The Pennsylvania State University
The Graduate School
Department of Computer Science and Engineering

Multimethod Solvers: Algorithms, Applications And Software

A Thesis in
Computer Science and Engineering
by
Sanjukta Bhowmick

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2004

The thesis of Sanjukta Bhowmick was reviewed and approved* by the following:

*Signatures are on file in the Graduate School.

Abstract

The solution of large sparse linear systems is a fundamental problem in scientific computing. A variety of solution schemes are available reflecting a wide range of performance and quality trade-offs. The "best" solution method can vary across application domains and often even across different phases in a single application. As noted by Ern et al. "The impossibility of uniformly ranking linear system solvers in any order of effectiveness....is widely appreciated." In this thesis, we attempt to deliver the benefits of the variety of sparse linear solution techniques to the application community, by developing multimethod solvers, i.e. solvers that use more than one basic sparse solution scheme.

More specifically, the thesis concerns the development of two types of multimethod sparse linear solvers, namely, composite and adaptive solvers. We develop a composite solver to provide highly reliable solution with low memory requirements by applying a sequence of limited memory iterative solution schemes to the same linear system. We develop an adaptive solver to dynamically select a linear solution scheme to match changing linear system attributes and thus reduce the time required for linear system solution.

Our main contributions are the development and analysis of algorithms to construct composite and adaptive multimethod solvers, and the design and implementation of software for their automatic instantiation. A central result includes the development of an optimal composite solver with increased reliability, low memory requirements, and minimal worst case time, using a combinatorial framework. Another contribution is the formulation of heuristics to enable dynamic linear solution method selection. Finally, we design a software architecture for providing multimethod solver service to the application community. We implement and test our new schemes on two applications; our results demonstrate that our multimethod schemes can significantly improve the reliability of linear system solution while reducing total application time.

Table of Contents

List of Tables

List of Figures

Acknowledgments

First of all, I would like to thank my adviser Dr. Padma Raghavan for introducing me to my thesis problems and for her invaluable guidance throughout my graduate studies. She not only advised me on relevant scientific disciplines but also trained me in other important aspects for a future academic career such as technical writing and interaction with colleagues. She encouraged me to participate in many conferences and introduced me to many other researchers associated with the field.

I would also like to thank Dr. Lois Curfman McInnes and Dr. Boyana Norris who were our collaborators in many of our papers and my advisers during my internship at Argonne National Laboratory. Their advice and guidance has helped me to better understand concepts in scientific computing and develop my programming skills in scientific computing software.

I like to thank Dr. David Keyes and Dr. Dinesh Kaushik for their helpful insight to CFD applications. I used their simulation codes for my experiments. They also helped me with references to better understand and validate my experiments with previously existing results.

I would like to thank Dr. Paul Plassmann, Dr. Mahmut Kandemir and Dr. Lyle Long for being on my thesis committee and for their suggestions towards improving my thesis.

I would like to thank my colleague and friend Keita Teranishi for introducing me to Padma (who I think is the best adviser one could have). In my early days when I was a novice in scientific computing, Keita helped me understand many of the concepts. I also thank my other friends Ivana Veljkovich, Amitayu Das and Indrani Halder, who were a great help to me in the "postprocessing" stage when I commuted between New York and State College to finish my thesis.

Finally, I would like to thank my family for their constant support. When things looked really bad they always said that everything is going to be alright and thanks to my extensive circle of friends, family and well wishers it has!

Dedication

To Babaiya

Chapter 1

Introduction

Scientific computing concerns algorithms, analysis and software to enable the modeling and simulation of physical phenomena from many disciplines, such as heat flow, chemical combustion, atmospheric turbulence, etc. Such modeling and simulation applications require the solution of many core scientific computing problems, such as linear system solution, nonlinear system solution and eigenvalue computations. It is important to note that such core problems typically have several competing alternative solution schemes and the choice of a scheme can affect the performance of the overall application. Furthermore, for a given problem, it is often neither practical nor possible to determine a priori a single "best" solution scheme across applications. Applications from different domains may have vastly different requirements and problem characteristics. Even a single application may produce core problems with different attributes at different phases. We therefore conjecture that application performance could potentially be improved by using more than one solution scheme.

This thesis concerns method selection and composition techniques to improve the performance of large-scale modeling applications. We focus on developing "multimethod solvers" i.e., enhanced solvers that combine (or select from) more than one basic method for sparse linear system solution. Although our algorithms can be applied to different problems we chose to focus on sparse linear system solution because it forms a core computational problem in many modeling applications. In a large class of modeling applications such sparse linear solution can be the dominant cost. Consequently, by improving the performance of the sparse linear system solution component we also improve the performance of the overall simulation.

Consider, for example, modeling applications where nonlinear partial differential equations (PDEs) are solved using implicit or semi-implicit schemes with Newton's method [18, 32, 39]. The application time can be dominated by the time to solve sparse linear systems generated at each Newton (nonlinear) iteration. There are a large number of sparse linear solution methods from broad classes such as direct, iterative, multilevel-multigrid and domain decomposition [24, 29, 32, 43, 46, 47, 48]. Furthermore, variants of methods from all these classes can be combined through preconditioning to accelerate the convergence of iterative methods [7]. Each class of methods has different characteristics with respect to robustness, memory requirements, parallel scaling, speedup, etc. Additionally, the numerical properties of the linear systems may change as the nonlinear iterations progress. Thus a solver that reflected a good balance of accuracy and computational costs at a certain stage may fail to be suitable at a later stage. We therefore seek to develop multimethod schemes that can effectively leverage the strengths of the large number of available solution schemes to improve application performance.

This thesis concerns the development of *multimethod solvers* by composing and selecting sparse linear solvers with attributes that best match problem characteristics and application requirements. We focus on non-symmetric large-scale sparse linear systems arising from the solution of PDE-based applications. Our goal is to improve the performance of such applications by improving the performance of the dominant step, that of solving sparse linear systems.

We develop, implement and test (i) composite solvers to improve reliability with scalable memory and (ii) adaptive solvers to reduce execution time by dynamic method selection. Our schemes to construct both types of multimethod solvers are based on metrics such as computation costs, execution time, failure rates and convergence rates, i.e., metrics that can be obtained analytically or through experiments.

We define a composite solver [12, 13, 14, 15] as an ordered sequence of a set of basic solution schemes. The failure of one method in the sequence results in the invocation of the next method until either the system is solved successfully or all methods have been attempted without success. On practical problems, we expect that the reliability of the composite will be much higher than the reliability of any of its constituent methods. We thus aim to provide a highly robust solution with very limited memory overheads beyond those required for the coefficient matrix. Our contributions include: (i) developing composite solvers that provide highly robust linear system solution with scalable memory, (ii) constructing the optimal composite with minimum worst case running time by using combinatorial frameworks, and (iii) experimental evaluation of the performance of composite solvers in computational fluid dynamics (CFD) applications.

We also develop an adaptive solver [13, 37] to dynamically select the most appropriate linear solver that matches the properties of linear systems generated in the course of nonlinear iterations. In contrast to composite solvers, *only one* base solver is used per linear system. Our goal is to adapt linear solvers to better fit the changing linear system attributes to improve the execution time. Our contributions include: (i) developing of “sequence based” heuristics for dynamic method selection using metrics such as the convergence rates of the linear and nonlinear systems or the execution time of the linear solution scheme, (ii) developing of "non-sequence based" heuristics to be used when the change in the linear system characteristics is more complex, and (iii) experimental evaluation of the performance of adaptive solvers in computational fluid dynamics (CFD) applications.

Instantiating a multimethod solver is a complicated task and this thesis would not be complete without addressing this issue. Instantiation of multimethod solver requires the reuse of a variety of sparse solver implementations in different languages. An additional requirement is that the interface to the multimethod solver service should be easy to use for the application developer. The interface should not reveal lower level implementation details such as integration of the linear solvers into applications or how the solvers are invoked. We propose the design of a multimethod software architecture which can use sparse linear solvers from a large number of linear solver packages and provide an easy to use interface to application developers. We present two different implementations of this software architecture, a simpler model using PETSc (Portable Extensible Toolkit for Scientific Computing) [5, 6] and a more complex component-based model conforming to the currently emerging CCA (Common Component Architecture Forum) [4, 11] standards.

The remainder of the thesis is organized as follows. In Chapter 2, we provide background material on the solution of partial differential equations, Newton's method for nonlinear system solution, sparse linear solvers, and recent related research on the use of multiple linear solution methods. Chapters 3, 4, and 5 contain our main contributions. Chapter 3 concerns the development and analysis of composite solvers and includes an empirical evaluation of their performance in CFD applications on sequential and parallel architectures. Chapter 4 concerns the development of adaptive solver heuristics and an empirical study of their performance. Chapter 5 concerns the design of our multimethod solver and its implementation in software component framework. Chapter 6 contains concluding remarks and potential directions for future research.

Chapter 2

Background

In this chapter, we present a brief review of the background material. We begin by discussing partial differential equations and their numerical solution. We focus in particular on Newton's method for nonlinear system solution. We then present the software infrastructure for implementing these techniques. We then provide an overview of sparse linear solvers along broad categories such as direct, iterative, multilevel and domain decomposition solvers. In the final section of this chapter, we present a review of related research that use more than one linear solution method to solve linear systems.

2.1 The Numerical Solution of Partial Differential Equations

Most physical phenomena evolve continuously in time and space and their mathematical models can often be described by partial differential equations (PDEs) involving the partial derivatives of several independent variables [32, 43]. Of particular interest are second order PDEs because the mathematical models of many applications fall in this category.

Some partial differential equations can be solved analytically by obtaining the series or integral representation for the continuous function. However, such analytical solutions, while providing valuable insight, are typically feasible only for simpler models. Consequently most applications with PDE based models are solved numerically by discretizing the equations to generate an approximate solution. The standard techniques for discretizing PDEs are (i) finite difference, (ii) finite elements and (iii) finite volume [10, 18, 26, 32, 42, 43]. The discretization of partial differential equations from such schemes gives rise to systems of linear or nonlinear equations which are then numerically solved.

2.1.1 The Solution of Nonlinear Systems

A nonlinear system of m equations with n unknowns has the form $f(x) = 0$, where $f : R^n \rightarrow R^m$. The root of the system of nonlinear equations is the vector x for which all component functions of f are zero simultaneously [32]. Systems of nonlinear equations are often solved using the Newton's method [18, 32, 43] and its variations.

We first consider Newton's method applied to solve $f(x) = 0$, where x is scalar and f is a scalar function. Newton's method starts with an initial guess, x_0 and at each iteration k finds the approximate value of the function at x_k. The value of $f(x_k)$ and its derivative is used to calculate the new estimate of x, i.e., the value of x_{k+1}.

The value of x_{k+1} is calculated as follows. The truncated Taylor series expansion gives, $f(x_k + h) = f(x_k) + f'(x_k)h$, where $x_{k+1} = x_k + h$. If x_{k+1} was the root of the nonlinear equation, then $f(x_k + h) = 0$ and $h = -\frac{f(x)}{f'(x)}$, assuming $f'(x) \neq 0$. This value of h is used to calculate the next approximation of x. This can be expressed as the iteration, $x_{k+1} \leftarrow x_k - f(x_k)/f'(x_k)$. The iterations are continued until a sufficiently good approximation is obtained (as determined by a tolerance specified by the user) [18, 32, 43]. Geometrically Newton's method is equivalent to drawing a tangent through the point $(x_k, f(x_k))$. The intersection of the tangent with the x-axis gives the new estimate.

Newton's method can be generalized to cases where x is a vector and f is a vector function. Now, $f(x)$ represents a system of equations and the corresponding Taylor's expansion is $f(x+h) = f(x) + J(x)h$, where $J(x)_{ij} = \partial f_i(x)/\partial x_j$ is the Jacobian matrix of f. At each iteration the linear system of equations given by $J(x_k)h = -f(x_k)$ is solved to obtain h which is then used to update x as $x_{k+1} \leftarrow x_k + h$ [18, 32, 39, 43].

Newton's method shows quadratic convergence for a simple root but it might not converge if the initial guess is far away from the solution [32]. This method is also expensive because it involves recalculating the Jacobian matrix and solving the associated linear system at each iteration. The calculation costs can be mitigated by using variations of the basic algorithm. Inexact Newton methods [30, 39] solve the system nonlinear equations $f(x) = 0$, by finding an approximate solution to $J(x_k)h = -f(x_k)$, such that $f(x_k) + J(x_k)h \leq \eta_k r_k$ with $(0 < \eta_k < 1$ for all $k)$. The Jacobian can also be approximated by premultiplying the linear system with a matrix B, where B^{-1} approximates the action of the Jacobian, at a smaller cost. The value of the next iterate is then obtained as follows, $x_{k+1} = x_k + \alpha h_k$; $0 < \alpha \leq 1$.

2.2 Software for Scientific Computing

In this section we will discuss some of the advanced software systems that are used to implement scientific computing applications. We will discuss PETSc [5, 6], a problem solving environment for scientific computing and the Common Component Architecture [4, 11], the emerging standard for scientific computing components.

PETSc: A Portable Extensible Toolkit for Scientific Computing The Portable Extensible Toolkit for Scientific Computing (PETSc) [5, 6], developed at the Argonne National Laboratory, consists of a suite of data structures and functions for the scalable solution of scientific applications. PETSc is composed of hierarchical libraries. These include low-level data structures for representing objects such as mesh, matrix, vector, and these data structures are subsequently integrated to form higher level libraries for implementing linear and nonlinear solvers. The higher level abstractions facilitate better understanding and reuse of individual code segments. The design of PETSc also introduces architecturally important features such as re-organizing code for strong cache locality and preallocation of memory. PETSc also includes functions and data structures to gather a limited set of performance data without significant runtime overheads.

Figure 2.1 shows the calling sequence of a typical PETSc nonlinear solver. The application driver performs I/O for initialization, restart and post processing. The application driver also calls routines to create data structures for matrices and vectors and also initiates the nonlinear solver. The nonlinear solver routine invokes the linear solver which is then executed. An iterative solver routine would further call the specified Krylov subspace solver and preconditioner.

Component-Based Software Architectures Current application codes are no longer relying on a single software package. Use of a single package generally leads to a monolithic code which gets unwieldy with each new addition. There is often no efficient mechanism for auxiliary functions such as collection and storage of performance data. Scientific computing codes therefore are veering towards a modular structure where different numerical or database related software are integrated into a single application. The software codes can potentially differ in the language they are implemented, their supporting architecture, etc. The challenge lies in effectively integrating the codes by bridging these differences.

Component based software design tries to overcome the problems occurring from variations of languages, libraries and architectures, by incorporating features such as object-oriented code, language interoperability and the ability for dynamic composition [4, 11, 40]. When codes are written in different languages, combining them to form an integrated application software poses a serious challenge. A software component architecture defines a set of application-level software components, their structural relationships, and their behavioral dependencies. Component based architectures may thus

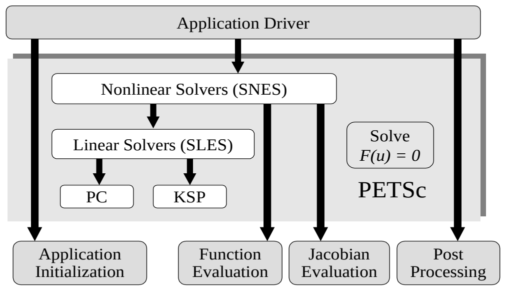


Fig. 2.1. A schematic diagram of the calling sequence of a PETSc Nonlinear Solver (Source: PETSc 2.1.3 tutorial).

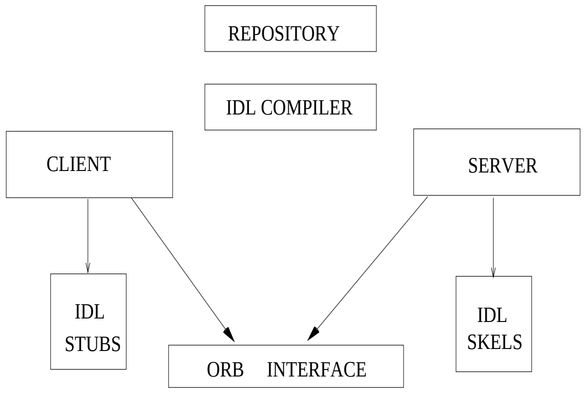


Fig. 2.2. CORBA Architecture (Source: Corba 3: Fundamentals and Programming).

allow the user to "plug-and-play" with different components when creating a complex software. A well known example is the Common Object Request Broker Architecture (CORBA) [45]. In this model, the users (called clients) request services from servers through an interface specified by an IDL (Interface Definition Language). A client can access an object by issuing a request. The central component of CORBA is the Object Request Broker (ORB). It encompasses all of the communication infrastructure necessary to identify and locate objects, handle connection management and deliver data [45].

Scientific computing however, deals generally with large-scale applications and it is a common practice to use parallel algorithms. Parallelization involving large scale computations in an heterogeneous multiprocessor environment lead to complex problems, such as effective partitioning of data, communication between processors, etc., which are not encountered in sequential codes. The Common Component Architecture Forum (CCA) [4, 11], specifically aims to develop a scientific component model while remaining compatible with other component architectures.

The Common Component Architecture (CCA), an extension of component-based models like CORBA, is designed to support large-scale high-performance software on parallel, distributed, and hybrid frameworks [4, 11]. The components are defined through the Scientific Interface Definition Language (SIDL). The SIDL files are processed by Babel [19], a SIDL based language interoperability tool, to generate components for the specified software library in the specified language. Components form the basic units of CCA and are integrated into the larger applications. The application views each component as a black-box and interacts with components via abstract interfaces called ports in a special runtime system. The *uses ports* specifies a functionality required by the component and the *provides ports* specifies a functionality implemented by the components and to be used through the ports of other components. Components are connected through a framework, such as CCAFFEINE. The framework connects matching uses and provides ports, without going into the implementation details of either component. The framework also provides a standard set of services available to all components. These services are cast as ports for uniformity.

2.3 Sparse Linear Systems and their Solution

Linear systems are of the form $Ax = b$, where A is a $m \times n$ coefficient matrix, b is a known vector of length m and x is an unknown vector of length n. We limit our discussions to linear systems associated with square $n \times n$ coefficient matrices with full rank having a unique solution. Linear systems are sparse when A has very few nonzeros cn, where c is a small constant. Most sparse matrices generated from discretization of PDEs on a spatial domain are structured, that is they exhibit a pattern of the non-zero elements.

2.3.1 Direct Methods for Sparse Linear Systems

Direct methods [24] involve factoring the coefficient matrix A into an upper triangular matrix (U) and lower triangular matrix (L), such that $A = LU$. Now, the system of linear equations can then be solved by solving the following two triangular systems in order, $Ly = b$ (by forward substitution)and $Ux = y$ (by backward substitution). If A is symmetric positive definite (SPD) then a symmetric format of LU factorization exists such that $U = L^T$ and $A = L^T L$. This is known as *Cholesky factorization* [29, 32, 43, 47, 48].

Sparse linear factorization is associated with an intrinsic problem of fill-in. A fill-in occurs when a zero element in the coefficient matrix is transformed into non-zero as a result of factorization. Fill-in depends on the structure of the matrix rather than the values. It has been proved that the minimization of fill-in is an NP-complete problem [49]. However, fill-in can be reduced by reordering the rows and columns of the matrices. The minimum degree heuristic [22], a greedy strategy for minimizing fill-in, starts with the column that will potentially incur the least fill in. Divide and conquer strategies include nested dissection, class of ordering schemes based on vertex separators. This is one of the most successful ordering heuristics. Factoring matrix represented by a planar graph with n nodes requires $O(n^{3/2})$ time and $O(nlogn)$ memory [36].

Direct methods can compute the solution in a finite number of steps and guarantee a solution if one exists. However, this reliability comes at the cost of extra memory due to fill-in. Direct methods are also difficult to parallelize as factorization involves extensive communication between the columns in the matrix.

2.3.2 Iterative Methods for Sparse Linear Systems

Iterative methods start with an initial estimate of the solution x_0 and refine it at each iteration. In practice the iterations are continued till a given criterion is achieved; for example limits may be placed on the number of iterations, or when the error is sufficiently low as given by the norm of the residual $b - Ax$. Iterative methods retain the sparsity structure and require very little extra memory in addition to that needed by the coefficient matrix. However, for certain problem instances, the convergence might be slow or the method might not converge at all. It is common practice to use preconditioners to change the numerical characteristics of the linear system to be solved and thereby improve the reliability of the method. In this section, we will concentrate on two classes of iterative methods, based on (i) Fixed Point iterations and (ii) projections on to Krylov subspaces [29, 32, 43, 48].

2.3.2.1 Fixed Point Methods

Fixed point iteration schemes transform the linear equation $Ax = b$ into the form $x_{k+1} = M^{-1}Nx_k + M^{-1}b$, where $A = M - N$. Some fixed point iterative schemes include,

$M = D; N = -(L + U)$ (Jacobi)

$M = (D + L); N = -U$ (Gauss-Seidel (GS))

$M = \frac{1}{\omega}D + L; N = (\frac{1}{\omega} - 1)D - U$ (Successive Over Relaxation (SOR))

L, U and D are respectively the lower triangular, upper triangular and diagonal parts of A and ω ranges from 0 to 2. The convergence of fixed point methods depend on the spectral properties of the matrix G; the method converges if the *spectral radius* of G, $\rho(G)$, is less than one. The convergence of SOR depends on the value of ω and its convergence for an optimal ω is an order of magnitude higher than Gauss-Seidel and Jacobi [32].

Block relaxation schemes are a generalized form of fixed-point iterative solvers. They involve updating components of the vectors instead of each single entity. The matrix A is divided into blocks and the corresponding vectors are updated accordingly [43].

2.3.2.2 Krylov Subspace Methods

Krylov subspace methods are based on projections onto Krylov subspaces. A Krylov subspace of dimension m for a matrix A, and an associated vector b is defined by $K_m(A, b) \equiv span\{b, Ab, A^2b, \ldots, A^{m-1}b\}$.

A set of orthogonal vectors $\{q_1, q_2, \ldots q_m\}$ that span the same space as $K_m(A, b)$ can be built using the Arnoldi Method [29, 43, 47, 48]. This method is based on reducing the matrix A to a Hessenberg matrix H, such that $A = QHQ^T$, where $Q \equiv \{q_1, q_2, \ldots q_n\}$ is an unitary matrix; $AQ = QH$. The Lanczos method [29, 43, 47, 48] is the symmetric equivalent of the Arnoldi method that reduces a symmetric matrix A to a tridiagonal matrix T, such that $AQ = QT$. Lanczos Biorthogonalization, a modification of the Lanczos algorithm, reduces a non-symmetric matrix to a tridiagonal matrix. Lanczos biorthogonalization [29, 43] builds Krylov spaces, $K_i(A, v_1)$ and $K_i(A^T, w_1)$. The corresponding vector sets are $V_i \equiv \{v_1, v_2, \ldots, v_i\}$ and $W_i \equiv \{w_1, w_2, \ldots, w_i\}$. The matrices V_i and W_i are not unitary, but the columns of V_i are orthogonal to those of W_i.

The Generalized Minimal Residuals(GMRES) [29, 43, 47, 48] linear solution technique is based on projection of vector b onto the Krylov space $K_i(A, b)$ and orthogonal to $AK_i(A,b)$. The approximate solution x_i is the vector that minimizes the residual $r_i = b - Ax_i$. Expressing $x_i = x_0 + Q_i y$, the corresponding least squares problem is, $\|b - Ax_i\|_2 = \|\beta e_1 - H_i y\|_2$, where $\beta = \|r_0\|_2$, $AQ_i = Q_i H_i$, and the columns of Q_i span the same subspace as $K_i(A,b)$. The GMRES algorithm computes y that minimizes the corresponding least square expression and then calculates x_i from the obtained y. The approximate solution is improved for higher values of i, i.e. for a larger set of orthogonal vectors. For matrices of high ranks, the expenses of storing the entire subspace is prohibitive. This limitation is overcome by restarted GMRES or GMRES(k) where Arnoldi iterations for creating the Krylov subspace are restarted after k iterations, setting $x_0 = x_k$ [43].

The Conjugate Gradient(CG) method [29, 43, 47, 48] is essentially a Lanczos scheme to reduce SPD matrices into a tridiagonal form. Since the reduced matrix is tridiagonal, the values of only three consecutive orthogonal vectors need to be saved and no restart value is required. Mathematically this linear solution algorithm is similar to GMRES, solving a least square problem and is the method of choice for solving linear systems when a is SPD. The approximate value of x at the ith iteration is obtained by $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$, where α_{i-1} is a scalar and p_{i-1} represents a new search direction.

The Biconjugate Gradients(BiCG) method [29, 43, 47, 48] uses biorthogonalization to reduce the co-efficient matrix. This method is also based on finding the least square solution. The vector b is projected onto $K_i(A, v_1)$ and orthogonal to $K_i(A^T, w_1)$. The approximate value of x_i can be calculated from the previous value x_{i-1} in a process similar to CG by specifying a new search direction p_{i-1}. Since two sets of mutually orthogonal vectors V and W are used, BiCG can simultaneously solve the dual problem $A^T x^* = b^*$ [43]. The transpose free variant of this algorithm is BiCGSTAB which avoids the calculations with A^T.

The Quasi-Minimal Residual (QMR) method [29, 43, 47, 48] for solving linear systems is based on reduction by biorthogonalization. In the i-th reduction step $AV_i = V_{i+1}\bar{T}_i$, where $\bar{T}_i = (T_i, \delta_{i+1} e_i^T)$ and $\delta_i = T(i, i-1)$. The residual norm is of the form, $\|b - Ax\|_2 = \|V_{i+1}(\beta e1 - \bar{T}_i y)\|_2$, where $\beta = \|r_0\|_2$. In a procedure similar to GMRES the value of y that minimizes the equation above is calculated to obtain the value of x_i. The transpose free variant of QMR is TFQMR [28].

2.3.3 Multigrid Methods

The residual, $r = Ax - b$, can be considered to be composed of high-frequency (oscillatory) and low-frequency(smooth) components. Fixed point iterative methods reduce the high frequency error rapidly, but are slow to reduce the low frequency components. It has been observed that a low frequency component becomes one with a high frequency in a coarser grid. This is the motivating principle for multigrid methods [32, 43].

The basic steps of a multigrid method are as follows. On the original mesh, the system $Ax = b$ is approximately solved for a few iterations. This operation is known as smoothing and different iterative methods can be used as smoothers. The residual r is computed and restricted to a coarser grid. The equation $Ae = r$ is solved on the coarser grid to get an approximation of the error, e. This value is then interpolated back to the finer mesh to obtain an improved approximation of x. The number of levels of coarsening varies according to the application requirements or user specified parameters. The transition from finer to coarse grids and back to finer grids is known as a cycle. Different cycling strategies such as V-cycle, W-cycle, Full multigrid, can be used [29, 32, 43].

2.3.4 Domain Decomposition Methods

Domain decomposition methods are often used for solving large-scale PDE-based models on two and three dimensional domains. These methods employ a divide and conquer strategy [43, 46]. The basic principle of domain decomposition is to divide the solution space Ω into s sub-domains Ω_i, such that $\Omega = \bigcup_{i=1}^{s} \Omega_i$ and solve the problem on the entire domain by utilizing solutions from the sub-domains. There are many variations of domain decomposition based on the type of partitioning, amount of overlap, accuracy of the sub-domain solutions, etc. Domain decomposition methods can be classified into direct (using the Schur Complement) and iterative (Schwarz alternating procedures) methods. Schwarz methods are used for solving systems with overlapping sub-domains. The values at the boundary points of each domain are updated based on their solution in the neighboring domains. The updates can be done after an entire iteration cycle through all domains is completed (Additive Schwarz Procedure). This is analogous to the block Jacobi iteration. Another choice is to perform the updates after solving the system in each domain (Multiplicative Schwarz Procedure). This is analogous to block Gauss-Seidel iteration.

2.4 Preconditioning

Preconditioning [7, 29, 32, 43, 47, 48] is used to modify the matrix A so that the numerical characteristics of the new matrix will lead to faster convergence when a KSP iterative method is applied. The linear system to be solved is transformed to $M^{-1}Ax = M^{-1}b$, where M^{-1} is the mapping representing the preconditioner. Use of preconditioners entail additional computation costs and thus there exists a trade-off between cost of using the preconditioner and the increase in convergence rate.

Some of the simpler preconditioners are based on fixed-point iterative methods, like the Jacobi preconditioner and the Symmetric Successive Over Relaxation (SSOR) [7, 29, 32, 43, 47]. *Block Jacobi* [43] preconditioning is a generalization of the Jacobi preconditioner and uses block diagonal elements.

Preconditioners can also be obtained from the incomplete factorization of the coefficient matrix. Incomplete LU (ILU) [7, 29, 43] factorization generates the triangular matrices $\hat{L}$ and $\hat{U}$, where some of the new nonzero values are ignored according to a given criteria [43]. ILU factorization with threshold ILUT(τ) [7, 29, 43], is a method where elements whose values are less than the threshold τ are eliminated from the factors L and U.

Preconditioners built from incomplete factorizations are difficult to parallelize. Sparse Approximate Inverse is a more scalable form of generating preconditioners. This class of preconditioners are based on calculating M, M is approximately equal to A^{-1}, by minimizing the Frobenius norm of $I - AM$ [7, 29, 43].

2.5 Towards Using Multiple Linear Solution Schemes

As observed in the last two sections, there exist several competing methods for solving sparse linear systems representing improvements in the execution time, the memory required, the robustness of the solver, etc. For linear systems arising in the course of solving a modeling application, it is often difficult to predict the single best linear solver to be used. This has motivated us and other research groups to design algorithms that use more than one linear solver to solve a given problem. However, until now, the research in this area relied on the user to experimentally combine solvers from a predetermined set. Our approach differs significantly in formulating algorithms to automatically construct multimethod solvers. The rest of the section describes related research that advocate or use multiple linear solvers.

2.5.1 Algorithmic Bombardment for the Iterative Solution of Linear Systems: A PolyIterative Approach [8]

In this paper Barrett et al. define and implement a “polyiterative” linear solver. A polyiterative linear solver consists of a set of iterative linear solvers that are applied simultaneously to the same system. The authors describe a parallel implementation of the polyiterative approach as follows.

The linear system is distributed among the nodes of the multiprocessor system. The following variants of the conjugate gradient method; BiCGSTAB, QMR and CGS are used. The operations required by these iterative methods are of two classes; (i) those that do not need any global communication and (ii) those requiring a communication stage. Operations, such as vector updates, that do not require any communication are performed in each processor on its local data. These operations differ according to the method used and are performed in sequence for each method in each processor. The communications across processors is however not done sequentially for each method. Global communication is optimized by aligning the linear solution methods at the communication stages and packing all the data required by the different methods in a single buffer. Though the data for all methods is sent together, the methods do not share data amongst themselves as this can adversely affect the Krylov subspace built for each of the methods [8]. In course of these computations if a solver fails to converge, it is removed from the set. The poly-iterative process is terminated when convergence is achieved by any one method.

The paper reports the solution of a 2D-Poisson's equation using the following variants of the conjugate gradient method; BiCGSTAB, QMR and CGS. The authors solve the problem on different grid sizes to demonstrate that the most effective solver (one which gives the most accurate solution in the least time) varies as the grid sizes are modified. The suite of experiments in [8] is an example that motivates the use of multiple algorithms in solving variants of the same problem. The main advantage of the polyiterative approach is that all likely linear solvers can be tried at the same time. However, the execution time is increased since the methods are aligned for each communication. This results in the execution time of an iteration being at least as much as that of the slowest solver.

2.5.2 Towards Polyalgorithmic Linear System Solvers for Nonlinear Elliptic Problems [25]

In this paper, the authors Ern et al. investigate the performance of linear solvers while solving a nonlinear elliptic problem representing flame sheet simulation. The simulation is divided into three phases; the initial, medium transient phases and the final steady phase. The linear solution methods used in the experiments were SOR with $\omega = .85$, BiCGSTAB and GMRES(20). The last two methods were preconditioned using block line Gauss Seidel(GS), symmetric block-line Gauss Seidel(SGS) and block incomplete LU decomposition(ILU).

Experiments on a 41×41 grid showed that BiCGSTAB and GMRES each with GS preconditioner required the least execution time and BiCGSTAB required less memory than GMRES. At the initial transition stages the performance of SOR was comparable to the Krylov methods and the method required less memory, but at the steady state the relaxation factor had to be switched to 0.3 to achieve convergence.

Experiments on a 81×81 grid, showed that once again the best methods were Krylov solvers with GS preconditioner. The memory requirement of GMRES depends on the restart value. A smaller restart value can ameliorate storage problems, but this might lead to stagnation. The relaxation method SOR performed favorably in the the initial transition phases, but was not competitive in the steady phase.

These results illustrate the factors that should be considered while choosing a solver. Time is of course a primary issue, but as seen in the case of Krylov methods storage space is also an important concern. These experiments also demonstrate that the performance of the solvers vary at different stages of the simulation.

2.5.3 The Linear System Analyzer [16]

The Linear System Analyzer (LSA) developed by Bramley et al. is a component-based problem-solving environment where the user can simply use a specific combination of preconditioner and linear solver without being required to know the details of the implementation.

From the user's perspective, the linear system analyzer is viewed as a GUI which shows a list of machines and components available on them. The user can choose solvers or other components on a specific machine by clicking a button and selecting a machine. Then the LSA starts the execution of the component on the machine specified.

The main server of the Linear System Analyzer comprises, (i) *the User Control*, a GUI interface to the end user, (ii) *the LSA Manager* which deals with resource management and the construction of a component network, (iii) *the Communication Subsystem* which supports communication between processors during parallel applications, and (iv) *the Information Subsystem* which provides information about the solution process and returns a summary of results and the performance metrics.

Details on the implementation and behavior of LSA can be found in [16]. The software advocates the use of components, an implementation strategy that is gaining prominence in recent scientific computing applications. LSA also provides the facility for simultaneously experimenting with multiple linear solvers by running them on multiple machines. However, LSA requires the user to specify the linear solvers needed and does not have a mechanism to automatically select or combine linear solution methods.

2.5.4 Self-Adapting Numerical Software [23]

Self-Adapting Numerical Software (SANS) by Dongarra et al. aims for "successful management of the complex grid environment while delivering the full power of available algorithmic alternatives" [23]. The software is composed of the SANS agent for automating algorithm selection and management of computational resources. A metadata vocabulary implemented in XML, is used to store information on different data and algorithms in an associated database. SANS components also include a runtime adaptive scheduler and prototype libraries to optimize performance on different architectures. The implementation of SANS encapsulates several interesting problems such as, selection of the most appropriate method for the problem, efficient storage and retrieval of data, effective scheduling policies, heuristics for compromising between trade-offs, etc. The design of SANS has been proposed and its development is still in its early stages. The current work includes designing a database to collect performance metrics of a variety of linear solvers and use of statistical and data-mining techniques to select the appropriate solver as required.

Chapter 3

Composite Solvers

In this chapter, we will define and develop a class of multimethod solvers that we call "*Composite Solvers.*" We develop a combinatorial scheme for constructing composite solvers and report on their performance in CFD applications. Some of these results were reported earlier in a journal article and conference proceedings [12, 13, 14, 15].

A composite solver is designed to provide a highly reliable sparse linear system solution scheme with limited memory requirements. A composite comprises a set of basic preconditioned iterative sparse linear solution methods that we henceforth call "base methods". A composite solver is defined by a predetermined sequence of these base methods. Initially, the first method in the sequence is used to solve the given linear system. If convergence is not achieved then the next method in the sequence is used. This process is continued until the solution is obtained or all methods in the sequence are exhausted. We thus attempt to produce composite solvers that are highly robust while using significantly less memory than a sparse direct solver.

Each base method is associated with a performance metric, generally taken to be the execution time, and a reliability/failure metric. The failure rate of a base method is the probability that the method would not converge. We define the reliability as 1-failure rate. The cumulative failure rate of the composite is calculated as the product of individual failure rates and the cumulative reliability is 1-cumulative failure rate. This value of the composite is independent of the ordering of the methods in the sequence. The cumulative execution time of the composite, however, does depend on the ordering. This is because in a composite solver a method is applied only if the preceding methods have failed. Therefore the cumulative execution time is calculated as the sum of the execution time of the individual methods weighted by the cumulative failure rates of the methods preceding it in the sequence.

Consider an example with two base methods M_1 and M_2 as shown in Figure 3.1. Methods M_1 and M_2 are associated with execution times $t_1 = 2$ and $t_2 = 3$ and reliabilities $r_1 = .02$ and $r_2 = .80$ respectively. Two different composites $M_1; M_2$ and $M_2; M_1$ can be obtained. The execution time of the composite represented by the first sequence is thus 4.94, whereas that of the second composite is 3.04. The reliabilities of both the composites are same, .984, which is much higher than the reliabilities of the individual methods. Our goal is to construct an optimal composite with minimal execution time while retaining the properties of high reliability and low memory requirements.

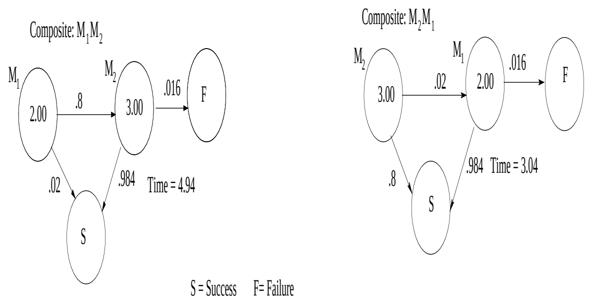


Fig. 3.1. Composites of two methods $M_1, t_1 = 2.0, r_1 = .02$ and $M_2, t_2 = 3.0, r_2 = .80$; both composites have reliability .984 but the composite M_2M_1 has lower execution time.

3.1 A Mathematical Representation

Consider a working set of n distinct methods $M_1, M_2, \ldots, M_n$. Each method M_i, is associated with its normalized execution time t_i (performance metric) and failure rate is f_i; and its reliability is given by $r_i = 1 - f_i$. Let $\mathbf{P}$ represent the set of all permutations (of length n) of $\{1, 2, \ldots, n\}$. For a specific $\hat{P} \in \mathbf{P}$, the associated composite is denoted by $\hat{C}$, that is the permutation $\hat{P}$ defines the sequence of the methods $M_1, M_2, \ldots, M_n$ in the composite $\hat{C}$. If $\hat{P}_k$ denotes the kth element of $\hat{P}$, the composite $\hat{C}$ consists of methods $M_{\hat{P}_1}, M_{\hat{P}_2}, \cdots, M_{\hat{P}_n}$. For a set of methods whose failure rates, and consequently reliabilities, are mutually independent, the total reliability of any composite $\hat{C}$ is $1 - \Pi_{i=1}^{i=n}(f_i)$. This value is independent of the ordering and has a higher value than the reliability of any single base solver. The worst case execution time, $\hat{T}$, of $\hat{C}$ occurs when all the methods of the sequence have to be considered. The value of $\hat{T}$ is:

$$\hat{T} = t_{\hat{P}_1} + f_{\hat{P}_1} t_{\hat{P}_2} + \cdots + f_{\hat{P}_1} f_{\hat{P}_2} \cdots f_{\hat{P}_{n-1}} t_{\hat{P}_n}.$$

We define the optimal composite to be the composite with minimal worst-case execution time.

A subsequence of the form $\hat{P}_k, \hat{P}_{k+1}, \cdots, \hat{P}_l$ ($\hat{P} \in \mathbf{P}$) is denoted by $\hat{P}_{(k:l)}$ and can be associated with a composite comprising $l - k + 1$ methods using the notation $\hat{C}_{(k:l)}$. The total reliability of $\hat{C}_{(k:l)}$ is $\hat{R}_{(k:l)} = 1 - \prod_{i=k}^{i=l} f_{\hat{P}_i}$ and the cumulative failure is $\hat{F}_{(k:l)} = \prod_{i=k}^{i=l} f_{\hat{P}_i}$. As observed previously, both these quantities depend only on the underlying set of methods specified by $\hat{P}_k, \hat{P}_{k+1}, \cdots, \hat{P}_l$ and are invariant under all permutations of these methods. The worst case time of $\hat{C}_{(k:l)}$ is defined as $\hat{T}_{(k:l)} = \sum_{i=k}^{i=l}[t_{\hat{P}_i} \prod_{m=k}^{i=l-1} f_{\hat{P}_m}]$. A final term we introduce is the *utility ratio* which is the ratio of the execution time to the reliability. For a method M_i the utility ratio $u_i = t_i/r_i$. By generalizing, we define the total utility ratio of $\hat{C}_{(k:l)}$ as $\hat{U}_{(k:l)} = \hat{T}_{(k:l)}/\hat{R}_{(k:l)}$.

Henceforth we will not explicitly reference $\hat{P}_i$ in expressions for $\hat{R}$, $\hat{F}$, $\hat{T}$, and $\hat{U}$ for a specific $\hat{C}$ and $\hat{P}$. Thus the expression for $\hat{T}_{(k:l)}$ simplifies to $\sum_{i=k}^{i=l}[t_i \prod_{m=k}^{i=l-1} f_m]$. Additionally, in an attempt to make the notation consistent, we will treat $\hat{C}_{(k:k)}$ specified by $\hat{P}_{(k:k)}$ as a (trivial) composite of one method and use related expressions such as $\hat{T}_{(k:k)}$, $\hat{R}_{(k:k)}$, $\hat{F}_{(k:k)}$ $\hat{U}_{(k:k)}$ (where $t_k = \hat{T}_{(k:k)}$,$r_k = \hat{R}_{(k:k)}$, $f_k = \hat{F}_{(k:k)}$, and $u_k = \hat{U}_{(k:k)}$).

3.2 Analytical Results

At first glance, it might seem that the optimal composite can be obtained by arranging the methods either in increasing order of time, or in decreasing order of reliability. However our analysis shows that neither of these strategies produce the optimal composite. We will proceed to show that a composite is optimal if and only if its underlying methods are in increasing order of the utility ratio.

3.2.1 Constructing an Optimal Composite with Mutually Independent Failure Rates

We will first consider composites formed from a set of methods with mutually independent failure rates, i.e., the failure of one method does not depend on the failure of another. We begin by observing that for any $\hat{P} \in \mathbf{P}$ the execution of the composite $\hat{C}$, can be mimicked by subsequent execution of two composites, $\hat{C}_{(1:r)}$ and $\hat{C}_{(r+1:n)}$. Consequently $\hat{T}_{(1:n)} = \hat{T}_{(1:r)} + \hat{F}_{(1:r)}\hat{T}_{(r+1:n)}$.

Lemma 3.1. *For any $\hat{P} \in \mathbf{P}$, the utility ratio of the composite $\hat{C}$ satisfies $\hat{U} \leq \max_{i=1}^{i=n} \hat{U}_{(i:i)}$.*

Proof: It is easy to see that the statement is true for the base case consisting two methods ($n = 2$). As the inductive hypothesis, we assume that the statement is true for any composite of $n-1$ methods, that is, $\hat{U}_{(1:n-1)} \leq \max_{i=1}^{i=n-1} \hat{U}_{(i:i)}$. Consider $\hat{C}$, a composite of n methods associated with the permutation $\hat{P}$, as a composite of *two methods* with execution times $\hat{T}_{(1:n-1)}$ and $\hat{T}_{(n:n)}$, reliabilities $\hat{R}_{(1:n-1)}$ and $\hat{R}_{(n:n)}$, and utility ratios $\hat{U}_{(1:n-1)}$ and $\hat{U}_{(n:n)}$.

If $\hat{U}_{(1:n-1)}$ is less than $\hat{U}_{(n:n)}$, then by the base case, $\hat{U}_{(1:n)}$ is less than $\hat{U}_{(n:n)}$. We also have $\hat{U}_{(n:n)} \leq \max_{i=1}^{i=n} \hat{U}_{(i:i)}$. By transitivity we can conclude that, $\hat{U}_{(1:n)} \leq \max_{i=1}^{i=n} \hat{U}_{(i:i)}$.

On the other hand if $\hat{U}_{(n:n)}$ is less than $\hat{U}_{(1:n-1)}$, then $\hat{U}_{(1:n-1)} \leq \max_{i=1}^{i=n} \hat{U}_{(i:i)}$, and by the base case $\hat{U}_{(1:n)}$ is less than $\hat{U}_{(1:n-1)}$. Therefore by transitivity $\hat{U}_{(1:n)} \leq \max_{i=1}^{i=n} \hat{U}_{(i:i)}$ $\square$.

Theorem 3.1. *Let $\tilde{C}$ be the composite given by the sequence $\tilde{P} \in \mathbf{P}$. If $\tilde{U}_{(1:1)} \leq \tilde{U}_{(2:2)} \leq \ldots \leq \tilde{U}_{(n:n)}$, then $\tilde{C}$ is the optimal composite, i.e., $\tilde{T} = \min\{\hat{T} : \hat{P} \in \mathbf{P}\}$.*

Proof: The base case would consist of a composite of two methods, and we can easily see that in this case the statement is indeed true.

We next assume that the statement is true for composites of $n-1$ methods. The optimal composite of $n-1$ methods is now extended to include the last method; and let this be represented by the sequence $\tilde{P}$ and the composite by $\tilde{C}$. For the sake of contradiction, let there be a permutation $\acute{P} \in \mathbf{P}$, such that $\acute{T} \leq \tilde{T}$ and the utility ratios $\{\acute{U}_{(i:i)} : 1 \leq i \leq n\}$ are not in increasing order of magnitude.

Let the k-th method in $\acute{C}$ be the n-th method in $\tilde{C}$. Therefore $\acute{T}_{(k:k)} = \tilde{T}_{(n:n)}$ and $\acute{F}_{(k:k)} = \tilde{F}_{(n:n)}$. Using the earlier observations:

$$\tilde{T} = \tilde{T}_{(1:k)} + \tilde{F}_{(1:k)}\tilde{T}_{(k+1:n-1)} + \tilde{F}_{(1:k)}\tilde{F}_{(k+1:n-1)}\tilde{T}_{(n:n)} \tag{3.1}$$

$$\begin{aligned}\acute{T} &= \acute{T}_{(1:k-1)} + \acute{F}_{(1:k-1)}\acute{T}_{(k:k)} + \acute{F}_{(1:k-1)}\acute{F}_{(k:k)}\acute{T}_{(k+1:n)} \\ &= \acute{T}_{(1:k-1)} + \acute{F}_{(1:k-1)}\tilde{T}_{(n:n)} + \acute{F}_{(1:k-1)}\tilde{F}_{(n:n)}\acute{T}_{(k+1:n)} \end{aligned} \tag{3.2}$$

According to the inductive hypothesis we can claim that $\tilde{T}_{(1:n-1)}$ is the optimal time over all composites of $n-1$ methods. This composite is thus lower than or equal to the time for the composite obtained by excluding the n-th method in $\tilde{C}$, which is the same as the k-th method in $\acute{C}$. Mathematically this can be expressed as, $\acute{T}_{(1:k-1)} + \acute{F}_{(1:k-1)}\acute{T}_{(k+1:n)} \geq \tilde{T}_{(1:k)} + \tilde{F}_{(1:k)}\tilde{T}_{(k+1:n-1)}$, to yield,

$$\acute{T}_{(1:k-1)} + \acute{F}_{(1:k-1)}\acute{T}_{(k+1:n)} - \tilde{T}_{(1:k)} - \tilde{F}_{(1:k)}\tilde{T}_{(k+1:n-1)} \geq 0 \tag{3.3}$$

According to our assumption $\acute{T} \leq \tilde{T}$; we expand this relation,

$\acute{T}_{(1:k-1)} + \acute{F}_{(1:k-1)}\tilde{T}_{(n:n)} + \acute{F}_{(1:k-1)}(1 - \tilde{R}_{(n:n)})\acute{T}_{(k+1:n)} \leq$

$\tilde{T}_{(1:k)} + \tilde{F}_{(1:k)}\tilde{T}_{(k+1:n-1)} + \tilde{F}_{(1:k)}\tilde{F}_{(k+1:n-1)}\tilde{T}_{(n:n)}$.

We can then rearrange the terms on either side to show that the left-hand side of Equation 3.3 is less than or equal to

$\tilde{F}_{(1:k)}\tilde{F}_{(k+1:n-1)}\tilde{T}_{(n:n)} - \acute{F}_{(1:k-1)}\tilde{T}_{(n:n)} + \acute{F}_{(1:k-1)}\tilde{R}_{(n:n)}\acute{T}_{(k+1:n)}$. Thus,

$$0 < \tilde{F}_{(1:k)}\tilde{F}_{(k+1:n-1)}\tilde{T}_{(n:n)} - \acute{F}_{(1:k-1)}\tilde{T}_{(n:n)} + \acute{F}_{(1:k-1)}\tilde{R}_{(n:n)}\acute{T}_{(k+1:n)}.$$

By rearranging terms and using the equation $\tilde{F}_{(1:k)}\tilde{F}_{(k+1:n-1)} = \acute{F}_{(1:k-1)}\acute{F}_{(k+1:n)}$ to simplify, we obtain,

$$\acute{F}_{(1:k-1)}\tilde{T}_{(n:n)} - \tilde{F}_{(1:k)}\tilde{F}_{(k+1:n-1)}\tilde{T}_{(n:n)} \leq \acute{F}_{(1:k-1)}\tilde{R}_{(n:n)}\acute{T}_{(k+1:n)}.$$

$$\text{implies, } \acute{F}_{(1:k-1)}\tilde{T}_{(n:n)} - \acute{F}_{(1:k-1)}\acute{F}_{(k+1:n)}\tilde{T}_{(n:n)} \leq \acute{F}_{(1:k-1)}\tilde{R}_{(n:n)}\acute{T}_{(k+1:n)}.$$

Canceling the common terms on either side yields,

$\tilde{T}_{(n:n)}(1 - \acute{F}_{(k+1:n)}) \leq \tilde{R}_{(n:n)}\acute{T}_{(k+1:n)}$ which is equivalent to $\tilde{U}_{(n:n)} \leq \acute{U}_{(k+1:n)}$. By the definition of $\tilde{C}$, $\tilde{U}_{(n:n)}$ is the largest utility ratio among all the n methods. But if $\tilde{U}_{(n:n)} \leq \acute{U}_{(k+1:n)}$, there is a composite whose overall utility is higher than the maximum utility ratio of its component methods, thus contradicting Lemma 1. This contradiction occurred because our assumption that $\acute{T} \leq \tilde{T}$ is not true; hence the proof. □

THEOREM 3.2. *If $\tilde{C}$ is the optimal composite then the utility ratios are arranged in increasing order, i.e., $\tilde{U}_{(1:1)} \leq \tilde{U}_{(2:2)} \leq \ldots \leq \tilde{U}_{(n-1:n-1)} \leq \tilde{U}_{(n:n)}$.*

Proof: The proof is based on finding the shortest path in a directed weighted graph. We construct the graph with unit vertex weights and positive edge weights as follows. The vertices are arranged in levels with edges connecting vertices from one level to the next. There are $n+1$ levels numbered 0 through n. Each vertex at level l $(0 \leq l \leq n)$ represents a subset of l methods out of n methods. Vertices are labeled by the sets they represent. Directed edges connect a vertex V_S at level l to a vertex $V_{\bar{S}}$ only if $|\bar{S} \setminus S| = 1$ and $\bar{S} \cap S = S$, i.e., the set $\bar{S}$ has exactly one more element than S. Let F_S denote the total failure rate over all methods in the set S. If $\bar{S} \setminus S = \{i\}$, the edge $V_S \rightarrow V_{\bar{S}}$ is weighted by $F_S T_{(i:i)}$, the time to execute method i after failing at all previous methods. This construction enforces the property that any path from V_0 (representing the empty set) to $V_{\{1,2,\cdots n\}}$ represents a particular composite, one in which methods are selected in the order in which they were added to sets at subsequent levels. It is easy to verify that the shortest path represents the optimal composite. Consider a fragment of the graph, as shown in Figure 3.2. We assume that V_S is a node on the shortest path, and $V_{\hat{S}}$ is also a node on the shortest path, such that $\hat{S} - S = \{i, j\}$. There will be only 2 paths from V_S to $V_{\hat{S}}$, one including the node $V_{\bar{S}}$ $(\bar{S} - S = \{i\})$ and the other including the node V_{S*} $(S^* - S = \{j\})$. Without loss of generality, assume V_{S*} is the node on the shortest path; thus method j was selected before method i in the sequence. Let the time from V_0 to V_S be denoted by T_S and the failure rate by F_S. Using the optimality property of the shortest path:

$$T_S + F_S T_{(j:j)} + F_S F_{(j:j)} T_{(i:i)} \leq T_S + F_S T_{(i:i)} + F_S F_{(i:i)} T_{(j:j)}.$$

After canceling common terms we get $T_{(j:j)} + F_{(j:j)} T_{(i:i)} \leq T_{(i:i)} + F_{(i:i)} T_{(j:j)}$. This can be simplified further using the relation $F_{(j:j)} = 1 - R_{(j:j)}$ to yield: $R_{(j:j)} T_{(i:i)} \geq R_{(i:i)} T_{(j:j)}$ and thus $U_{(j:j)} \leq U_{(i:i)}$. This relationship between utility ratios holds for any two consecutive vertices on the shortest path. Hence, the optimal composite given by the shortest path is one in which methods are selected in increasing order of the utility ratio. □

The two theorems provide us with two different algorithms for constructing the optimal composite. Theorem 3.2 is easier to derive, but Theorem 3.1 is cheaper to implement. The algorithm given by Theorem 3.2 concerns construction of a permutation graph and then finding the shortest path. The execution time of the algorithm is proportional to the number of vertices in the graph, of the order of $O(2^n)$. In contrast, Theorem 3.1 involves simply sorting the individual utility ratios and takes only time $O(nlogn)$ which is polynomial to the number of methods, n. We will be using the algorithm of Theorem 3.1 (sorting by utility ratios) for constructing composite solvers.

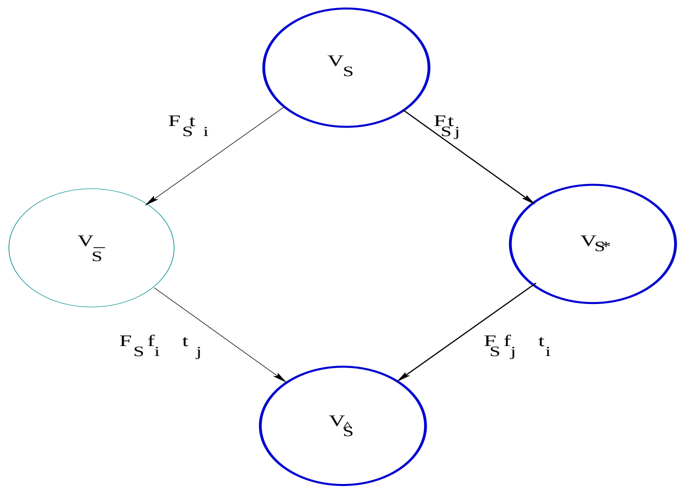


Fig. 3.2. Segment of the graph used in the proof of Theorem 2.

3.2.1.1 Composites From Smaller Subset of All Base Methods

An interesting variant to the problem of constructing optimal composites concerns building an optimal composite using k methods from a total collection of n methods; $k \leq n$. Observe that now there exist two classes of such k-method subset composites: (i) subset composite with the highest reliability of any composite with k methods and (ii) subset composite with the minimal worst case execution time of any composite with k methods.

A composite with the highest reliability is equivalent to the one with the lowest failure rate. We sort the base methods in the increasing order of their failure rates and select the first k methods from the list. It is easy to see that the optimal composite composed of these methods will have the lowest cumulative failure and thus the highest reliability.

Building a subset composite with minimal worst case time requires a more complex algorithm because it is not possible to select the required k methods by simply arranging the base methods in order of execution time, reliability or utility ratios. However, once the base methods of the subset are determined, the optimal composite can be obtained by arranging them in the increasing order of their utility ratios. That is, if methods $M_1, M_2, ..., M_k$ form the subset, and $\hat{P}$ is the optimal permutation, then $U_{\hat{P}1} \leq U_{\hat{P}2} \leq \dots U_{\hat{P}k}$. Therefore if a method can be placed in the r^{th} position of a subset sequence, then its utility ratio is less than that of at least $k - r$ methods and greater than that of at most $r - 1$ methods. Consequently, for each position r our choice is limited only to $n - k + 1$ methods. We define this candidate set of methods for each position as *levels*. There will be k levels corresponding to each position in the sequence, each containing $n - k + 1$ methods.

The algorithm to construct the optimal composite is as follows. For each method M_j in level i, we construct an optimal composite of $k-i+1$ methods with M_j as the first method. There might be more than one sequence of methods that start with M_j and are in the increasing order of the utility ratio. We select the sequence with the minimum worst case execution time. Since there are $n - k + 1$ methods at each level, we obtain $n - k + 1$ composites. Each of these composites can be combined with a method from level p; $p < i$. We continue to build such subset composites until level 1 is reached. We select the sequence with minimum execution time from the $n - k + 1$ sequences built in this level.

We will now prove that this algorithm indeed yields the optimal subset composite. Let us suppose that the permutation obtained by the above procedure is Q, and there exists another composite associated with the permutation P, whose execution time is less than the obtained composite. This can occur only if a sub-sequence associated with permutation P was discarded while constructing the composite, which means that at the corresponding level the sequence associated with Q had lower execution time. Thus, our assumption that the execution time of P is lower, is false.

3.2.2 Composites when Failure Rates of Methods are Corelated

In the previous section we discussed the construction of optimal composites for base methods with mutually independent failure rates. We will now consider a more generalized scenario of base methods whose failure rates are not mutually independent. We now define the cumulative failure of a composite $\hat{C}_{(k:l)}$, as the conditional failure of method M_k to M_l, given that methods M_1 to M_{k-1} have been executed. By Bayes' law the cumulative failure can be expressed as $\hat{F}_{(k:l|1:k-1)} = \frac{\bigcap_{i=1}^{i=l} f_{\hat{P}_i}}{\bigcap_{i=1}^{i=k-1} f_{\hat{P}_i}}$. The cumulative reliability is $\hat{R}_{(k:l|1:k-1)} = 1 - \hat{F}_{(k:l|1:k-1)}$. Let the cumulative time be defined as $\hat{T}_{(k:l|1:k-1)}$. Then the cumulative utility ratio is $\hat{U}_{(k:l|1:k-1)} = \frac{\hat{T}_{(k:l|1:k-1)}}{\hat{R}_{(k:l|1:k-1)}}$.

We now obtain a generalized version of Theorem 3.2 which gives the condition to be satisfied by an optimal composite. The theorem is as follows, THEOREM 3.3. *If $\tilde{C}$ is the optimal composite then each pair of consecutive methods are arranged in the increasing order of their utility ratios, i.e.,* $\tilde{U}_{(1:1)} \leq \tilde{U}_{(2:2)}; \tilde{U}_{(2:2|1)} \leq \tilde{U}_{(3:3|1)}; \ldots \tilde{U}_{(k:k|1:k-1)} \leq \tilde{U}_{(k+1:k+1|1:k-1)}; \ldots \tilde{U}_{(n-1:n-1|1:n-2)} \leq \tilde{U}_{(n:n|1:n-2)}$.

The proof can be constructed using arguments similar to those in Theorem 3.2.

Theorem 3.3 provides a necessary, but *not sufficient* condition. Table 3.1 gives an example of three methods with corelated failure rates. It can be seen from the values of the utility ratios two composites (1,2,3 and 2,3,1) can be formed that satisfy the condition in Theorem 3.3. The execution time of composite 1,2,3 is 1.52, this is less than the execution time(1.6) of the second composite 2,3,1.

Methods	Time	Individual Failures	Individual Utilities	Conditional Failures	Conditional Utilities
M1	1.0	.4	1.67	$F(1\|2)$=.3	$U(1\|2)$=1.4
M2	1.0	.5	2	$F(2\|1)$=.3	$U(2\|1)$=1.4
M3	1.0	.6	2.5	$F(3\|1)$=.6; $F(3\|2)$=.2	$U(3\|1)$=2.5; $U(3\|2)$=1.25

Table 3.1. An example of three methods with conditional failure rates

However in practice, methods whose failure rates are dependent generally belong to the same class of solvers. With this assumption, we classify the set of methods into groups according to the following conditions.

- Failure rates of methods in the same group are non-independent.
- Failure rates depend only on the *number* of methods previously executed from that group, i.e., the failure rate after execution of any i methods in group G_j is f_i^j; $r_i^j = 1 - f_i^j$.
- Reliability in any group G_j increases with the number of methods executed; $r_0^j < r_1^j < \ldots < r_n^j$.

- Failure rates of methods across groups are mutually independent.

For methods in the same group, the necessary condition in Theorem 3.3 can now be expressed as follows,

$$\tilde{T}_{(1:1)}/(1-f_0) \leq \tilde{T}_{(2:2)}/(1-f_0);$$

$$\tilde{T}_{(2:2)}/(1-f_1) \leq \tilde{T}_{(3:3)}/(1-f_1);$$

$$\vdots$$

$$\tilde{T}_{(k+1:k+1)}/(1-f_{k-1}) \leq \tilde{T}_{(k+1:k+1)}/(1-f_{k-1});$$

$$\vdots$$

$$\tilde{T}_{(n:n)}/(1-f_{n-2}) \leq \tilde{T}_{(n:n)}/(1-f_{n-2}).$$

The above relation is equivalent to, $\tilde{T}_{(1:1)} \leq \tilde{T}_{(2:2)} \leq \ldots \leq \tilde{T}_{(n:n)}$, i.e. arranging the methods in increasing order of time. Therefore, in a same group, the optimal composite is obtained by arranging the methods in increasing order of time. Observe that the utility ratio of methods change according to their position in the sequence. The utility ratio of method M_i at position j is defined as $u_i^j = t_i/(1-f_{j-1}) = t_i/r_{j-1}; 1 \leq i, j \leq n$.

We will now build the composite across groups $G_1, G_2, \ldots, G_k$. Within each group the optimal composite is obtained by arranging the methods in increasing order of time as shown above. Our goal is to create an integrated optimal composite by merging the optimal sequences of each group. We will prove that if the utility ratios of the methods are sufficiently low then merging produces an unique optimal composite.

Consider a group G_j with p elements; let method M_i be in group G_j. The utility ratio of M_i, $u_i^{max(j)}$, is maximized when it is the first method in the sequence comprising only of methods in that group. Therefore $u_i^{max(j)} = t_i/r_0^j$. We define $\Delta t = min\{t_a - t_b : \forall M_a, M_b\}$ as the minimum difference between the execution times of any two methods. For a particular group G_j, this value is $\Delta t^j = min\{t_a - t_b : M_a, M_b \in G_j\}$. Within a group G_j the highest reliability is given by r_n^j and the minimum by r_0^j. Since the failure rates of methods across groups are mutually independent, therefore the maximum reliability across all groups $G_1, G_2, \ldots G_k$, will be $r_{max} = 1 - \prod_{i=1}^{i=k} f_n^i$. The minimum reliability will be $r_{min} = 1 - \max_{i=1}^{i=k} f_0^i$. We define $\Delta r = r_{max} - r_{min}$, as the maximum difference between cumulative reliabilities across groups. For group G_j the corresponding value is $\Delta r^j = r_n^j - r_0^j$. We state that method M_i satisfies *Property X* if $u_i^{max} < \Delta t / \Delta r$. Observe that Property X is invariant over subsets of groups. That is if a method M_i satisfies Property X and is a method in G_g, where G_g is any subset of the groups $G_1, G_2, \ldots, G_k$, then $u_i^{max(g)} < \Delta t^g / \Delta r^g$.

THEOREM 3.4. *If all methods $M_1, M_2, \ldots M_n$, satisfy Property X then, (i) the optimal sequence over all methods across all groups is obtained by arranging methods in the increasing order of their utility ratios and (ii) there is a unique ordering specifying the optimal composite.*

Proof. We first prove that methods in the same group are arranged in the increasing order of their utility ratios. Consider two methods M_a and M_b from the same group G_j at positions x and $x+i$, and $t_a < t_b$. From the optimality condition in Theorem 3.3, we know that M_a is placed earlier in the sequence than M_b. If M_b satisfies Property X then,

$$
\begin{aligned}
u_b^{max(j)} &< \frac{\Delta t^j}{\Delta r^j} \\
\text{implies, } \frac{t_b}{r_0} &< \frac{\Delta t^j}{\Delta r^j} \\
\text{implies, } \frac{t_b}{r_n} &< \frac{t_b - t_a}{r_n - r_0} \qquad \text{as } r_0 < r_n \\
\text{implies, } t_a/r_0 &< t_b/r_n \\
\text{implies, } \frac{t_a}{r_{x-1}} &< \frac{t_b}{r_{x+i-1}} \\
\text{implies, } u_a^x &< u_b^{x+i}
\end{aligned}
$$

This ordering is true for methods in the same group. Because the failure rates of methods across groups are mutually independent, therefore this relative ordering is maintained during merging.

We now prove that there exists a unique ordering. It is easy to see from Theorem 3.3 that the ordering is unique within a group. Let us consider inserting a method M_a, with execution time t_a and failure rate f_a, from group G_1 into the optimal sequence of group G_2. Let the method be inserted between M_i and M_{i+1} which are methods of group G_2, arranged in the optimal sequence. According to Theorem 3.3, the following condition should be satisfied, $\tilde{U}_{(i:i|1:i-1)} \leq \tilde{U}_{(a:a|1:i-1)}$ and $\tilde{U}_{(a:a|1:i)} \leq \tilde{U}_{(i+1:i+1|1:i)}$. This is simplified to, $\tilde{T}_{(i:i)}/(1-f_{i-1}) \leq \tilde{T}_{(a:a)}/(1-f_a) \leq \tilde{T}_{(i+1:i+1)}/(1-f_i)$ which is the increasing order of the utility ratio of the methods.

Let us suppose that there exists another position between methods M_j and M_{j+1} in the optimal sequence of group G_2, where method M_a can be inserted while satisfying the same conditions. Therefore, $\tilde{T}_{(j:j)}/(1-f_{j-1}) \leq \tilde{T}_{(a:a)}/(1-f_a) \leq \tilde{T}_{(j+1:j+1)}/(1-f_j)$. Since M_a can be inserted between M_i and M_{i+1} as well as between M_j and M_{j+1}, the following condition is satisfied, $\tilde{T}_{(j:j)}/(1-f_{j-1}) \leq \tilde{T}_{(i+1:i+1)}/(1-f_i)$. This condition is equivalent to $\frac{\Delta t^2}{\Delta r^2} < u_{i+1}^{max(2)}$. This contradicts our assumption that method M_{i+1} in group G_2 satisfies Property X. Therefore M_a can be inserted only in a unique position.

Observe that after merging, the methods still satisfy Property X. Therefore for any method M_i in the merged sequence $u_{i+1}^{max(1+2)} < \frac{\Delta t^{(1+2)}}{\Delta r^{(1+2)}}$. Using a proof similar to that of merging two groups we can show that multiple groups can be merged in a unique sequence. $\square$

3.3 The Design and Implementation of Composite Solvers

The implementation of composite solvers raises some issues which were not evident in the preceding analysis. For example, consider a situation when a base method fails to converge for a certain problem instance. According to the definition of a composite solver, the next method in the sequence is invoked. At this stage, the original linear system has been partially solved by the first method. The new method can be applied to the partially solved linear system or it could start afresh by solving the original linear system. The second method has the advantage that its execution time can be accurately determined through the preceding analytical results. However the first method is likely to have lower execution time. Most of our experiments have used the first approach.

Another issue is the implementation of parallel composite solvers. The efficient parallelization of the composite depends on whether the individual base methods can be parallelized. To construct a parallel composite solver for solving a linear system of the form $Ax = b$, we assume that the base methods are fully parallel. We also assume that all of these methods use the same data distribution across P processes for A, x, and b. Under this assumption, upon failure of a base method, the next method in the sequence can be easily invoked on the same system without any extra data redistribution. The remaining construction details are similar to that in the sequential case, the data structures used by the previous method are released and those for the next method in the sequence are constructed, etc. We use this approach because it allows a scalable implementation with the coefficient matrix distributed across processors.

3.4 Empirical Results

In this section, we report on our experiments for evaluating composites in sequential (uniprocessor) and parallel (multiprocessor) executions. We obtained the values of the utility ratios by sampling relevant performance data such as reliability and execution time. A method was considered to fail if it did not converge within a fixed number of iterations. The execution time was measured either as the overall time required for the linear solution or as the time per iteration of the linear solver. For sampling we took a representative subset of problem instances and calculated the values for each base method. The mean values over the instances in the sample set were taken to be the metrics associated with the methods. The composites were then constructed by arranging the methods in the increasing order of their utility ratios.

We will first give a brief description of the Chiba and Jazz multiprocessor clusters on which most of our experiments were performed. We will then present our initial experimental results based on solving linear systems arising from different applications. The rest of our experiments were based on simulation of the driven cavity flow, which is a model CFD application. We will the briefly review this model problem. Then we will present sequential results on a 96 by 96 mesh and a 128 by 128 mesh. In the final subsection, we will report on the parallel implementation of composite solvers.

3.4.1 Multiprocessor Environment

Chiba City Cluster [1] at the Argonne National Laboratory is built to support research on scalable library development, scientific visualization, distributed computing, systems software and cluster management, etc. The cluster consists of 256 computing nodes with 500 MHz dual-cpu Pentium III running Red Hat Linux operating system. Each computing node is 512 MB RAM and 9G local disk storage. In addition to the computing nodes, the cluster has 4 login nodes with the same specifications. The cluster also contains 32 visualization nodes with G400 graphics cards and 8 storage nodes with 500MHz Xeon processors with 512 RAM and 200GB disk storage. There are two network systems in the cluster, the high performance network through a 64-bit Myrinet and the management network based on a fast Ethernet.

Jazz Cluster is a part of the Laboratory Computing Resource Center(LCRC) at the Argonne National Lab. LCRC is built to "promote the widespread use of high-performance computing technologies across the laboratory in support of scientific research" [2]. The Jazz LCRC cluster consists of 350 computing nodes with 2.4 GHz Pentium Xeon running Red Hat Linux operating system. Half of the computing nodes (175) have 2GB RAM and the rest have 1GB RAM. The storage infrastructure consists of a 20TB cluster wide disk. The network is supported by a Myrinet2000 interconnect. According to the top 500 supercomputers list in June2004 [33], the Jazz cluster was ranked 235.

3.4.2 Performance Across Systems from Several Applications

In this set of experiments, we used a collection of linear systems arising from different applications. We used a suite of nine preconditioned Conjugate Gradient methods labeled $M_1, \ldots, M_9$. M_1 represents Conjugate Gradient method without any preconditioner. M_2 and M_3 use Jacobi and SOR preconditioning schemes respectively. Methods M_4 through M_7 use incomplete Cholesky preconditioners with 0, 1, 2 and 3 levels of fill. Methods M_8 and M_9 use incomplete Cholesky preconditioners with numerical drop threshold factors of .0001 and .01. The experiments were performed on a SUN Ultra workstation with a 296MHz UltraSPARC-II processor and 384 MB of RAM. We obtained the linear solvers from PETSc.

Our first set of experiments were aimed to demonstrate that the composite where methods are arranged in the increasing order of the utility ratios takes the least execution time. We executed each method over all the problems in the sample set. We then normalized the execution time of each method by dividing it by the time required for a sparse direct solver. The geometric mean of the normalized running time was used as our estimate of t_i for each M_i. We assumed that the method was unsuccessful if it failed to converge in 200 iterations. We used the success rate as the reliability metric r_i for method M_i. These two measures were used to compute the utility ratio $u_i = t_i/r_i$ for each method M_i.

For our first experiments we used a set of six `bcsstk` sparse matrices from finite element methods in structural mechanics [15]. The values of the metrics of the eight base methods were obtained as described above. Four different composite solvers C_T, C_R, C_X, C_O were created representing orderings in increasing order of normalized execution time, in decreasing order of reliability, a random ordering and in increasing order of the utility ratio respectively. The overall reliability of each composite was analytically computed to be 1. If the method with highest reliability M_8 is excluded, the composite reliability is .99, a value significantly higher than the reliability of the remaining underlying methods. We applied these four composite solvers to the complete set of matrices and calculated the total time for each composite over all the test problems. The results are shown in Table 3.2; our optimal composite C_O has the least total time.

In our second set of experiments, we considered a larger suite of test problems consisting of matrices from five different applications [15]. We obtained the performance metrics using a sample set of 10 matrices consisting of two matrices from each application type. We constructed four composites solvers C_T, C_R, C_X, C_O as previously defined, with the same reliability. Results in Table 3.3 indicate that our composite solver still has the least total execution time over all problems in the test suite. As in the previous problem instance, the reliability of all composites (after M_8 is excluded) is .99. The total execution time of C_O is less than half the execution time for C_T, the composite obtained by selecting underlying methods in increasing order of time.

Methods and metrics

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9
Time	1.01	.74	.94	.16	1.47	2.15	3.59	5.11	2.14
Reliability	.25	.50	.75	.25	. 50	. 50	.75	1.00	.25
Ratio	4.04	1.48	1.25	.63	2.94	4.30	4.79	5.11	8.56

Composite solver sequences

C_T	M_4	M_2	M_3	M_1	M_5	M_9	M_6	M_7	M_8
C_R	M_8	M_3	M_7	M_2	M_5	M_6	M_1	M_4	M_9
C_X	M_9	M_8	M_1	M_5	M_3	M_2	M_7	M_6	M_4
C_O	M_4	M_3	M_2	M_5	M_1	M_6	M_7	M_8	M_9

Execution time (in seconds)

Problem	Rank	Non-zeroes (10^3)	C_T	C_R	C_X	C_O
bcsstk14	1,806	63.4	**.25**	.98	1.19	.27
bcsstk15	3,908	117.8	1.88	5.38	9.45	**1.22**
bcsstk16	4,884	290.3	1.05	6.60	2.09	**.98**
bcsstk17	10,974	428.6	57.40	**12.84**	16.66	37.40
bcsstk18	11,948	149.1	4.81	5.70	12.40	**2.80**
bcsstk25	15,439	252.2	1.60	21.93	36.85	1.59
Total execution time			66.99	53.43	78.64	**44.26**

Table 3.2. Results for the `bcsstk` test suite.

Methods and metrics

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9
Time	.77	.73	.81	.20	1.07	1.48	2.10	.98	.76
Reliability	.50	.60	.90	.50	. 70	.60	.60	1.00	.40
Ratio	1.54	1.23	.90	.40	1.53	2.47	3.50	.98	1.91

Composite solver sequences

C_T	M_4	M_2	M_9	M_1	M_3	M_8	M_5	M_6	M_7
C_R	M_8	M_3	M_5	M_2	M_6	M_7	M_1	M_4	M_9
C_X	M_9	M_8	M_7	M_6	M_5	M_3	M_2	M_1	M_4
C_O	M_4	M_3	M_8	M_2	M_5	M_1	M_9	M_6	M_7

Execution time (in seconds)

Problem	Rank	Non-zeroes (10^3)	C_T	C_R	C_X	C_O
bcsstk14	1,806	63.4	**.31**	1.06	1.18	.37
bcsstk16	4,884	290.3	**.97**	6.35	2.07	.99
bcsstk17	10,974	428.6	35.7	**13.3**	16.3	23.4
bcsstk25	15,439	252.2	1.61	22.8	36.8	**1.60**
bcsstk38	8032	355.5	35.8	33.5	51.5	**2.39**
crystk01	4875	315.9	**.44**	4.03	.84	.47
crystk03	246,96	1751.1	**2.55**	35.8	5.45	2.56
crystm02	139,65	322.90	.32	.40	5.38	**.32**
crystm03	246,96	583.77	.60	.72	.73	**.60**
msc00726	726	34.52	.13	1.39	.23	**.13**
msc01050	1050	29.15	.80	**.10**	.23	.27
msc01440	1440	46.27	2.91	.79	2.39	**.5**
msc04515	4515	97.70	10.5	**1.95**	6.10	4.45
msc10848	10848	1229.77	75.6	101	163	**26.3**
nasa1824	1824	39.21	2.46	**1.15**	1.3	1.80
nasa2146	2146	72.25	**.09**	.64	2.29	.09
nasa2910	2910	174.29	10.9	**2.34**	6.69	2.80
nasa4704	4704	104.756	13.4	13.4	13.40	**4.61**
xerox2c1	6000	148.05	.27	1.92	**.23**	18.1
xerox2c2	6000	148.30	**.24**	.41	.48	.25
xerox2c3	6000	147.98	.27	.41	**.21**	.24
xerox2c4	6000	148.10	.23	.40	**.22**	.23
xerox2c5	6000	148.62	.25	.42	.24	**.23**
xerox2c6	6000	148.75	.29	.62	.90	**.23**
Total execution time			196.64	244.9	318.16	**90.6**

Table 3.3. Results for the test suite with matrices from five applications.

3.4.3 Composites for Driven Cavity Flow

3.4.3.1 Application Description

The driven cavity flow model [35] is an example of incompressible flow resulting due to the combined effects of lid-driven flow and buoyancy-driven flow in a two-dimensional rectangular cavity. The lid velocity is steady and spatially uniform and generates a principal vortex and subsidiary corner vortices. The principal vortex is opposed by the buoyancy vortex, which is induced by the differentially heated lateral cavity walls.

The governing differential equations are obtained by using the Navier-Stokes and energy equations. The equations are given in terms of u, v, which are the velocities in the (x, y) directions respectively. The vorticity ω on a domain Ω is defined as $\omega(x, y) = -\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}$. The following elliptical PDEs are generated:

$$-\Delta u - \frac{\partial \omega}{\partial y} = 0,$$

$$-\Delta v + \frac{\partial \omega}{\partial x} = 0,$$

$$-\Delta \omega + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} - Gr\frac{\partial T}{\partial x} = 0,$$

$$-\Delta T + Pr(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}) = 0,$$

where $T(x, y)$ is the temperature, Pr is the Prandtl number, and Gr is the Grashof number. The boundary conditions are $\omega(x, y) = -\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}$. The continuous domain is then discretized using a finite difference scheme with a five-point stencil for each component on a uniform Cartesian coordinate system.

3.4.3.2 Implementation and Validation

We test the role of composites in a driven cavity flow application. We performed the sequential experiments on a workstation with a dual-CPU 500 MHz Pentium III with 512 MB of RAM. We employed a cluster with a Myrinet 2000 network and 2.4 GHz Pentium Xeon processors with 1-2 GB of RAM for our parallel experiments.

We used the driven cavity implementation by Keyes et al. which is an example in the PETSc tutorial [5]. In this code, the governing nonlinear equations of the driven cavity flow is solved using an inexact Newton method [30, 39]. The linearized Newton system is approximately solved by an iterative Krylov solver. For a fixed grid size, the nonlinearity of the system is determined by the values of the Grashof number and the lid velocity. It has been observed that Newton's method often struggles at higher values of these parameters. This problem can be overcome by using a globalization technique known as pseudo-transient continuation. The application employs several iterations of an inexact Newton method, where each nonlinear iteration requires a linear system solution. The linear solver can use one of several underlying base preconditioned Krylov methods or their composites.

In the driven cavity flow application with a fixed mesh (and linear system size), the convergence of the nonlinear solver is affected mainly by two parameters that determine the degree of nonlinearity of the system: the Grashof number and the lid velocity. At higher values of either or both parameters, the application typically produces linear systems that are more difficult to solve using Krylov methods with mild to medium degrees of preconditioning. Consequently, most underlying linear solvers have high failure rates. At significantly lower values of the two parameters, both linear and nonlinear iterations converge readily, and linear solver failures seem to have a negligible effect on the convergence of the nonlinear solver. Thus these low and high parameter values define the range of values that are relevant for our experiments; our experiments were limited to those values where the nonlinear solver converged while incurring failures for several linear system solution instances. In the following section we will present the performance data obtained by applying composites on driven cavity application with varying mesh sizes.

Driven cavity flow is a well known model problem in CFD and has been widely used as "a suitable vehicle for testing and validating computer codes" [20]. Vahl Davis et al. compared 37 different numerical solutions from 30 different contributors for solving a particular instance of this problem [21]. It was observed that despite the differences in implementation the numerical results were "substantially in agreement with each other" [21]. However the execution time of the implementations varied considerably. By using composite solvers we were able to provide highly robust linear solutions, which in turn lowered the nonlinear solution time. We used the driven cavity implementation in PETsc, which is in agreement with the implementation by Bennet et al. [9] which in turn has been validated with the results in [21].

3.4.3.3 Results for a 96×96 Mesh

Our first set of experiments [14] used a 96×96 mesh with Grashof numbers in the range [500, 1000] and lid velocities in the range [10, 20]. We detected convergence of

Newton's method when $||f(u)|| < \epsilon||f(u^0)||$, where $\epsilon = 1.e^{-8}$. To obtain initial sample observations, we fixed the lid velocity at 20 for Grashof numbers 820, 840, and 1000. Table 3.4 summarizes performance measures for the four base linear solvers B1, B2, B3, and B4; all methods use GMRES with different values of the restart parameter and preconditioners with level of fill ILU and an RCM ordering or drop-threshold ILU and a QMD ordering. The optimal composite, CU, comprises base methods B3, B1, B4, and B2, arranged in increasing order of the utility ratio. The second composite, labeled CT, comprises methods B1, B3, B4, and B2, in increasing order of time. The third composite, CR, has methods in a random order B3, B4, B1, and B2. The solution from a failed base method becomes the initial guess for a subsequent method, thus naturally allowing reuse. We detected convergence of each linear solve (whether composite or not) when the relative reduction in residual norm fell below $1.e^{-5}$.

We report on the performance of the four base methods and three composites for nine simulations. Table 3.5 summarizes the empirical data related to this set of experiments. The cumulative failure rate here and in other experiments is calculated as the product of the failure rates of the base methods. We introduce the term *simulation point* to designate each set of nonlinearity parameters. We use a series of figures with a stacked bar for each method; the height of a bar indicates the cumulative value over all simulation points, while a single segment corresponds to the value observed at a simulation point. In our figures, we indicate the parameters for each simulation point (and thus a segment of stacked bar) in the form *i:j*, where *i* denotes the Grashof number and *j* denotes the lid velocity.

Figure 3.3 shows the total number of failures and the reliability for base methods B1, B2, B3, and B4 and composites CU, CT, and CR. Based on our model, all composites should have a worst-case failure rate of .19, a value significantly lower than that of a base method. Likewise, the reliability of a composite is .81 and thus higher than that of a base method (see Table 3.5). The observed values of composite reliability were ideal and better than the predicted values. All composites successfully solve all linear systems, and no failures were observed (although composites typically use more than one underlying method).

Figures 3.4 and 3.5 show the total number of linear and nonlinear iterations and the time per iteration for each method. The product of these two measures is approximately equal to the total time for linear (or nonlinear) solution. These results show the benefits of taking into account both failure rates and execution times to develop composites. The time per linear iteration is the lowest for the composite CT, which is

Base Methods	B1	B2	B3	B4
GMRES Restart	30	60	45	30
Preconditioner	ILU	ILUT	ILUT	ILUT
ILU: Incomplete LU with 1 level of fill and an RCM ordering.				
ILUT: Incomplete LU with drop threshold .01 and a QMD ordering.				
Linear Solver				
Iteration count	2114	2135	2191	2188
Time for all iterations (sec)	1001	1512	1252	1400
Mean time per iteration (sec)	1.42	2.12	1.71	1.92
Failure rate	0.75	0.75	0.50	0.67
Reliability	0.25	0.25	0.50	0.33
Utility ratio	4004	6049	2503	4664
Failure rate of a composite is .19 (.75 $\times$ .75 $\times$ .50 $\times$ .67).				
Reliability of a composite is .81 (1.00 $-$ failure rate).				
Nonlinear Solver				
Iteration count	12	12	12	12
Time for all iterations (sec)	1049	1562	1300	1448
Mean time per iteration (sec)	262	390	325	362
Failure rate	0	0	0	0
Reliability	1	1	1	1

Table 3.4. The cumulative performance of four base methods for three driven cavity flow simulations with a 96×96 mesh, a lid velocity of 20, and Grashof numbers 820, 840, and 1000.

Metric	Base Methods				B1-B4	Composites		
	B1	B2	B3	B4	Total	CU	CT	CR
Cumulative Performance Data over 9 simulations								
Time(Linear) (secs)	3371	3916	3258	3640	14185	3105	4668	3299
Time(Nonlinear) (secs)	3563	4044	3387	3769	14763	3224	4795	3417
Iterations(Linear)	7160	5571	5698	5696	24125	4966	8332	5157
Iterations(Nonlinear)	40	32	32	32	136	27	29	27
Failures	31	23	18	22	94	0	0	0
Failure Rate(%)	77.5	71.9	56.2	67.7	21.2	0	0	0
Average Performance Data over 9 simulations								
	B1	B2	B3	B4	Mean	CU	CT	CR
Time (Linear) (secs)	374.5	435.1	362	404.4	394.02	345	518.6	366.5
Time (Nonlinear) (secs)	395.8	449.3	376.3	418.7	410.08	358.2	532.7	379.6
Iterations (Linear)	795.5	619	633.1	632.8	670.13	551.7	925.7	573
Iterations (Nonlinear)	4.4	3.5	3.5	3.5	3.7	3	3.2	3
Failures	3.4	2.5	2	2.4	2.6	0	0	0

Table 3.5. Summary of performance measures for 9 simulations of driven cavity flow on a 96 by 96 mesh. The column labeled (B1-B4) gives the cumulative and mean values across four base solution schemes. The optimal composite CU is 12% faster than the average execution time of the base methods.

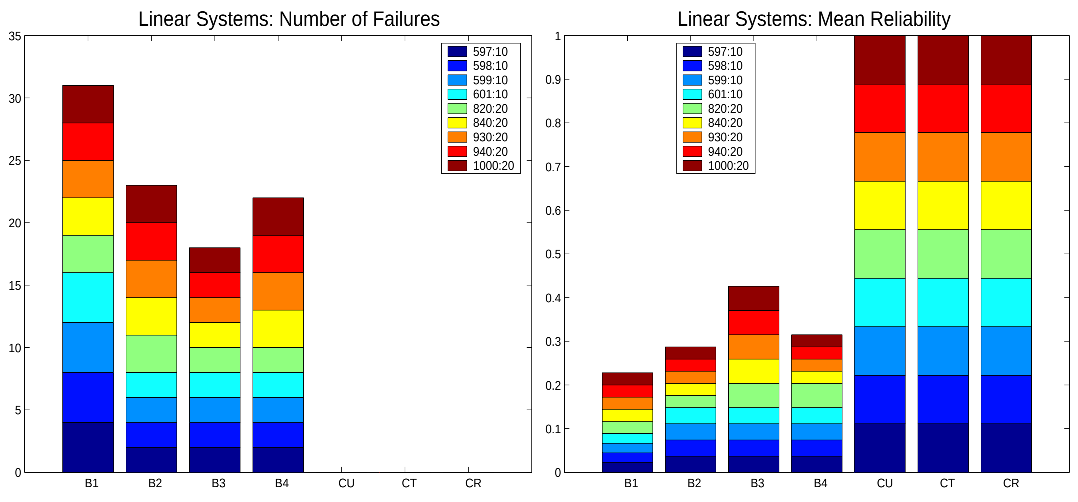


Fig. 3.3. The number of failures and the mean reliability of the linear solver using base linear solution methods B1, B2, B3, and B4 and composites CU, CT, and CR.

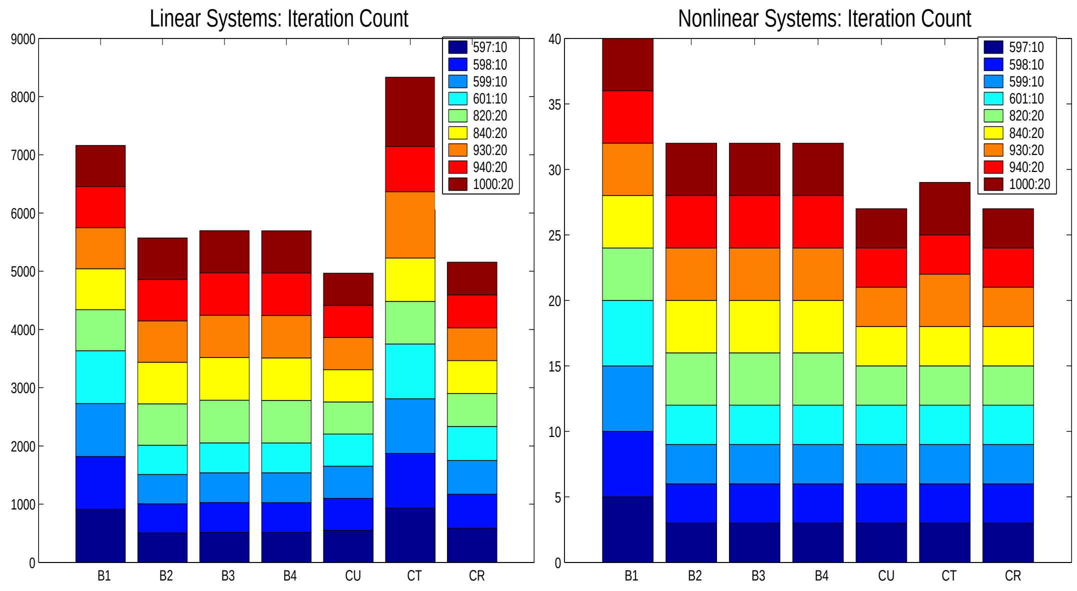


Fig. 3.4. Iteration counts for linear and nonlinear solution using base methods B1, B2, B3, and B4 and composites CU, CT, and CR.

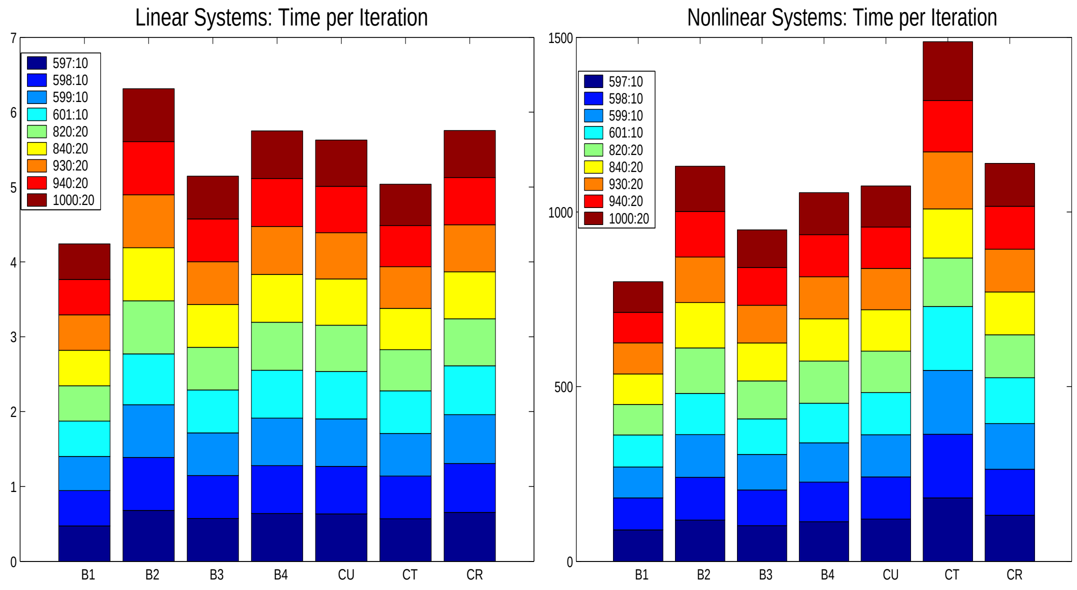


Fig. 3.5. Average time per iteration for linear and nonlinear solution using base methods B1, B2, B3, and B4 and composites CU, CT, and CR.

based on least time. Since the initial methods in the composite often fail, however, the overall number of linear solver iterations for CT correspondingly increases, as does the time per nonlinear iteration. On the other hand, for the utility ratio composite CU, although the time per linear iteration is higher than for CT, the number of linear iterations and hence the time per nonlinear iteration are significantly smaller. Observe, too, that the total number of nonlinear iterations is lower for all composites than for the base methods. We conjecture that this is a consequence of the improved reliability of the composite linear solvers. We expect this effect to be more pronounced in applications where the convergence of the nonlinear solver depends more critically on accurate linear system solution. This relationship is somewhat weak for our application for the selected range of parameters.

Figure 3.6 shows the total linear and nonlinear solution time over all nine simulations, and Table 3.5 summarizes the results shown in detail in Figures 3.3 through 3.6. The execution time is the least for composite CU, in which the underlying methods are in increasing order of the utility ratio. However, these execution times are not vastly different from those for the base method B1, even though the number of linear solver iterations for CU is significantly lower (see Figure 3.4). This situation occurs partly because the decrease in nonlinear iterations from accurate linear system solution using CU is offset by the lower time per linear iteration of base method B1. Another reason is that although base method B1 fails the most number of times (least reliable), the failures do not translate into a proportional increase in the nonlinear iterations. We conjecture that the potential benefits of robust linear solution through composites would be even more dramatic for applications in which the linear solver failures lead to significantly slower convergence (or failure) of the nonlinear solver. We expect this situation to be especially relevant in the latter iterations of Newton's method, when relatively accurate linear solves are often needed to achieve quadratic convergence.

3.4.4 Results for a 128×128 Mesh

In our second set of experiments [13] we solved a larger sparse linear system of rank $260,100$ (with approximately 5.2 million non zeroes), resulting from a 128×128 mesh. The base solution methods used were (B1) GMRES(30) and an ILU preconditioner with fill level 1 with a quotient minimum degree (qmd) ordering, (B2) TFQMR with an ILU preconditioner with drop threshold 10^{-2} and a reverse Cuthill Mckee (rcm) ordering, (B3) GMRES(30) and an ILU preconditioner with fill level 0 and an rcm ordering, and (B4) TFQMR with an ILU preconditioner with fill level 0 and an rcm ordering [24, 27, 41].

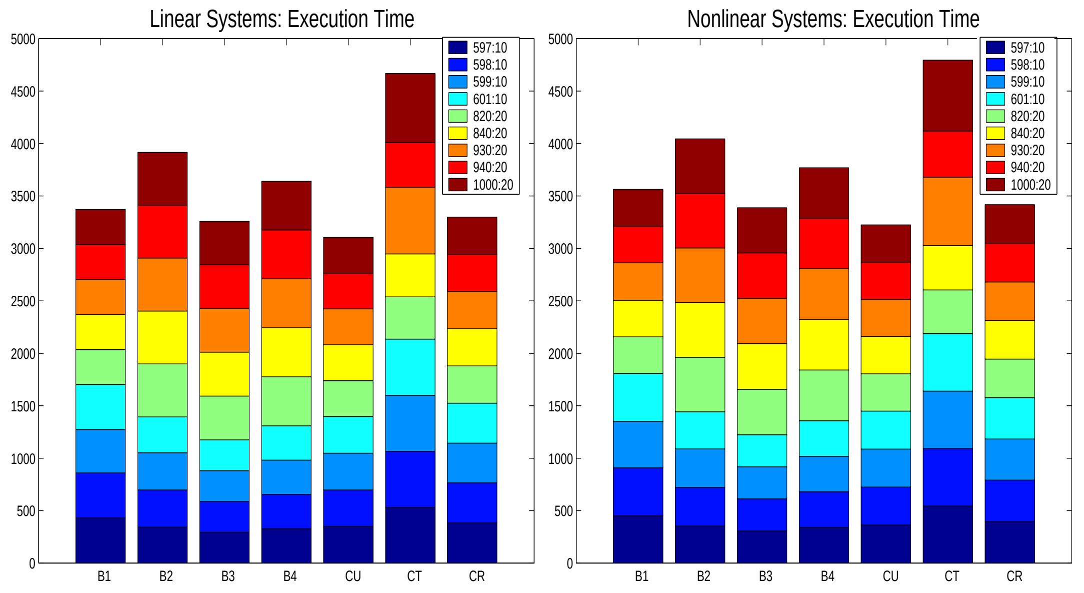


Fig. 3.6. Total time for linear and nonlinear solution using base methods B1, B2, B3, and B4, and composites CU, CT, and CR.

As our sample set, we used a Grashof number of 660 with a lid velocity of 10 and a Grashof number of 620 with lid velocities 13, 16, and 20. We observed the failure rates (f_i) and the mean time per iteration (t_i) of the linear solver and used these metrics to compute the utility ratio of each method; these metrics are shown in Figure 3.7. The optimal composite is denoted as CU and correspond to base methods in the sequence: 2, 3, 1, 4. We also formed three other composites using arbitrarily selected sequences: C1: 3, 1, 2, 4; C2: 4, 3, 2, 1; and C3: 2, 1, 3, 4.

We ran a total of 24 simulations with six Grashof numbers (580, 620, 660, 700, 740, and 780) and four lid velocities (10, 13, 16, and 20). We report on the performance of the four base and four composite methods using several stacked bar graphs. Each stacked bar in Figures 3.8 through 3.11 corresponds to one of the eight methods, and each segment of the stack represents a simulation point corresponding to a Grashof:lid velocity pair; the segments are arranged in increasing order of Grashof number and lid velocities (per Grashof number) with the bottom representing 580:10 and the top 780:20. Thus, starting at the top or bottom, each patch of four contiguous segments represents results for a specific Grashof value. The results are summarized in Table 3.6.

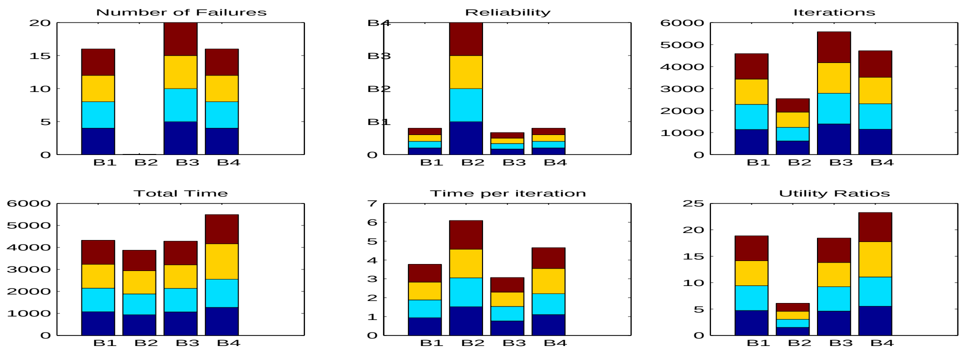


Fig. 3.7. Observed performance and the computed utility ratios of the four base linear solution methods at the sample simulation.

Metric	Base Methods				B1-B4	Composites			
	B1	B2	B3	B4	Total	C1	C2	C3	CU
Cumulative Performance Data over 24 simulations									
Time in 10^4 secs; Linear Iterations(Its) of the order 10^4									
Time(Linear)	2.6	2.4	2.5	3.1	10.6	2.8	3.5	1.4	1.3
Time(Nonlinear)	2.7	2.5	2.6	3.2	11	2.9	3.6	1.5	1.45
Its(Linear)	2.7	1.5	3.4	2.8	10.4	3.0	3.1	.94	.94
Its(Nonlinear)	125	75	140	120	460	76	76	65	56
Failures	98	10	120	100	328	0	0	0	0
Failure Rate(%)	78	13.3	85.7	83.3	7.4	0	0	0	0
Average Performance Data over 24 simulations: Time given in seconds									
	B1	B2	B3	B4	Mean	C1	C2	C3	CU
Time(Linear)	1,083	,1000	1,041	1,291	1,104	1,166	1,458	583.3	541.6
Time(Nonlinear)	1,125	1,041	1,083	1,333	1,145	1,208	1,500	625	604
Its(Linear)	1,125	625	1,416	1,166	1,083	1,250	1,291	391.6	391.6
Its(Nonlinear)	5.2	3.1	5.8	5	4.7	3.1	3.1	2.7	2.3
Failures	4.0	.41	5	4.1	3.41	0	0	0	0

Table 3.6. Summary of performance measures for 24 simulations of driven cavity flow on a 128 by 128 mesh. The column labeled (B1-B4) gives the cumulative and mean values across four base solution schemes. The optimal composite is 49% faster than the average execution time of the base methods.

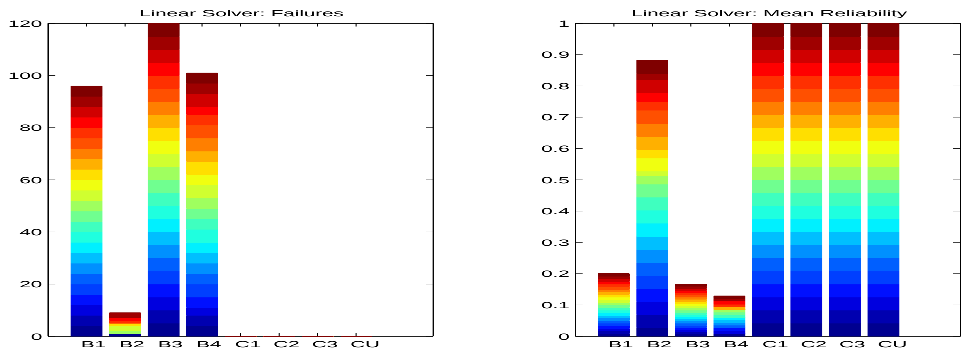


Fig. 3.8. The number of failures and the mean reliability of the linear solver for each problem instance in a driven cavity problem with mesh size 128 by 128.

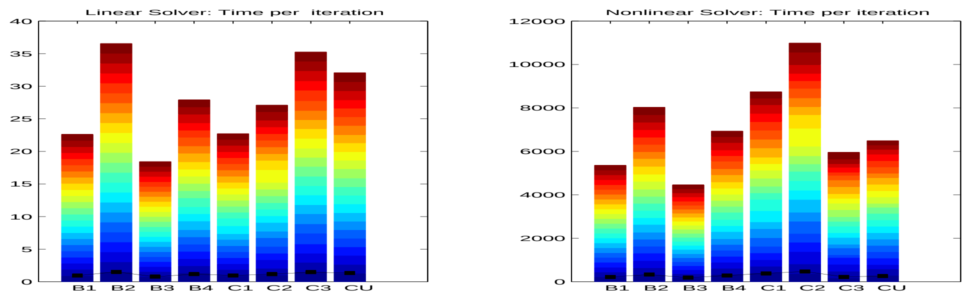


Fig. 3.9. Time per iteration (in seconds) for linear and nonlinear solution for each problem instance in a driven cavity problem with mesh size 128 by 128 ; the plot shows the mean time per iteration over all simulations.

Figure 3.8 shows the number of failures and the reliability of the linear solver; each base methods suffers some failures, while all composites are robust. Figure 3.9 shows the time per iteration, which is nearly invariant across simulation points for a given method. As expected, the composites typically require greater time per iteration than the least expensive base method, B3. Figures 3.10 and 3.11 show the total iteration counts and time for linear and nonlinear solution; the latter is the the total application time. In these figures, the plots that interlace the stacked bars show cumulative values at the completion of simulations corresponding to each Grashof number. All composites show a reduction in total nonlinear iterations as a consequence of improved linear solution; CU requires only 75% (63%) of the nonlinear (linear) iterations required by the fastest base method, B2. The composite based on the utility ratio, CU, incurs the least total linear solution time, which is approximately 56% of the time required by the best base method B2. The linear solution time comprises on average 96% of the nonlinear solution time (total simulation time) and consequently, CU requires approximately 58% of the total simulation time of the best base method B2.

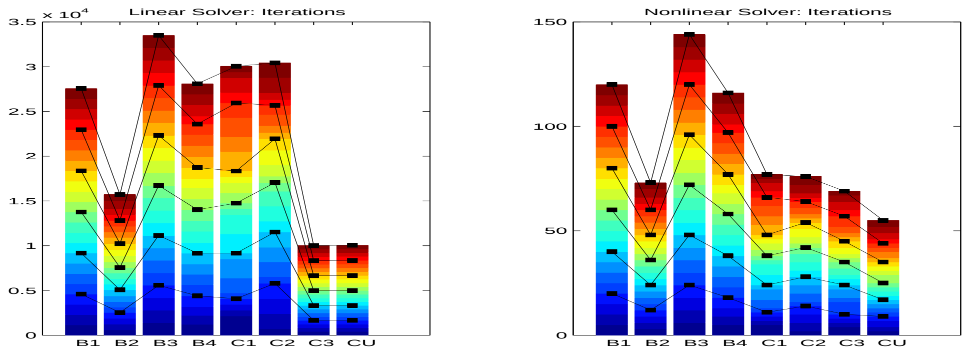
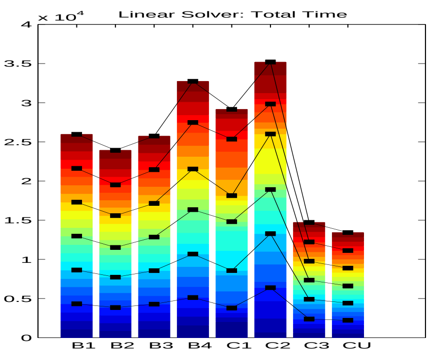


Fig. 3.10. Iteration counts for linear and nonlinear solution; the plots highlight cumulative values after simulations for each Grashof number.



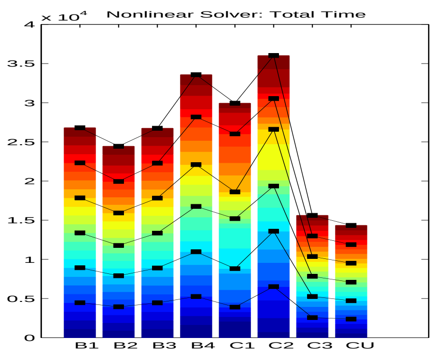


Fig. 3.11. Total time for linear and nonlinear solution (in seconds); the plots highlight cumulative values after simulations for each Grashof number.

3.4.4.1 Parallel Performance on a 128×128 mesh

We now consider results of experiments using our parallel composite solvers. We used a 128×128 mesh for discretizing the driven cavity flow model with pseudo-transient continuation. At each nonlinear iteration, the discretized sparse linear system had rank 65,536 with approximately 1,302,528 nonzeros. We set the maximum number of linear iterations to 200.

As the preconditioner we used the restricted additive Schwarz method (RASM) [17] with varying subdomain solvers and varying degrees of overlap (e.g., overlap of 1 is denoted by RASM(1)). We combined various Krylov methods [27, 41], subdomain solvers, and degrees of overlap to get the following set of base solution methods: (B1) GMRES(30), RASM(1), Jacobi subdomain solver; (B2) GMRES(30), RASM(1), SOR subdomain solver; (B3) TFQMR, RASM(3), no-fill ILU subdomain solver; and (B4) TFQMR, RASM(4), no-fill ILU subdomain solver. We composed a sample set of the following problem instances of Grashof numbers and lid velocities: (700:85), (800:83), (900:80), (950:73). We distributed the matrix across multiple processors and used each of the four base methods to solve the sparse linear systems that were generated by these problem instances. We obtained the resulting utility ratios by computing the ratio of the linear time per iteration to the reliability of each method and then taking the average over the four problems in the sample set. Based on metrics obtained from the sample set, we formed the optimal composite, CU, which contained the following sequence of base methods: 3,4,2,1. We also created 3 arbitrary composites using random sequences, denoted by C1 (3,2,4,1), C2 (2,1,4,3) and C3 (4,1,2,3). We performed these experiments on a fixed size problem while varying the number of processors from 2, 4, 8 to 16. We ran 24 simulations, using six different Grashof numbers, [700, 750, 800, 850, 900, 950], and four lid velocities, [73, 80, 83, 85].

Our results show that the composites achieve improved reliability and good parallel performance. Method B1 never converges, while other base methods have varying degrees of failure. The composites show near ideal reliability. Furthermore, the simulation time with the optimal composite, CU, is approximately 40% – 48% of the worst base method. We also consider the speedup and efficiency of the solvers on different processors. We use T_1 as the best estimate of the time for the full simulation on one processor using the fastest base method. Thus, we set T_1 to two times the time for method B3 on two processors (it is not meaningful to use the additive Schwarz method for a single processor with a single subdomain). We calculate the speedup $S = \frac{T_1}{T_p}$, where T_p is the observed time on p processors; the corresponding efficiency is calculated as $E = \frac{S}{p}$. The results shown in Figure 3.12 indicate that the speedups of the composites CU, C1, and C3 are almost as good as that of the best base method, with near ideal efficiencies. Table 3.7 gives the data for other parameters such as running time, number of failures, etc.

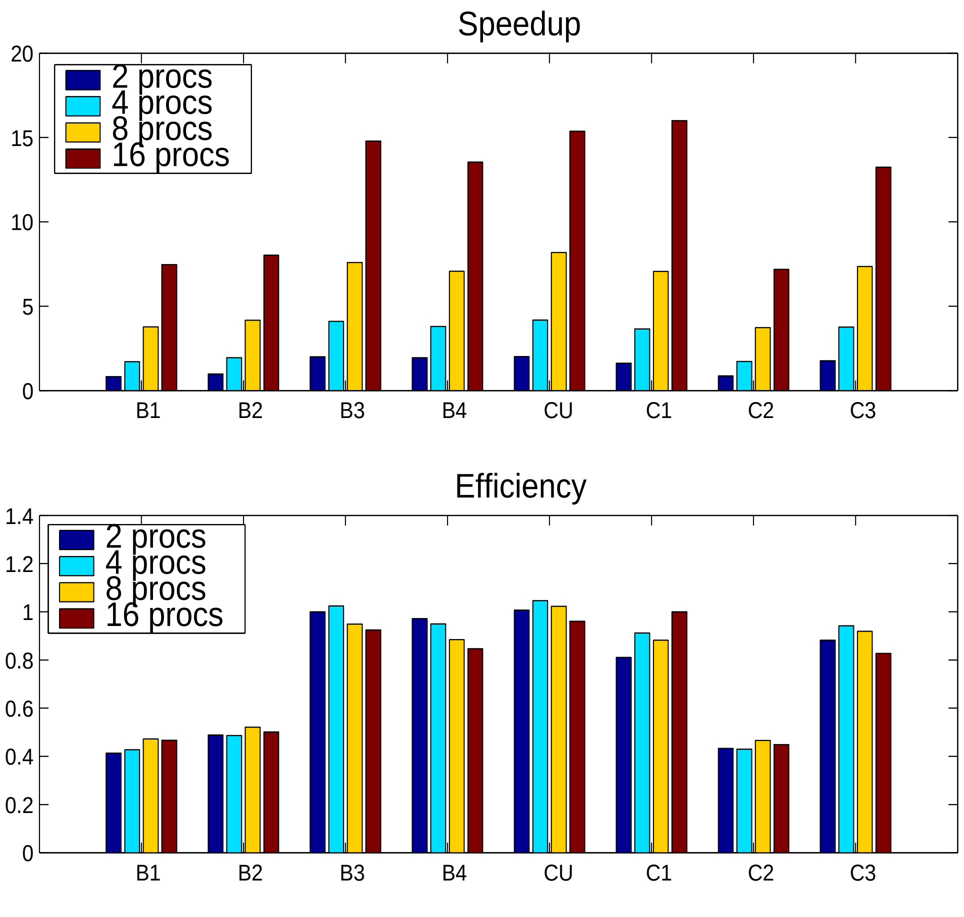


Fig. 3.12. Parallel performance of the driven cavity application on 2 – 16 processors.

Metric	Base Methods				B1-B4	Composites
	B1	B2	B3	B4	Total	CU
Cumulative Performance Data over 24 simulations; Time in seconds						
Time (Linear)	470.17	434.78	243.58	262.05	1410.58	224.90
Time (Nonlinear)	521.43	472.79	259.41	278.31	1531.94	240.71
Linear Iterations	113,600	80,719	20,813	21,441	236,572	19,367
Nonlinear Iterations	568	422	140	141	1271	123
Number of Failures	568	398	36	34	1036	0
Failure Rate(%)	100	94.3	25.7	24.1	5.8	0
Efficiency	.47	.52	.95	.88		1.02
Speed Up	3.11	4.16	7.59	7.07		8.18

Average Performance Data over 24 simulations; Time in seconds						
	B1	B2	B3	B4	Mean	CU
Time (Linear)	19.59	18.11	10.14	10.9	14.6	9.3
Time (Nonlinear)	21.72	19.7	10.8	11.59	15.9	10.02
Linear Iterations	4733.3	3363.3	867.2	893.4	2464.3	807
Nonlinear Iterations	23.66	17.5	5.83	5.87	13.23	5.1
Number of Failures	23.6	16.58	1.5	1.41	10.8	0

Table 3.7. Summary of performance measures for 24 simulations of driven cavity flow on a 128 by 128 mesh on 8 processors. The column labeled (B1-B4) gives the cumulative and mean values across four base solution schemes. The optimal composite is 36% faster than the average execution time of the base methods. Speedup and efficiency values are based on execution times for one processor implementation with the best base method (B3).

3.5 Summary

We have developed a combinatorial model for analyzing and developing composite solvers. We have developed algorithms for constructing optimal composites when the failure rates of base methods are mutually independent and we extended them to apply to more general situations when the failure rates of methods are corelated. We have implemented and tested our composites using a test-suite of linear systems spanning several applications, and on sequential and parallel implementation in a model CFD code for driven cavity flow. Our results on uniprocessors and multiprocessors demonstrate that composites achieve near ideal reliability. Furthermore, the improved reliability can reduce the execution time for an application by reducing the number of nonlinear iterations. More specifically, for our largest driven cavity flow experiment on a 128×128 grid (summarized in Table 3.6) our optimal composite requires only 56% (42%)of the execution time of best(worst) base method. Our optimal composite requires 49% of the average execution time across the base methods (column "B1-B4" in Table 3.6). Our parallel composite solvers also exhibit near ideal speedups and efficiencies because they successfully retain the scalability of the base methods while improving the reliability (see Table 4.1). It is interesting to observe that composite solvers showed improved performance even when the convergence of the nonlinear solution was not greatly affected by the linear system solution. We conjecture that the benefits would increase when the nonlinear convergence is more dependent on the linear solution.

Chapter 4

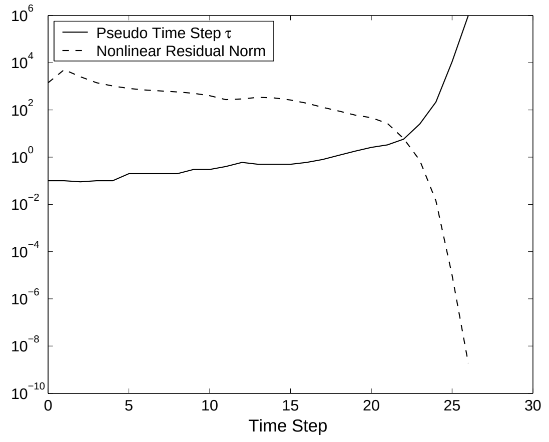
Adaptive Solvers

In this chapter, we introduce our second type of multimethod solvers, namely adaptive solvers. Adaptive solvers are useful in simulations which require the solution of a series of linear systems with changing numerical properties. The goal of adaptive solvers is to reduce the execution time of the simulation by dynamically selecting the most appropriate base method to match the characteristics of the current linear system. In contrast to composite solvers, adaptive solvers use one base method per linear system. In this chapter, we present our adaptive heuristics [13, 37] for selecting appropriate solvers and report on their performance in CFD applications.

As we have observed in the previous chapter, the execution times for applications based on nonlinear PDE-based equations is dominated by the time for solving linear systems generated at each nonlinear iteration. The numerical characteristics of the linear systems may change in course of the simulation. For example, the use of pseudo-transient continuation generates linear systems that get progressively difficult to solve. In simulations involving spatial or temporal discontinuities, such as the occurrence of shock waves, the changes in the linear systems might reflect the changing nature of the application. In these cases where the linear systems characteristics are not constant, changing the linear solver can potentially improve the performance.

An example of changing linear systems is demonstrated in Figure 4.1. The graphs plot the performance data from the simulation of a driven cavity flow application. The left-hand graph in Figure 4.1 plots the growth of the pseudo time step and the convergence of the nonlinear residual norm. The right-hand graph depicts the time spent in solving linear systems during each time step, using a fixed linear solver. As the pseudo-time step increases, the corresponding linear system becomes more difficult to solve. This is reflected by the increase in the linear solution time. Our goal in building adaptive solvers is to restrict or lower the growth in the linear solution time, by dynamically switching linear solvers and thereby reducing the time required for the total simulation.

An efficient adaptive solver must be able to accurately determine when to change linear solvers. Our adaptive heuristics monitor "indicators" to detect the switching point. Some common examples of indicators are increase in linear (nonlinear) solution time, rate of change of linear iterations, convergence rate, etc. It has been observed that designating a single parameter as the indicator is not a good strategy. A judicious combination of several indicators generally leads to better results.



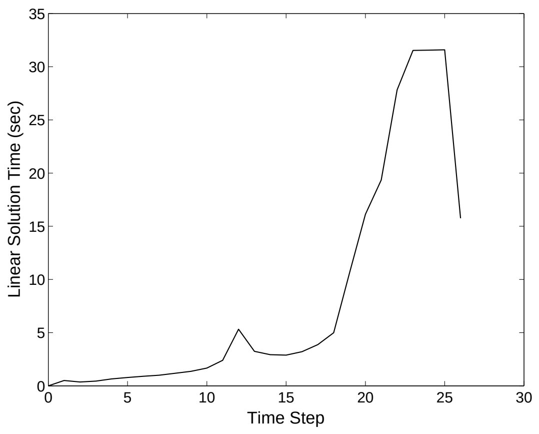


Fig. 4.1. Left-hand graph plots growth of pseudo time step size and convergence of residual norm for the pseudo-transient continuation version of the driven cavity flow. Right-hand graph plots time spent in solving linear systems during each nonlinear iteration; this illustrates that different amounts of work are needed throughout the simulation to solve the linear systems.

4.1 A Mathematical Model for Adaptive Method Selection

In this section, we will demonstrate how adaptively selecting linear solvers can reduce the execution time. Let there be a set of k distinct linear solvers represented by $M_1, M_2, \ldots, M_k$ where t_i is the time per iteration for M_i. At each nonlinear iteration j, the number of linear iterations executed by M_i is given by I_j^i. If M_i was the only linear solver used, the total linear solution time would be $T_i = t_i \sum_{j=1}^{i=n} I_j^i$. We now define the following terms which are relevant for developing our selection procedure.

- **Minimum Method** For each non-linear iteration j, the minimum method, M_m^j, is such that $t_m I_j^m = \min_{i=1}^{i=n}\{t_i I_j^i\}$.
- **Crossover** If the minimum method of a non-linear iteration j is different from the minimum method of the non-linear iteration $j-1$, then j is a crossover. That is $M_m^{j-1} \neq M_m^j$. For ease of calculation, we define iterations 0 and n as crossovers.
- **Phase** A phase is the range of non-linear iterations between two crossovers. A phase, $P(a, b)$ means that:

$$M_m^{a-1} \neq M_m^a, \text{ and}$$
$$M_m^a = M_m^x; a \leq x \leq b, \text{ and}$$
$$M_m^b \neq M_m^{b+1}$$

Consider a simulation with crossovers at iterations, $0, c_1, c_2, \ldots c_{l-1}, n$, corresponding to phases $P(0, c_1), P(c_1, c_2), \ldots, P(c_{l-1}, n)$. We prove that selecting the minimum method at each phase gives an optimal selection strategy. Let the total linear solution time be $T = t_x \sum_{j=1}^{i=n} I_j^x$, where x can be any method in the set $M_1 \ldots M_k$. Since T is a summation of positive quantities, therefore T can be minimized by minimizing each $t_x I_j^x$. This is equivalent to selecting $t_x I_j^x = \min_{i=1}^{i=n}\{t_i I_j^i\}$, for each non-linear iteration j, i.e. the definition of minimum method. By definition, the minimum method remains constant within a phase, and the result follows.

4.2 Adaptive Solvers

Adaptive solvers are defined by the heuristic employed for method selection. Ideally, the selection heuristic should be applied at each nonlinear iteration. However this could be potentially expensive due to the added overheads of method selection. The switching costs can be amortized by observing parameters over a fixed number of nonlinear iterations, defined as a "window". The heuristic is invoked at the end of each window.

Adaptive heuristics can be sequence-based, that is, rely on a predetermined sequence of linear solvers. The linear solvers are arranged in increasing order of "strength" as defined by the time taken per iteration. We assume that a solver which requires more time per iteration is stronger, i.e., more likely to achieve convergence. The adaptive solver could potentially "switch up" to a more powerful linear solver or "switch down" to a less powerful one in course of the simulation. Only three methods are compared in this class of heuristics; the current method and the methods preceding and succeeding it in the sequence.

Adaptive heuristics can also be nonsequence-based where all the methods in the set are compared. This class of adaptive methods requires more time for selection, but they have greater flexibility. Sequence-based methods are used in simulations where linear systems get progressively difficult or easier and nonsequence-based strategies are used in applications where such monotonic pattern is missing and can be used to manage highly irregular behavior. We have developed the following adaptive heuristics.

4.2.1 Adaptive Method 1 (Based on Convergence Rates)

This heuristic is based on the following three indicators measured over a window w of nonlinear iterations:

- The convergence rate R_L of previous linear solutions defined as, $R_L = \frac{1}{w}\sum_{k-w<i<=k} \frac{r_{i,n_i} - r_{i,0}}{n_i}$, where k is the current nonlinear iteration; $r_{i,j}$ is the relative residual of the linear system solution at the i-th time step and j-th linear iteration; and n_i is the total number of linear iterations performed at the i-th time step.

- The convergence rate R_N of the nonlinear solution, $R_N = \frac{\|f_k\| - \|f_{k-w}\|}{\|f_{k-w}\|}$, where $\|f_k\|$ is the function norm at the k-th time step.

- The rate of increase in the number of iterations for linear system solutions, $R_I = \frac{n_k - n_{k-1}}{n_{k-1}}$, where n_i is the number of linear iterations performed at the i-th time step.

These indicators are evaluated at each window and the actual switching is based on user specified parameters λ, w, β as follows,

- If $|R_N + R_L| \leq \lambda$, use the same solver in the next time step.

- If $R_N + R_L > \lambda$, select a less powerful solution method (switch down).

- If $R_N + R_L < -\lambda$ or $R_I > \beta$, select a more powerful solution method (switch up).

4.2.2 Adaptive Method 2 (Based on Execution Time)

The goal of adaptive solvers is to reduce the execution time, therefore, linear solution time is a natural choice for an indicator. Let method M_j be used to solve linear systems in window i. The corresponding number of linear iterations be $I(i,j)$ and the time per iteration be t_j. Now the time taken to solve the linear systems generated in window i is $T(i,j) = I(i,j)t_j$. It is intuitive that the adaptive solver will switch up when $T(i,j) > T(i,j+1)$ or switch down when $T(i,j) > T(i,j-1)$. We approximate the number of linear iterations that would have been by required method M_{j+1} to solve the same linear system as $\alpha I(i,j)$ and by method M_{j-1} as $\beta I(i,j)$.

Let $T(i-1,j)$ and $T(i,j)$ be the observed times for all the linear system solutions in windows $i-1$ and i. If each nonlinear iteration generates a progressively difficult linear system, then $T(i,j+1) > T(i-1,j+1)$. The switching heuristic can now be expressed as follows,

Method $j+1$ is selected for window $i+1$ if $T(i,j) > T(i,j+1)$.

$$T(i,j) > T(i,j+1) > T(i-1,j+1)$$

$$\text{implies, } I(i,j)t_j > \alpha I(i-1,j)t_{j+1}$$

$$\text{implies, } \frac{I(i,j)}{I(i-1,j)} > \frac{t_{j+1}}{t_j}\alpha$$

Method $j-1$ is selected for window $i+1$ if $T(i,j) > T(i,j-1)$.

$$T(i,j) > T(i,j-1) > T(i-1,j-1)$$

$$\text{implies, } I(i,j)t_j > \beta I(i-1,j)t_{j-1}$$

$$\text{implies, } \frac{I(i,j)}{I(i-1,j)} > \frac{t_{j-1}}{t_j}\beta$$

4.2.3 Adaptive Method 3 (Based on Polynomial Interpolation)

Adaptive heuristics are based on accurate prediction of future performance data. The trend of the indicators can be estimated by fitting a function to the known data points. This is the motivation for designing adaptive heuristics based on polynomial interpolation. The value of the indicator associated with method M_j at nonlinear iteration t can be calculated using an n degree polynomial as follows, $P_n^j(t) = \sum_{i=0}^{i=n} a_i^j t^i$; where a_i^j are coefficients of the polynomial. The choice of indicators may vary according to the nature of the problem. In our experiments, the indicator was taken to be the time required to complete the simulation. We first estimate the number of nonlinear iterations n required to achieve convergence, and then calculate the total time required to solve linear systems that will be generated from the current nonlinear iteration to nonlinear iteration n. We select the method requiring the least solution time.

4.3 Empirical Results

We evaluated the performance of the three adaptive heuristics on the following CFD applications–the compressible Euler equation for airflow on a ONERA M6 wing on the PETSc-FUN3D implementation and the driven cavity flow. We performed the experiments on the Jazz cluster at the Argonne National Laboratory, which has a Myrinet 2000 network and 2.4 GHz Pentium Xeon processors with 1-2 GB of RAM (detailed description in Chapter 3). We report our observations and results in this section.

4.3.1 FUN3D Code for Euler Equations on Unstructured Grids

FUN3D is a simulation code for compressible and incompressible Euler equations originally developed by W. K. Anderson of the NASA Langley Research Center [3]. This code uses a finite volume discretization with a variable order Roe scheme on a tetrahedral, vertex-centered unstructured mesh [3].

Gropp et al. have incorporated PETSc functionality into the FUN3D code [31]. The parallel PETSc-FUN3D code has been used to solve above application of subsonic flow for an ONERA M6 wing and the results show improvement in scalability and the performance. The unstructured mesh was partitioned using MeTiS [34] and the set of nonlinear equations were solved using the ψNKS [35] family of implicit solution schemes. ψNKS uses inexact Newton's method to solve the nonlinear equations. The linear systems generated at each iterative step are solved by combining a Schwarz preconditioner with a Krylov iterative method. Pseudo-transient continuation is employed for ensuring robustness in problems involving shocks or turbulence.

We have used examples from PETSc-FUN3D as test applications for our multi-method adaptive solvers. We used Newton's method for solving the nonlinear systems and our linear solvers were variations of Krylov methods. Neilsen et al. [38] have demonstrated that use of Newton-Krylov methods within the FUN3d code maintains the accuracy of the results while lowering the execution time. In addition, we also validated the results of the relevant physical quantities, such as pressure along the wing span, with the authors of the PETSc-FUN3D code (through personal communication).

4.3.2 Experiments on PETSc-FUN3D

The PETSc-FUN3D code is used to implement a compressible flow for an ONERA M6 wing. The application is associated with a shock discontinuity. Initially the PDEs are discretized using first-order discretization, but once the shock position has settled down second order discretization is applied. This change in discretization affects the nature of the linear system. Our experiments were limited to the available three dimensional problem instance designated as 1Grid. This represented a matrix of rank approximately 1.8×10^6 with 1×10^9 nonzeros. The simulations were done using 4 processors. We used Block Jacobi preconditioner and varied the Krylov solvers and the subdomain preconditioners. The base methods used were (B1) GMRES with SOR, (B2) TFQMR with ILU(0), (B3) BiCG with ILU(0)and (B4) FGMRES with ILU(1). The adaptive methods were based on the heuristics discussed earlier. In addition we also evaluated the performance of an optimal adaptive solver by forcing the solver to adapt at each phase, as per the mathematical formulation. It is generally not possible to automate such an optimal adaptive, but the results give an estimate of extent of potential improvement. Figure 4.2 shows the convergence rates of the base and adaptive solvers.

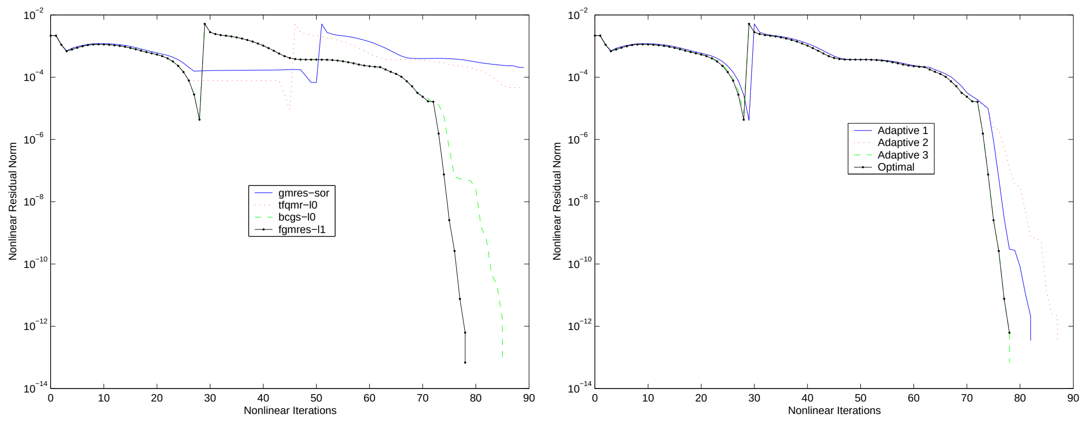


Fig. 4.2. Convergence of residual norm for 1Grid. Left-hand graph plots the curve for the base solvers. Right hand graph plots the curve for the adaptive solvers.

Observe that the convergence rate of adaptive heuristic with polynomial interpolation is similar to that of the optimal adaptive solver. Adaptive method 3 converges faster than the other two adaptive methods. This is because the polynomial interpolation strategy has the flexibility of selecting from any base method rather than being confined to its immediate neighbors in the sequence. The superiority of adaptive method 3 is also evident in Figure 4.3 which shows the time taken to solve linear systems. Adaptive solvers 1 and 2 take significantly more time, while the plot for adaptive 3 is close to the optimal adaptive solver. Table 4.1 compares the performance of the base and adaptive methods. The execution time of the adaptive method based on polynomial interpolation is within one percent of the execution time of the best base method B4 and 32% better than that of the worst one B2. The execution time of the optimal adaptive is 7% better than that of the best base method and 42% better than execution time of the worst one. Execution time taken by any adaptive solver is lower than or equal to the average time taken by the base methods as shown in Table 4.1.

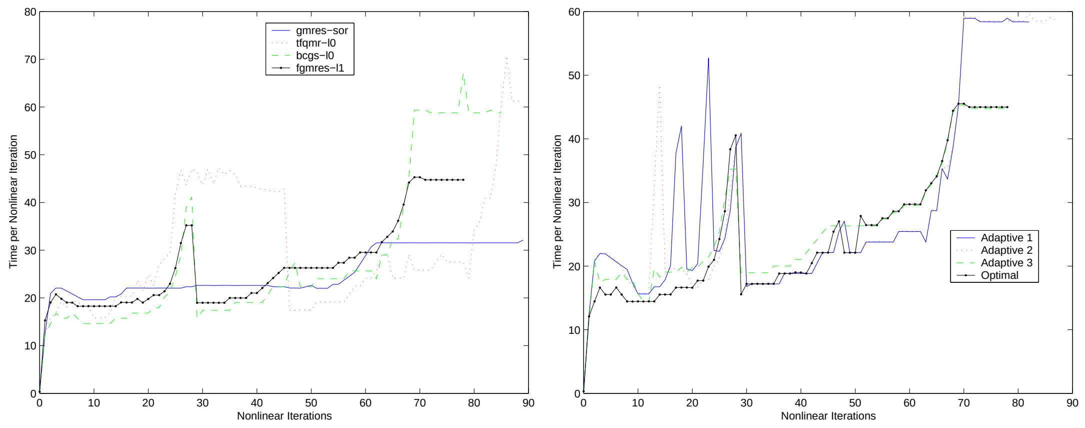


Fig. 4.3. Time spent in solving linear systems during each nonlinear iteration. Left-hand graph plots the curve for the base solvers. Right hand graph plots the curve for the adaptive solvers.

Metric	Base Methods				B1-B4	Adaptives			
	B1	B2	B3	B4	Mean	A1	A2	A3	AO
Cumulative Performance Data									
Time in seconds; Linear Iterations of order 10^3									
Solution Time	2268	2790	2535	2123	2420	2461	2708	2103	1961
Linear Iterations	2.56	1.45	1.16	1.25	1.6	1.25	1.35	1.24	1.04
Nonlinear Iterations	89	89	85	78	85.2	82	87	78	77

Table 4.1. Summary of performance measures of adaptive solvers for 1Grid. The column labeled mean gives the average performance over four base methods. The third adaptive method (A3) performs 13% better than the average execution time of the base methods.

4.3.3 Experiments on Driven Cavity Flow

The PETSc implementation of the model Driven Cavity flow (described in Chapter 3) is solved using pseudo transient continuation. This is an example of a simulation where as a result of using a globalization technique, the linear systems become more difficult to solve at later nonlinear iterations. Therefore adaptive solvers based on incrementally switching up perform well in such simulations.

We experimented on a suite of 30 problem instances on a 128×128 mesh with Grashof numbers [400, 450, 550, 580, 600, 650] and lid velocities [10, 13, 15, 20, 25]. The base methods were (B1) BiCG with ILU(0), (B2) GMRES with ILU(0), (B3) TFQMR with ILU(0), and (B4) BiCG with ILU and drop threshold 10^{-4}.

Figure 4.4 depicts the simulation with Grashof 600 and lid velocity 20. The convergence rates of all the base methods and adaptive solvers are equivalent. However, the adaptive solvers are successful in maintaining a lower value of time per linear solver as shown in the right hand graph. Figure 4.5 gives the cumulative time over 30 problem instances. Each colored patch represents a simulation instance specified by a Grashof number and lid velocity. The segments are arranged in increasing order of Grashof number and lid velocities (per Grashof number) with the bottom representing 400:10 and the top 650:25. Thus, starting at the top or bottom, each patch of five contiguous segments represents results for a specific Grashof value. The execution time of sequence-based adaptive solvers which depend on the convergence rate or linear solution time are 17% better the best base method BiCG with drop threshold and 50% better than the worst method GMRES with ILU(0). The nonsequence-based approach based on polynomial interpolation fares less well compared to the other adaptive solvers. It is 6% better than the best base method and 44% better than the worst one.

4.4 Summary

We have developed adaptive strategies that can dynamically select the best suited linear solver for a given linear system. Our experiments demonstrate that use of adaptive solvers can significantly reduce the execution time for large-scale modeling applications. Our adaptive solvers can reduce the application time to be 39% better than the average performance of four base solvers (see Table 4.2) and by 50% of the worst method. These results are indeed promising; however it is important to note significant differences in the performance of our three adaptive strategies. For example, simpler strategies based on linear and nonlinear convergence rates or execution time have lower associated overheads and are successful in adapting to applications requiring solvers of gradually increasing (or decreasing) strength. Our adaptive solver based on polynomial-interpolation incurs higher overheads but is more successful in adapting to applications where the variations in the numerical properties of linear systems are more complex.

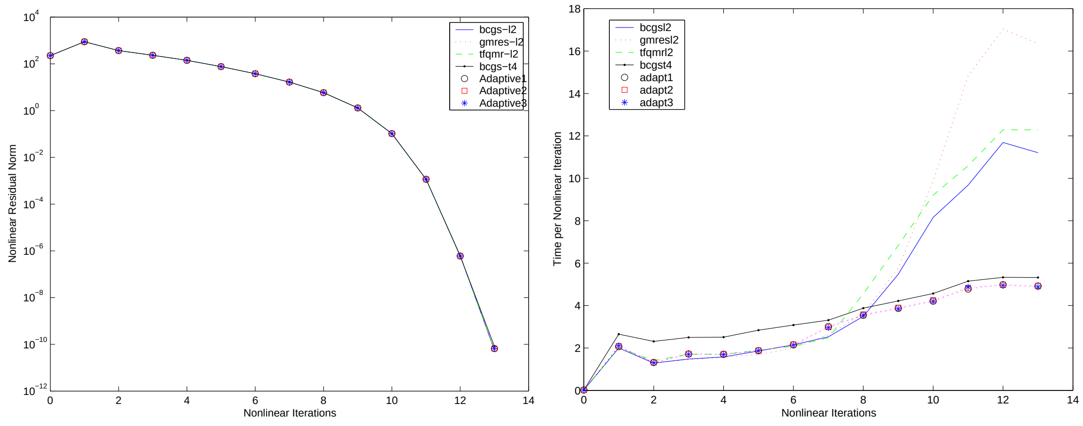


Fig. 4.4. Left-hand graph plots convergence of residual norm for the pseudo-transient continuation version of the driven cavity flow(600:20). Right-hand graph plots of time spent in solving linear systems at each nonlinear iteration.

Metric	Base Methods				B1-B4	Adaptives		
	B1	B2	B3	B4	Total	A1	A2	A3
Cumulative Performance Data over 30 simulations								
Solution Time (seconds)	1824.5	2372.7	2130.6	1409.7	7737.5	1195.6	1189.4	1311.5
Linear Iterations	2987	7109	3438	1122	14656	1116	1116	1487
Nonlinear Iterations	450	450	450	450	1800	450	450	450
Average Performance Data over 30 simulations								
	B1	B2	B3	B4	Mean	A1	A2	A3
Solution Time (seconds)	60.81	79.09	71.02	46.99	64.45	39.85	39.6	43.7
Linear Iterations	99.56	236.96	114.6	37.4	122.13	37. 2	37. 2	49.56
Nonlinear Iterations	15	15	15	15	15	15	15	15

Table 4.2. Summary of performance measures of adaptive solvers for 128 by 128 driven cavity flow application. The column labeled (B1-B4) gives the cumulative and mean values across four base solution schemes. The adaptive solvers are 39% to 32% faster than the average execution time of the base methods.

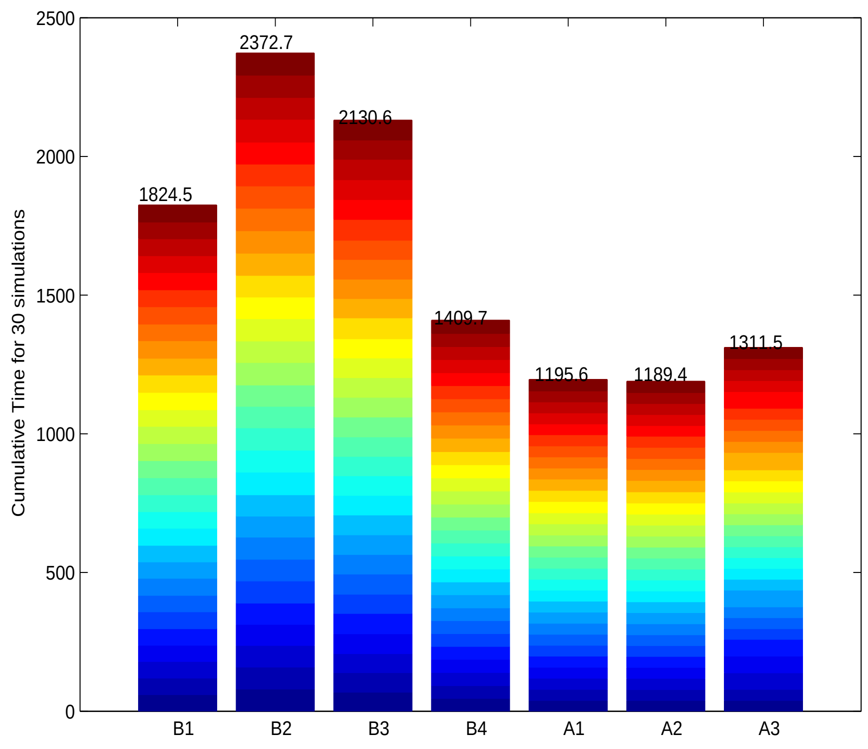


Fig. 4.5. Cumulative time required by simulations on 30 problem instances on driven cavity application

Chapter 5

Toward Multimethod Solver Components

In this chapter, we consider the development of a software system that can automatically instantiate a composite or adaptive solver in response to a specific user request. We start with an overview of the system requirements and the design issues involved in creating such software. We then present a high-level description of our multimethod software architecture, followed by descriptions of two alternate implementations. We end with remarks on our contributions and plans for further development.

5.1 Design Requirements

Our multimethod software system should be capable of serving the needs of both the application community as users of multimethod solvers and the scientific computing community as developers of new techniques for multimethod solver creation. In this section, we summarize the key features of a multimethod software architecture which would meet the needs of both communities.

We expect the user community to consist of scientists and engineers who wish to solve large-scale modeling applications. From their perspective, ease of use is a primary issue. Ideally, our multimethod software system should be no harder to use than the standard linear solver packages. Additionally, the software should also allow the user to specify details that could influence the instantiation of multimethod schemes and their performance in specific applications. For example, the user might need to specify the desired level of reliability of a composite or wish to add new solvers (not defined in the underlying linear solver packages) and use it towards building adaptive or composite solvers for his application. The multimethod software system should be able to efficiently accommodate such user-specific requests.

We expect the scientific computing community to be concerned primarily with the efficient implementation of multimethod solvers. Multimethod solvers are composed of several individual base solvers which can differ in the language they are implemented, the design of their data structures, etc. Our multimethod software system should be capable of effectively bridging the difference in the various implementations to provide access to the majority of available packages that implement the base methods. Another issue is that of the reuse of base solvers. The same linear solver can be used in different multimethod solver instances. Providing a uniform abstract interface to base method implementation (for example, through extra interface wrappers) will ensure easy generation of multimethod instances and promote reusability and portability.

Building an ideal multimethod software system also involves implementing data collection, storage and retrieval techniques. The multimethod software system should be equipped with functionalities for efficiently monitoring the performance of the solvers it provides to the users. The system should be associated with a database that would store information regarding individual base methods such as their intrinsic computational costs or observable metrics such as failure rates, convergence, etc. after each execution. Such data will be useful in determining the arrangement of the base methods in composite or adaptive solvers. The database could also be used to store specific instances of multimethod solvers. Future instances of the same application could potentially utilize existing multimethod solvers thus saving the time for constructing new adaptive or composite solvers.

The study of multimethod solvers is a relatively new area and researchers from the scientific computing community would be interested in exploring further extensions to this project. The multimethod software system should be extensible and allow the user to experiment by specifying different combinations of base solvers or by including new multimethod algorithms. Such functionality in conjunction with those for performance data collection will facilitate further developments in the use and construction of multimethod schemes.

5.2 Software Architecture for Multimethod Solvers

In this section, we propose a software architecture for a multimethod solver system that addresses the design requirements discussed above. The software architecture provides a platform-independent, object-oriented characterization that can be used to hide lower level implementation details and used towards providing specific implementations of multimethod solver instances.

Consider a multiprocessor system consisting of computing nodes each comprising a set of high performance CPUs and communicating through fast interconnects such as Myrinet. Assume that several software services are available on this system and the user can access them through interaction with a suitable runtime environment. Our software architecture is designed to provide a multimethod solver service in such a system. This service would be registered through the runtime system and accessed by the user through interaction with the runtime system.

Our design is based on a client-server model where the application (client) can request a specific multimethod solver (service) from the runtime system. This request is then transmitted to our multimethod software (server) which then provides the service. Figure 5.1 depicts the conceptual structure of our multimethod software and its function in relation to the other computational entities in the system. In general terms, the main computational entities involved include the following.

- A central runtime system through which applications can request and receive appropriate multimethod solvers. The runtime system also provides "yellow pages", i.e., directory services that can be used to identify base linear solver packages.
- Applications that request multimethod solvers from the software system.
- Linear solver packages that provide solvers that can be used to construct composite or adaptive multimethod solvers.
- The multimethod software system, composed of implementations for multimethod algorithms, functions for performance monitoring and a database that stores information about the performance of linear solvers in different applications. The multimethod software system creates multimethod instances to be used by the requesting application and optionally evaluates the performance of multimethod solvers in the application.

Consider an application requiring a linear solver for obtaining the solution of a sparse linear system. The application would send a request to the runtime system along with information regarding the type of solver needed, for example a composite, and other user data, such as the structure of the matrix, desired reliability, etc. This request and information is transmitted to the multimethod software. The multimethod algorithms in the software use this information and look up the database to determine the appropriate set of linear solution methods to be used. A request is then sent from the multimethod software system to the runtime system to locate and use these solvers. The linear solvers are arranged in the increasing order of their utility ratios, as obtained from the database, to form a composite solver instance tailor-made for the requesting application. This ends the "creation service". Next the application accesses the multimethod solver through a "user request" via the runtime system. Figure 5.2 shows the main steps of the process. After the linear system is solved, relevant performance data is collected in the multimethod software database. The database can also store the multimethod solver instance which can be reused for solving similar linear systems.

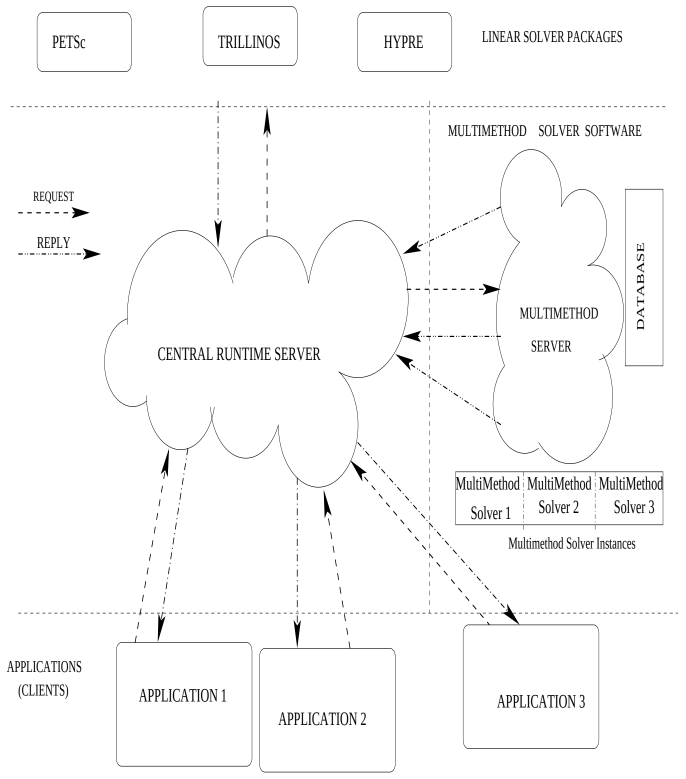


Fig. 5.1. Multimethod Software Architechture.

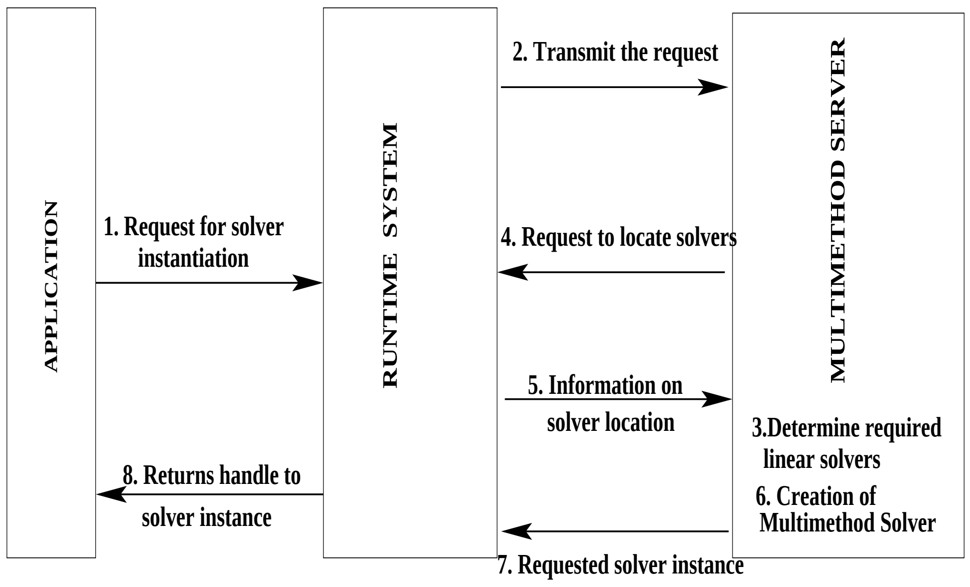

APPLICATION
RUNTIME SYSTEM
MULTIMETHOD SERVER
1. Request for solver instantiation
2. Transmit the request
3.Determine required linear solvers
4. Request to locate solvers
5. Information on solver location
6. Creation of Multimethod Solver
7. Requested solver instance
8. Returns handle to solver instance

Fig. 5.2. The main steps in processing a request to create a specific multimethod solver instance.

5.3 An Implementation Using PETSc

A simple way to implement the architecture defined above is through extension to PETSc, the Portable Extensible Toolkit for Scientific Computing(PETSc) [5, 6]. We incorporated extensions to provide multimethod solver services as follows. The base methods were implemented using PETSc functions for linear solvers. The multimethod composite or adaptive solvers were represented by suitable algorithms to invoke the base methods. Then, the section defining the linear solver in the original application was replaced by a call to functions implementing multimethod solvers.

This implementation is a very simple and static representation of the multimethod software architecture described earlier. Even in this simple model many design issues were addressed. We were able to include multimethod solvers into the application code without any significant modification. The use of abstract interfaces in PETSc provide uniform access to a variety of base methods. This promoted flexibility of algorithm selection and also allowed the reuse of base methods in different multimethod solvers instances. The performance data was collected with in-built PETSc monitoring functions.

However, this simple implementation entailed several limitations. The set of solvers was limited to only those included in PETSc. In the absence of a runtime server, each multimethod solver instance had to be statically built for each application and were coded as separate functions in the application. This resulted in a monolithic code which tended to get unwieldy with the addition of every new multimethod solver instance. The performance metrics of base methods had to be individually acquired and processed and there was no efficient mechanism for storing past results, especially across applications.

Our goal is to build a multimethod solver with a much wider scope that extends beyond these limitations. Our initial implementation has provided us with the valuable understanding of several issues concerning multimethod solvers. We utilize this experience to address and overcome these problems when constructing a component-based multimethod software.

5.4 An Implementation Using Component Software Framework

In our second implementation of multimethod solvers, we employed a "component-based approach" as described in Chapter 2. In a component-based model, each component is independent of the larger application and is viewed as a black-box by the rest of the code. Components provide the ability to plug-and-play thus allowing the user to experiment with different code segments. In recognition of these advantages many scientific computing projects, such as climate modeling, are using component-based software. In particular, the Common Component Architecture Forum(CCA) [4, 11, 40] focuses on developing a scientific computing component model specification. Our design of multimethod software conforms to the CCA standards.

We used SIDL (Scientific Interface Definition Language), an IDL built with emphasis on scientific computing and Babel, a language interoperability tool, to create components for the multimethod software. The linear solvers from different packages are encapsulated into components. This enables solvers of different implementations to have compatible abstract interfaces and they can be easily combined to form multimethod solver instances. There exists a well-defined central server in this model which processes requests and services as described earlier. Functionalities for collecting performance data can be obtained from performance monitoring packages like TAU [44]. These functions are also implemented as components, are independent of the applications or linear solver packages used.

Our work on multimethod solvers contributes to the current research in developing efficient component-based high-performance software. In the next subsection we will describe how components can be created and integrated in a generalized application.

5.4.1 Creating Components within CCA Framework

A specification in SIDL is used to create CCA components. The functionalities of each component are specified in an associated SIDL file. The SIDL files have an object-oriented structure. The highest level is a package. Each package is formed of interfaces and classes, which in turn contain variables and methods. The interface and classes may depend on other interfaces from the same or different packages thus creating a complex dependency structure.

Next, a given SIDL file, is transformed into a component encoded in an user-specified language, through Babel, a SIDL based language interoperability tool [19]. Babel reads the interface and class specifications in the SIDL file to generate an intermediate XML representation. The code generator then reads the XML descriptions and automatically generates code for the specified software library. Finally, the application developer integrates the components generated by Babel into the main application.

Figure 5.3 shows the steps for generating component-based code using Babel. The user defines one or more SIDL files. The SIDL files are then parsed into associated XML representations by invoking specific Babel command options. After generating the XML files, the user can create server and client files using Babel commands. The server and

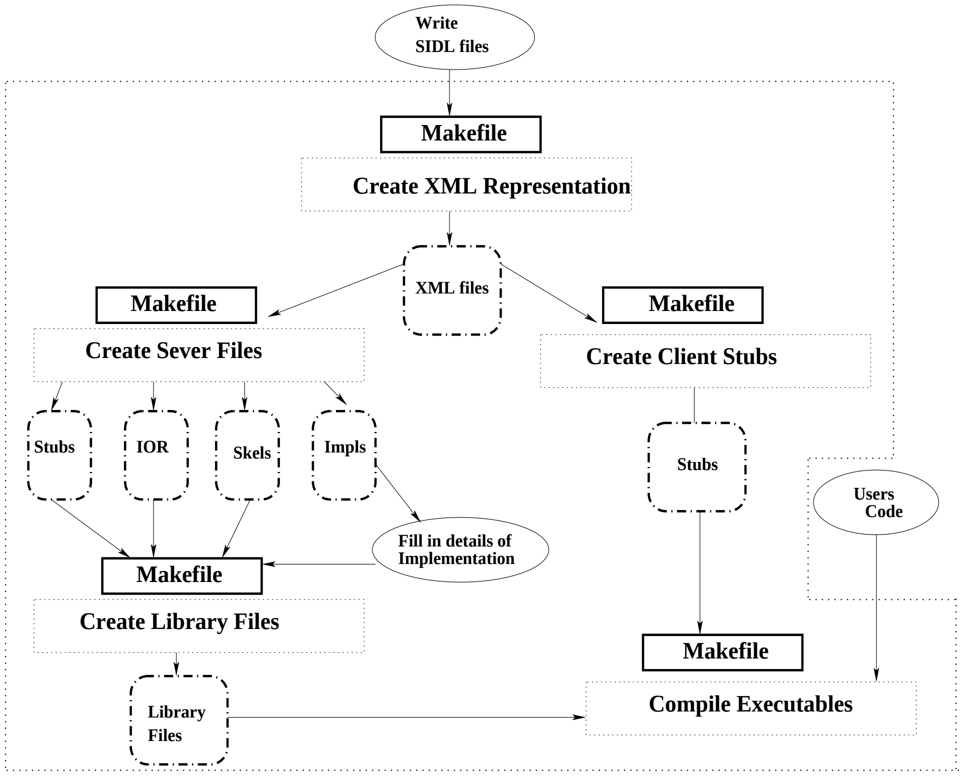


Fig. 5.3. Component creation with Babel.

client files generated are stubs, skeletons (Skel) and Babel internal representation (IOR) files. In the case of the server, implementation prototype (Impl) files are also created. The implementation files contain prototypes for the methods that are filled in by the application developers. Once the implementation details have been filled in, the user compiles the server codes to generate library files. The user can now write a client code which uses the methods described in the server. The client code and the client stubs are linked with the library files generated from the server code and compiled to obtain the executable.

The process of generating server and client codes from the SIDL definition requires active user participation. In Figure 5.3, the ovals represent the files containing user defined information. The dotted boxes represent scripts to be written by the user to generate the intermediate Babel files (each such operation has an associated makefile) and the dashed boxes represent intermediate files that have been generated by Babel commands. In order to proceed from one step to another, the user has to be aware of what files were generated, their location, their mutual dependency patterns and how to effectively link them. The makefile scripts have to be changed each time a new SIDL file is incorporated or a new client code is written. Writing the makefiles and changing them for each new addition in the project is, if not difficult, at the very least time consuming. In an ideal situation the user should only have to know what is pertinent to the project, the client codes, the packages used in writing them, the SIDL files needed and implementation details of the methods i.e. the parts denoted by the ovals in the diagram. The user should be able to add, modify or delete components by filling in only the relevant details and the rest of the operations (within the dotted box) should be taken care of automatically. Our goal is to construct an interface over Babel functions that allows the users to access the benefits of language interoperability without getting involved into the intricacies of writing complicated makefiles.

5.4.2 Towards an Automated Interface

In the course of our implementation, we discovered that with a few modifications to Babel, user effort in creating the components can be significantly reduced. This motivated us to extend Babel and develop an automated interface for creating components. The interface has the ability to support and track dependencies across components and requires only minimum information regarding the component from the application developer. The interface is compatible with the CCA-based framework and retains all the associated advantages.

The two main operations in constructing the automated interface were : (i) building templates of makefiles, and (ii) generating data dependency trees. We observe that the makefiles generally follow a "linear" approach; there are rarely any branching commands. The user has to specify the relevant libraries, the source files, the commands for creation of the object files and their subsequent linking and compilation to form the executables. There is not much variance in this pattern and we can substitute individual makefiles for templates. The appropriate template can be chosen according to the application need or user specifications. An important requirement for creating such templates is that the locations of the files should be known. Our interface solves this problem by fixing the locations of all intermediate files. To ensure fixed locations, we restrict the user interaction with Babel. The user can access the Babel commands only via the interface. This restriction does not entail any loss of functionality, and if needed the user can move files across directories.

The complex dependency structure between packages requires maintaining a dependency tree. Inputs to some Babel commands require the user to arrange the SIDL files in the order of their dependencies. Thus the user has to know the dependency patterns of all the SIDL files he is using and this might be difficult, especially in a collaborative project. In addition the effects of modifications in one package would cascade down to its children. In absence of a dependency tree this might create many unforeseen and complicated problems in the code. We maintain two sets of dependency trees in the form of databases, one based on packages and the second based on SIDL files. This is because parts of the same package may be defined across different SIDL files. Users can view the dependency structure if they wish, but the databases function transparent to the user. The dependency structure generates an ordered list of SIDL files when needed and also makes necessary modifications to relevant packages when the parent package is modified.

Figure 5.4 demonstrates our design for this automated system. Notice that each combination of a dotted and solid box in Figure 5.3 (which represented makefile scripts to be written by the user) has a corresponding solid box in Figure 5.4 that does exactly the same thing with only a single command. The only inputs required from the user are the names of the SIDL files, the packages used and the language for generating servers and clients. For users who would like to use finer level options, this system provides the flexibility of selecting the choice of compilers(at configure), type of libraries (static/dynamic), and also allows the user to change the makefiles if they wish to do so.

The basic installation steps sets the appropriate libraries, packages and compilers best suited to the architecture in the directory HOME, next, the user can add new SIDL files or modify existing ones using the following command options.

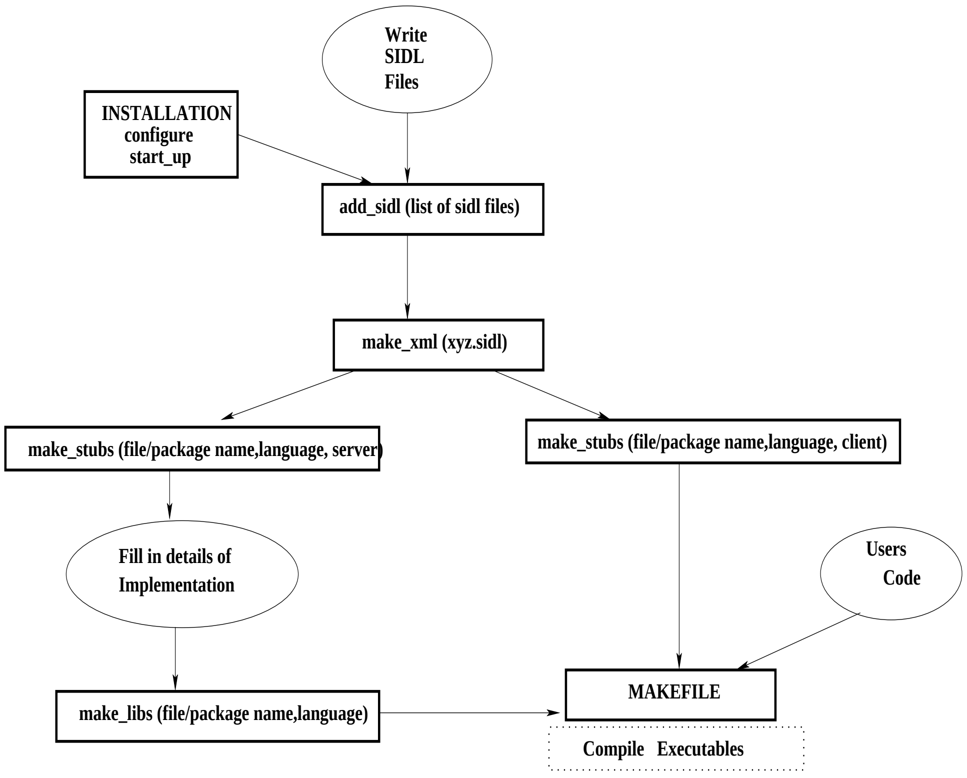

Write
SIDL
Files
INSTALLATION
configure
start_up
add_sidl (list of sidl files)
make_xml (xyz.sidl)
make_stubs (file/package name,language, server)
make_stubs (file/package name,language, client)
Fill in details of
Implementation
Users
Code
make_libs (file/package name,language)
MAKEFILE
Compile Executables

Fig. 5.4. Automated component creation with the extended interface to Babel.

- `add-sidl:` This command allows the user to specify the SIDL file to be added. The original file can be at any location, but after execution of `add-sidl` the file is copied to `HOME/SIDL` and *this* copy is accessed for all subsequent calls. This operation involves parsing the SIDL file to construct databases representing dependency trees. The databases provide a list of packages defined in the file, their constituent classes, and external packages required by classes within the parsed packages. The user can obtain the information in the database either by referencing the name of the SIDL file or just by the package name.

- `make-xml:` This command forms an interface to the Babel function for creating XML representations of the SIDL files. The XML version is stored in `HOME/SIDL/xml`. Using the Babel command directly might generate error messages if the class being parsed either (i) depends on an external package that does not yet have an XML representation or (ii) already has an XML representation.
  The script `make-xml` eliminates these situations by creating a dependency tree for each SIDL file. Each SIDL file is now associated with a list of parent files that contain packages required by the classes in this SIDL file and children files that require the packages defined in this SIDL file. The XML files are generated in order of dependency from the topmost parent file (one which does not have any dependency) to the SIDL file given in input argument. If the XML version of any file in this sequence exists, then that file is skipped and we move on to the next one in the list.

- `make-stubs:` This script creates server/client files for the packages. The user can specify any particular package or a SIDL file (in which case files for all the packages would be generated). The user specifies the language and the type (client or server) of the files. Once the files are generated the user accesses the prototypes in the *Impl* files (generated only for servers) to fill in the implementation details.

- `make-libs:` This script is used for generating libraries from the server/client files. By default a shared library is created, though the user can build a static library by adding the library type argument. The package itself might depend on several external packages that should be included in the makefile. The script reads through the database to create a list of the parent packages. The libraries after being generated are stored in `HOME/lib`.

- `rewrite-sidl:`The function of this script is almost the same as that of `add-sidl`, except as the name suggests this function is used for "rewriting" SIDL files that have already been added. To rewrite, the function first eliminates the given SIDL file and all its associated files, i.e. XML representations, database entries, client and server codes as well as SIDL files that are dependent on it. Then the file is added again (as by using `add-sidl`).

5.5 Summary

In this chapter, we discussed the primary requirements to be satisfied by a multimethod software system. We then presented a software architecture design that would address these issues. We have implemented limited versions of the architecture in two forms. The first implementation, using PETSc, while providing the basic requirements, suffers many limitations. We overcome some of these limitations in our second implementation using a component-based approach which conforms to the CCA standards. The main advantage of CCA is the use of object-oriented components that allow easy integration of different software packages. However, building these components is as yet a non-trivial task and we have developed an interface to Babel which facilitates component creation. We continue to implement our component-based multimethod solver system. In the near future we plan to develop and add performance monitoring components and databases for efficient storage, retrieval and use of the performance metrics.

Chapter 6

Conclusions and Future Research

This thesis concerns the development, design and implementation of multimethod solvers to enable robust, limited memory and efficient solution of sparse linear systems. We have developed two classes of multimethod solvers (i) composite solvers that automatically compose a sequence of sparse linear solution schemes to be applied to the same system to provide highly reliable scalable solution and (ii) adaptive solvers which dynamically select the linear solver that best matches linear system characteristics to reduce solution time. We have implemented our schemes and evaluated their performance in solving linear systems generated by computational fluid dynamics applications. Additionally, we have designed a client-server multimethod software architecture that enables easy creation and use of our solvers in high-performance scientific computing applications.

We developed multimethod composite solvers that can solve linear systems with high reliability and have low memory requirements. We provided analytical results, based on a combinatorial framework, for instantiating an optimal composite. We demonstrated that the use of composite solvers can significantly increase the reliability of linear system solution schemes and thereby improve the execution time of applications. For example, on the driven cavity flow application, our optimal composite solver is 50% better on average compared to the base methods, using 1 processor, and 30% better using 8 processors.

We developed heuristics for adaptive selection of linear solution schemes using polynomial interpolation and based on metrics such as linear and nonlinear convergence rates, execution times, etc. Our emphasis is on reducing the execution time of the application by dynamically solver properties to the system attributes. Adaptive solvers have shown 39% improvement in execution time (average) when applied to the driven cavity flow application.

We designed a component-based client-server architecture for multimethod solver software system that can automatically instantiate multimethod solvers on request from the user. The software system also aims to provide application developers with an easy to use interface to multimethod solvers. We have implemented the major features of our architechture as CCA-compliant software, thus contributing to the on-going community efforts in creating component based applications. As part of this effort, we have also developed an interface to Babel (a language interoperability tool) that simplifies component creation. This interface is not specific to multimethod solvers and can be used to aid the development of other component-based software.

In the near future, we plan to investigate further extensions to adaptive and composite solvers. An important research problem is to understand and formulate a mathematical relation between the application type and the corresponding adaptive heuristic. Composite solvers are in a sense less problem dependent. However, the dependence of the failure rates among different methods lead to interesting combinatorial problems which can potentially lead to improved composite solvers. We also plan to expand our application domain and experiment with larger practical problems. We conjecture that the improvements observed in the our initial experiments would be magnified for larger applications.

Finally, we note that the multimethod algorithms need not be confined to sparse iterative linear solvers. Other problems, with a set of competitive solution methods and no obvious "best method" have the potential to benefit from our techniques. For example, multimethod algorithms can be applied to reduce fill-in for sparse matrix factorization by utilizing multiple methods for ordering [22, 36]. Other examples include, eigenvalue computations, mesh generation and optimization algorithms. Multimethod algorithms can also be potentially applied to other areas beyond scientific computing, for example, in scheduling or resource allocation.

References

[1] The Chiba City Project, URL: http://www.mcs.anl.gov/chiba.

[2] LCRC Argonne National Laboratory Computing Project, URL:http://www.lcrc.anl.gov/jazz/index.php.

[3] W.K. Anderson and D. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23(1):1–21, 1994.

[4] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L.C. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. *Proceedings of High Performance Distributed Computing*, pages 115–124, 1999.

[5] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L.C. McInnes, B. Smith, and H. Zhang. The PETSc users manual. Technical Report ANL-95/11, Argonne National Laboratories, Argonne, IL, 2002.

[6] S. Balay, W. Gropp, L.C. McInnes, and B. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E.Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[7] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V.Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of Linear Systems: Building Blocks for Iterative Methods.* Siam, 1994.

[8] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic bombardment for the iterative solution of linear systems: A polyiterative approach. *Journal of Computational and applied Mathematics*, 74:91–110, 1996.

[9] B. A. V. Bennet and M. D. Smooke. Local rectangular refinement with application to nonreacting and reacting fluid flow problems. *Journal of Computational Physics*, 151:648–727, 1999.

[10] Marshall W. Bern and Paul E. Plassmann. Mesh generation. In Jörg Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 291–332. Elsevier Scientific, 2000.

[11] D. Bernholdt, B. A. Allen, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krisman, G. Kumfert, J W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, ACTS Collection Special Issue, submitted, 2004.

[12] S. Bhowmick, L. McInnes, B. Norris, and P.Raghavan. Robust algorithms and software for parallel PDE-based simulations. *Proceedings of HPC 2004, The Twelfth Special Symposium on High Performance Computing at the 2004 Advanced Simulation Technologies Conference, Arlington, VA, April 18-22, 2004*, pages 37–42, 2004.

[13] S. Bhowmick, L. C. McInnes, B. Norris, and P. Raghavan. The role of multi-method linear solvers in pde-based simulations. *Lecture Notes in Computer Science, Computational Science and its Applications-ICCSA 2003*, 2667:828–839, 2003.

[14] S. Bhowmick, P. Raghavan, L. McInnes, and B. Norris. Faster PDE-based simulations using robust composite linear solvers. *Future Generation Computer Systems*, 20:373–386, 2004.

[15] S. Bhowmick, P. Raghavan, and K. Teranishi. A combinatorial scheme for developing efficient composite solvers. *Lecture Notes in Computer Science, Computational Science- ICCS 2002*, 2330:325–334, 2002.

[16] R. Bramley, D. Gannon, T. Stuckey, J. Balasubramanian J. Villacis, E. Akman, F. Berg, S. Diwan, and M. Govindaraju. The linear system analyzer. In *Enabling Technologies for Computational Science*. Kulwer, 2000.

[17] X. C. Cai and M. Sarkis. A restricted additive schwarz preconditioner for general sparse linear systems. *SIAM Journal of Scientific Computing*, pages 792–797, 1999.

[18] T.J. Chung. *Computational fluid dynamics*. Cambridge University Press, 2002.

[19] T. Dahlgren, T.Epperly, and G.Kumfert. Babel Users' Guide, Lawrence Livermore National Laboratory. URL: http://www.llnl.gov/CASC/components/babel.html.

[20] G. De Vahl Davis. Natural convection of air in a square cavity: A bench mark numerical solution. *International Journal for Numerical Methods in Fluids*, 3:249–264, 1983.

[21] G. De Vahl Davis and I. P. Jones. Natural convection of air in a square cavity: a comparison exercise. *International Journal for Numerical Methods in Fluids*, 3:227–248, 1983.

[22] D.J.Rose, R.E.Tarjan, and G.S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput*, 5:266–283, 1976.

[23] J. Dongarra and V. Eijkhout. Self adapting numerical algorithm for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–132, 2003.

[24] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[25] A. Ern, V. Giovangigli, D. E. Keyes, and M. D. Smooke. Towards polyalgorithmic linear system solvers for nonlinear elliptic problems. *SIAM J. Sci. Comput.*, 15, No 3:681–703, 1994.

[26] Joel. H. Ferziger and Miloran Peric. *Computational Methods for Fluid Dynamics.* Springer-Verlag, 1983.

[27] R. Freund, G. H. Golub, and N. Nachtigal. Iterative solution of linear systems. *Acta Numerica*, 1992.

[28] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermetian linear systems. *SIAM J. Sci. Stat.Comp*, 14:470–481, 1993.

[29] G.H. Golub and C.F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1989.

[30] W.D. Gropp, D.E. Keyes, L.C. McInnes, and M.D. Tidriri. Globalized newton-krylov-schwarz algorithms and software for parallel implicit cfd. *Int. J. High Perform. Comput. Appl*, 14:102–136, 2000.

[31] Wiiliam D. Gropp, Dinesh K. Kaushik, David E. Keyes, and Barry F. Smith. High performance parallel implicit cfd. *Parallel Computing*, 2000.

[32] Michael T. Heath. *Scientific Computing:An Introductory Survey, Second Edition.* McGraw Hill, 2002.

[33] H.Meuer, E.Strohmaier, J. Dongarra, and H.D.Simon. Top 500 supercomputer sites. URL: http://www.top500.org.

[34] G. Karypis and V. Kumar. A fast and high quality scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20:359–392, 1999.

[35] C. T. Kelley and D. E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM Journal on Numerical Analysis*, 35:508–523, 1998.

[36] R. J. Lipton, D. J. Rose, and E. Tarjan R. Generalized nested dissection. *Siam Journal of Numerical Analysis*, 16(2):346–358, 1979.

[37] L. McInnes, B. Norris, S. Bhowmick, and P. Raghavan. Adaptive sparse linear solvers for implicit cfd using newton-krylov algorithms. *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*, June 17-20,2003.

[38] Eric J. Nielsen, W. Kyle Anderson, Robert W. Walters, and David E. Keyes. Application of newton-krylov methodology to a three-dimensional unstructured euler code. Technical Report AIAA-95-1733, 1995.

[39] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer-Verlag, New York, 1999.

[40] B. Norris, S. Balay, S. Benson, P. Hovland L. Freitag, L. McInnes, and B. F. Smith. Parallel components for pdes and optimization: Some issues and experiences. *Parallel Computing*, 28(12):1811–1831, 2002.

[41] O.Axelsson. A survey of preconditioned iterative methods for linear systems of equations. *BIT*, 1987.

[42] Roger Peyret and Thomas D. Taylor. *Computational Methods for Fluid Flow.* Springer-Verlag, 1983.

[43] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company,, 1995.

[44] S. Shende, A.D. Malony, J. Cuny, K.Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel scientific applications using c++. *Proceedings of the SIGMETRICS Symposium on Parallel and DIstributed Tools (SPDT'98)*, pages 134–145, 1998.

[45] Jon Siegel. *Corba 3: Fundamentals and Programming.* John Wiley and Sons, 2000.

[46] Barry Smith, Peter Bjørstad, and William Gropp. *Domian Decomposition.* Cambridge University press, 1996.

[47] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[48] Llyod. N. Trefethen and David Bau III. *Numerical Linear Algebra.* SIAM, 1997.

[49] Mihalis Yannakakis. Computing the minimum fill-in is np-complete. *SIAM J. Alg. Disc. Meth*, 2:77–79, 1981.

Vita

Sanjukta Bhowmick was born in Pilani, Rajasthan, India on February 20, 1978. She went to high school in Kolkata, India. She received the National Scholarship for achievement in the Higher Secondary Examination from the West Bengal Higher Secondary Education Board. Sanjukta did her undergraduate studies in the Haldia Institute of Technology, Haldia, India. She received her B.Tech (Honours) degree in Computer Science and Engineering in 2000 along with a medal for securing the highest grades from the Vidyasagar University. In the same year she enrolled in the Ph.D. program in the department of Computer Science and Engineering in the Pennsylvania State University. She received the Wallace Givens Research Associate Fellowship to work as an intern at the MCS division of the Argonne National Laboratory. She received the CSE Best Research Assistant Award from the Department of Computer Science and Engineering, the Pennsylvania State University in 2004. Sanjukta's thesis concerns the development of multimethod solvers for fast and reliable solution of large scale sparse linear systems and their application in scientific and engineering problems. She is also interested in the design and implementation of component software. Her research includes scalable parallel algorithms, numerical methods, particularly iterative solvers and also non-numerical topics like combinatorics and graph theory.