Parallel Simulation of Compressible Flow Using Automatic Differentiation and PETSc

Paul D. Hovland and Lois C. McInnes

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439-4844
email: {hovland,mcinnes}@mcs.anl.gov

Abstract

Many aerospace applications require parallel implicit solution strategies and software. We consider the use of two computational tools, PETSc and ADIFOR, to implement a Newton-Krylov-Schwarz method with pseudo-transient continuation for a particular application, namely, a steady-state, fully implicit, three-dimensional compressible Euler model of flow over an M6 wing. We describe how automatic differentiation (AD) can be used within the PETSc framework to compute the required derivatives. We present performance data demonstrating the suitability of AD and PETSc for this problem. We conclude with a synopsis of our results and a description of opportunities for future work.

Key words: Compressible Euler, PETSc, Nonlinear PDEs, Automatic Differentiation

1 Introduction

Parallel implicit solution strategies are important in aerodynamic applications modeled by PDEs with disparate temporal and spatial scales. Within this family of techniques, Newton-Krylov methods have been shown to be widely applicable, as these offer the advantage of rapid convergence when an iterate is near a problem's solution, and can incorporate pseudo-transient continuation,

⋆ 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.

line search, or trust region approaches that extend the radius of convergence of basic Newton techniques. This article considers issues in the application of such methods to a particular aerospace application, namely, a steady-state, fully implicit, three-dimensional compressible Euler model of flow over an M6 wing. Our emphasis in this work is exploring the utility of automatic differentiation (AD) technology to provide derivative information required by these techniques, and understanding the benefits of using AD in conjunction with numerical libraries that provide encapsulated expertise for parallel inexact Newton methods.

We apply a Newton-Krylov-Schwarz technique with pseudo-transient continuation (ΨNKS) and adaptive advancement of the CFL number to a discretization within a legacy CFD code (the JULIANNE code of Whitfield and Taylor [29]), which employs a C-H mesh with a second-order finite volume discretization. We use the nonlinear solvers within PETSc (the Portable, Extensible Toolkit for Scientific computing) [1–3], which provide a robust and flexible suite of data-structure-neutral numerical routines for Newton-like methods. These solvers generally require the action of a Jacobian matrix of derivatives, which can be approximated with finite differences. However, the convergence rate and robustness are typically improved if the application programmer supplies code to compute the derivatives analytically. Unfortunately, such code is often very complicated and difficult to program correctly by hand, especially in parallel.

We describe how automatic differentiation (AD) can be used within the PETSc framework to automatically generate code for derivative computations. In particular, we highlight how the use of high-level abstractions for mathematical objects and parallel communication enables us to decouple issues of parallelism from the local nonlinear function computations and thus simplifies the task of applying AD.

Section 2 provides an overview of the Euler model, while Section 3 discusses the pseudo-transient Newton-Krylov-Schwarz algorithmic framework. Section 4 introduces the primary software tools used in this work, namely, the numerical PDE software within PETSc and the automatic differentiation technology within ADIFOR [4]. Section 5 discusses the use of finite differences and automatic differentiation for computing derivatives within such simulations. Section 6 presents experimental results that demonstrate the suitability of AD and PETSc within this parallel transonic Euler flow model. We conclude in Section 7 with a synopsis of our results and a discussion of opportunities for future work in integrating PETSc and AD.

2 Compressible Inviscid Flow Model

To illustrate our approaches in parallel algorithms and software, we solve the steady-state, three-dimensional compressible Euler equations on mapped, structured meshes using a second-order, Roe-type, finite-volume discretization. The governing system of PDEs can be expressed in coordinate-invariant form by

$$\nabla \cdot (\rho \mathbf{u}) = 0, \tag{1}$$
$$\nabla \cdot (\rho \mathbf{u}\mathbf{u} + pI) = 0, \tag{2}$$
$$\nabla \cdot ((\rho e + p)\mathbf{u}) = 0, \tag{3}$$

where ρ, $\mathbf{u}$, and e represent the density, three-dimensional velocity, and energy density fields, respectively, and the pressure field p is determined by an algebraic equation of state, $e = \frac{p}{\gamma - 1} + \frac{1}{2}\rho(u^2 + v^2 + w^2)$, where γ is the ratio of specific heats.

As described in [29], this system and its discretization are standard and produce a nonlinear system of the form

$$f(u) = 0, \tag{4}$$

where u is a vector of unknowns representing the state of the system, and $f(u)$ is a vector-valued function of residuals of the governing equations. The basis for our implementation is a sequential Fortran 77 code from D. Whitfield called JULIANNE, which includes a discrete Newton-relaxation pseudo-transient continuation solver with explicit enforcement of boundary conditions. We reused valuable discretization modules within this sequential legacy code in the form of flux balance routines for computation of steady-state residuals and finite-difference Jacobians. The function evaluations are undertaken to second order, while the Jacobian matrices (used only for preconditioning in this work) are evaluated to first order. We replaced the explicit characteristic boundary conditions with fully implicit characteristic variants; see [28] for the derivation of the explicit forms of the boundary conditions and [26,27,29] for the importance of using an implicit form of these boundary conditions to maintain stability as timesteps are adaptively increased. We also added the differentiable Van Albada limiter option to three existing limiters [12].

3 Algorithmic Framework

Pseudo-transient continuation methods are designed to solve discretized steady-state systems of nonlinear boundary value problems of the form given by equation (4). As discussed in [16], these methods solve a sequence of problems derived from the model $\frac{\partial u}{\partial t} = -f(u)$, namely,

$$g_\ell(u) \equiv \frac{1}{\tau_\ell}(u - u^{\ell-1}) + f(u) = 0, \ \ell = 1, 2, \ldots. \tag{5}$$

When the timestep τ_ℓ is sufficiently small, the physical transient is followed so that a feasible sequence of states results. In addition, the Jacobians associated with $g_\ell(u) = 0$ are well conditioned when τ_ℓ is small. The timestep τ_ℓ is advanced from $\tau_0 \ll 1$ to $\tau_\ell \to \infty$ as $\ell \to \infty$, so that the iterate u^ℓ approaches the root of $f(u) = 0$. Details of our timestepping scheme, which employs modified Successive Evolution-Relaxation (SER) [19], are discussed in [12]. In particular, after an initial frozen phase, the timestep grows in inverse proportion to residual norm progress:

$$\tau_\ell = \tau_{\ell-1} \cdot ||f(u^{\ell-2})||/||f(u^{\ell-1})||, \tag{6}$$

within bounds relative to the current step.

We use an inexact Newton method (see, e.g., [20]) to solve $f(u) = 0$ through the two-step sequence of (approximately) solving the Newton correction equation

$$f'(u^{\ell-1})\ \delta u^\ell = -f(u^{\ell-1}), \tag{7}$$

in the sense that the linear residual norm $||f'(u^{\ell-1})\ \delta u^\ell + f(u^{\ell-1})||$ is sufficiently small, and then updating the iterate via

$$u^\ell = u^{\ell-1} + \alpha \cdot \delta u^\ell, \tag{8}$$

where α is a scalar such that $0 < \alpha \leq 1$. The right-hand side of the linear Newton correction equation, $f(u^{\ell-1})$, is evaluated to full discretization order, so the inexactness arises from incomplete convergence or from the employment of an inexact Jacobian. The Jacobian may be approximated at various levels: by evaluating it to a lower discretization order, by lagging it to a previous iteration, or by performing a finite difference approximation of the differentiation.

To achieve efficient performance on parallel architectures with multilevel memory hierarchies, we use Newton-Krylov-Schwarz methods, in which the Newton correction equation (7) is solved with a Krylov method that is preconditioned by a Schwarz technique. In other words, we increase the linear convergence rate at each nonlinear iteration by transforming the linear system (7) 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. Schwarz-type preconditioners (see, e.g., [25]) are applied on a subdomain-by-subdomain basis through a local Jacobian approximation; these methods provide good data locality for parallel implementations over a range of parallel granularities. In particular, we use the restricted additive Schwarz method (RASM) [9], which eliminates interprocess communication during the interpolation phase of conventional additive Schwarz, and for this Euler model converges more rapidly than conventional additive Schwarz [12].

Of particular interest in this Euler simulation are so-called matrix-free Newton-Krylov methods (see, e.g., [8]), where the Jacobian matrix $f'(u)$ need not be formed explicitly, since only the action of the Jacobian on a vector, $f'(u) * v$, is needed. The matrix-free approach enables cost-effective computation of up-to-date Jacobian-vector products at each nonlinear iteration. In contrast, the time for explicitly forming and storing the Jacobian matrix at each nonlinear iteration is often prohibitively expensive, so that we typically lag this computation. In other words, in practice we hold fixed for several nonlinear iterations the explicitly computed Jacobian, which may be used to define the Newton system and/or to precondition it. Within the context of fully implicit pseudo-transient Newton-Krylov methods with adaptively advancing CFL, the cost-effective matrix-free computation of up-to-date Jacobian-vector products is critical for achieving nearly quadratic convergence. Section 6.1 provides a brief review of investigations (e.g., [26,12]) of these issues for this compressible Euler model.

4 Numerical Software Tools

The high-performance computing community has explored a variety of approaches for implementing parallel scientific applications, including parallel languages, parallelizing compilers, automated source-to-source parallel translators, and parallel libraries. For large-scale PDE-based simulations, we advocate for the use of parallel libraries, which offer the encapsulation of expertise in forms that are directly usable as building blocks in large-scale applications. Libraries can of course be used in conjunction with these other approaches where appropriate. Moreover, when designed to exploit abstractions in mathematics and interprocess communication, parallel libraries offer the added benefit of facilitating the interface between numerical software and automatic

differentiation tools for derivative computations. As shown in Section 6, such capabilities can be quite powerful in helping to achieve good overall performance for applications involving nonlinear PDEs. This section introduces the tools used in our investigations of transonic Euler flow, namely, the PETSc library and the ADIFOR automatic differentiation software.

4.1 PETSc

Our strategy for parallelizing the legacy Euler model introduced in Section 2 and investigating ΨNKS algorithms is to use PETSc [1–3], a suite of data structures and routines for the scalable solution of scientific applications modeled by PDEs. The library handles in a highly efficient way, through a uniform interface, the low-level details of the distributed-memory hierarchy. Examples of such details include striking the right balance between buffering messages and minimizing buffer copies, overlapping communication and computation, organizing node code for strong cache locality, preallocating memory in sizable chunks rather than incrementally, and separating tasks into one-time and every-time subtasks 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.

The software integrates a hierarchy of components that range from low-level distributed data structures for meshes, vectors, and matrices to 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.

Figure 1 illustrates the calling tree of a typical ΨNKS application using the nonlinear solvers within PETSc. The top-level user routine performs I/O related to initialization, restart, and postprocessing; it also calls PETSc subroutines to create data structures for vectors and matrices and to initiate the nonlinear solver. Subroutines with the PETSc library call user routines for function evaluations $f(u)$ and (approximate) Jacobian evaluations $f'(u)$ at given state vectors.

Fig. 1. Schematic diagram of a ΨNKS application, showing a user-provided driver and user-provided callback routines for evaluating the nonlinear residual vector and corresponding Jacobian at PETSc-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 object-oriented techniques of encapsulation and polymorphism enable the

support of various storage schemes and solvers through a single user interface. Such flexibility is critical for enabling both library developers and application programmers to easily experiment with different algorithmic and data structure capabilities. For example, as mentioned in Section 3, of particular interest in this Euler simulation are matrix-free Newton-Krylov methods, where only the action of the Jacobian on a vector, $f'(u) * v$, is needed. As discussed within Section 6, we have investigated a range of matrix-free and matrix-explicit Newton-Krylov-Schwarz variants simply by specifying various options at runtime; no changes in the application code or library were required for these experiments.

4.2 Automatic Differentiation

We advocate the use of automatic differentiation for computing the Jacobian $f'(u)$. Automatic differentiation (AD) is a technique that, given a program or subprogram to compute a function, produces a program or subprogram to compute the derivatives of that function. It does so by augmenting the original program with instructions for computing partial derivatives and for propagating those derivatives according to the chain rule of differential calculus.

Unlike traditional symbolic manipulation, the techniques of automatic differentiation are directly applicable to computer programs of arbitrary length containing branches, loops, and subroutines. AD enables derivatives to be updated easily whenever the original code is modified. Unlike divided differences, there is no truncation error.

The derivative augmentation can be implemented in one of two ways: transforming the source code using a precompiler or overloading the arithmetic operators and intrinsic functions. Each approach has its own merits [5]; we focus on the source transformation approach. Among the available source transformation tools are ADIFOR [4], Odyssee [22], and TAMC [10] for Fortran 77 and ADIC [7] for C/C++. We used ADIFOR for this project.

5 Computing Derivatives

As discussed in Section 3, ΨNKS algorithms require derivatives of the residual, $f(u)$, either via explicit storage of a Jacobian matrix, $f'(u)$, or in the form of Jacobian-vector products, $f'(u) * v$, both of which may be approximated. The derivatives can be computed analytically (a task that is typically time-consuming and error prone in terms of programmer time), approximated via finite differences, or computed with automatic differentiation. Section 5.1 dis-

cusses finite differencing issues, while Sections 5.2 and Section 5.3 explain automatic differentiation for computing explicit Jacobians and Jacobian-vector products, respectively.

5.1 Computing $f'(u)$ and $f'(u)v$ Using Finite Differences

Matrix-free Jacobian-vector products can be defined by directional differencing of the form

$$f'(u)v \approx \frac{f(u+hv) - f(u)}{h},$$

where the differencing parameter h is chosen in an attempt to balance the relative error in function evaluation with the magnitudes of the vectors u and v. Selection of an appropriate parameter is nontrivial, as values either too small or too large will introduce rounding errors or truncation errors, respectively, that can lead to breakdowns. Investigators with relatively small, well-scaled discrete problems sometimes report satisfaction with a simple choice of h, approximately the square root of the "machine epsilon" for their machine's floating-point system. A typical double-precision machine epsilon is approximately 10^{-16}, with a corresponding appropriate h of approximately 10^{-8}. More generally, adaptivity to the vectors u and v is desirable.

We choose the differencing parameter dynamically using the technique proposed by Brown and Saad [8], namely,

$$h = \frac{w}{||v||^2} \max\left[|u^T v|, \hat{u}^T |v|\right] \operatorname{sign}(u^T v), \tag{9}$$

where $|v| = [|v_1|, ..., |v_n|]^T$, $\hat{u} = [|\hat{u}_1|, ..., |\hat{u}_n|]^T$ for $\hat{u}_j > 0$ being a user-supplied typical size of u_j, and $w \approx$ square root of relative error in function evaluations.

Determining an appropriate estimation of the relative noise or error in function evaluations is crucial for robust performance of matrix-free Newton-Krylov methods. Assuming double-precision accuracy (or $w \approx 10^{-8}$) is inappropriate for many large-scale applications. A more appropriate relative error estimate for the compressible flow problems considered in this work is $w = 10^{-6}$, as determined by noise analysis techniques currently under investigation by McInnes and Moré [17].

In addition to evaluating the Jacobian-vector products with directional differencing within Newton-Krylov methods, we can construct the preconditioner in a blackbox manner, without recourse to analytical formulae for the Jacobian

elements, by directional differencing as described in [29] and as provided in the JULIANNE code.

5.2 Applying AD to the Function

Applying AD to the user routine for evaluating $f(u)$ to produce the required Jacobian (or Jacobian-vector products) is challenging. In this application the user code is a mixed-language subprogram that invokes PETSc communication functions (which in turn invoke MPI [18] primitives). Although in principle it is possible to apply AD to mixed-language, parallel programs [14,15], at present there is no implementation of AD for such codes. Furthermore, applying AD to the entire PETSc toolkit in order to support the differentiation of the function is overkill. We therefore propose an alternative approach that relies upon some insight into the structure of the parallel function computation.

We use a distributed-memory computational paradigm, in which the mesh and associated data are partitioned at runtime across the participating processes so that each process "owns" a unique subset of the mesh and the corresponding unknowns of the problem. Discretization typically follows the basic philosophy of "owner-computes." For efficient distributed memory computations, the process that stores mesh and associated data for a particular region of the global problem domain calculates most, though not necessarily all, of the corresponding local part of the discretization. Typically, the computation of a nonlinear residual, or function, based on the discretization of nonlinear PDEs (including the Euler code considered here) exhibits the structure diagrammed in Figure 2. First, generalized vector scatters distribute ghost data, or the bordering portions of the vector that are owned by neighboring processors. This phase is followed by a local function evaluation involving no communication. Finally, the distributed vector containing the discretized nonlinear function is assembled in parallel. (See [2] for a detailed discussion of these communication issues, including message passing-oriented optimizations.)

To differentiate such a function, we explicitly separate the local function evaluation from the scattering and assembly phases. As illustrated in Figure 3, we apply AD to the local function evaluation subprogram to produce code for the local Jacobian evaluation. The scattering and assembly phases need only be modified slightly (at present, manually) to initialize the derivative values and assemble the distributed Jacobian matrix.

Fig. 2. Schematic diagram of a ΨNKS application, showing the various phases for parallel evaluation of the nonlinear residual vector and corresponding Jacobian.

Fig. 3. Schematic diagram of the use of automatic differentiation tools to generate the Jacobian routine for a nonlinear PDE computation.

5.3 Computing Jacobian-Vector Products using AD

To this point, our discussion has assumed that we are interested in computing a dense Jacobian. This is not the case. For this problem, $f'(u)$ is very sparse and if a matrix-free method is used, we need only Jacobian-vector products. Fortunately, AD (or, more precisely, the forward mode of AD [11]) provides the ability to compute not just the Jacobian matrix $f'(u)$ but also the product $f'(u) \times S$ where S is an arbitrary matrix, often called the seed matrix. Furthermore, the cost of computing $f'(u) \times S$ is roughly proportional to $p+1$, where p is the number of columns in S. Thus, the cost of computing a Jacobian-vector product $f'(u)v$ is very low relative to the cost of computing the full Jacobian.

The cost of computing $f'(u) \times S$ using the forward mode of AD can be bounded as $3p + 1$ times the cost of computing $f(u)$. The "+1" term comes from the fact that we must recompute $f(u)$, because intermediate values are used in computing derivatives. The multiplier 3 comes from the fact that computing the derivatives of highly nonlinear functions can be somewhat more expensive than computing the functions themselves. In practice, this multiplier is usually about 1, with an observed range of about 0.5 to 3. Thus, a good estimate of the cost of computing $f'(u)v$ is about 2 function evaluations. This is twice the cost of approximating $f'(u)v$ using finite differences, which requires only a single evaluation of the function $f(u + hv)$ (the value of $f(u)$ can be reused).

As discussed earlier, in this application we precondition with a lagged Jacobian computed using a lower-order discretization with some terms neglected. It is, however, possible to compute the full, sparse Jacobian using an appropriate seed matrix S based on coloring [6]. This reduces the cost of computing $f'(u)$ to appromately $k+1$ times the cost of computing $f(u)$, where k is the chromatic number for $f'(u)$.

6 Experimental Results

The following experiments with the Euler simulation modeled transonic flow over an ONERA M6 wing, a standard three-dimensional test case for which extensive experimental data is given in [24]. A frequently studied parameter combination combines a freestream Mach number of 0.84 with an angle of attack of 3.06°. This transonic case gives rise to a characteristic λ-shock, as depicted in Fig. 4.

Fig. 4. Mach number contours (the local tangential velocity magnitude divided by the local sound speed) of the converged flowfield on the upper wing surface for a mesh with 224,264 nodes.

6.1 Review of Previous Work

In previous work [12] we compared the convergence of various matrix-explicit and matrix-free pseudo-transient inexact Newton approaches: explicit boundary conditions (limiting CFL to 7.5), implicit boundary conditions with the same CFL, implicit boundary conditions with advancing CFL (using the strategy presented in Section 3), and implicit boundary conditions with advancing CFL and matrix-free application of the Jacobian. In all cases the nonlinear function was evaluated to second-order accuracy. Whenever required, the explicit Jacobian was computed to first-order accuracy in space via finite differences, refreshed once every ten pseudo-timesteps, and stored with a compressed sparse block scheme that handles five degrees of freedom per node (three-dimensional momentum, internal energy, and density). This approximate Jacobian served as the left-hand-side matrix of the Newton systems for the matrix-explicit methods or as the preconditioner for the matrix-free method. The frequency of Jacobian recomputation provides a reasonable trade-off in terms of convergence rate and the computational cost of matrix evaluation.

The key observation from this previous work was that the combination of implicit boundary conditions coupled with the higher-order Jacobian-vector product discretization enabled by the matrix-free technique solves the nonlinear problem to machine precision several times faster than do the other methods. The impact of using this combination increases with problem size and is consistent across various processor configurations. Neither the use of implicit boundary conditions alone nor the use of increasing CFL with a low-order Jacobian allows the approach of quadratic convergence. Consequently, all numerical experiments in this article employ matrix-free techniques with an adaptively advancing CFL number.

6.2 Matrix-Free Experiments Using Finite Differencing and Automatic Differentiation

All of the following numerical experiments used the pseudo-transient matrix-free Newton-Krylov-Schwarz algorithm, as discussed in Section 3. The linearized Newton correction equations were solved by using restarted GMRES [23] preconditioned with RASM. All experimental results presented here were run on an IBM SP with 120 MHz P2SC nodes with two 128 MB memory cards each and a TB3 switch; in addition, we have observed analogous qualitative behavior on a range of other current parallel architectures.

Figure 5 shows the convergence behavior for the matrix-free Newton method

Fig. 5. Comparison of convergence using pseudo-transient continuation of a matrix-free Newton method with finite differencing (FD) and automatic differentiation (AD); iterations (left) and time (right).

Table 1
Comparison of times for matrix-free ΨNKS using finite differences and automatic differentiation. We consider a mesh of dimension $98 \times 18 \times 18$ and one approximately eight times as large, with a dimension of $194 \times 34 \times 34$.

using Jacobian-vector products computed via finite differences and automatic differentiation. The finite difference derivatives were computed using a variety of step sizes according to equation (9), with estimates for the relative error w ranging from 10^{-7} to 10^{-5}. As the iteration history illustrates, the choice of w is critical to achieving the proper balance between roundoff and truncation error.

These experiments, as well as those whose results are summarized in Figures 6 through 8, employ a mesh of dimension $98 \times 18 \times 18$, which corresponds to a total system size of 158,760 unknowns (where there are five degrees of freedom per node: density, internal energy, and three-dimensional momentum). This mesh is sufficiently fine to resolve flow features and enable algorithmic analysis. All experiments presented here for this mesh used a four-processor distribution and a single degree of overlap for the RASM preconditioner.

If the value used for w is too small (10^{-7}), roundoff error (underflow) in the derivative approximation leads to stagnation of the algorithm, and convergence is never achieved. On the other hand, if the value used for w is too large (10^{-5}), truncation error in the derivative approximation slows the convergence. With an intermediate choice for w (10^{-6}), a good approximation to the derivatives is achieved, and the algorithm converges rapidly. Convergence is even more rapid when the analytic derivatives provided by AD are used.

While AD exhibits rapid convergence in terms of number of iterations, it is obvious from the time history that it does not fare so well in terms of time to solution. As discussed in Section 5.3, the cost of computing a Jacobian-vector product using AD is usually about twice the cost of computing it using finite differences. For this particular problem, independent of mesh size or number of processors, we observe a ratio of about 2.4 (see Table 1). This is somewhat amortized by the overhead of the linear solve (including preconditioning) and the update of the Newton iterate, so the time per pseudo-transient Newton iteration is about 70% longer for the AD case.

Figure 5 also reveals that accurate derivatives are important only toward the end of the solution process. The finite difference approximations based on a step size that is too large or too small converge at essentially the same rate as the analytic derivatives until the residual norm is reduced by about 10^4. This

Fig. 6. Comparison of convergence using matrix-free ΨNKS with finite differencing (FD), automatic differentiation (AD), and hybrid variants (FD/AD) with various switching parameters, s; iterations (left) and time (right).

Table 2
Comparison of total solution time using matrix-free ΨNKS with finite differencing (FD), automatic differentiation (AD), and hybrid variants (FD/AD) using a switching parameter $s = .005$ with various differencing parameters, w. The columns headed "Ratio" represent the ratio to the best method.

observation, coupled with the fact that the AD derivatives provide excellent convergence in terms of iterations but poorer performance in terms of time, led us to consider a hybrid strategy, in which finite differences are used for the early iterations and AD is used for the later iterations.

Many possible mechanisms exist for deciding when to switch from finite difference approximations to the analytic derivatives of AD. For this study, we selected a simple strategy based on relative reduction in the residual norm. Finite differences are used until the relative reductions drops below some threshold s, after which AD is used. As Figure 6 illustrates, this strategy is effective in achieving rapid convergence in terms of both time and iterations. As expected, the larger the value of s (thus, the sooner one switches from finite differences to analytic derivatives), the faster the convergence in terms of iterations. However, regardless of the value used for s, there is little difference in terms of time to solution.

A hybrid scheme is particularly attractive because of its resilience to an imperfect choice for w. Table 2 demonstrates that the hybrid strategy provides performance within 20% of the best finite difference scheme, for a variety of w. In contrast, straight finite difference schemes with the same values for w may take up to 40% longer, or fail to converge altogether.

Figure 7 shows the history of CFL advancement, which is computed according to equation (6). The CFL number exhibits the desirable trait of increasing monotonically from its starting point of 7.5 through its maximum of 10,000 for the AD and hybrid FD/AD approaches. In contrast, for all of the FD cases the CFL number encounters segments of decrease, which corresponds to an increase of the residual norm.

Fig. 7. Adaptively advancing CFL number for matrix-free ΨNKS with finite differencing (FD), automatic differentiation (AD), and a hybrid approach (FD/AD).

Figure 8 shows virtually identical convergence of various flow features as a function of pseudo-timestep number for the various matrix-free approaches. The left-hand graph illustrates the evolution of the shock structure (characterized by the number of supersonic mesh points on the upper surface of the airfoil). The right-hand graph illustrates convergence of the dimensionless

Fig. 8. Number of supersonic mesh points (left); coefficients of lift and drag (right) for matrix-free ΨNKS with finite differencing (FD), automatic differentiation (AD), and a hybrid approach (FD/AD).

Fig. 9. Convergence history of matrix-free ΨNKS with hybrid FD/AD across 8, 16, 32, and 64 processors; iterations (left) and time (right) for a mesh of dimension $194 \times 34 \times 34$.

Fig. 10. Scalability of matrix-free ΨNKS with hybrid FD/AD across 8, 16, 32, and 64 processors. Time (solid line) scales nearly linearly (dotted line) for a mesh of dimension $194 \times 34 \times 34$. The number of nonlinear iterations (dashed line) shows a modest increase as the number of processors increases.

aerodynamic functionals for lift and drag coefficients.

The graphs in Figures 9 and 10 show the scaling of the complete nonlinear simulation for a mesh of dimension $194 \times 34 \times 34$ (corresponding to 1,121,320 unknowns) using restarted GMRES on 8, 16, 32, and 64 processors. The restricted additive Schwarz preconditioner is used with one level of overlap for the 8-processor case and with two levels of overlap for the 16-, 32-, and 64-processor cases. Extensive experiments in [12] determined that these degrees of overlap are optimal for these processor configurations. The data indicate a modest decrease in nonlinear convergence rate as the number of processors grows; overall solution times scale reasonably well.

7 Summary and Future Work

We described a methodology for using PETSc and automatic differentiation to implement a parallel Newton-Krylov-Schwarz technique with pseudo-transient continuation. We examined the advantages and disadvantages of using AD relative to finite difference approximations, and we introduced a hybrid scheme that marries the efficiency of finite differences with the robustness of AD. We presented experimental results from the application of these techniques to a transonsic Euler flow model.

Three opportunities for future work are apparent: improving the performance of AD relative to finite differences, exploring new hybrid schemes, and automating much of the work that was performed manually for this application. Performance can be improved by using Krylov methods that require multiple matrix-vector products. Such methods are receiving increased attention because of their potential for overcoming the memory bandwidth limitations of modern computer architectures [13]. Performance can also be improved by eliminating in the AD-generated code statements used for computing $f(u)$ but not $f'(u)$; this task could be automated through the use of tools for program

chopping [21]. New hybrid schemes might involve the use of probing or other strategies to detect when finite differences break down. The work required to automate differentiation of the complete user routine for nonlinear function evaluation includes extending the AD tools to provide more support for mixed-language programming, enhancing the AD tools to recognize some of the library-provided functions, and automating the generation of code for the parallel Jacobian assembly.

References

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc Web page. http://www.mcs.anl.gov/petsc.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. 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.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, Apr. 1999.

[4] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[5] C. Bischof and A. Griewank. Tools for the automatic differentiation of computer programs. In *ICIAM/GAMM 95: Issue 1: Numerical Analysis, Scientific Computing, Computer Science*, pages 267–272, 1996. Special Issue of Zeitschrift für angewandte Mathematik und Mechanik (ZAMM).

[6] C. Bischof and P. Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Report ANL/MCS-TM-158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[7] C. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[8] P. N. Brown and Y. Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Scientific and Statistical Computing*, 11:450–481, 1990.

[9] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. Technical Report CU-CS 843-97, Computer Science Department, University of Colorado-Boulder, 1997. Accepted by SIAM J. Scientific Computing.

[10] R. Giering and T. Kaminski. Recipes for adjoint code construction. Technical Report 212, Max-Planck Institut für Meteorologie Hamburg, 1996.

[11] A. Griewank. On automatic differentiation. Technical Report MCS–P10–1088, Mathematics and Computer Science Division, Argonne National Laboratory, 1988.

[12] W. Gropp, D. Keyes, L. McInnes, and M. Tidriri. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. Technical Report 98-24, ICASE, Aug. 1998. To appear in the *International J. High-Performance Computing Applications*, 14(2).

[13] W. Gropp et al. Unpublished information, Mathematics and Computer Science Division, Argonne National Laboratory, 2000.

[14] P. Hovland and C. Bischof. Automatic differentiation of message-passing parallel programs. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[15] P. D. Hovland. *Automatic Differentiation of Parallel Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, May 1997.

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

[17] L. C. McInnes and J. J. Moré. Unpublished information, Mathematics and Computer Science Division, Argonne National Laboratory, 2000.

[18] MPI: A message-passing interface standard. *International J. Supercomputing Applications*, 8(3/4), 1994.

[19] W. Mulder and B. V. Leer. Experiments with implicit upwind methods for the Euler equations. *J. Computational Physics*, 59:232–246, 1985.

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

[21] T. Reps and G. Rosay. Precise interprocedural chopping. *ACM SIGSOFT Software Engineering Notes*, 20(4):41–52, 1995.

[22] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.

[23] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 7:856–869, 1986.

[24] V. Schmitt and F. Charpin. Pressure distributions on the ONERA M6 wing at transonic Mach numbers. Technical Report AR-138, AGARD, May 1979.

[25] B. F. Smith, P. Bjørstad, and W. D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[26] M. D. Tidriri. Schwarz-based algorithms for compressible flows. Technical Report 96-4, ICASE, Jan. 1996.

[27] M. D. Tidriri. Efficient preconditioning of Newton-Krylov matrix-free algorithms for compressible flows. *J. Computational Physics*, 132:51–61, 1997.

[28] D. L. Whitfield and J. M. Janus. Three-dimensional unsteady Euler equations using flux vector splitting. In *Proceedings of the AIAA 17th Fluid Dynamics, Plasma Dynamics, and Lasers Conference*, 1984.

[29] D. L. Whitfield and L. K. Taylor. Discretized Newton-relaxation solution of high resolution flux-difference split schemes. In *Proceedings of the AIAA Tenth Annual Computational Fluid Dynamics Conference*, pages 134–145, 1991.

Table 1:

	$98 \times 18 \times 18$ Mesh 4 Processors		$194 \times 34 \times 34$ Mesh 32 Processors	
	AD	FD	AD	FD
Time per f evaluation (sec)	.219	.218	.259	.259
Time per $f'(u)v$ evaluation (sec)	.531	.230	.614	.271
Ratio of time for $f'(u)v$ to f	2.42	1.06	2.37	1.05
Time per nonlinear iteration (sec)	6.84	4.07	7.81	4.65
Total nonlinear iterations	67	73	208	209
Total time (sec)	458	297	1624	971

Table 2:

		Nonlinear Iterations		Time to Solution	
Method	w	Number	Ratio	Time (sec)	Ratio
FD	1.e-5	89	1.33	408	1.38
FD	1.e-6	73	1.09	296	1.00
FD	1.e-7	–	∞	–	∞
FD/AD	1.e-5	69	1.03	341	1.15
FD/AD	1.e-6	67	1.00	315	1.06
FD/AD	1.e-7	71	1.06	356	1.20
AD		67	1.00	457	1.54