

A Quality-of-Service Architecture for High-Performance Numerical Components[†]

P. Hovland, K. Keahey, L. C. McInnes, B. Norris

Math. & Computer Science Division, Argonne National Laboratory, Argonne, IL USA

L. Freitag

Software and Computing Systems, Sandia National Laboratories, Albuquerque, NM USA

P. Raghavan

Dept. of Computer Science & Engineering, Penn. State Univ., University Park, PA USA

Abstract: We propose a model for QoS-based composition of high-performance numerical components. We define an architecture that relies on five key capabilities and services including component characterization, component proxy services, component replacement, a decision module, and archival run information processing. We describe quality metrics that are important for high-performance numerical simulations, including computational cost, accuracy, and rates of convergence and failure. We discuss the use of the architecture and quality metrics in the context of a driven cavity flow simulation, which has been shown to benefit from adaptive solution techniques that could be derived from a QoS architecture.

Introduction

Recent years have seen much research and development in the area of component-based software engineering (CBSE). CBSE enables programmers to represent independent pieces of functionality as entities that can be composed, configured, and installed to create applications both rapidly and robustly. This approach is attractive because it shields the application development process from complexities such as platform and language heterogeneity and resource location. By defining clear interfaces, it promotes reusability and interoperability among different projects and thereby helps accelerate and generally improve the process of software development and sharing. Examples of component models include the CORBA Component Model (CCM) [1,2], COM [3], and more recently the Common Component Architecture (CCA) [4], which specifically targets high-performance scientific applications.

The relative maturity of component-based software infrastructures encourages users to look beyond syntactically connecting components to using higher-level information about component properties to compose applications. Such properties include accuracy of results, reliability with which results can be delivered, and the costs associated with providing them. Ideally, a software infrastructure, or framework, should enable the developer to construct applications using components that satisfy a given set of properties without knowing the intrinsic merits and performance tradeoffs of the underlying algorithms and implementations. To meet this requirement, the framework must allow component developers to both provide and use *Quality-of-Service (QoS) component descriptions*. If such descriptions can be clearly specified, then one can automate the selection of the most appropriate component to solve a particular subproblem. Furthermore, such automation can take the form of adaptive algorithms that can deliver

[†] 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.

improved performance by using QoS data collected during the course of the application's life-span and through multiple invocations of a given class of components.

Previous work has examined the role of higher-level semantic information in component assembly [5—12]. Raje et al. [7] describe a QoS framework for distributed, heterogeneous components and provide a catalog of QoS metrics [8]. Furmento et al. discuss performance models and their use in overall component application assembly at run time [11,12]. In this paper, we present an approach that provides a QoS infrastructure suitable for high-performance numerical components. Section 2 describes a motivating scenario. In Section 3, we propose an architecture and set of facilities for combining components while providing the desired QoS over an entire application. In Section 4, we define component-level QoS metrics relevant to high-performance numerical applications. In Section 5, we illustrate how our architecture addresses the needs of numerical applications.

2 Scenarios and Infrastructure Requirements

In this section we motivate our approach with an example from the solution of a nonlinear partial differential equation and describe the infrastructure requirements for solving this problem using a QoS approach.

2.1 A Motivating Scenario

For simulations in areas such as fusion, astrophysics, and computational fluid dynamics, application scientists typically compose multiple existing numerical components for different facets of the computation: for example, mesh management, discretization, derivative computation, and the solution of linear and nonlinear systems of equations. Each component can have multiple implementations that represent different approaches to solving the same problem and that differ in qualities such as robustness, time to solution, and solution accuracy. Moreover, the optimal choice of a specific algorithm or implementation for a given task may change during the life of the simulation as the data changes. Current practice typically involves manually selecting particular components and running experiments to determine which algorithms and implementations are most effective for a given scenario; changing algorithms typically involves stopping the simulation and replacing existing components with others.

As a particular example, consider a driven cavity flow simulation. This problem incorporates numerical features that are common in many large-scale scientific applications, and a detailed problem description, including the governing differential equations, boundary conditions, and discretization, is given in [13]. It has been found that some form of continuation is often required to solve these problems when certain parameters are even moderately nonlinear [14]. Commonly used pseudo-transient continuation methods introduce into the model a false time-stepping term and the need to solve a nonlinear system of equations at each time step using Newton's method. This time step transitions from small to large and thereby controls the conditioning of the linearized Newton systems [14]. The linear systems are initially well conditioned and relatively easy to solve when the pseudo time step is small, although they transition to being much more challenging as the pseudo time step grows and the nonlinear function approaches that of the true model. Previous work has shown that this problem benefits

from the use of adaptive numerical strategies in the Newton solves [15], and it is therefore a natural candidate for motivating and evaluating QoS infrastructure.

Newton's method solves a nonlinear system of equations of the form $f(u) = 0$, where $f: R^n \rightarrow R^n$, through the following two-phase iterative process:

1. (Approximately) solve $f'(u^{i-1}) u^i = -f(u^{i-1})$
2. Update $u^i = u^{i-1} + \cdot u^i$,

where \cdot is determined by a line search such that $0 < \cdot < 1$, $f(u)$ is the nonlinear function or residual, and $f'(u)$ is the corresponding Jacobian matrix of derivatives.

The iterative nature of Newton's method can be modeled using a directed graph as shown in Figure 1. Vertices in the graph denote components that are invoked, and the edges denote the direction of information flow. The graph is rooted at the application, with the first level containing a Newton component (denoted NS) that the application invokes, followed by a second level containing components that evaluate the nonlinear function (FE) and Jacobian (JE), solve the resulting linearized Newton system using a preconditioned Krylov method (PKS), and apply a line search method (LS).

The edges of the directed graph may have annotations in the form of both static and dynamic QoS requirements, capabilities, and performance. For example, the forward edge between the nonlinear and linear solver may require a nonsymmetric linear solver that returns a solution with a relative linear residual norm reduction of 10^{-4} . The corresponding backward edge may also be annotated to state the type of solver that was actually used, a detailed convergence history, and so forth. In addition, some graph layers may reflect transitions among particular component implementations during an ongoing simulation. For example, a preconditioned Krylov component implementation that matches the original QoS metrics may be used initially; subsequently the dynamic QoS metrics may be adjusted, and this implementation may be replaced by another preconditioned Krylov solver as part of an adaptive algorithmic scheme. We note that Furmento et al. [11] use directed graphs in a similar fashion.

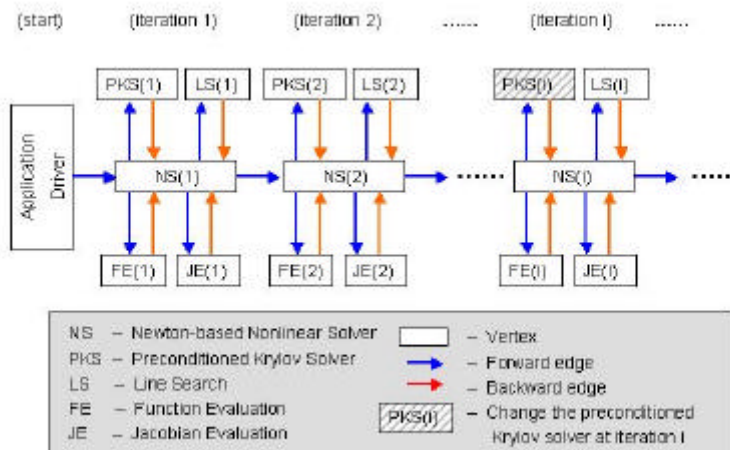


Figure 1: Directed graph for an application using a Newton-based nonlinear solver, showing the structure at the top level for several nonlinear iterations. Vertices at each iteration denote the components invoked. The nonlinear solver component exports both static and dynamic descriptions of QoS measures that it wants the simulation to satisfy.

2.2 Problem Definition and Infrastructure Requirements

In an ideal scenario, the application scientist would be relieved of managing much of the complexity discussed above by a component framework that incorporates QoS information into the component composition and management process. This could be achieved by defining descriptions representing both the meta-requirements for numerical components and the potential of particular numerical component implementations, based on historical data, algorithmic convergence theory, and so forth. To produce such descriptions would require transcribing intuitive knowledge, based on which many scientists currently formulate their simulations, into more qualitative and quantitative terms and then managing those descriptions. Where such knowledge is not easily captured in terms of absolute quality measures, it would be accompanied by a “measure of confidence” attached to a particular component implementation. In addition, a scientist would be able to specify the quality of simulation measures that could be adjusted at runtime.

As discussed in the preceding section, we are faced with a problem where we have *substitution sets* of numerical components that implement the same functionality (as expressed by the component interface) but are characterized by different semantic qualities, such as accuracy, robustness, and speed of convergence. We want to use the information of those semantic qualities in two ways. First, we want to use it to obtain the most optimal composition of components. Second, as the need arises during application execution, we want to seamlessly replace certain components by others chosen from their substitution set. Such adaptivity is an important theme guiding the development of our QoS framework. In broad terms, we use the word *adaptivity* to denote improved selection of methods (both algorithms and implementations), with the goal of decreasing execution time for the application while delivering the required quality of solution. This “learning” could occur within the application’s life-span (i.e., using information acquired during a single simulation), but also could represent experience combined across multiple past simulations. We introduce the term *dynamic component* to describe a component that can adapt its behavior in these ways. In addition, to the extent possible, we want the substitution to take place not only dynamically but also automatically, with suitable component selection and provisioning being performed by infrastructure services, rather than by the programmer.

3 Architecture

In the design of our QoS infrastructure we assume a component model as described by CORBA [1,2] and the CCA [4]. In the CCA model, a component exports two kinds of ports: *provides* ports, which describe what functionality a component implements, and *uses* ports, which describe what functionality a component requires. We further assume that the data descriptions used will allow a component to make its data available for interaction with other components.

As shown in Figure 2, our architecture relies on five key capabilities: (1) component characterizations, which describe component behavior and monitoring at runtime; (2) component proxies, which enable dynamic substitution; (3) a component replacement service, which locates and deploys new components at runtime; (4) a decision module, which decides which components need to be replaced; and (5) services that process

archival and run information. To provide these capabilities, we extend the CCA model by annotating the *uses* and *provides* ports with metadata. In addition, we designed several services that deliver the required functionality for adaptive, dynamic component substitution.

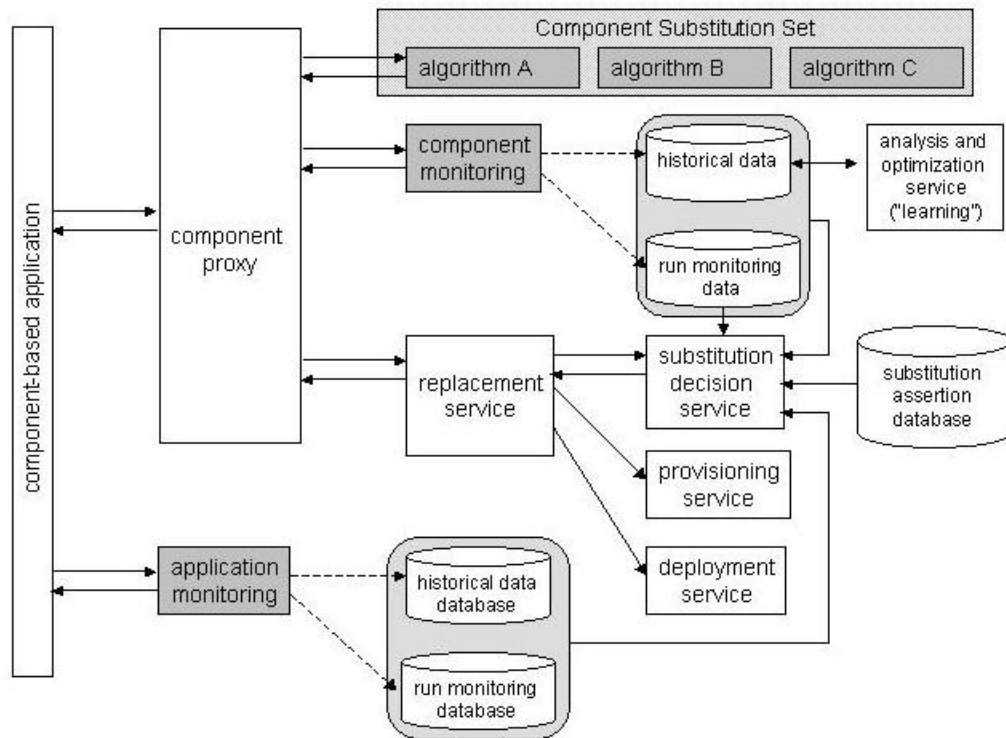


Figure 2: Interaction of an application with a dynamic component. User-provided components are marked in gray, while infrastructure services are in white.

Figure 2 shows how a component-based application interacts with one of its composite components in this architecture. Direct interaction with this component is replaced by interactions with a component proxy, which represents to the application one of the component implementations from the substitution set. The initial choice of a particular component implementation is typically based on static metadata specified as application requirements and the offering of a concrete algorithm.

Also based on information contained in the static metadata, the proxy chooses a monitoring component suitable for monitoring the run-time performance of that particular numerical component. This monitoring information may be logged at runtime to provide input on its performance as well as to enable further analysis later. In addition to monitoring the performance of the numerical component, the performance of the whole application is being monitored, also potentially logging its results.

Moreover, the component proxy keeps track of how the runtime *uses* metadata of an application relates to *provides* metadata of the actual component implementation. Whenever a particular component is found to be underperforming, a replacement action can be triggered either by an application scientist monitoring the execution or by a

decision service. Replacement decisions are made at the end of each iteration and are executed before the next iteration begins. In the meantime, the replacement service performs actions leading to provisioning and deploying the new implementation component. We now discuss the five key components in detail.

Metadata Component Characterizations. We extended the CCM/CCA model by *uses* and *provides* metadata associated with a given port. The *uses* metadata describes the QoS that a given *uses* port requires, while the *provides* metadata describes the QoS that a given port can provide. We also distinguish static metadata and dynamic metadata. Static metadata represents the knowledge that we have about the method implemented by a given port before it starts executing. It may be based on historical data (for example, previous component runs) as well as analytical data about expected performance scalability. Its main purpose is to provide initial selection guidance to the numerical scientist. The static metadata also includes information on how to monitor dynamic metadata. The dynamic metadata is updated after each iteration and indicates how well the selected method is performing in terms of qualities observable only at runtime. The updates are based on information obtained from a monitoring component invoked by the component proxy and logged in databases. Section 4 gives examples of such metadata for numerical applications. In addition to metadata associated with ports, we also define static component metadata describing implementation-related information (for example, deployment information).

Component Proxy. The component proxy implements the notion of a dynamic component by substituting at runtime components with better performance promise. To have a basis for this substitution, the proxy monitors component execution, compares the *uses* metadata of the calling application and the *provides* metadata of the component, potentially raises exceptions, and works with the application scientist or uses a replacement module to determine the right replacement. Substitution itself is performed by using the Dynamic Invocation Interface (DII) [1,2] mechanism. In addition to providing substitution for optimization, a component proxy provides a level of indirection allowing it to provide reliability in a manner transparent to the client [15,16].

Replacement Service. The dynamic replacement service arranges for the seamless between-iterations replacement of one module by another implementing the same functionality but exporting different metadata and potentially a different interface. It provisions and deploys the new component. The replacement service can be invoked by the proxy either on the programmer's behest or triggered by the substitution decision service. As part of provisioning decisions, this service also decides whether to keep the old component choice cached in memory and when to replace such components.

Substitution Decision Service. The substitution decision service automates decisions about component replacement by applying heuristics supplied by either an application scientist or a "learning module". These heuristics are very simple and have the form: "If the ratio of the residual norm reduction to the number of iterations is x , then substitute a more powerful method." As shown in Figure 2 the replacement decision will typically be based on the component's as well as overall application's run monitoring data (run

management data as well as historical data). It could, for example, happen that after a particular iteration a decision to replace multiple components will be made.

Data and Data Processing Modules. In our implementation we rely on three databases: (1) the historical data database that archives data on history of multiple runs, (2) the run-management database that contains information about a current run, and (3) the assertion database that contains substitution rules and heuristics. For reasons of efficiency, run-management logging is optional on any given run. The data in historical and run-management databases may be used by a learning system to recommend heuristics on when and how to substitute component implementations, and develop meta-data for similar or related components. This service closes the loop between generating execution data and steering the execution; its objective is to identify techniques for processing data about a run to improve policies and to formalize knowledge about how to steer a run to obtain the best possible performance.

4. Metrics for Numerical QoS

Our framework must provide a set of numerical QoS metrics for specifying component requirements and capabilities. These metrics can also be used to provide stopping criteria for iterative algorithms and to provide a measure of performance. For each metric, our framework must accommodate multiple sources of information (empirical, analytic) and varying levels of precision (qualitative, relative, quantitative). Furthermore, some metrics are static properties of a component, whereas other metrics, are dynamic properties that cannot be determined until run time. We propose computational cost, accuracy, failure rate, convergence rate, and preconditioner quality as metrics for numerical QoS. The first three can be mapped to the turn-around time (also called “end-to-end delay”), quality of result, and error rate metrics identified by Brahmamath et al. in their quality of service catalog [8]; we have chosen labels that more closely match the nomenclature of the computational science community.

Computational cost can be measured in a variety of ways, including wall clock time, CPU time, number of floating-point operations, or number of major/minor iterations of the numerical algorithm. On parallel platforms, computational cost is often a function of algorithmic scalability. In some cases, the expected performance degradation with number of processors can be accurately captured by models such as BSP or LogP [17,18]. Often, however, it is difficult to capture the impact of degradation in the numerical method; for example, additive Schwarz preconditioners typically become less effective as the local problem size decreases, and Newton-Krylov methods may require more iterations as the global problem size increases. For these and other reasons, accurate forecasts of computational costs are often difficult to obtain. Thus, this metric will often be used as a stopping criterion or measure of performance. Nonetheless, our framework must accommodate whatever models are available (see, e.g., [19,20]), so that an accurate prediction can be created using information, such as problem size or number of processors, that cannot be determined until runtime.

Accuracy can be measured in terms of residual norms or the asymptotic behavior of an approximation. Residuals provide a measure of how close an approximate solution is to the true solution, which has a residual of zero. For a linear system of the form $J(u)h = f$,

the residual is $f - J(u)h$. For a nonlinear system, the residual is simply the nonlinear function, $f(u)$. The residual may be reduced to a scalar value by computing its maximum value (infinity norm), the sum of its values (L_1 norm), or the square root of the sum of squares (L_2 norm). Furthermore, the residual norm of the final approximation may be measured in absolute terms or relative to the residual norm of the initial guess. Residual norms are often used as stopping criteria for iterative numerical algorithms. The asymptotic discretization error, expressed in big- O notation, provides a measure of the accuracy of a discretization by indicating how the error decreases as a function of the resolution. For example, doubling the resolution (halving the mesh “spacing”) would double the accuracy (halve the error) of a discretization with $O(h)$ accuracy but double the *number of digits* of accuracy (take the square root of the error) of a discretization with $O(h^2)$ accuracy. The truncation error in a finite difference approximation to derivatives can also be described in terms of its asymptotic behavior.

Failure rate is a measure of how frequently a component has failed to find a solution, either for previous iterations of the current problem or for a representative set of problems. Because failure rate is very data dependent, it can be difficult to predict. The asymptotic **convergence rate** measures the rate at which a given numerical algorithm approaches the solution; for example, near the solution Newton’s algorithm is quadratically convergent, meaning that the number of digits of accuracy doubles with each Newton iteration. Other algorithms are linearly or superlinearly convergent. Several metrics for **preconditioner quality** are possible. For incomplete factorizations, one possible metric is the amount of fill relative to a complete factorization, given a particular ordering. Other possible numerical QoS metrics include mesh quality, function smoothness, and stability.

5 Example of Architecture and Metric Use

We now show how the QoS architecture and metrics introduced in Sections 3 and 4 can be applied to manage the complexity of the Newton solves for the driven cavity simulation described in Section 2.

We first describe how a computational scientist could employ static metadata in a QoS historical performance database to aid in selecting a particular linear solver. The metrics of most interest in this case are *computational cost* (measured by wall clock time and number of nonlinear/linear iterations) and *accuracy* (measured by relative residual norm reduction). For example, the application driver specifies a nonlinear convergence criterion to the Newton solver (relative nonlinear residual reduction of $\|f\|/\|f_0\| < = 10^{-8}$), and the Newton solver specifies a fixed linear convergence criterion to the linear solver component proxy (relative linear residual reduction of $\|r\|/\|r_0\| < = 10^{-4}$ in a maximum of k_{\max} Krylov iterations). Then, the static metadata from simulations represented in the historical database captures the performance of a variety of base methods that used these metrics and indicates that a particular linear solver performs best for a set of relevant test problems. The computational scientist then conjectures that this method would also be a good choice for the current simulation, which uses, for example, the same nonlinearity parameters as the initial simulations but a more refined mesh.

We next demonstrate how a computational scientist could also employ dynamic metadata within the run monitoring database to adapt solution strategies during a given simulation.

Based on the analysis of static metadata discussed above, the scientist initiates a run on a driven cavity simulation using a particular method as the linear solver. During this simulation, he increases the model's nonlinearity parameters significantly and thus also incorporates pseudo-transient continuation to help handle this related but more difficult problem. As the simulation progresses, the run monitoring database incorporates data both from an application-based monitor of the nonlinear residual norm and pseudo time step and from a component-based monitor of the linear solver's performance. As the time step grows, the linear systems become less well conditioned and more difficult to solve, as evidenced by increases in the actual achieved metrics of wall clock time and Krylov iterations needed to reduce the linear residual norm by the specified amount. The analysis and optimization service deduces that the current linear solver is not sufficiently powerful anymore and thus recommends its dynamic replacement in the midst of the nonlinear simulation by another method, which has been shown in the historical database to converge more rapidly.

6 Conclusions

In this paper we described an architecture for a QoS-based system for composition of numerical applications. We first described the application that motivate this design and their features, and from them derived the requirements underlying our design. We then presented the design of an infrastructure fulfilling these requirements: allowing for QoS-based composition and monitoring of application components, and incorporating mechanisms necessary for their run-time replacement to improve QoS of any particular run. We also described mechanisms that could perform such substitution automatically, based on a combination of component and application monitoring data as well as mechanisms that could guide substitutions based on records of past runs. To make the operation of this infrastructure possible, we defined QoS measures specifically targeting the performance of numerical applications. Finally, we showed how our design could be used in the context of a real application.

We believe that this architecture could simplify and dramatically improve the efficiency of multi-component numerical applications. Where previously ad hoc methods for composition were used, primarily based on knowledge and experience of individual scientists, we now propose a methodical and automated approach to such compositions. We further believe that our solution is generic enough to be applied to other application domains with similar benefits.

References

- [1] O. M. Group, The Common Object Request Broker: Architecture and Specification, OMG Document, 1998. See <http://www.omg.org/corba/>.
- [2] The CORBA Component Model, <http://ditec.um.es/~dsevilla/ccm/>.
- [3] Microsoft COM Web page. See <http://www.microsoft.com/com/about.asp>.
- [4] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. C. McInnes, S. Parker, and B. Smolinski, Toward a common component architecture for high-performance scientific computing, in Proceedings of HPDC 1999, pp. 115—124.

- [5] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, Making components contract aware, *IEEE Computer*, (1999), pp. 38—45.
- [6] K. Keahey, P. Beckman, and J. Ahrens, Ligate: A component architecture for high-performance applications, *International Journal of High-Performance Computing Applications*, 14 (2000).
- [7] R. Raje, B. Bryant, A. Olson, M. Auguston, and C. Burt, A quality-of-service-based framework for creating distributed heterogeneous software components, *Concurrency Comput: Pract. Exper.*, 14 (2002), pp. 1009—1034.
- [8] G. J. Brahmamath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, C.C. Burt. A quality of service catalog for software components. *Proceedings of the Southeastern Software Engineering Conference*. <http://www.ndiatvc.org/SESEC2002/>, 2002.
- [9] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, Specifying and measuring quality of service in distributed object systems, in *Proceedings of ISORC '98*.
- [10] X. Gu and K. Nahrstedt, A scalable qos-aware service aggregation model for peer-to-peer computing grids, in *Proceedings of HPDC 2002*, 2002.
- [12] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington, Optimisation of component-based applications within a grid environment, in *Proceedings of SC2001*, 2001.
- [12] ICENI: Optimisation of component-based applications within a grid environment, *Journal of Parallel Computing*, (2002).
- [13] T. S. Coffey, C. T. Kelley, and D. E. Keyes, Pseudo-Transient Continuation and Differential-Algebraic Equations, submitted to the open literature, 2002.
- [14] C. T. Kelley and D. E. Keyes, Convergence analysis of pseudo-transient continuation. *SIAM Journal on Numerical Analysis* 1998; 35:508—52.
- [15] L. McInnes, B. Norris, S. Bhowmick, and P. Raghavan, Adaptive Sparse Linear Solvers for Implicit CFD Using Newton-Krylov Algorithms, To appear in the *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*, Massachusetts Institute of Technology, Boston, USA, June 17-20, 2003.
- [16] The Taming of the Grid: Virtual Application Services, K. Keahey and K. Motawi, submitted to the 12th IEEE International Symposium on High Performance Distributed Computing.
- [17] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of 4th PPOPP*, San Deigo, CA, May 1993.
- [18] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103—111, 1990.
- [19] A. Snively, L. Carrington and N. Wolter, “A Framework for Performance Modeling and Prediction,” *Proceedings of SC2002*, Baltimore, MD, Nov. 2002.
- [20] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala and B. Lee, 2002, Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply, *Proceedings of SC2002*, Baltimore, MD, Nov. 2002.