Computational Quality of Service in Parallel CFD *

L. McInnes[a], B. Norris[a], I. Veljkovic[b]

[a]Mathematics and Computer Science Division, Argonne National Laboratory,
9700 South Cass Avenue, Argonne, IL 60439-4844, *[mcinnes,norris]@mcs.anl.gov.*

[b]Department of Computer Science and Engineering, The Pennsylvania State University,
IST Building, University Park, PA 16802-6106, *veljkovi@cse.psu.edu*.

1. INTRODUCTION

Component-based environments provide opportunities to improve the performance, numerical accuracy, and other characteristics of parallel simulations in computational fluid dynamics (CFD). Because component-based software engineering combines object-oriented design with the powerful features of well-defined interfaces, programming language interoperability, and dynamic composability, it helps to overcome obstacles that make it difficult to share even well-designed traditional numerical libraries. Not only can applications be assembled from components selected to provide good algorithmic performance and scalability, but they can also be changed dynamically during execution to optimize desirable characteristics. This concept of the automatic selection and configuration of components to suit a particular computational purpose is called *computational quality of service* (CQoS) [7, 13]. CQoS embodies the familiar concept of quality of service in networking and the ability to specify and manage characteristics of the application in a way that adapts to the changing (computational) environment. The factors that affect performance are closely tied to a component's parallel implementation, its management of memory, the algorithms executed, the algorithmic parameters employed (for example, the level of overlap in an additive Schwarz preconditioner), and other operational characteristics. Scientific component software is also concerned with functional qualities, such as the level of accuracy achieved for a particular algorithm.

This paper presents an overview of new infrastructure for automated performance gathering and analysis of high-performance components, which is a key facet of our CQoS research, with emphasis on using these capabilities in parallel CFD simulations, such as flow in a driven cavity and compressible Euler flow. The remainder of this paper is organized as follows. Section 2 discusses parallel CFD applications and algorithms that motivate this infrastructure. Section 3 introduces the new framework for enabling CQoS in parallel nonlinear PDE-based applications, while Section 4 discusses conclusions and opportunities for future work.

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

2. MOTIVATING APPLICATIONS AND ALGORITHMS

Flow in a Driven Cavity. The first parallel application that we use to motivate and validate this work is driven cavity flow, which combines lid-driven flow and buoyancy-driven flow in a two-dimensional rectangular cavity. We use a velocity-vorticity formulation of the Navier-Stokes and energy equations, which we discretize using a standard finite-difference scheme with a five-point stencil for each component on a uniform Cartesian mesh; see [6] for a detailed problem description.

Compressible Euler Flow. Another motivating application is PETSc-FUN3D [2], which solves the compressible and incompressible Euler equations in parallel; the sequential model was originally developed by W. K. Anderson [1]. The code uses a finite-volume discretization with a variable order Roe scheme on a tetrahedral, vertex-centered unstructured mesh. The variant of the code under consideration here uses the compressible Euler equations to model transonic flow over an ONERA M6 wing, a common test problem that exhibits the development of a shock on the wing surface. Initially a first-order discretization is used, but once the shock position has settled down, a second-order discretization is applied. This change in discretization affects the nature of the resulting linear systems.

Newton-Krylov Algorithms. Both applications use inexact Newton methods (see, e.g., [12]) to solve nonlinear systems of the form $f(u) = 0$. We use parallel preconditioned Krylov methods to (approximately) solve the Newton correction equation $f'(u^{\ell-1})\ \delta u^{\ell} = -f(u^{\ell-1})$, and then update the iterate via $u^{\ell} = u^{\ell-1} + \alpha \cdot \delta u^{\ell}$, 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. Our implementations use the Portable, Extensible Toolkit for Scientific Computation (PETSc) [3], a suite of data structures and routines for the scalable solution of scientific applications modeled by PDEs.

Pseudo-Transient Continuation. For problems with strong nonlinearities, Newton's method often struggles unless some form of continuation is employed. Hence, we incorporate pseudo-transient continuation [9], a globalization technique that solves a sequence of problems derived from the model $\frac{\partial u}{\partial t} = -f(u)$, namely,

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

where τ^ℓ is a pseudo time step. At each iteration in time, we apply Newton's method to equation (1). As discussed by Kelley and Keyes [9], during the initial phase of pseudo-transient algorithms, τ^ℓ remains relatively small, and the Jacobians associated with equation (1) are well conditioned. During the second phase, the pseudo time step τ^ℓ advances to moderate values, and in the final phase τ^ℓ transitions toward infinity, so that the iterate u^ℓ approaches the root of $f(u) = 0$.

Adaptive Solvers. In both applications the linearized Newton systems become progressively more difficult to solve as the simulation advances due to the use of pseudo-transient continuation [9]. Consequently both are good candidates for the use of *adaptive* linear solvers [4, 11], where the goal is to improve overall performance by combining more robust (but more costly) methods when needed in a particularly challenging phase of solution with faster (though less powerful) methods in other phases. Parallel adaptive solvers are designed with the goal of reducing the execution time of the simulation by dynamically selecting the most appropriate method to match the characteristics of the current linear system.

A key facet of developing adaptive methods is the ability to consistently collect and access both runtime and historical performance data. Our preliminary research in adaptive methods [4, 11], which employed ad hoc techniques to collect, store, and analyze data, has clearly motivated the need for a framework to analyze performance and help to manage algorithmic adaptivity.

3. COMPONENT CQOS FOR PARALLEL CFD CODES

For a given parallel fluids problem, the availability of multiple solution methods, as well as multiple configurations of the same method, present both a challenge and an opportunity. On one hand, an algorithm can be chosen to better match the application's requirements. On the other hand, manually selecting a method in order to achieve good performance and reliable results is often difficult or impossible. Component-based design enables us to automate, at least partially, the task of selecting and configuring algorithms based on performance models, both for purposes of initial application assembly and for runtime adaptivity. Algorithmic parameters and different metrics of expressing CQoS properties that affect the performance of an algorithm will be referred to as *metadata*.

The Common Component Architecture (CCA) specification [5] defines a component model that specifically targets high-performance scientific applications, such as parallel CFD. Briefly, CCA components are units of encapsulation that can be composed to form applications; *ports* are the entry points to a component and represent public interfaces through which components interact; *provides* ports are interfaces that a component implements, and *uses* ports are interfaces that a component uses. A runtime framework provides some standard services to all CCA components, including instantiation of components, and *uses* and *provides* port connections. Components can be instantiated/destroyed and port connections made/broken at runtime, thereby allowing dynamic adaptivity of CCA component applications and enabling the implementation of the adaptive linear solver methods introduced above.

In this paper, we present a CCA component infrastructure that allows researchers to monitor and adapt a simulation dynamically based on two main criteria: the runtime information about performance parameters and the information extracted from metadata from previous instances (executions) of a component application. This infrastructure includes components for performance information gathering, analysis, and interactions with off-line databases. Figure 1 shows the principal components and their relationships in a typical application. This design makes development of adaptive algorithms easier and less error-prone, by separating as much as possible unrelated concerns from the adaptive strategy itself. By contrast, because the implementation was interleaved with unrelated functionality, our initial implementation of adaptive linear solvers ([4, 11]) became difficult to maintain (mixed code for multiple heuristics) and extend (e.g., adding new adaptive heuristics). This motivated us to define a simple interface that is flexible enough to enable the implementation of a wide range of adaptive linear solver heuristics. We have been able to reproduce some of our original results using this new infrastructure.

We briefly introduce the terminology used in our CQoS infrastructure. We collectively refer to performance-relevant attributes of a unit of computation, such as a component, as performance *metadata*, or just metadata. These attributes include algorithm or application parameters, such as problem size and physical constants, compiler optimization options, and execution information, such as hardware and operating system information. Performance metrics, also referred to as CQoS metrics, are also part of the metadata, for example, execution time and

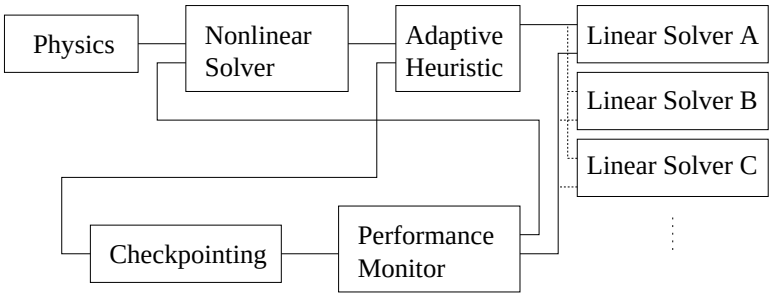

Figure 1. Components of CQoS framework.

convergence history of iterative methods. Ideally, for each application execution, the metadata should provide enough information to be able to repeat the run; we collectively refer to these metadata as an application instance, or *experiment*.

Our initial implementation of this infrastructure uses the Tuning and Analysis Utilities (TAU) toolkit [14] and the Parallel Performance Data Management Framework (PerfDMF) [8]. A summary of the components used for run-time adaptivity follows.

- **TAU Measurement Component**. This component collects runtime data from hardware counters, timing, and user-defined application-specific events. This component was provided by the developers of TAU, and complete implementation details can be found in [10].

- **Checkpoint Component**. This component checkpoints and stores the collected data into a runtime database that can be queried efficiently during the execution for the purpose of runtime performance monitoring and adaptation. The TAU profiling API can only give either callpath-based or cumulative performance information about an instrumented object (from the time execution started). Hence, we have introduced the Checkpoint component to enable us to store and retrieve data for the instrumented object during the application's execution (for example, number of cache misses for every three calls of a particular function). The period for checkpointing can be variable; the component can also be used by any other component in the application to collect and query context-dependent and high-level performance information. For example, a linear solver component can query the checkpointing component for performance metadata of the nonlinear solver (the linear solver itself has no direct access to the nonlinear solver that invoked it). We can therefore always get the latest performance data for the given instrumented object from the database constructed during runtime.

- **Metadata Extractor**. This component retrieves metadata from the database at runtime. After running several experiments, analyzing the performance data, and finding a common performance behavior with some parameter values, we store data summarizing this behavior in the database. An example of derived metadata is the rate of convergence of a nonlinear or a linear solver. During runtime, these data are used in adapting our parameter and algorithm selection, and the Metadata Extractor component can retrieve compact metadata from the database efficiently.

- **Monitor Component**. This component monitors the application and the algorithm and parameter selection based on runtime performance data and stored metadata.

4. CONCLUSIONS AND FUTURE WORK

This work has introduced new infrastructure for performance analysis and adaptivity of parallel CFD applications. The full paper will discuss in more depth the use of this tool in parallel fluids simulations. Future work will include extending the infrastructure to include off-line analysis of performance data and new adaptive heuristics, as well as evaluating these capabilities in additional large-scale applications.

REFERENCES

1. 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.
2. 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. In *Proceedings of Supercomputing 1999*. IEEE Computer Society, 1999. Gordon Bell Prize Award Paper in Special Category.
3. S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. McInnes, Barry F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.2.1, Argonne National Laboratory, 2004. http://www.mcs.anl.gov/petsc .
4. 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.
5. CCA Forum homepage. http://www.cca-forum.org/ , 2005.
6. T. S. Coffey, C.T. Kelley, and D.E. Keyes. Pseudo-transient continuation and differential algebraic equations. *SIAM J. Sci. Comp*, 25:553–569, 2003.
7. P. Hovland, K. Keahey, L. C. McInnes, B. Norris, L. F. Diachin, and P. Raghavan. A quality of service approach for high-performance numerical components. In *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference*, Toulouse, France, 20 June 2003.
8. K. Huck, A. Malony, R. Bell, L. Li, and A. Morris. PerfDMF: Design and implementation of a parallel performance data management framework. 2005. To appear.
9. C. T. Kelley and D. E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM Journal on Numerical Analysis*, 35:508–523, 1998.
10. A. Malony, S. Shende, N. Trebon, J. Ray, R. Armstrong, C. Rasmussen, and M. Sottile. Performance technology for parallel and distributed component software. *Concurrency and Computation: Practice and Experience*, 17:117–141, Feb 2005.
11. 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.
12. J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, 1999.
13. B. Norris, J. Ray, R. Armstrong, L. McInnes, Bernholdt, W. Elwasif, A. Malony, and S. Shende. Computational quality of service for scientific components. *Proceedings of the International Symposium on Component-Based Software Engineering, (CBSE7), Edinburgh, Scotland*, 2004.
14. Department of Computer, University of Oregon Information Science, Los Alamos National Laboratory, and Germany Research Centre Julich, ZAM. *TAU User's Guide (Version 2.13)*, 2004.