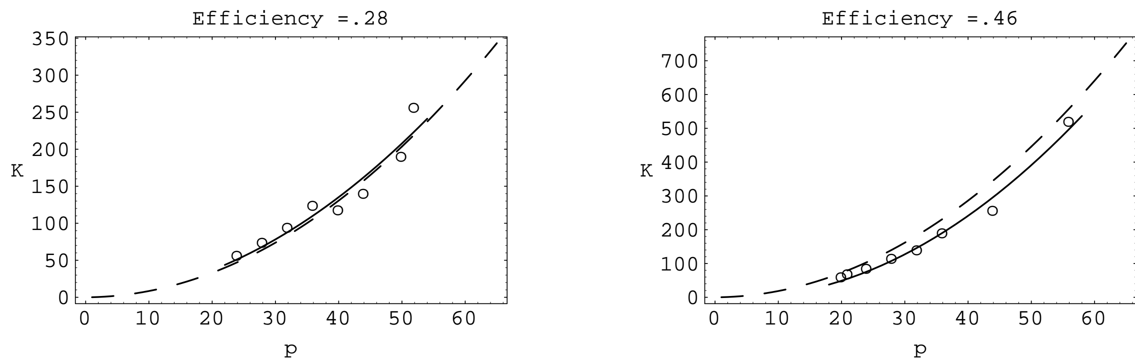

FIG. 29. *Isoefficiency curves for HG_P with k = 35. Dashed line — theoretical; solid line — fit to data.*

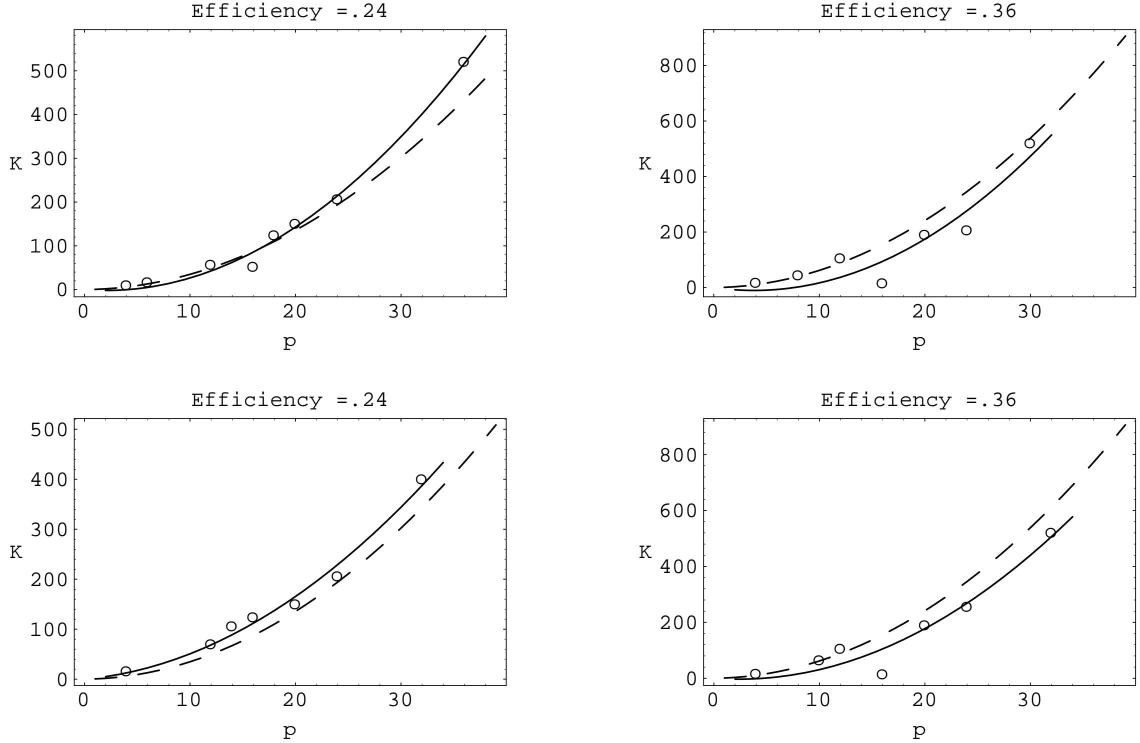

FIG. 30. *Isoefficiency curves for HG_T (k = 15, top) and MHG_T (k = 15, bottom). Dashed line — theoretical; solid line — fit to data.*

Predicted isoefficiency functions for the same efficiences, respectively, are

$$0.33p^2 + 0.18p \quad \text{and} \quad 0.59p^2 + 0.32p.$$

For the Cray T3E parallel systems, each least squares fit grows similarly to the corresponding predicted isoefficiency.

Since the architectures considered here have different values of machine-dependent constants, the corresponding isoefficiency functions have different coefficients, even though they include terms of the same order. For example, the isoefficiency function f_{E_T} on the Cray T3E differs significantly from the isoefficiency function f_{E_S} on the IBM SP2. In particular, for $E = 0.42$,

$$f_{E_T} = 0.76p^2 + 0.48p \quad \text{and} \quad f_{E_S} = 0.58p^2 + 0.33p.$$

The coefficient of the leading term of f_{E_T} is larger than that of f_{E_S} since the computing rate $1/t_c$ on the Cray T3E is faster than on the IBM SP2, while the improvement in t_s and t_w is not sufficient to hide the communication latency. Hence, the idling time on the Cray T3E is larger than on the IBM SP2. The isoefficiency function of the Paragon architecture presents the case when moderate values for *all* the machine-dependent constants involved result in good isoefficiency scalability characteristics. However, with scaling of the problem size, less powerful architectures, such as the Intel Paragon, tend to exhaust their resources faster than more powerful ones, such as the IBM SP2 and the Cray T3E.

6.7.4. Speed-up as an indication of resource saturation.

Comparison of speed-ups obtained on each parallel architecture (Figure 31) also pinpoints rapid resource saturation on the Intel Paragon, where for large problem sizes, the superlinear speed-ups are due to the poor performance of a sequential algorithm. (The values of k and K in Figure 31 have no special significance, other than that they represent data generated for earlier figures.) Such performance degradation is explained by extensive memory hierarchy operations. For example, while solving a medium size problem ($K = 206$) on an Intel Paragon node, the main memory exchanged (on average) 192 pages with disk memory every five seconds. On the IBM SP2, the superlinear speed-up manifests itself for larger problem sizes and has a smaller magnitude than on the Intel Paragon. On the other hand, the speed-ups on the Cray T3E show that the problems considered here may be solved successfully on a single T3E processor and that Amdahl's Law governs the performance analysis. In such a case, when the sequential timing results are not contaminated by resource exhaustion, the isoefficiency analysis is meaningful. Therefore, for a given parallel system there is an upper bound on the problem size, beyond which the isoefficiency prediction is unreliable. Clearly, this bound is much smaller if it is based on an actual sequential execution; thus, some scalability analyses consider an estimate of the sequential time instead of actually measuring it. Such an approach is taken, for example, by Sun and Rover [63] in a scalability analysis which does not require solving large problems on a single processor.

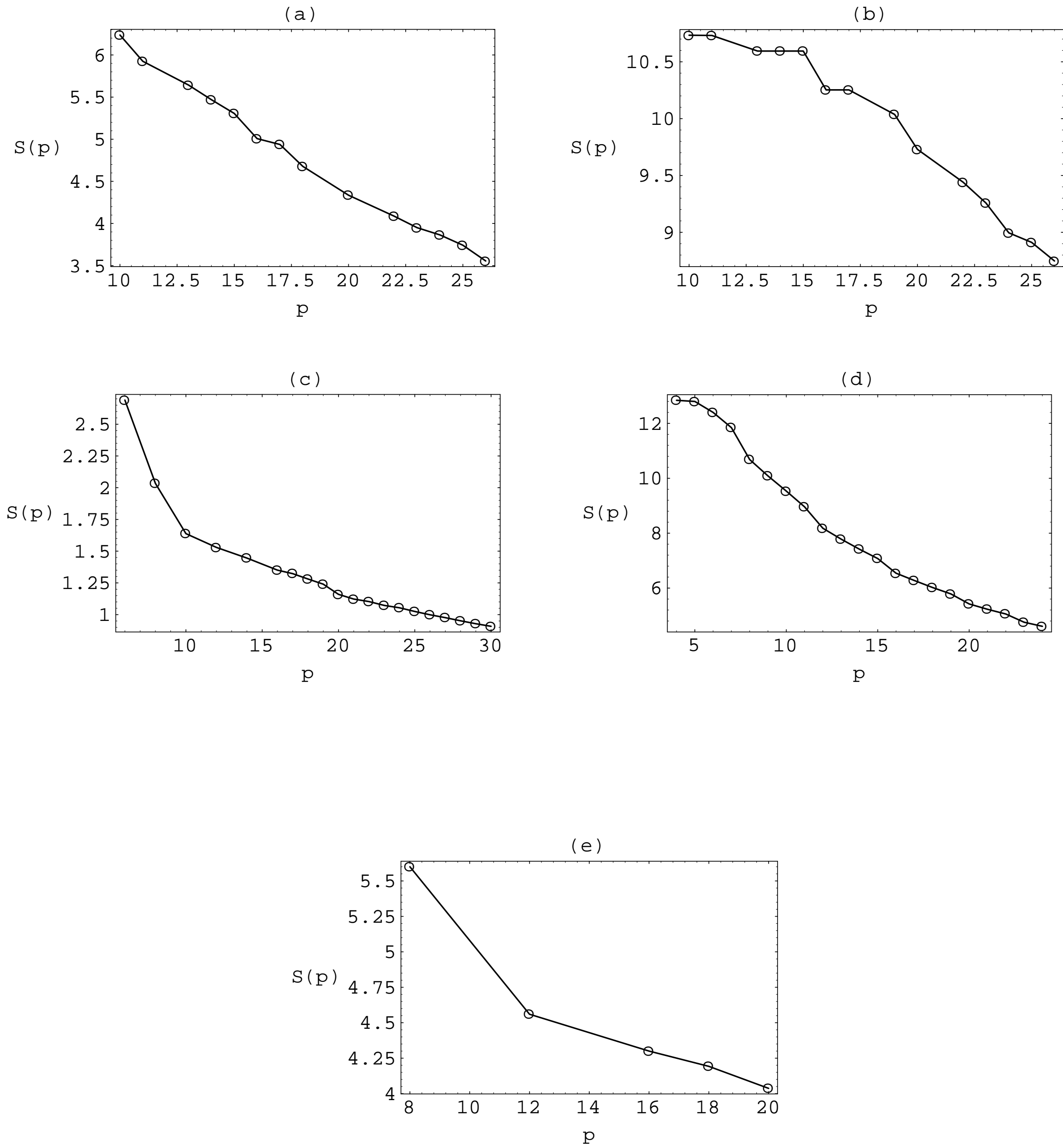

FIG. 31. *Speed-up curves with $k = 15$ for HG_S: (a) $K = 44$, (b) $K = 120$; HG_P: (c) $K = 10$, (d) $K = 118$; HG_T: (e) $K = 124$.*

6.8. Conclusions.

In this chapter, the isoefficiency analysis is carried out for parallel versions of GMRES(k)

with Householder reflections orthogonalization implemented on the IBM SP2, Intel Paragon, and Cray T3E. The theoretical part of this analysis not only establishes an asymptotic relation between the increase in problem size and number of processors, but also provides an analytic expression for the isoefficiency function. Communication and nonessential work overheads are identified for parallel GMRES(k) applied to solve a real-world circuit simulation problem distributed among processors in a block-striped fashion (see Chapter 5). For vertex-based partitioning of the problem, the theoretical overhead is claimed to be the same under certain assumptions on load balancing by a partitioning algorithm. Thus, the same isoefficiency function is derived for the parallel GMRES(k) implementations used with each of these two partitioning schemes. On target parallel architectures, experimental results support the claim and match closely predicted isoefficiency functions. Both theoretical and experimental results show that the isoefficiency function is a quadratic polynomial with a small leading term coefficient, which implies that the given parallel GMRES(k) implementations are reasonably scalable on the given parallel architectures, namely, on an IBM SP2, Intel Paragon, and Cray T3E. In general, a parallel algorithm is considered scalable if the isoefficiency function (in any form) exists; that is, given a parallel architecture and a fixed efficiency value, the amount of useful work in a given algorithm can be determined such that a given parallel system achieves the desired efficiency.

It has also been observed that, because of resource saturation, there is an upper bound on the problem size beyond which the sequential execution time is affected by extensive memory hierarchy operations. In this case, the time measurement of solving large problems on a single processor may impede the isoefficiency analysis, since the efficiency $E = T_1/pT_p$ becomes greater than one and the relation between the useful work and the parallel overhead $W(\bar{N}) = eV(W(\bar{N}), p, \bar{h})$ cannot be applied to study scalability of an algorithm-machine combination.

7. SCALABILITY ANALYSIS OF ADAPTIVE GMRES(k).

The performance of parallel adaptive GMRES(k) was studied experimentally and compared with the performance of sequential adaptive GMRES(k) to obtain efficiency characteristics of the parallel version. In the experiments, the Krylov subspace dimension varied from $k = 15$ to $k = 50$ by adding four Krylov subspace vectors. The threshold parameters bgv and smv of the adaptive strategy remained constant.

7.1. Efficiency of Adaptive GMRES(k).

The erratic behavior of the efficiency values for adaptive GMRES(k) (Figure 32 top and bottom left plots) suggests that not only the convergence for adaptive GMRES(k) differs from that for the restarted GMRES (see Chapter 4), but also their parallel algorithm-architecture performances differ [60].

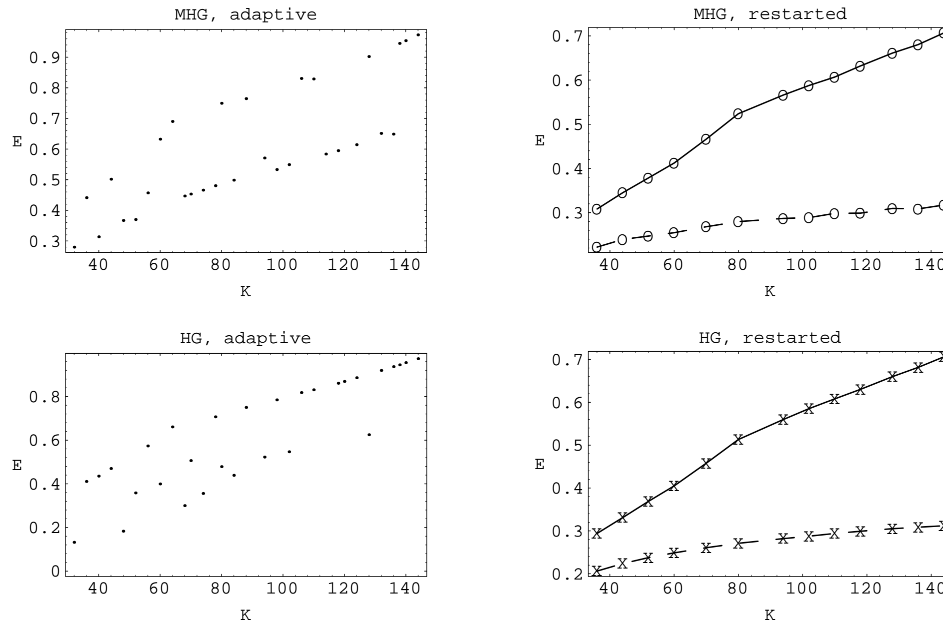

FIG. 32. *Efficiency on 21 processors, GMRES(kmax) (solid lines), GMRES(k) (dashed lines).*

In the case of problem size scaling considered here, changing matrix dimensions (K times) can lead to a variation in test problem difficulty as well as in its size. This often happens in practical applications, such as postbuckling analysis of structures, described in Chapter

4, with increase in problem size. By definition, the adaptive GMRES(k) algorithm proceeds qualitatively differently on different problems. Therefore, for the adaptive GMRES(k) algorithm, the operation count of the full convergence history (or of the convergence history until the maximum subspace dimension is reached) represents the useful work to be considered in the isoefficiency analysis. Also, from the equation

$$W(Kn, k) = eV(W(Kn, k), p, \bar{h}),$$

stated in the previous chapter, the parallel overhead includes terms depending on k as a parameter. In the previous chapter, several numerical experiments deal with computing the isoefficiency function for different fixed values of k. These experiments indicate that, as predicted theoretically, the isoefficiency function increases with increase in k. However, if it is not known in advance when increases in k occur during the solution process, as, for example, in adaptive GMRES(k), the isoefficiency characteristics of adaptive GMRES(k) are unpredictable. Variation in the Krylov subspace dimension makes the scalability analysis of the adaptive GMRES(k) algorithm more complicated than the analysis of the conjugate gradient method [29], where an operation count and parallel overhead per iteration is sufficient to derive an isoefficiency function. By the same token, the definition of both useful work and parallel overhead differ from their expressions in the analysis of the GMRES(k) method, which can be performed for a particular subspace dimension k and a particular operation count of a single GMRES iteration (see Chapter 6).

One approach to estimate the efficiency of the adaptive GMRES(k) algorithm on scaled problems is to introduce a measure of problem difficulty into the problem size definition. Since it is hard to predict convergence of the adaptive GMRES(k) method on an arbitrary problem, making problems more uniform by preconditioning is another way of dealing with the issue of useful work scaling for the purpose of an isoefficiency analysis.

7.2. Conclusions.

For adaptive GMRES(k), the results (Figure 32) indicate a strong dependency of the efficiency of a parallel version of the method on the variability in problem difficulty when problems are scaled. Thus a measure of the problem difficulty has to be a parameter of a scaling procedure, or scalable preconditioning has to be applied to normalize the problem difficulty with increasing problem size, or isoefficiency is not a meaningful concept for adaptive algorithms (cf. Figure 32).

8. RELATION BETWEEN PARALLEL SPARSE LINEAR ALGEBRA, STEPPING ALGORITHM, AND USER-DEFINED FUNCTIONS IN PARALLEL HOMPACK90.

The ultimate goal of the parallel sparse linear algebra methods proposed in this research is to enable a practical application of homotopy algorithms to solve large scale problems; that is, to be a part of a distributed version of HOMPACK90. Two major stages are considered in achieving this goal. First, the requirements of zero curve tracking necessitate the adaptation of efficient sparse linear algebra techniques to solve sequences of linear systems of varying difficulty and to achieve high accuracy in the solution. Second, the mechanism of integrating parallel sparse linear algebra methods into ||HOMPACK90 is specified in this chapter. At the second stage, the following design issues are addressed: internal sparse matrix format in ||HOMPACK90 (Section 8.1), processing the output of user-defined functions (Section 8.2), and the outline of the overall design of a distributed version of HOMPACK90 (Section 8.3).

8.1. Internal Data Format in ||HOMPACK90.

Similar to the sequential version, ||HOMPACK90 is intended to be used as a black-box algorithm. This view of a software package is quite natural for sophisticated optimization techniques. However, it differs from packages comprised of a collection of routines with a variety of "stand alone" routines for a particular purpose, say matrix factorization. Consider, for example, such software packages as PSPARSLIB [45] and ScaLAPACK [8], in which there are many combinations of routines designed for a linear system solution. In practice, the distinction between a black-box software package and a suite of routines usually reveals itself in the way data-dependent and data-independent routines interact and in the amount of control allowed over data structures. The "reverse call" approach, where a subroutine returns to the calling program whenever the subroutine needs certain information (e.g., a function value) or a certain operation performed (e.g., a matrix-vector multiply), can be handled only by an expert user. It makes a called subroutine independent of any particular

data structure and requires the user to provide all the necessary matrix storage and handling information.

Whereas a version of HOMPACK90 uses the reverse communication approach for numerical linear algebra [73], this approach is not used to pass data within the sparse numerical linear algebra routines in HOMPACK90. Thus, the question of a ||HOMPACK90 internal sparse matrix data format needs resolution. There are two general guidelines in choosing this format. From a user viewpoint, ||HOMPACK90 must be able to handle the most common formats; from a parallel system viewpoint, the internal format must support efficient parallel implementations of linear system solvers on targeted parallel computing platforms. A vast majority of engineering applications based on the finite element method (see, e.g., shell and truss problem descriptions in Chapter 4) represents the matrix in the *skyline format*, such as

$$\begin{pmatrix}
\bullet_{1} & \bullet_{3} & & & & & & & & \\
\bullet & \bullet_{2} & \bullet_{5} & & \bullet_{11} & \bullet_{16} & & & & \\
 & \bullet & \bullet_{4} & \bullet_{7} & \bullet_{10} & \bullet_{15} & & & & \bullet_{31} \\
 & & \bullet & \bullet_{6} & \bullet_{9} & \bullet_{14} & & & & \bullet_{30} \\
 & \bullet & \bullet & \bullet & \bullet_{8} & \bullet_{13} & & & & \bullet_{29} \\
 & \bullet & \bullet & \bullet & \bullet & \bullet_{12} & \bullet_{18} & & & \bullet_{28} \\
 & & & & & \bullet & \bullet_{17} & \bullet_{20} & \bullet_{23} & \bullet_{27} \\
 & & & & & & \bullet & \bullet_{19} & \bullet_{22} & \bullet_{26} \\
 & & & & & & \bullet & \bullet & \bullet_{21} & \bullet_{25} \\
 & & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet_{24}
\end{pmatrix}.$$

The packed skyline format consists of a real array and an integer array. The upper triangle is stored in a one-dimensional real array indexed as shown above. The auxiliary integer array (1, 2, 4, 6, 8, 12, 17, 19, 21, 24, 32) of diagonal indices is also required. By convention the auxiliary integer array has length $n+1$ with the $(n+1)$st element containing the length of the packed real array plus one.

The skyline format is supported in the serial HOMPACK90 package. Alternatively for general matrices, sparse linear algebra routines operate with the *compressed sparse row* format (CSR) in HOMPACK90. This compressed sparse row storage format is intended for arbitrary sparsity patterns, and stores only the structural nonzeros of the matrix (i.e., elements that happen to be zero by accident should be treated as nonzero). The data

structure consists of three arrays: a real one-dimensional array q holding only the structural nonzero elements stored by row in row order (the elements within a row need not be in column order), an integer array r with r_i giving the location within q of the beginning of the elements for the ith row, and an integer array c with c_j giving the column index of the jth element of q. By convention, the diagonal elements of the matrix A are always structurally nonzero (for invertible A there always exists a permutation matrix P such that PA has no zeros on the diagonal), and r_{n+1} is the length of q plus one. For example, the data structure for

$$A = \begin{bmatrix} 4 & 0 & 0 & 2 & 0 \\ 7 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 6 \\ 2 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 5 \end{bmatrix}$$

is (here $n = 5$)

$$q = (4, 2, 7, 1, 3, 0, 6, 1, 2, 1, 5),$$

$$r = (1, 3, 5, 8, 10, 12),$$

$$c = (1, 4, 1, 2, 3, 4, 5, 4, 1, 2, 5).$$

The zero in location (3,4) is stored to illustrate a structural nonzero, q and c have the same dimension, and $r_{n+1} = \dim q + 1 = 12$.

In the sequential environment, not only is the skyline format a convenient representation of a matrix arising from the finite element method, but also reduces computational and storage complexity when numerical algorithms (e.g. matrix-vector multiplication) are tuned to take advantage of the matrix skyline structure. However, in parallel computations involving sparse matrices, the skyline format seems inappropriate, for at least two reasons. Savings in storage are not preserved or they incur extra communication among processors that are missing the matrix data because only one half of the matrix is stored due to the matrix symmetry. Computational savings are diminished because of the same problem of excessive communications. On the other hand, data storage in the CSR format does not hinder the performance of parallel algorithms. With row partitioning of a matrix (say vertex-based partitioning) among processors, the local portion of the matrix on each processor is also

stored in the CSR format, so there is no mixture of local and global data structures, and efficient matrix-vector multiplication routines, like that described in Chapter 5, can be easily implemented. Furthermore, the CSR format is also commonly used to represent sparse matrix data in various scientific applications, so this storage format is also quite popular. Format conversion routines can be introduced into ||HOMPACK90. In particular, there should be a routine converting from the skyline storage format to the compressed sparse row format. By setting a value of a ||HOMPACK90 input parameter to a format code corresponding to a given data structure, the user can request the format conversion.

8.2. User-Defined Function Evaluations in ||HOMPACK90.

To obtain the function and its derivative values at a point, ||HOMPACK90 requires user-defined routines. In general, they can be parallel or sequential. Following a suggestion of an earlier effort in developing parallel HOMPACK [12], the user specifies whether the user-defined function evaluations are done in parallel or sequentially by setting a proper input parameter to "parallel" (PF) or "sequential" (SF). For sparse matrix computations, when this parameter is set to PF, the user also has to provide an array specifying the matrix row mapping to processors; that is, the ith element of this array is the processor to which the ith row of the matrix is assigned. When this parameter is set to SF, ||HOMPACK90 computes the function and derivative values on a single processor, calls a partitioning algorithm, say Metis [41], and distributes the matrix rows (possibly after creating a structurally symmetric matrix) in accordance with the output of this partitioning algorithm. Note that since a sequence of linear systems with structurally equivalent matrices needs to be solved in homotopy zero curve tracking, a graph partitioning algorithm is called only once (on the first step of curve tracking). Thus, the cost of partitioning is amortized over the course of a parallel homotopy method.

The foregoing tacitly assumes the Jacobian matrix will be computed by rows. There are problems (e.g., the adjoint equations for an ODE) where the Jacobian matrix is naturally computed one entire column at a time. There are essentially two choices: calculate the Jacobian matrix by columns and provide column oriented linear algebra, or calculate the Jacobian matrix by columns and redistribute the data for row oriented linear algebra. The latter choice may be viable whenever graph partitioning needs to be used for matrix partitioning.

8.3. Global View of ||HOMPACK90.

Curve tracking is an inherently sequential process, although some of its vector operations may be executed on multiple processors in parallel. Such parallelism is well suited to a shared memory environment. However, the fine granularity of some curve tracking tasks is not suitable for distributed memory computers, which incur communication overhead in exchanging data. Hence, the remainder of this section considers only coarse-grain parallelism of the computationally intensive parts of ||HOMPACK90 that benefit from parallel execution in distributed memory environments: namely, numerical linear algebra and function evaluation.

In general, the user-defined routines may evaluate the function and derivatives serially or in parallel and dictate either sequential or parallel execution for the linear algebra routines. For a multiprocessor environment, the choices of sequential or parallel execution for the numerical linear algebra routines are illustrated as a decision tree in Figure 33. (The rationale for sequential execution of dense linear algebra routines is carefully explained in [12]). Figure 33 shows six logical combinations denoting parallel (combinations 1, 3, 5, and 6) and sequential (combinations 2 and 4) solution of a linear system. A further complicating detail, whether the Jacobian matrix is calculated by rows or by columns, is omitted from Figure 33.

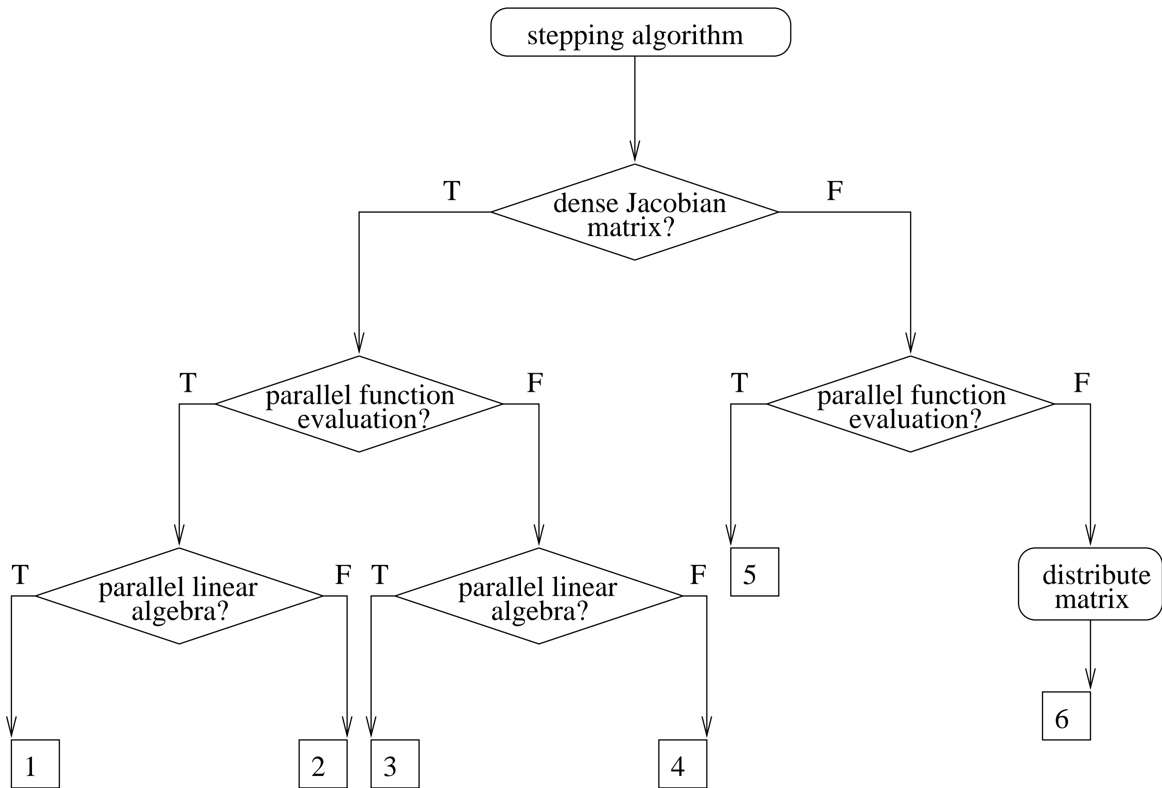

FIG. 33. *Decision tree for implementing numerical linear algebra in parallel.*

9. CONCLUSION.

The parallel sparse linear algebra techniques considered in this research present practical opportunities for application of homotopy methods to large scale real-world problems. These techniques ensure robustness of homotopy zero curve tracking by using a self adaptive strategy based on the given problem characteristics and by utilizing Householder reflections orthogonalization in achieving high accuracy of the linear system solution.

The results of this research lead to a conclusion that adapting the Krylov subspace dimension in restarted Krylov subspace methods well suits solving a sequence of general linear systems of varying difficulty. Comparison of the proposed adaptive GMRES(k), several GMRES variants, and QMR on difficult real-world problems supports this conclusion, since adaptive GMRES(k) outperforms other iterative techniques in the context of homotopy zero curve tracking. The experimental study of adaptive GMRES(k) employed by homotopy methods highlights several practical cases when the Householder reflections orthogonalization in GMRES(k) is clearly a better choice than the commonly used modified Gram-Schmidt orthogonalization. These cases indicate that accurate orthogonalization improves reliability of homotopy zero curve tracking, especially for the curves with sharp turning points. Thus, the extra cost of Householder reflections is well justified. Monitoring the condition number of the GMRES least squares problem is shown to be a good indication of the orthogonalization accuracy provided the condition number of the input matrix is known.

A parallel implementation of adaptive GMRES(k) using Householder reflections has been developed, such that it requires no all-to-all communications and, at the same time, preserves orthogonality of a Krylov subspace basis leading to a highly accurate solution. The idea of performing Householder reflections with a high degree of parallelism, which is due to Chu et al. [16] and Sidje [58], has been extended to the case of an arbitrary matrix row distribution. In particular, this extension deals efficiently with graph partitioned data; that is, it causes no data redistribution to operate with the coefficient matrix at each iteration. For parallel architectures, such as Intel Paragon, the experiments conducted in this research indicate that the advantage of graph partitioning in reducing parallel communication overhead becomes especially noticeable for a large number of processors.

This research studies the isoefficiency scalability of parallel implementations of a Krylov subspace method and, in this sense, is a continuation of the work by Gupta et al. [29], who considered preconditioned conjugate gradient methods. However, a scalability analysis of a Krylov subspace method for general linear systems, such as GMRES(k), requires a somewhat different approach, as shown in this research. Specifically, the whole restart cycle of GMRES(k) rather than a single iteration needs to be considered in determining scalability. The presented isoefficiency analysis of parallel versions of GMRES(k) with Householder reflections orthogonalization establishes both an asymptotic relation between the increase in problem size and number of processors and an analytic expression for the isoefficiency function. For the Cray T3E, IBM SP2, and Intel Paragon, this function is a quadratic polynomial with a small leading term. In contrast, measuring efficiency of adaptive GMRES(k) reveals a strong dependency of its efficiency on the variability in problem difficulty with scaling. Thus a measure of the problem difficulty has to be a parameter of a scaling procedure, or scalable preconditioning has to be applied to normalize the problem difficulty with increasing problem size, or isoefficiency is not a meaningful concept for adaptive algorithms.

Considering conclusions drawn by Chakraborty [12] for parallel dense linear algebra in homotopy methods, this research outlines the design of parallel HOMPACK90 in the form of a decision tree for either sequential or parallel execution of its major components, such as stepping algorithm, user-defined function evaluation, and numerical linear algebra routines.

REFERENCES.

[1] 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 Conf. on Hypercube Concurrent Computers and Applications, Pasadena, CA, (1988), pp. 1463–1472.

[2] D. C. S. Allison, K. M. Irani, C. J. Ribbens, and L. T. Watson, *High-dimensional homotopy curve tracking on a shared-memory multiprocessor*, J. Supercomput., 5 (1992), pp. 347–366.

[3] Z. Bai, D. Hu, and L. Reichel, *A Newton basis GMRES implementation*, IMA J. Numer.Anal., 4 (1994), pp. 563–581.

[4] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, In Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset and H. P. Langtangen, Ed., Birkhauser Press, 1997, to appear.

[5] S. T. Barnard and R. L. Clay, *A portable MPI implementation of the SPAI preconditioner in ISIS++*, In Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, 1997.

[6] M. W. Benson and P. O. Frederickson, *Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems*, Utilitas Math., 22 (1982), pp. 127–140.

[7] Å. Björck, *Solving linear least squares problems by Gram-Schmidt orthogonalization*, BIT, 7 (1967), pp. 1–21.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petiet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK User's Guide*, SIAM, Philadelphia, PA, 1997.

[9] C. H. Bischof and P. T. P. Tang, *Robust incremental condition estimation*, Tech. Rep. CS-91-133, LAPACK Working Note 33, Computer Sci. Dept., Univ. of Tennessee, May, 1991.

[10] P. N. Brown and H. F. Walker, *GMRES on (nearly) singular systems*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 37–51.

[11] D. Calvetti, J. Petersen, and L. Reichel, *A parallel implementation of the GMRES method*, In Numerical Linear Algebra, pp. 31–46, Berlin, 1993.

[12] A. Chakraborty, *Parallel homotopy curve tracking on a hypercube*, Ph.D. Thesis, Computer. Sci. Dept., Virginia Tech, Blacksburg, VA 24061.

[13] A. Chakraborty, D. C. S. Allison, C. J. Ribbens, and L. T. Watson, *The parallel complexity of embedding algorithms for the solution of nonlinear equations*, IEEE Trans. on Parallel and Distributed Systems, 4 (1993), pp. 458 – 465.

[14] E. Chow and Y. Saad, *Approximate Inverse preconditioners for general sparse matrices*, Tech. Rep., UMSI 94-101, Univ. Minnesota Supercomputer Institute, Minneapolis, MN.

[15] 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), pp. 887–899.

[16] E. Chu and A. George, *QR factorization of a dense matrix on a shared-memory multiprocessor*, Parallel Comput., 11 (1989), pp 55–71.

[17] E. De Sturler, *Truncation strategies for (nested) Krylov methods*, In Proc. Copper Mountain Conf. on Iterative Methods, Copper Mtn, CO, 1996.

[18] E. De Sturler and H. A. van der Vorst, *Communication cost reduction for Krylov methods on parallel computers*, In Lecture Notes in Computer Science 797, HPCN, Springer-Verlag, 1994.

[19] J. J. Dongarra, *Performance of various computers using standard linear equations software, (LINPACK Benchmark Report)*, Tech. Report, CS-89-85, Computer. Sci. Dept., Univ. Tennessee, April 1997.

[20] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, PA, 1979.

[21] I. Duff, R. G. Grimes, and J. G. Lewis, *Users' guide for the Harwell-Boeing sparse matrix collection (Release I)*, Tech. Report TR/PA/92/86, CERFACS, France, October 1992

[22] J. Erthel, *A parallel GMRES for general sparse matrices*, ETNA, 3 (1995), pp. 160–176.

[23] R. W. Freund and N. M. Nachtigal, *An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices, part II*, Tech. Rep. 90.46, RIACS, NASA Ames Research Center, November, 1990.

[24] R. W. Freund and N. M. Nachtigal, *QMR: a quasi-minimal residual method for non-Hermitian linear systems*, Numer. Math., 60 (1991), pp. 315–339.

[25] P. E. Gill and W. Murray, *Newton-type methods for unconstrained and linearly constrained optimization*, Math. Programming, 7 (1974), pp. 311–350.

[26] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 2nd ed., 1989.

[27] A. Y. Grama, A. Gupta, and V. Kumar, *Isoefficiency: Measuring the scalability of parallel algorithms and architectures*, IEEE Parallel and Distrib. Technol., 1 (1993), pp. 12–21.

[28] A. Greenbaum, M. Rozlôžník, and Z. Strakoš, *Numerical behaviour of the modified Gram-Schmidt GMRES implementation*, BIT, 37 (1997), pp. 706–719.

[29] A. Gupta, and V. Kumar, *Performance and scalability of preconditioned conjugate methods on parallel computers*, IEEE Trans. Parallel and Distrib. Systems, 6 (1995), pp. 455–469.

[30] J. L. Gustafson, G. R. Montry, and R. E. Benner, *Development of parallel methods for a 1024-processor hypercube*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 609–638.

[31] B. A. Hendrickson and R. W. Leland, *A multilevel algorithm for partitioning graphs*, Tech.Rep., SAND92-1301, Sandia National Laboratories, November 1993.

[32] S. M. Holzer, R. H. Plaut, A. E. Somers, Jr., and W. S. White, *Stability of lattice structures under combined loads*, J. Engrg. Mech., 106 (1980), pp. 289–305.

[33] E. N. Houstis and J. R. Rice, *Parallel ELLPACK: A development and problem solving environment for high performance computing machines*, In Programming Environments for High-Level Scientific Problem Solving, pp. 229–241, Noth-Holland, 1992

[34] T. Huckle and M. Grote, *A new approach to parallel preconditioning with sparse approximate inverses*, Tech. Rep., SCCM 94-03, Scientific Computing and Computational Mathematics Program, Stanford Univ., Stanford, CA, 1994.

[35] S. A. Hutchinson, J. N. Shadid, R. S. Tuminaro, *Aztec User's Guide*, Tech.Rep., SAND95-1559, Sandia National Laboratories, Albuquerque, NM 87185, October 1995.

[36] 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., 1 (1991), pp. 222–251.

[37] M. T. Jones and P. E. Plassmann, *A parallel graph coloring heuristic*, SIAM J. Sci. Comput., 14 (1993), pp. 654–669.

[38] M. T. Jones and P. E. Plassmann, *BlockSolve95 users manual: scalable library software for the parallel solution of sparse linear systems*, Tech. Report ANL-95/48, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439, December 1995.

[39] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear systems*, Parallel Comput., 20 (1994), pp. 753–773.

[40] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, Tech. Report, 95-036, Computer Sci Dept., Univ. Minnesota, Minneapolis, MN 55455, October, 1996.

[41] G. Karypis and V. Kumar, *MeTiS: Unstructured graph partitioning and sparse matrix ordering system*, User's Guide—Version 2.0, Dept. of Computer Sci., Univ. of Minnesota, MN 55455, 1995.

[42] R. K. Kapania and T. Y. Yang, *Formulation of an imperfect quadrilateral doubly curved shell element for postbuckling analysis*, AIAA J., 24 (1986), pp. 310–311.

[43] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to parallel computing*, The Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[44] C. Lanczos, *Solution of systems of linear equations by minimized iterations*, J. Research of the National Bureau of Standards, 46 (1952), pp. 33–53.

[45] G.-C. Lo and Y. Saad, *Iterative solution of general sparse linear systems on clusters of workstations*, Tech. Report, UMSI 96/117, Supercomputer Institute, Univ. Minnesota, 1200 S. Washington Ave., Minneapolis, MN 55415, August 1996.

[46] W. D. McQuain, R. C. Melville, C. J. Ribbens, and L. T. Watson, *Preconditioned iterative methods for sparse linear algebra problems arising in circuit simulation*, Comput. Math. Appl., 27 (1994), pp. 25–45.

[47] R. C. Melville, S. Moinian, P. Feldmann, and L. T. Watson, *Sframe: an efficient system for detailed DC simulation of bipolar analog integrated circuits using continuation methods*, Analog Integrated Circuits Signal Processing, 3 (1993), pp. 163–180.

[48] R. C. Melville, Lj. Trajković, S.-C. Fang, and L. T. Watson, *Artificial parameter homotopy methods for the DC operating point problem*, IEEE Trans. Computer-Aided Design, 12 (1993), pp. 861–877.

[49] Message Passing Interface Forum, *MPI: A message-passing standard*, Int'l J. of Supercomput. Applic., 8 (1994).

[50] D. M. Nicol and W. Mao, *On bottleneck partitioning of k-ary n-cubes*, Parallel Programming Letters, 6 (1996), pp. 389–399.

[51] D. A. Reed, L. M. Adams, and M. L. Patrick, *Stencils and problem partitionings: Their influence on the performance of multiple processor systems*, IEEE Trans. Comput., C-36 (1987), pp. 845–858.

[52] Y. Saad, *A flexible inner-outer preconditioned GMRES algorithm*, SIAM J. Sci. Comput., 14 (1993), pp. 461–469.

[53] Y. Saad, *Iterative methods for sparse linear systems*, PWS Publishing Company, 1996.

[54] Y. Saad and A. V. Malevsky, *P-SPARSLIB: A portable library of distributed memory sparse iterative solvers*, Tech. Rep., UMSI 95/180, Supercomputer Institute, Univ. Minnesota, 1200 S. Washington Ave., Minneapolis, MN 55415, September 1995.

[55] Y. Saad and M. H. Schultz, *GMRES: a generalized minimal residual method for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856 – 869.

[56] Y. Saad and K. Wu, *DQGMRES: a direct quasi-minimal residual algorithm based on incomplete orthogonalization*, Tech Report UMSI-93/131, Minnesota Supercomputer Institute, Univ. Minnesota, 1200 S. Washington Ave., Minneapolis, MN 55415, July 1993.

[57] L. F. Shampine and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[58] R. B. Sidje, *Alternatives for parallel Krylov subspace basis computation*, Numer. Linear Algebra Appl., 4 (1997), pp. 305–331.

[59] J. P. Singh, J. L. Hennessy, and A. Gupta, *Scaling parallel programs for multiprocessors: methodology and examples*, Computer, 7 (1993), pp. 42–50.

[60] M. Sosonkina, D. C. S. Allison, and L. T. Watson, *Scalability of an adaptive GMRES algorithm*, In Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, 1997.

[61] M. Sosonkina, L. T. Watson, and D. E. Stewart, *Note on the end game in homotopy zero curve tracking*, ACM Trans. Math. Software, 22 (1996), pp. 281–287.

[62] M. Sosonkina, L. T. Watson, R. K. Kapania, and H. F. Walker, *A new adaptive GMRES algorithm for achieving high accuracy*, Numer. Linear Algebra Appl., submitted.

[63] X.-H. Sun and D. T. Rover, *Scalability of parallel algorithm-machine combinations*, IEEE Trans. on Parallel and Distributed Systems, 5 (1994), pp. 599 – 612.

[64] X.-H. Sun and J. Zhu, *Performance prediction: a case study using a multi-ring KSR-1 machine*, NASA CR-195066 ICASE Rep. 95-24 , Institute Comp. Applic. Sci. Eng., Mail Stop 132C, NASA Langley Research Center Hampton, VA 23681-0001, April 1995.

[65] H. A. van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems*, SIAM J. Sci. Stat. Comput., 12 (1992), pp. 631–644.

[66] H. A. van der Vorst and C. Vuik, *The superlinear convergence behaviour of GMRES*, J. Comp. Appl. Math., 48 (1993), pp. 327–341.

[67] H. F. Walker, *Implementation of the GMRES method using Householder Transformations*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 152–163.

[68] H. F. Walker, private communication, 1996.

[69] L. T. Watson, *Globally convergent homotopy methods: a tutorial*, Appl. Math. Comput., 31BK (1989), pp. 369–396.

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

[71] 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.

[72] L. T. Watson, S. M. Holzer, and M. C. Hansen, *Tracking nonlinear equilibrium paths by a homotopy method*, Nonlinear Anal., 7 (1983), pp. 1271–1282.

[73] L. T. Watson, M. Sosonkina, R. C. Melville, A. P. Morgan, and H. F. Walker, *HOMPACK90: A suite of FORTRAN 90 codes for globally convergent homotopy algorithms*, ACM Trans. Math. Software., to appear.

[74] D. B. West, *Introduction to graph theory*, Prentice Hall, Upper Saddle River, NJ 07458, 1996.

[75] S. Wolfram, *Mathematica: A system for doing mathematics by computer*, Addison-Wesley, 2nd ed., 1991.

[76] T. Y. Yang, R. K. Kapania, and S. Saigal, *Accurate rigid-body modes representation for a nonlinear curved thin-shell element*, AIAA J., 27 (1989), pp. 211–218.

VITA.

Maria (Masha) Sosonkina Driver obtained her B.S. and M.S. degrees in Applied Mathematics and Theory of Programming from Kiev State University, Ukraine, in 1990 and 1992, respectively. A year later, she entered the graduate program at Virginia Tech to pursue a Ph.D. degree in Computer Science and Applications, which she completed in Fall, 1997. Her research interests include sparse matrix computations and high performance computing.