A Microkernel Design for Component-based Parallel Numerical Software Systems *

Satish Balay, Bill Gropp, Lois Curfman McInnes, and Barry Smith †

Abstract

What is the minimal software infrastructure and what type of conventions are needed to simplify development of sophisticated parallel numerical application codes using a variety of software components that are not necessarily available as source code? We propose an opaque object-based model where the objects are dynamically loadable from the file system or network. The microkernel required to manage such a system needs to include, at most

- a few basic services, namely,
  - a mechanism for loading objects at run time via dynamic link libraries, and
  - consistent schemes for error handling and memory management; and
- selected methods that all objects share, to deal with
  - object life (destruction, reference counting, relationships), and
  - object observation (viewing, profiling, tracing).

We are experimenting with these ideas in the context of extensible numerical software within the ALICE (Advanced Large-scale Integrated Computational Environment) project, where we are building the microkernel to manage the interoperability among various tools for large-scale scientific simulations. This paper presents some preliminary observations and conclusions from our work with microkernel design.

1 Introduction

The complexity of large-scale scientific simulations, often collaborative efforts among scientists and engineers, necessitates the combined use of multiple software packages developed by different groups. For example, several projects that motivate our current work are the modeling of microstructural evolution in sintering [31, 32, 33], astrophysical thermonuclear simulations [27], and multi-model aerodynamic computations [23, 16, 22]. These investigations involve a range of computational areas such as discretization, partitioning, load balancing, adaptive mesh manipulations, scalable algebraic solvers, optimization, parallel input/output, performance diagnostics, computational steering, and visualization; moreover, the state-of-the-art within each of these areas is constantly evolving, necessitating frequent software updates within the lifetime of a given application.

*The authors were supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†balay@mcs.anl.gov, gropp@mcs.anl.gov, curfman@mcs.anl.gov, bsmith@mcs.anl.gov, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4844.

A pressing question now facing the computational science community is how to effectively leverage the varying expertise of all team participants in such multidisciplinary projects. The limitations of the current generation of software tools and infrastructure, even those that employ modern software design techniques, cause us to fall short of where we must move in order to have a hope of exploiting forthcoming teraflop-scale computational resources for meaningful scientific gains. The high-performance computational software community is thus faced with the charge of developing more effective ways for multiple groups with different areas of expertise to encapsulate their knowledge within extensible, reusable, interoperable software tools, and thereby to raise the levels of abstraction used by application scientists.

The mainstream computing community has developed interoperability mechanisms (e.g., distributed object technology such as the COM family and CORBA, and portable languages such as Java) to address similar levels of complexity within their applications. Our approach is to leverage parts of this work when appropriate, recognizing that the features of large-scale scientific computation present different challenges and thus demand different solutions. One challenge is the need for efficient and scalable performance on ever evolving distributed-memory architectures such as symmetric multi-processor and workstation clusters. Also, the culture of research computing differs from that of the business world; scientists need to be able to explore their ideas without requiring legions of programmers to translate from scientific abstractions to actual code, and without becoming overwhelmed with a myriad of details (e.g., security) that are not of primary interest. These issues are further discussed in Section 2.2.

Many open research questions must be explored to determine good interoperable software management techniques for high-performance numerical simulations. These questions include: What are appropriate performance-sensitive abstractions and operations? What are appropriate data exchange formats among particular components? How can we develop bridges between data structures for performance optimization?

To enable exploration of these issues, we are building infrastructure within a microkernel to manage tool coordination. Our approach focuses on a base object class coupled with dynamic loading of software components. The remainder of this paper motivates these choices and explains our design strategy. Section 2 defines in more detail the scope of the issues under consideration, briefly surveys existing solution techniques, and discusses the limits of our technical approach. Section 3 defines various terms used in the remainder of the discussion and introduces our solution strategy. Section 4 presents an overview of the microkernel design, including the core infrastructure it provides and the base object model. Preliminary observations and directions of future work are discussed in Section 5.

2 Interoperability Issues

As we investigate techniques for dynamic component-based interactions in scientific computing, we must bear in mind our target customers [14]. The microkernel work presented here forms part of a flexible architecture within the Advanced Large-scale Integrated Computational Environment (ALICE) [1], under development at Argonne National Laboratory. We aim to provide low-overhead mechanisms to enable different groups to contribute and maintain their own libraries (as painlessly as possible) in a distributed fashion. The goal of ALICE is to leverage the strengths of different high-performance toolkits, *not* to develop a single massive library into which everyone contributes code.

2.1 A Motivating Simulation

As a motivating example, we consider simulations of microstructural evolution in sintering via the method of lines, under investigation by W. Zhang and J. Schneibel [31, 32, 33]. At the heart of these simulations is the solution of a set of coupled ordinary differential equations (ODEs), which requires the solution of nonlinear systems, which in turn can be addressed by preconditioned Newton-Krylov methods. For large-scale parallel simulations, we have incorporated the complementary capabilities of (1) mathematical sintering models of Zhang and Schneibel; (2) ODE solvers within the PVODE [17] software of Hindmarsh et al. of Lawrence Livermore National Laboratory, and (3) preconditioning techniques, matrix coloring tools for finite differencing Jacobian approximation, and parallel problem decomposition infrastructure within the PETSc software [4, 5]. PVODE and PETSc complement each other well because PVODE provides higher-order, adaptive ODE schemes and robust nonlinear solvers tailored for ODE solution, but does not focus on parallel preconditioners and coloring tools; likewise, PETSc does not provide higher-order ODE integrators. Due to data-structure-neutral design [28] with well-defined application programming interfaces (APIs), this interoperability required no changes to the source code of either PVODE or PETSc.

Although the bilateral PVODE/PETSc interfacing has been a good step forward in bridging the gap between what scientific simulations need and what numerical libraries provide, it raises questions about broader interoperability issues. What we really need is the *dynamic* combined use of various packages, including multiple tools that provide identical functionality as well as those with different capabilities. For example, the sintering simulations could potentially benefit from algorithmic alternatives for preconditioning and ODE solution provided by different software packages as well as complementary capabilities such as derivative-enhanced sensitivity analysis [18], visualization, and steering.

One-to-one interfacing is excessively burdensome, and the current generation of infrastructure is inadequate, even for software packages that individually have been built using object-oriented design. To enable more dynamic component-based interactions, we must consider issues such as mechanisms for specifying software APIs (input/output parameters) and behavior (services that a toolkit provides as well as those it requires), approaches for establishing dynamic connections among tools, "standard" interfaces for particular functionalities (see, e.g., [9] for linear algebra interface work), and language interoperability (see, e.g., [10],[21]). Such issues are under consideration by a variety of researchers, including [2, 20, 26, 25, 30, 3].

2.2 Comparison of Approaches

There are two main alternatives to the component model that we propose in this paper. One is a more library-oriented approach, with a long and successful history in numerical computing. The other is a commercial component or distributed object solution, which has been successful in mainstream computing. To understand why we propose dynamically loaded components, it is important to remember the problem that we are trying to solve: managing the growing complexity of numerical libraries, particularly collections of libraries, each representing the unique expertise of a research or product group, while maintaining the performance that scientific/engineering application developers expect and require.

One common approach to developing software within the numerical computing community is the unsatisfactory "flattening" of all code within a single application; this approach fails because of the inherent lack of scalability in terms of development group size. In addi-

tion, because the necessary skills cannot reside in any one group, additional problems can arise in managing a distributed development effort.

More successful has been the use of full-scale class libraries, which require all users and software developers to write code to these class libraries (e.g., the standard template libraries in C++). However, this approach still does not work well for scientific computing because of the wide variety of needs within meaningful applications.

Templates [6] (algorithmic, not C++ templates) have been proposed as a method for providing advanced numerical methods to applications writers without dictating a particular choice of data structure or even interface. This approach suffers from the fact that there is no executable code; the user is still faced with all of the issues of implementation and testing. With today's high-performance parallel machines, these issues are just as difficult as the mathematical and algorithmic development.

These considerations have led us to explore a model for numerical software components that draws on the success of the component model in commercial information processing. However, there are a number of differences between the needs of scientific and commercial applications. Primarily, many of the commercial component solutions (e.g., CORBA and the COM family) are targeted at the very difficult problem of managing distributed objects on a wide-area network. While these approaches have proven revolutionary in domains where interoperability is paramount (e.g., client-server interactions), they have not been designed to address the complexity of interactions and performance issues for large-scale, distributed-memory numerical applications. We cannot simply use COM or CORBA to connect these components because we must address performance problems, e.g., by allowing some objects to access the memory of others. In addition, some component systems are based on an event-driven model that, while natural and effective in commercial applications, is not appropriate for many scientific applications.

Perhaps most fundamental, our microkernel work focuses on the design of the base objects themselves, along with a minimal infrastructure that allows experimentation with different lightweight approaches for accessing components in an environment where security is not an issue (i.e., security is handled separately, as it is within a single high-performance computer). As other component infrastructures begin to address the issue of high performance, we can move our object design to those platforms. For example, an implementation in Java using something like InfoBus [19] to communicate data and Java's mechanisms for managing components (or remote method invocation for more distributed applications) would still need to provide the component operations that we describe here.

3 Scope of Solution

We specify two design requirements for this work: basic functionality should be as efficient as standard procedural code (Fortran/C/C++), and no "run-time" system (e.g., threads) should be needed. Bearing these in mind, we propose a model for object interaction that consists of two orthogonal submodels: (1) a synchronous function call-based model (e.g., in C++ calling methods on objects) for numerical algorithm implementation and (2) an asynchronous remote function call-based model for accessing remote (or local) objects for such tasks as monitoring, steering, and visualization. Model 1 meets our basic performance requirement, while model 2 provides additional functionality, especially for remote operations.

We begin by stating some definitions for use in the remainder of this discussion.

- *object* - encapsulated data and methods that operate on that data.

- *standardized object* - object that supports a set of predefined methods.

- *object toolkit* - software package that provides code (source or binary) for one or more (generally related) types of objects, including constructors to generate the object instantiations. For example, PETSc [4] provides a variety of C objects, while ISIS++ [8] provides a variety of C++ objects.

- *standardized object toolkit* - object toolkit that supports some standardized objects. X-windows and MPI [24] are examples of standards for object toolkits, while MPICH [15] and LAM [7] are examples of standardized object toolkits that implement the MPI standard.

- *component* - an encapsulated software object that provides a certain set of functionalities or services and can be used in conjunction with other components to build applications. A component consists of an API and one or more component implementations, and conforms to a prescribed behavior within the context of a given framework.

  This is our working definition of the term *component*; however, since this term is so overloaded with partial meanings (see, e.g., [29]), and since the remainder of this discussion focuses on functionality within encapsulated software *objects* that in part comprise components, the remainder of this discussion will not employ the term *component*.

Traditional large-scale numerical simulations are almost always implemented in a procedural style, where subroutines are called in a well-defined order to implement a deterministic numerical algorithm. When object-oriented techniques are used in numerical computing, the standard approach (used in, for example, PETSc and ISIS++) is to encapsulate the data structures in objects, while still allowing the application programmer to write procedural code; that is, he or she calls a sequence of functions that operate on the objects to perform the desired calculations. This programming style is used for the numerical computations within the sintering model discussed in Section 2.1.

On the other hand, the programming of graphical user interfaces and transaction processing systems has moved away from the expression of a computation as a linear list of functions that are called. Rather, (possibly distributed) objects are viewed as making requests of and serving requests of other objects.

We propose to recognize the differences between these two models and adopt different mechanisms to handle the two types of interactions:

1. synchronous, local-address-space function calls, intended for implementing numerical algorithms; and

2. asynchronous, possibly remote, transactions such as accessing an array of variables for visualization. (Here we use the term asynchronous to mean that the object may serve a request while simultaneously performing a numerical calculation.)

Model 1 can be implemented with no run-time environment. Handling the requirements of model 2 requires a relatively large run-time infrastructure to support marshaling of arguments, communicating among remote processes, locking and unlocking data structures to allow access from multiple threads, and so forth. The remainder of this paper explains model 1 in more depth and presents ideas about the design of a microkernel to support it.

Infrastructure for model 2 and the integration of the two submodels are currently under investigation but are beyond the scope of this paper.

In the procedural object model, the user writes code to create objects and then calls methods on those objects in (more or less) a linear fashion. It is assumed that the caller and the callee share the same memory address space. Multiple threads may be used both within an object (transparently as it handles a member function call) and explicitly outside the objects by the user; however, the objects can handle only one member function request at a time. For example, a matrix object would not be able to insert new matrix entries while it is performing a matrix-vector product.

Data in the object can be accessed only through arguments to member functions, either by explicitly passing data into or out of the member function or by passing memory references by address. The latter is frowned upon if it exposes any of the underlying object data structures to the caller.

Member functions may be multistaged; the object developer selects which operations require a split-phase implementation and simply provides the corresponding multiple member function interface. For example, in PETSc we perform vector scatters with the two-stage routines `VecScatterBegin(VecScatter,Vec,Vec)`, which begins the parallel communication involved in performing the scatter, and `VecScatterEnd(VecScatter,Vec,Vec)`, which completes the operation [5]. This approach allows efficient overlapping of communication and computation without requiring threads or application-specific coding.

Standardized objects can be developed in C++ by defining standardized abstract base classes and inheriting from them to create a standardized object toolkit. Likewise, in C standardized objects can be implemented by defining a standard format for function tables for each type of object. Cross-language portability can be obtained by creating object wrappers that convert method calls in one language (e.g., a class member function call in C++) to another (e.g., a function table lookup in C).

4 Microkernel Design

This section presents a microkernel design for the synchronous function call-based model discussed in Section 3. We propose an opaque object-based model where the objects are dynamically loadable from the file system or network. That is, the executable code that creates an object and fills its member functions can be loaded into memory by the application as it runs. To limit complexity and overhead, the software for managing such a system should be lightweight; we thus propose a microkernel that includes, at most

- a few basic services, namely,
  - a mechanism for loading new objects at run time via dynamic link libraries, and
  - consistent schemes for error handling and memory management; and
- selected methods that all objects share, to deal with
  - object life (destruction, reference counting, relationships), and
  - object observation (viewing, profiling, tracing).

We have chosen these areas of functionality based on our experiences in PDE software development. While some of these mechanisms, namely the variants of object observation, might seem of secondary importance on initial consideration, they have proven invaluable in practice for debugging, analysis, and performance optimization.

4.1 Basic Microkernel Services

For different software toolkits to interoperate smoothly, they must employ consistent schemes for error handling and memory management. These basic services, as well as dynamic linking capabilities, should be supported within the microkernel as objects so that various implementations can be introduced as needed. Details of these APIs and our base implementations are beyond the scope of this paper, but will be presented elsewhere.

Support for dynamic linking is one of the key features needed to achieve true separation of implementations from mathematical abstractions. By enforcing programming discipline so that we can verify appropriate *interface* definitions for particular functionalities, dynamic linking enables us to avoid the web of interdependent complexity that arises in traditional class hierarchies. By separating software binding time from an application's compilation, we no longer care when particular toolkit implementations are written, as they can be imported at any time. Hence, dynamic linking capabilities provide an effective means to test proper modularity. This feature enables new software injection without maintenance intervention, and thereby makes it possible for applications to exploit new advances in algorithms and architecture-specific performance optimizations, since these may be introduced at run time without requiring code recompilation. For example, within the sintering simulation discussed in Section 2.1, new adaptive ODE techniques and cache-sensitive matrix data structures provided by different toolkits compliant with the basic object model specified above could be seamlessly introduced. In addition, dynamic linking ensures name space separation, which is not resolved in C and Fortran, although it is somewhat resolved in C++ and Java.

4.2 Common Object Functionality

How can a software component efficiently interact with another that it did not know about at compilation time? The answer is to choose a universal object memory layout that defines the representation in memory of object methods. Within C they are represented as groups of structures, while within C++ they are represented as abstract base classes. Two issues must be considered here: methods common to all objects, and the details of memory layout.

Common Object Header. The following presents the memory layout within a common header for all objects that are implemented using C; C++ classes could also be used.

```
struct _Object {
   int      cookie;
   BaseOps  *bops;
   ClassOps *cops;
}
```

Here all objects are pointers to a particular structure, which contains a cookie to indicate object type; a pointer to a basic set of operations, `BaseOps`; and a pointer to object-specific operations, `ClassOps`.

Common Object Functions. We use a common function table to specify data manipulations within the basic object, `BaseOps`. These core methods deal with two categories of activity: object life (destruction, reference counting, relationships) and object observation (viewing, profiling, tracing). The following figure presents an implementation using C; other language implementations could employ corresponding language features, for example, virtual functions in C++.

```
typedef struct {
  int (*getcomm)(Object,MPI_Comm*);                - get MPI communicator from object
  int (*view)(Object,Viewer);                      - visualize, serialize
  int (*reference)(Object);                        - increase reference count by one
  int (*destroy)(Object);                          - decrease reference count by one
  int (*attach)(Object,char*,Object);              - attach another object (interface)
  int (*query)(Object,char*,Object*);              - get an attached object (interface)
  int (*attachfunction)(Object,char*,char*,void*); - attach a method to an object
  int (*queryfunction)(Object,char*, void**);      - get a method from an object
  int (*querylanguage(Object,Language,void**);     - get a representation of an object
                                                     (interface) in another language
} BaseOps;
```

We now provide slightly more information on some of the basic methods.

- `attach(Object,char *name,Object)` - attaches the second object to the first object and increases the reference count of the second object. This mechanism allows one toolkit to carry references to another toolkit's objects; these references are transparent to the carrying toolkit (and invisible to the end user). For example, a vector object may carry a reference to a related grid object, and a Jacobian object may carry a reference to a discretization object that computes Jacobian entries. This capability has enabled interfacing between the unstructured meshing tools of SUMAA3d [13] and the algebraic solvers of PETSc, without introducing a single change to either software package [12].

- `query(Object,char *name,Object *)` - retrieves an object that has been attached to the first object via `attach()`.

- `attachfunction(Object,char *name,char *fname)` - attaches a function pointer to an object. The string `fname` is the character string name of the function; it may include the path name or URL of the dynamic library where the function is located. For example, `fname` may be `libpetscsles:PCCreate_LU` or `http://www.mcs.anl.gov/petsc/libpetscsles:PCCreate_LU`. The argument `name` is a "short" name of the function to be used with the `queryfunction()` call. This provides a mechanism for specifying object methods at run time. For example, a user can, at run time, select a particular preconditioner, say, a drop tolerance ILU technique; then all methods controlling the factorization are dynamically added at that time.

- `queryfunction(Object,char *name,void **func)` - retrieves a function pointer that has been associated with the object via `attachfunction()`. If dynamic libraries are used, the function is loaded into memory at this time (if it has not been previously loaded), not when the `attachfunction()` routine was called. This mechanism enables access to an object's dynamic methods.

- `querylanguage(Object obj,Language lang,void **interface)` - requests an interface to an object's data from a language other than the one in which it is implemented, (e.g., a C++ class representation of a C object for use in the C++ portion of a multilanguage application).

5 Preliminary Observations and Conclusions

To support scientific computing components, we have presented ideas for minimal infrastructure in the form of basic microkernel services and a common object memory layout. Key features include support for runtime binding of different toolkits, dynamic addition of methods, and object attachment through a dynamically loaded library approach. A base implementation of the microkernel, recently introduced into PETSc, has greatly increased software extensibility by facilitating the use of new external components.

We are currently experimenting with the microkernel as part of a flexible multilevel ALICE architecture that supports interoperability among a range of computational software. The microkernel coordinates interaction among tools with complementary capabilities (e.g., ODE solvers within PVODE, unstructured mesh tools within SUMAA3d, and algebraic solvers within PETSc) and provides a foundation for investigating broader issues in high-performance component design. For example, a great strength of the dynamically loaded component approach presented here is that it promotes well-designed interfaces that are completely separate from implementations, so that various external toolkits can be introduced and the community can begin to work toward defining sets of canonical interfaces. Our experiences thus far are merely the first steps toward much larger computing and component sharing.

Acknowledgments

We thank Wen Zhang for working with us on the parallelization of her sintering simulations. For PVODE/PETSc interoperability we acknowledge support from the DOE2000 Initiative [11] and the work of Liyang Xu. In addition, we thank Paul Hovland, Boyanna Norris, and our colleagues in the DOE Common Component Architecture (CCA) Working Group [3] for stimulating discussions of issues in component design for high-performance scientific computing.

References

[1] *ALICE Web page*. http://www.mcs.anl.gov/alice, Mathematics and Computer Science Division, Argonne National Laboratory.

[2] R. Armstrong and A. Cheung, *POET (Parallel Object-oriented Environment and Toolkit) and frameworks for scientific distributed computing*, in Proceedings of HICSS97, 1997.

[3] R. Armstrong et al., *Common Component Architecture Working Group Web page*. http://z.ca.sandia.gov/~cca-forum.

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

[5] ——, *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.

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

[7] G. Burns, R. Daoud, and J. Vaigl, *LAM: An open cluster environment for MPI*, in Proceedings of Supercomputing Symposium '94, J. W. Ross, ed., University of Toronto, 1994, pp. 379–386.

[8] R. L. Clay, K. Mish, and A. B. Williams, *ISIS++ Web page*. http://ca.sandia.gov/isis.

[9] R. Clay et al., *Equation Solver Interface Working Group Web page*. http://z.ca.sandia.gov/esi.

[10] A. Cleary, S. Kohn, S. Smith, and B. Smolinski, *Language interoperability mechanisms for high-performance scientific applications*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Oct. 21–23 1998. (to appear).

[11] *DOE2000 Initiative Web page*. http://www.mcs.anl.gov/DOE2000.

[12] L. Frietag, M. Jones, and P. Plassmann, *Component integration for unstructured mesh algorithms and software*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Oct. 21–23 1998. (to appear).

[13] ——, *The scalability of mesh improvement algorithms*, in Algorithms for Parallel Processing, M. T. Heath, A. Ranade, and R. S. Schreiber, eds., vol. 105 of The IMA Volumes in Mathematics and Its Applications, Springer-Verlag, 1998, pp. 185–212.

[14] W. Gropp, *Exploiting existing software in libraries: Successes, failures, and reasons why*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Oct. 21–23 1998. (to appear).

[15] W. Gropp and E. Lusk, *MPICH Web page*. http://www.mcs.anl.gov/mpi/mpich.

[16] W. D. Gropp, D. E. Keyes, L. C. McInnes, and M. D. Tidriri, *Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD*, Tech. Rep. 98-24, ICASE, Aug. 1998.

[17] A. Hindmarsh et al., *PVODE Web page*. http://www.llnl.gov/CASC/PVODE.

[18] P. Hovland, B. Norris, and B. Smith, *Developing a derivative-enhanced object-oriented toolkit for scientific computations*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Oct. 21–23 1998. (to appear).

[19] *InfoBus Web page*. http://www.java.sun.com/beans/infobus.

[20] *Infospheres Web page*. http://www.infospheres.caltech.edu/.

[21] B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi, *Inter-Language Unification reference manual*. ftp://ftp.parc.xerox.com/ilu/ilu.html, Xerox Corporation.

[22] D. Kaushik, D. Keyes, and B. Smith, *On the interaction of architecture and algorithm in the domain based parallelism of an unstructured grid incompressible flow code*, in Domain Decomposition Methods 10, AMS, 1998, pp. 287–295.

[23] D. E. Keyes et al., *Multi-Model Multi-Domain Computational Methods in Aerodynamics and Acoustics Web page*. http://www.cs.odu.edu/~keyes/nsf.

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

[25] S. G. Parker, D. W. Weinstein, and C. R. Johnson, *The SCIRun computational steering system*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997.

[26] *PSEware Web page*. http://www.extreme.indiana.edu/pseware/.

[27] R. Rosner et al., *University of Chicago Center on Astrophysical Thermonuclear Flashes Web page*. http://www.asci.uchicago.edu.

[28] B. F. Smith and W. D. Gropp, *The design of data-structure-neutral libraries for the iterative solution of sparse linear systems*, Scientific Programming, 5 (1996), pp. 329–336.

[29] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ACM Press, New York, 1997.

[30] S. Weerawarana, E. N. Houstis, J. R. Rice, A. C. Catlin, C. Crabill, C. C. Chui, and S. Markus, *PDELab: An object-oriented framework for building problem solving environments for PDE based applications*, Tech. Rep. CSD-TR-94-021, Department of Computer Sciences, Purdue University, 1994.

[31] W. Zhang, *Using MOL to solve a high order nonlinear PDE with a moving boundary in the simulation of a sintering process*, Appl. Numer. Math., 20 (1996), pp. 235–244.

[32] W. Zhang and I. Gladwell, *The sintering of two particles by surface and grain boundary diffusion - a three-dimensional numerical study*, Comp. Mater. Sci. (in press).

[33] W. Zhang and J. H. Schneibel, *Microstructural Evolution in Sintering - an Experimental, Mathematical and Numerical Study*. http://www.oakland.edu/~w2zhang/sintering.htm.