Faster PDE-Based Simulations Using Robust Composite Linear Solvers

S. Bhowmick,[1] P. Raghavan [*,1]

Department of Computer Science and Engineering, The Pennsylvania State University, 220 Pond Lab, University Park, PA 16802-6106.

L. McInnes,[2] B. Norris [2]

Mathematics and Computer Sciences Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne IL 60439-4844.

Abstract

Many large-scale scientific simulations require the solution of nonlinear partial differential equations (PDEs). The effective solution of such nonlinear PDEs depends to a large extent on efficient and robust sparse linear system solution. In this paper, we show how fast and reliable sparse linear solvers can be composed from several underlying linear solution methods. We present a combinatorial framework for developing optimal composite solvers using metrics such as the execution times and failure rates of base solution schemes. We demonstrate how such composites can be easily instantiated using advanced software environments. Our experiments indicate that overall simulation time can be reduced through highly reliable linear system solution using composite solvers.

Key words: large-scale PDE-based simulations, composite methods, multi-algorithms, sparse linear solution, Newton-Krylov methods

* Corresponding author.
Email addresses: `bhowmick@cse.psu.edu` (S. Bhowmick,), `raghavan@cse.psu.edu` (P. Raghavan), `mcinnes@mcs.anl.gov` (L. McInnes,), `norris@mcs.anl.gov` (B. Norris).

[1] This work was supported in part by the Maria Goeppert Mayer Award from the U. S. Department of Energy and Argonne National Laboratory and by the National Science Foundation through grants ACI-0102537, CCR-0075792, and ECS-0102345.

[2] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U. S. Department of Energy under Contract W-31-109-Eng-38.

1 Introduction

Computational science and engineering advance the state of knowledge across disciplines by means of modeling and simulation. This endeavor relies to a large extent on advances in algorithms and software to meet the underlying computational challenges. Recent trends include the development of multi-algorithm and composite solution schemes that attempt to provide faster solutions with greater reliability [6,7]. At the same time, the efficient implementation of such methods and their use in large-scale applications are becoming more feasible through recent advances in specifying component architectures as well as sets of domain-specific abstract interfaces. In this paper, we demonstrate that simulation times can be significantly reduced by developing composite solvers tailored to match application attributes and by readily implementing these solvers using advanced software systems that allow flexible composition of algorithms and their implementations.

Many fundamental problems in scientific computing have several competing solution methods. For example, PDE-based simulations frequently involve the solution of linear systems for which two broad classes of methods are often considered, direct and iterative, with a variety of algorithms within each class. The performance of a specific algorithm often depends on the numerical properties of the problem instance. The choice of a particular algorithm may depend on many factors, such as its computational cost, its memory requirements, the likelihood that it computes a solution without failure, and the level of scalability of a parallel implementation. One can therefore view each method as reflecting a certain tradeoff among several metrics of performance and reliability. Even with a very limited set of metrics (for example, the time to compute a solution and the probability of failure), it is often neither possible nor practical to predict a priori which algorithm will perform best for a given suite of problems. For a hard problem instance, an expensive but reliable method might be required, while a simpler problem instance might be solved easily with a faster method with potentially poor reliability. We observe that even a single simulation can produce problem instances with varying attributes, such as the degree of nonlinearity and the conditioning of the operator. Furthermore, significant variations in attributes are natural when considering problems across different application domains. Consequently, recent efforts have focused on developing multimethod and composite schemes that utilize several underlying methods to improve reliability while reducing total execution time. The idea of multi-algorithms has been explored earlier in conjunction with a multiprocessor implementation [6]; the multi-algorithm comprises several algorithms that are simultaneously applied to the problem by exploiting parallelism. More recently, Bhowmick et al. have provided a combinatorial formulation to develop a composite solver as a special sequence of constituent methods [7].

Traditionally, software libraries (collections of functions) have been the mechanism for encapsulating and disseminating advanced numerical and scientific algorithms. However, the traditional library model does not provide the flexibility, interoperability and extensibility required for large-scale scientific simulations with composite or multi-method solvers. Advanced object-oriented software systems, such as PETSc [4], with well-defined abstract interfaces and dynamic method selection, allow the instantiation of composite methods without large software development overheads.

In this paper, we focus on potential performance improvements for PDE-based simulations by developing and implementing composite linear solvers using a flexible and extensible software environment. More specifically, we study the role of composite linear solvers for driven cavity flow, which involves the solution of a nonlinear PDE. Section 2 provides a combinatorial framework for developing composites that can minimize execution times while maximizing reliability. Section 3 discusses typical PDE-based applications using a driven cavity flow simulation as an example. Section 4 shows how composite linear solvers can be developed and implemented within the PETSc library [4,5] and then used within Newton-type methods to solve nonlinear PDE-based applications. Section 5 demonstrates how application time can be potentially reduced by using robust composite linear solvers and their flexible instantiation through PETSc. Section 6 contains concluding remarks and future research directions.

2 A Combinatorial Scheme for Optimal Composite Solvers

The solution of large, sparse systems of linear equations is a fundamental problem in scientific computing. Such systems arise in a wide variety of PDE-based simulations, for example, when the underlying physical models are discretized by using finite element and finite difference methods. Despite active research in the development of efficient algorithms over the past several decades, no single solution method is robust and consistently the best in performance across application domains. Thus, sparse linear system solution has been viewed more recently as a problem for which potential performance and robustness gains may be achieved through a combination of several base methods. In this section, we provide some background on sparse linear system solution methods and discuss our combinatorial approach for developing efficient composite solvers.

We provide a brief overview of sparse linear solution techniques to provide a context for the development of multi-algorithm composites; some general references include [3,11,13,14]. In broad terms, linear solution methods can be categorized into Krylov subspace iterative methods and direct methods. The former consist of the method of conjugate gradients (CG) and nonsymmetric forms such as BICG and GMRES. The methods converge to a solution by refining an initial guess through iterations in a certain subspace. These methods are scalable, but convergence can be slow or fail altogether; convergence depends on the numerical properties of the matrix. The class of direct methods relies on the sparse factorization of the coefficient matrix followed by triangular solution with the factors. These methods are typically much harder to describe and implement because symbolic techniques have to be used to limit fill-in during factorization; fill-in depends on the zero-nonzero structure of the matrix and not on actual numeric values. The main limitation of these methods is that they are not memory scalable. Direct methods can fail if there is insufficient memory; a solution will be computed only if the fill-in can be accommodated within available memory limits. Both classes contain several different algorithms with significantly different performance and reliability attributes. Furthermore, algorithms from the two classes can be combined through preconditioning; in its more general form, incomplete variants of a direct method are used to accelerate the convergence of the iterative method by providing a linear system with better spectral properties. More recently, significant research has focused on domain-decomposition, multilevel, and multigrid methods, and many of these techniques are highly scalable; however, they are typically linked to the discretization of the underlying PDE and not easily packaged as black-box solvers. Given the variety of algorithms, their implementations, and performance-robustness tradeoffs, researchers have naturally explored the benefits of combining several methods to deliver improved performance for large-scale applications [6,7,9]. The challenges lie in formulating techniques for meaningful composition and in developing high-performance software.

We now consider a combinatorial framework for developing a composite linear solver from several base methods [7]. Assume that each candidate method can be evaluated by using two metrics: (i) performance or cost, and (ii) reliability. The former represents execution time or the number of operations, while the latter reflects the rate of success. It may be possible to derive analytic expressions for both metrics; failing that, the metrics can be computed by empirical means. One approach is to observe the performance of each method on a representative set of sample problems and use the observed values to predict average metrics for the problem population. We assume that the two metrics can be represented in a normalized form for the set of selected methods. For example, if the performance metric is execution time, then the values can be scaled by the time for the fastest method to provide values bounded below by 1. The reliability is naturally a number in the range $[0, 1]$, reflecting the probability of success. For example, if an iterative linear solver fails to converge on average in one-third of the sample set of problems, its failure rate is 0.33 and its reliability 0.67.

Method 1	Method 2	Method 3
t_1, r_1, f_1	t_2, r_2, f_2	t_3, r_3, f_3
1, 0.1, 0.9	1.5, 0.7, 0.3	3.0, 0.8, 0.2

Permutation	Composite Time
1, 2, 3 'least time'	$3.16 = 1 + .9 \times 1.5 + .9 \times .3 \times 3.0$
3, 2, 1 'least failure'	$3.36 = 3 + .2 \times 1.5 + .2 \times .3 \times 1.0$
2, 1, 3	$2.61 = 1.5 + .3 \times 1 + .3 \times .9 \times 3.0$
2, 3, 1	$2.46 = 1.5 + .3 \times 3 + .3 \times .2 \times 1.0$
Failure rate of any composite is 0.054.	

Table 1
Example illustrating the effect of different sequences of base methods on the worst-case time of the composite.

Consider developing a composite using n base methods labeled $B_1, B_2, \ldots, B_n$. Let t_i be the performance measure of method B_i with success rate r_i (failure rate $f_i = 1 - r_i$). The composite then consists of all the n methods in a sequence; failure of one method will result in the execution of the next method in the sequence. Such a composite is significantly more robust than its component methods. If failures of the methods occur independently, then the reliability of the composite is $1 - \Pi_{i=1}^{i=n} f_i$; that is, its failure rate is much smaller than the failure rate of any component method. All such composites are equally reliable; however, the exact sequence in which the base methods are selected can affect the total execution time. Let the set $\mathbf{P}$ contain all permutations (of length n) of $\{1, 2, \ldots, n\}$. Let $\hat{P}_i$ denote the ith method in $\hat{P} \in \mathbf{P}$. The composite $\hat{C}$ is specified by $\hat{P}$, and it executes methods in the order specified by $\hat{P}$. Assuming partial results of a failed method are not reused, the worst case execution time of $\hat{C}$ is

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

Different permutations can lead to wide variations in execution time, as shown by the example in Table 1 for composites of three methods.

In our earlier paper [7], we considered the problem of determining $\tilde{P} \in \mathbf{P}$ such that the associated composite $\tilde{C}$ is optimal, that is, its worst case time $\tilde{T} = \min\{\hat{T} : \hat{P} \in \mathbf{P}\}$. We proved that the optimal composite is one in which the component methods are arranged in *increasing order* of the ratio $u_i = \frac{t_i}{r_i}$ [7]. Such an optimal composite is obviously easy to build; the component methods are sorted in increasing order of the ratios of their performance and reliability metrics. Interesting, the optimal composite is not realized through sequences in increasing order of time or failure rates ("least time first" or "least failure first" greedy heuristics).

The proof that a composite is optimal if and only if its component methods are in increasing order of the ratio $\frac{t_i}{r_i}$ is rather complicated [7]. The intuition behind the proof, however, is easily revealed by considering shortest paths in a graph. Consider a layered graph with vertices at $n+1$ levels, where a vertex at level i represents a *subset* of i methods. The vertex V_0 at level 0 denotes the empty set. An edge connects vertex $\bar{S}$ to $\hat{S}$ if they are on adjacent levels, $|\hat{S} \setminus \bar{S}| = 1$, and $\hat{S} \cap \bar{S} = \bar{S}$. For the subset of methods S, $F_S = \Pi_{B_i \in S} f_i$ and $F_\Phi = 1$; edge $\bar{S} \rightarrow \hat{S}$ has weight $F_{\bar{S}} t_i$ if $\hat{S} \setminus \bar{S} = \{i\}$. In this graph, it is easily verified that a path from V_Φ to $V_{\{1,2,\cdots n\}}$ denotes a composite in the order in which methods were added to vertex sets (on the path). Additionally, the length of the path is the time for the composite, as specified in Equation 1. Thus, the optimal composite corresponds to the shortest path.

Examining the shortest path in further detail reveals the ordering of the ratio $\frac{t_j}{r_j}$ and $\frac{t_i}{r_i}$ when method j precedes i in the optimal composite. Assume that V_S and $V_{\hat{S}}$ are on the shortest path and $\hat{S} - S = \{i, j\}$. Now there are only two paths from V_S to $V_{\hat{S}}$, one with $V_{\bar{S}}$ ($\bar{S} - S = \{i\}$) and another with V_{S*} ($S^* - S = \{j\}$). Assume without loss of generality that the latter segment is on the shortest path. Thus, $T_S + F_S t_j + F_S f_j t_i \leq T_S + F_S t_i + F_S f_i t_j$. This simplifies to $t_j(1-f_i) \leq t_i(1-f_j)$; now $\frac{t_j}{r_j} \leq \frac{t_i}{r_i}$, that is, $u_j \leq u_i$. Figure 1 illustrates the relationship between shortest paths and optimal composites for the example with three components methods introduced in Table 1. As mentioned earlier, the graph and its shortest paths are not required to construct the optimal composite.

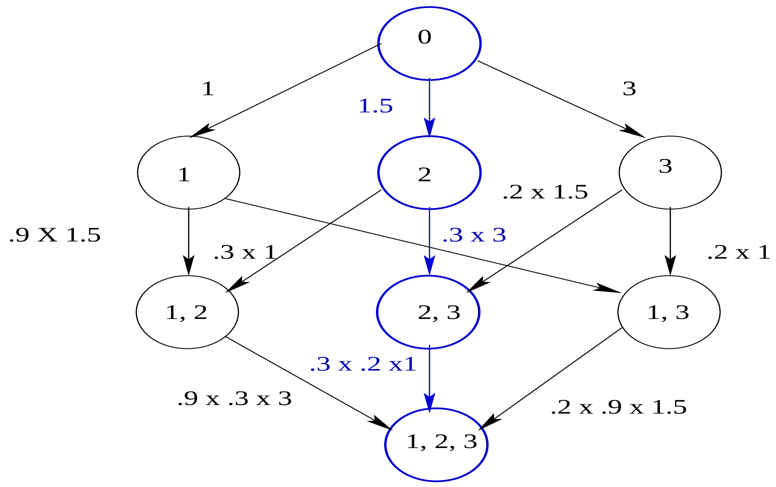

Fig. 1. Example with three component methods ($\frac{t_1}{r_1} = 10.0, \frac{t_2}{r_2} = 2.14, \frac{t_3}{r_3} = 3.75$) illustrating the relationship between the optimal composite and the shortest path.

3 PDE-Based Simulations

This work is motivated by the solution of nonlinear PDEs of the form $f(u) = 0$, where $f : \Re^n \to \Re^n$, which arise in a variety of large-scale scientific simulations. Some applications that motivate this work involve computational aerodynamics [1], astrophysics [12], and fusion [18]. In conjunction with theoretical and experimental research, such simulations are playing increasingly important roles in overall scientific advances, particularly when physical experiments are prohibitively expensive, time consuming, or in some cases impossible. One of the most computationally intensive phases within semi-implicit and implicit strategies for solving nonlinear PDEs is the solution of discretized linear systems, which are typically very large and have sparse coefficient matrices. The focus of this work is the development of composite linear solvers that increase algorithmic robustness and decrease overall time to solution.

We consider a driven cavity flow simulation to illustrate the benefits of composite solvers and their instantiation in flexible software. This model has the characteristics of the motivating simulations mentioned above, yet it is sufficiently compact to allow a simple description. The driven cavity flow model is a combination of lid-driven flow and buoyancy-driven flow in a two-dimensional rectangular cavity. The lid moves with a steady and spatially uniform velocity and thus sets a principal vortex and subsidiary corner vortices. The differentially heated lateral walls of the cavity induce a buoyant vortex flow, opposing the principal lid-driven vortex. See [10] for a detailed description of the problem and solution method.

The two-dimensional driven cavity flow uses the velocity-vorticity formulation of the Navier-Stokes and energy-governing equations, in terms of the velocity u_x, u_y in the (x, y) directions and the vorticity ω on a domain Ω defined as $\omega(x, y) = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}$. The governing differential equations are

$$-\Delta u_x + \frac{\partial \omega}{\partial y} = 0, \tag{2}$$

$$-\Delta u_y + \frac{\partial \omega}{\partial x} = 0, \tag{3}$$

$$-\Delta \omega + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} - \mathrm{Gr}\frac{\partial T}{\partial x} = 0, \tag{4}$$

$$-\Delta T + \mathrm{Pr}(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}) = 0, \tag{5}$$

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

4 Algorithms and Software

We use inexact Newton methods (see, e.g., [16]) to simultaneously solve the driven cavity flow equations (2) through (5), which have the combined form $f(u) = 0$. We use a Krylov iterative method to (approximately) solve the Newton correction equation

$$f'(u^{\ell-1})\ \delta u^{\ell} = -f(u^{\ell-1}), \tag{6}$$

in the sense that the linear residual norm $||f'(u^{\ell-1})\ \delta u^{\ell} + f(u^{\ell-1})||$ is sufficiently small. We then update the iterate as follows,

$$u^{\ell} = u^{\ell-1} + \alpha \cdot \delta u^{\ell},$$

where α is a scalar determined by a line search technique such that $0 < \alpha \leq 1$. We terminate the Newton iterates when the relative reduction in the residual norm falls below a specified tolerance, namely, when $||f(u^{\ell})|| < \epsilon ||f(u^0)||$, where $0 < \epsilon < 1$.

We precondition the Newton-Krylov methods, whereby we increase the linear convergence rate at each nonlinear iteration by transforming the linear system (6) into the equivalent form

$$B^{-1} f'(u^{\ell-1})\ \delta u^{\ell} = -B^{-1} f(u^{\ell-1}),$$

through the action of a preconditioner, B, whose inverse action approximates that of the Jacobian, but at smaller cost. As further discussed in Section 5, we consider Krylov methods such as restarted GMRES, with preconditioners including both drop tolerance and level of fill variants of incomplete factorizations (see, e.g., [14]).

We instantiate the solution of our driven cavity flow models through the Portable, Extensible Toolkit for Scientific computation (PETSc) [4,5], a suite of data structures and routines for the scalable solution of scientific applications modeled by PDEs. The software integrates a hierarchy of libraries that range from low-level distributed data structures for meshes, vectors, and matrices through high-level linear, nonlinear, and timestepping solvers. The algorithmic source code is written in high-level abstractions so that it can be easily understood and modified. This approach promotes code reuse and flexibility and in many cases helps to decouple issues of parallelism from algorithm choices.

The toolkit attempts to handle in a highly efficient way, through a uniform interface, the low-level details of assembling and invoking a large number of methods. Examples of such details include organizing code for strong cache locality, preallocating memory in sizable chunks rather than incrementally, and separating tasks into one-time and every-time subtasks by using the inspector/executor paradigm. The benefits to be gained from these and from other numerically neutral but architecturally important techniques are so significant that it is efficient in both programmer time and execution time to express them in general-purpose code.

Figure 2 illustrates the calling tree of a typical nonlinear PDE application using preconditioned Newton-Krylov solvers within PETSc. The top-level application driver performs I/O related to initialization, restart, and postprocessing; it also calls PETSc routines to create data structures for vectors and matrices and to initiate the nonlinear solver. In turn, the nonlinear solver library calls user-defined routines for function evaluations $f(u)$ and (approximate) Jacobian evaluations $f'(u)$ at given state vectors.

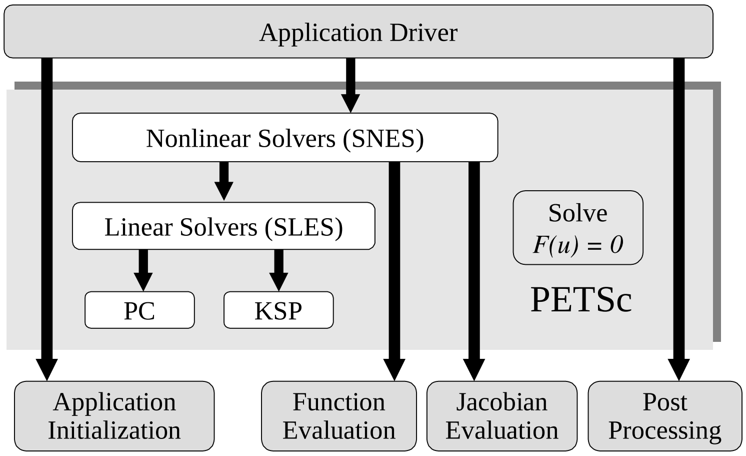

Fig. 2. Schematic diagram of a nonlinear PDE application using preconditioned Newton-Krylov methods, showing a user-provided driver and user-provided callback routines for evaluating the nonlinear residual vector and corresponding Jacobian at solver-requested states.

The Newton-based methods within PETSc are written in a data-structure-neutral form that uses abstractions for vectors, matrices, and linear solvers. The techniques of encapsulation and polymorphism enable the support of various matrix storage schemes and linear solvers through a single user interface. Such flexibility is critical for enabling easy experimentation with different algorithms and data structures. For example, we implemented the composite methods presented in Section 2 by defining a new linear solver that encompasses the set of possible algorithms in the composite. We then investigated a range of composite variants simply by specifying particular runtime options; no changes in the application code were required for these experiments.

5 Experiments

We report on our experiments with single and multimethod (composite) linear solvers for the driven cavity flow application. We performed our experiments on a workstation with a dual-CPU 500 MHz Pentium III with 512 MB of RAM. As described in Section 4, 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. Our initial experiments were directed toward evaluating the potential benefits of robust composite linear solvers for reducing total application time over a set of several related simulations.

In our 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. Furthermore, the nonlinear iterations often fail even with relatively low failure rates of the linear solver. 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.

Our experiments used a 96×96 mesh with Grashof numbers in the range [500, 1000] and lid velocities in [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 2 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. Failure rates, reliability and utility ratios were computed as specified in Section 2 and used to construct three composite linear solvers. The "optimal" composite, U, comprises base methods B3, B1, B4, and B2, arranged in increasing order of the utility ratio. The second composite, labeled T, comprises methods B1, B3, B4, and B2, in increasing order of time. The third composite, R, has methods in a random order B3, B4, B1, and B2. In our composites, 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 $1.e^{-5}$.

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
Utility ratio	4198	6242	2599	4824
Utility ratio was obtained using reliability measure of the linear solver.				

Table 2
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.

We report on the performance of the four base methods and three composites for nine simulations. 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

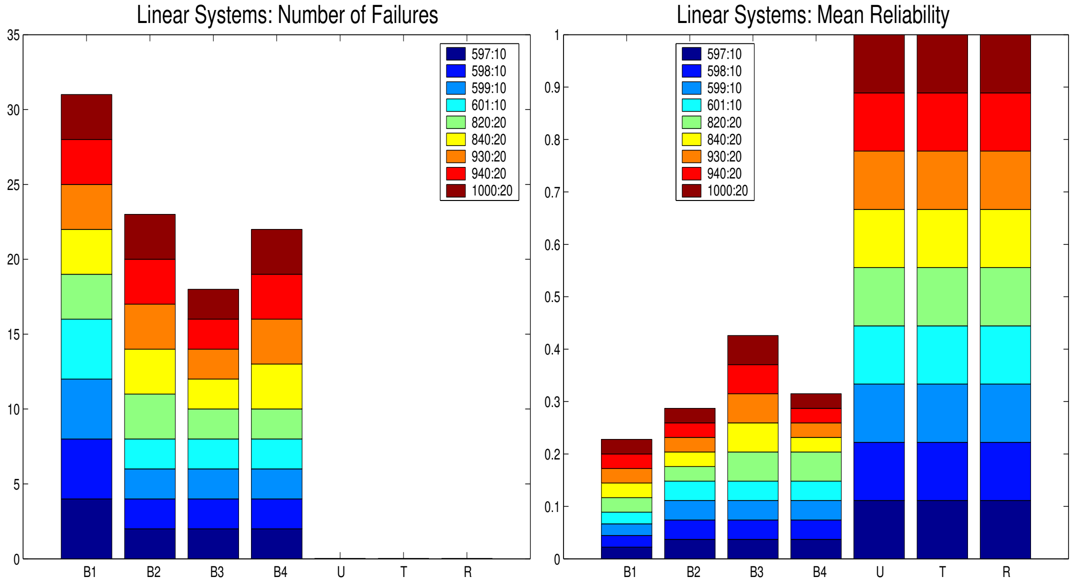

Fig. 3. The number of failures and the mean reliability of the linear solver using base linear solution methods B1, B2, B3, and B4 and composites U, T, and R.

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 shows the total number of failures and the reliability for base methods B1, B2, B3, and B4 and composites U, T, and R. 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 2). 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 4 and 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 T, which is based on least time. Since the initial methods in the composite often fail, however, the overall number of linear solver iterations for T correspondingly increases, as does the time per nonlinear iteration. On the other hand, for the utility ratio composite U, although the time per linear iteration is higher than for T, 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

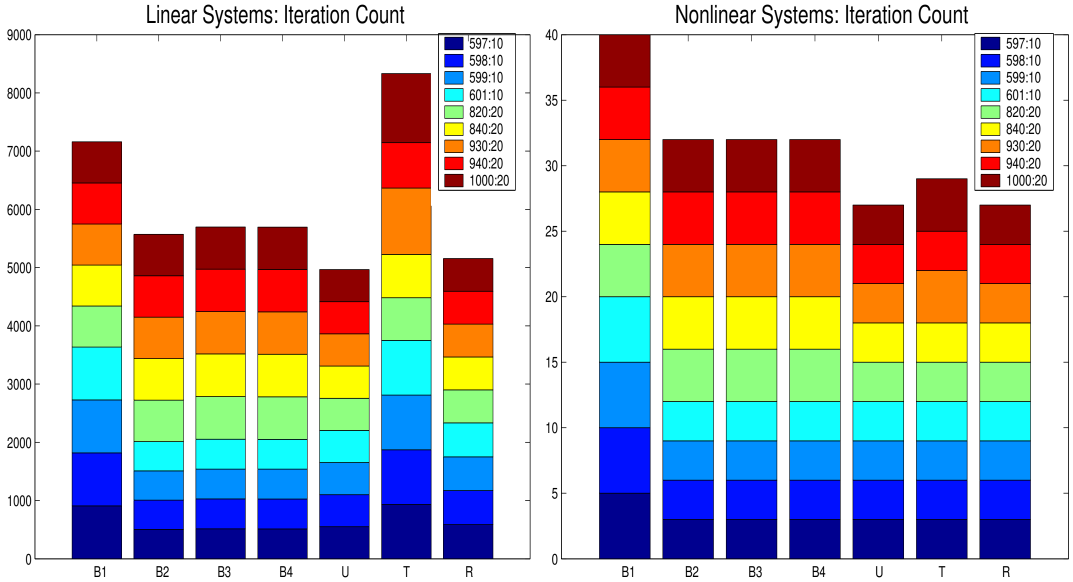

Fig. 4. Iteration counts for linear and nonlinear solution using base methods B1, B2, B3, and B4 and composites U, T, and R.

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 6 shows the total linear and nonlinear solution time over all nine simulations, and Table 3 summarizes the results shown in detail in Figures 3 through 6. These times are the least for composite U, in which the underlying methods are in increasing order of the utility ratio. However, these execution

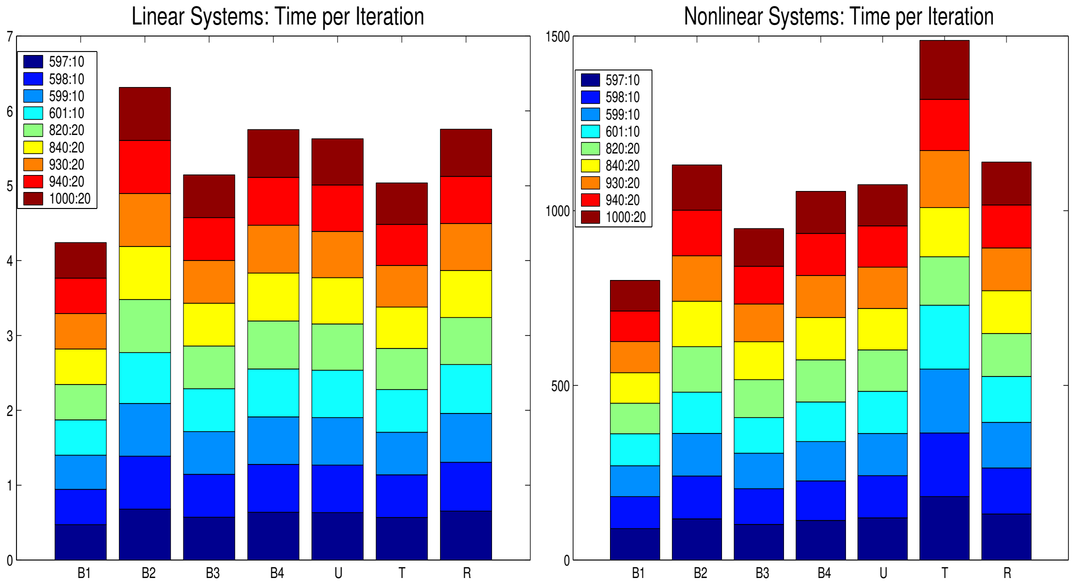

Fig. 5. Average time per iteration for linear and nonlinear solution.

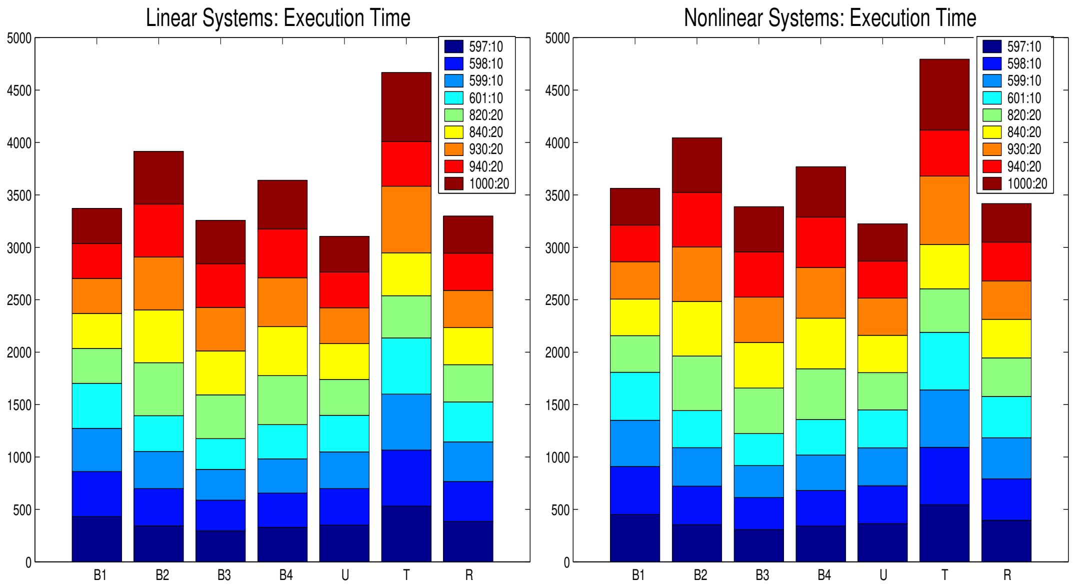

Fig. 6. Total time for linear and nonlinear solution using base methods B1, B2, B3, and B4, and composites U, T, and R.

times are not vastly different from those for the base method B1, even though the number of linear solver iterations for U is significantly lower (see Figure 4). This situation occurs partly because the decrease in nonlinear iterations from accurate linear system solution using U 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.

6 Conclusions

We have provided a combinatorial model for developing multimethod composite solvers that provide highly reliable solution while minimizing worst-case average execution time. We have shown how such composites can be instantiated with relative ease using advanced software environments. As demonstrated by our experiments for the PDE-based driven cavity flow simulation, our composite solvers can indeed provide improved performance and reliability compared with any single method.

Our composites were based on a static ordering of underlying methods. We

Metric	Base Methods				Composites		
	B1	B2	B3	B4	U	T	R
	Linear solver						
Time (seconds)	3371	3916	3258	3640	3105	4668	3299
Iteration count	7160	5571	5698	5696	4966	8332	5157
Number of Failures	31	23	18	22	0	0	0
Failure Rate(%)	77.5	71.9	56.2	67.7	0	0	0
	Nonlinear solver						
Time (seconds)	3563	4044	3387	3769	3224	4795	3417
Iteration count	40	32	32	32	27	29	27

Table 3
Summary of cumulative performance measures for nine simulations.

conjecture that long-running simulations can potentially benefit from adapting the composite to match evolving problem characteristics more accurately. We are currently investigating practical methods for developing such dynamic composites.

The traditional library approach for scientific computing software would impose prohibitive software development costs for instantiating composite methods. Consequently, advanced software environments such as PETSc are critical for implementing both static and dynamic composites. A key feature of such systems is that they provide a hierarchy of abstract interfaces, whose implementations can be selected at runtime. A more general component approach for high-performance scientific software is currently under development by the Common Component Architecture Forum (CCA) [2]. Recently, Norris et al. have developed some prototype CCA-compliant component interfaces for PDEs and optimization [17]; these interfaces are at moderate granularities such as a linear solve or a gradient evaluation with support for multiple underlying component implementations. These developments in advanced software architectures for scientific computing make composite methods both timely and practical. We plan to implement CCA-compliant components that would allow both static and dynamic algorithm composition to improve application performance. A longer-term goal is to develop component interfaces for composites that simplify their assembly and invocation.

Acknowledgments

We gratefully acknowledge the assistance of Barry Smith, who enhanced PETSc functionality to facilitate support of composite methods.

References

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, Achieving High Sustained Performance in an Unstructured Mesh CFD Application, Proceedings of SC99, 1999.

[2] 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, 1999, pp. 115-124, (see `www.cca-forum.org`).

[3] O. Axelsson, A Survey of Preconditioned Iterative Methods for Linear Systems of Equations, BIT, 25 (1987), pp. 166–187.

[4] S. Balay, W. Gropp, L.C. McInnes, and B. Smith, Efficient Management of Parallelism in Object Oriented Numerical Software Libraries, in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[5] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L.C. McInnes, B. Smith, and H. Zhang, PETSc Users Manual, Tech. Rep. ANL-95/11 - Revision 2.1.3, Argonne National Laboratory, 2002 (see `www.mcs.anl.gov/petsc`).

[6] 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 (1996), pp. 91–110.

[7] S. Bhowmick, P. Raghavan, and K. Teranishi, A Combinatorial Scheme for Developing Efficient Composite Solvers, Lecture Notes in Computer Science, vol. 2330, eds. P. M. A. Sloot, C. K. Tan, J. J. Dongarra, and A. G. Hoekstra, eds., Springer Verlag, 2002, pp. 325–334.

[8] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, A Component Based Services Architectures for Building Distributed Applications, Proceedings of High Performance Distributed Computing, 2000.

[9] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Berg, S. Diwan, and M. Govindaraju, The Linear System Analyzer, Enabling Technologies for Computational Science, Kluwer, Amsterdam, 2000.

[10] T. S. Coffey, C. T. Kelley, and D. E. Keyes, Pseudo-Transient Continuation and Differential-Algebraic Equations, submitted to the open literature, 2002.

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

[12] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, and H. Tufo, FLASH: An Adaptive-Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes, Astrophysical Journal Supplement, 131 (2000), pp. 273–334, (see `www.asci.uchicago.edu`).

[13] A. George, J. R. Gilbert, and J. W. H. Liu, Graph Theory and Sparse Matrix Computation, Springer-Verlag, 1993.

[14] G. Golub and D. O'Leary, Some History of the Conjugate Gradient and Lanczos Methods, SIAM Review, 31 (1989), pp. 50–102.

[15] G. H. Golub, C. F. Van Loan, Matrix Computations, 3rd ed. The John Hopkins University Press, Baltimore, Maryland, 1996.

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

[17] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, and B. F. Smith, Parallel Components for PDEs and Optimization: Some Issues and Experiences, To appear in Parallel Computing, 2002.

[18] X. Z. Tang, G. Y. Fu, S. C. Jardin, L. L. Lowe, W. Park, and H. R. Strauss, Resistive Magnetohydrodynamics Simulation of Fusion Plasmas, PPPL-3532, Princeton Plasma Physics Laboratory, 2001.