LA-UR-00-1519

Approved for public release;
distribution is unlimited.

Title: Ligature: Component Architecture for High-Performance Applications

Author(s): K. Keahey, P. Beckman and J. Ahrens

Intended for: International Journal of High-Performance Computing Applications—Special Issue on Performance Modeling, Vol 14. No. 4, Winter 2000

Los Alamos NATIONAL LABORATORY EST. 1943

Ligature: Component Architecture for High-Performance Applications

Katarzyna Keahey
Peter Beckman
James Ahrens
{*kate,beckman,ahrens*}*@acl.lanl.gov*
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545

Abstract

The increasing feasibility of developing applications spanning nationwide supercomputing resources makes possible the creation of simulations composed of multiple interdisciplinary components and capable of modeling natural and social phenomena of national importance with unprecedented speed and accuracy. However, the potential offered by hardware technology often fails to be fully realized due to the lack of software environments supporting such efforts. Furthermore, the complexity of combining within one application components with different performance characteristics often prevents such applications from achieving required performance levels. The Ligature project at LANL addresses the issue of designing a software infrastructure enabling fast and efficient development of multi-component applications, and that of providing performance guidance to the programmer using this infrastructure. Ligature allows the programmer to define component interfaces specifying how heterogeneous, distributed components can interact within a larger system and provides a reusable infrastructure capable of connecting these components. These interfaces, as well as information about component performance are accessible through a database. Within this framework we are trying to understand how information about the performance of individual components, and information about performance of the framework can be combined to develop a performance-aware multi-component application.

1. Introduction

Present day computing infrastructure [4] enables the development of scientific simulations on an unprecedented scale. Implementing those simulations requires interdisciplinary collaborations involving aspects of scientific computing, visualization and data management. Although the hardware technology required by such applications is becoming increasingly more common, the development of environments which would enable programmers to efficiently use this technology lags behind. Due to the lack of clearly defined interfaces, and execution paradigms between project components, collaborations often become a bottleneck of project development and rarely result in efficient solutions.

It is clear that as our potential for solving complex problems increases with the developing hardware technology, we will need a breakthrough in software design to match these new capabilities. A similar breakthrough introduced high-level languages as an alternative to machine instruction programming and revolutionized software development by allowing programmers to develop software much faster, make it more reliable, and by clearly defining areas capable of performance improvement (code generation for example). Today we need to think at an even higher level of abstraction and develop novel programming techniques, which would deal with the heterogeneity and distributed computing aspects of modern programming. If we want the development of scientific simulations to win the race with constantly advancing science, and growing hardware capabilities, this development has to be automated, accelerated and provide a focus for performance improvement.

This paper will describe Ligature, a component architecture defining a set of abstractions and techniques for high-performance scientific component programming. Ligature in music is a symbol that combines within itself several notes to be sung together; similarly we are trying to develop a system capable of combining several independently developed high-performance components so that they efficiently work together in one application. We believe that by defining programming abstractions at the level of heterogeneous program components, and providing a reusable infrastructure which can combine those components, we will create an environment enabling quick and efficient creation of multi-component interdisciplinary applications. In addition to enabling the programmer to think at a higher level and therefore develop applications faster, defining abstractions at this level will also enable him or her to reuse components across applications and through reusability also contribute to faster development. Defining clear component specifications will allow us also to reuse the combining infrastructure and will thereby make its optimization more cost-effective. Finally, through defining those abstractions Ligature will enable run-time inclusion of components.

The latter feature is particularly important in modeling real-time phenomena such as crisis forecasting and management in the DELPHI project at Los Alamos. A statically developed simulation will not suffice here as new factors may arise at run-time and will need to be incorporated into the simulation as the situation unfolds. Furthermore, new managing algorithms can be developed concurrently with crisis modeling and incorporated into the simulation on the fly. Another example of the application of component-based programming is the need for refinement in large, complex simulations. For example having implemented a traffic modeling system, a complex task in itself, we may want to use it in order to model emissions. This may also include modeling air movement. The complexity of these components, as well as their reusability potential, make component-based programming the most cost-effective solution.

The need for component programming has been recognized by the business world and resulted in the development of systems such as CORBA [11], DCOM [12], Active X and others. However, these systems were designed primarily for sequential applications and are not necessarily suitable for high-performance programming. Similarly, the need for more complex abstractions than those offered by sequential languages lead to the development of parallel languages and libraries. From the point of view of parallel programming, the most important shortcoming of the existing component systems is that they don't support abstractions necessary for high-performance programming [8] and they don't place enough emphasis on performance crucial in parallel programming [5]. In the case of high-performance systems, the latter problem is of particular importance; it is crucial that the multi-component application developer has access to not only efficient components and efficient means of combining them, but also can quickly and reliably assess and predict performance of a multi-component application. Finally, the existing component architectures are invasive, that is, they require substantial modifications to existing applications which may not be acceptable to the developer of high-performance components. It is crucial that the abstractions and techniques comprised in a component architecture for scientific computing are based on a clear understanding of the needs of all the components which can be involved in a high-performance multi-component application containing elements of parallelism and concurrency. Only a clear formulation of such abstractions will allow us to develop efficient multi-component application and focus on developing algorithms and schemes for optimizing component interactions.

The rest of the paper is organized as follows. In section 2, we introduce a general set of requirements defining the design of Ligature, and in section 3 describe an architecture designed based on those requirements. In section 4 we will highlight performance issues arising when combining multiple components in a distributed environment and propose how to deal with them. We conclude in section 5 and present future directions.

2. Characteristics of Component Architecture for High Performance Applications

The introduction explained the motivation for Ligature and provided a rough outline of its functionality. Before we proceed to consider a concrete design, we will briefly summarize the characteristics that a component architecture should possess in order to effectively stage interactions of high-performance components.

- *Addressing different paradigms and levels of granularity*. High-Performance components are implemented according to varying paradigms, each defining different interactive requirements. For example a component based on a SPMD implementation requires interaction with all the processes taking part in computation, while a multi-threaded component won't necessarily pose such a requirement. Furthermore, the efficiency of interactions between some of the components relies on recognizing very fine level of granularity. A multi-component application built in this way can be wrapped as a component itself and later take part in coarser grain interactions with other components.

- *Integrating local components, as well as remote components in a seamless manner*. Interacting components may be local (share an executable) or remote (distinct executables) with respect to each other. A component architecture should accommodate both cases in a seamless manner, that is, little or no change should have to be made to a user's code in order to switch between these configurations. It is up to the framework to take advantage of locality, for example by avoiding making copies of data shared between components whenever possible. Remote components could be interacting over wide-area networks or local-area networks; in either case the architecture should be capable of orchestrating efficient communication between the components. In particular, assuming interaction over fast local networks, where network latency is often comparable with internal latencies of component hosts, puts additional emphasis over the high-performance aspect of interaction with the framework.

- *Heterogeneity in all aspects of the architecture*. As in many traditional component architectures, multi-component applications will incorporate different levels of heterogeneity ranging from hardware (different host architectures, different networks), through software (for example Fortran, C++, different C++ based libraries, etc.). However, given the fact that many high-performance components are implemented using sophisticated run-time systems with differing properties interoperability of local components will be much harder and sometimes impossible to achieve (for example a threaded component may be incompatible with a component which is not thread-safe).

- *Automation and ease of programming*. The abstractions and mechanisms provided by a component architecture should be high-level enough to to significantly contribute to accelerating and automating the development of a multi-component application even for a relatively simple application. For example, a mechanism should be provided for ensuring the consistency of actual component implementation and interfaces that are exported for use of different multi-component applications. This mechanism could take the form of interface generation based on highlighted features of the implementation or run-time checks.

- *Non-invasive integration*. We will call a system invasive if it requires substantial changes to an existing code in order to interoperate with it. Traditional component-oriented systems (such as CORBA) are often invasive, that is they require the programmer to write code in terms of data structures supported by them. There are two reasons to make the requirement for non-invasiveness in the context of high-performance components. First, they are complex in nature which makes modification difficult and error-prone, especially as it often requires the programmer to learn new names for implemented functions. Second, the efficiency of many applications relies on features such as a specific memory allocation scheme; modifications requiring a different scheme, or copying to a different scheme will affect the performance of a component.

- *Dynamic structure of applications*. Traditional applications are usually statically composed of some functional units. Partially due to the dynamic nature of a distributed environment composed of many contended heterogeneous resources, and partially due to the increasing need for flexibility, the multi-component applications have to be dynamic. This means that their interactions should be orchestrated in such a way as to allow the programmer to add or remove components from the application at run-time. Such decision could be influenced both by necessity and the changing performance characteristics of the application.

- *Design for efficiency*. Performance is much more critical in the system we are trying to build than in traditional component architectures. This should not only be acknowledged in the API design, but also deserves a treatment of its own. Since by necessity the components will be much more independent of each other than functional units in a monolithic application, the programmer will need not only efficient tools, but also information about what role the performance characteristics of specific components, such as scalability, numeric scalability, resource availability for a particular component, predictability and execution paradigm could play in interactions with other units. This information has to be harvested and provided as part of the design.

These characteristics are often interdependent; for example the requirement for non-invasiveness is caused both by the requirement for ease of use and efficiency. Also, not all of these requirements can be satisfied fully in the context of high-performance applications. For example, combining two components based on incompatible run-time systems within one executable is a whole separate issue; this research is pursued [7].

3. Component Architecture Design

So far we have been informally using the notion of *component* to describe a self-contained functionality or application which can interact with other components. In the context of component framework we will make this requirement of interactive capabilities more specific, and say that a component defines its inputs and outputs in some standardized manner, as well as implements a set of functionalities common to all components, such as behavior on error.

The component architecture we will describe now is a set of abstractions and techniques using these standardized features of independently developed components to implement efficient interactions between them. In addition, the framework may contain some standard components facilitating interactions. The design presented in this section builds on our experiences with PARDIS [9] and PAWS [2].

3.1. Elements of the Architecture

The interaction of elements of Ligature is depicted in figure 1. We will now describe these elements:

- *Component Description Syntax (CDS)*. This syntax enables components to define their inputs and outputs to each other and the combining framework independent of the language in which they were implemented. We will call a set of of such definitions pertaining to one component a *component interface*. The existence of common syntax not only satisfies the heterogeneity requirement (by allowing the components speak the same language), but also since it can be interpreted by the framework at run-time, new components can be added dynamically to an executing application. CDS is crucial for performance and functionality of the component architecture since the quality and detail of the abstractions it includes will define the possibilities of component interactions.

  Currently, Ligature considers two modes of component interaction: data-centric interactions, and invocation-centric interactions. In data-centric interactions the input and output specifications of components consist entirely of data that the component consumes and produces. This style of interaction is most popular in building dataflow interactions. Its main advantage is generality: the input or inputs can be provided by different components and the outputs likewise directed to different recipients, which makes interchanging components easy. Our data-centric interactions are based on the experiences of PAWS, but extend the PAWS model by introducing an ”execute” function whose task is to ”fire” a component. This extension was made in order to enable seamless transition between building interactions of local and remote components (ie. if two components are local with respect to each other, not only data, but also control must be passed from one to the other in order to enable a component to execute). The current design assumes only one standard ”execute” method per component, but may be extended in the future. Furthermore, we allow a component with an independent thread of control to ”fire” whenever it receives all necessary inputs.

  In invocation-centric interactions the input and output specifications of components consist of a list of methods and arguments used by those methods rather than purely of data. Invoking methods on a component enables a much more closely defined interaction, in which groups of arguments can be sent out as well as received, but at the same time by being more specific limits the number of components which can participate.

  Following PARDIS, we use an extended version of CORBA IDL as the CDS defining component interfaces. The extensions include:

  – Clear specifications of component inputs, outputs and their relationships or *contracts*. Component contracts describe which inputs need to be provided in order for the component to produce specific output or group of outputs. A component interface can contain several such contracts.

  – Multi-dimensional distributed grids and distribution specifications similar to those introduced in [8]. Both PARDIS and PAWS showed how the knowledge of distribution can be used to speed up data transfer between interacting SPMD components. Furthermore, [10] shows that knowing component distribution at the time of building an application (as opposed to finding out at run-time) allows the programmer to make efficient choices.

  – Restricted and unrestricted templates. CDS allows templated types similar to those in C++, however the definitions of templates can restricted to a set of types, constants and expressions actually implemented by the component. For example a particular distributed grid type produced by a component may be either checkerboard or blockwise, but not otherwise.

- *Reusable Combining Infrastructure*. The reusable combining infrastructure contains all functionality necessary to locate, move, and activate components, as well as libraries including support for local and remote interactions between components. Locating and activating components is performed in cooperation with other elements of the architecture (see repository and combining mechanism). The infrastructure also contains explicit interfaces to those elements, for example it allows the programmer to browse repositories. Implementing interactions involves not only communication and data translation; the infrastructure must also handle restrictions imposed on interactions, such as security and access levels, and must include elements of fault tolerance and error handling. Furthermore we require that the infrastructure allow the programmer to dynamically include and delete components from interactions based on their interface descriptions. In order to satisfy performance requirements the infrastructure may or may not interface with the run-time system underlying a particular component implementation. These issues will be further discussed in the next section.

- *Binding*. The binding is what gives the component interface semantics relative to a particular software system. More specifically, binding between a component CDS and its implementation is the code that for a particular software system will bind (or translate) a CDS definition into format used by the component implementation. For example, a binding to a POOMA [1] system might translate a CDS definition of a multi-dimensional grid into a multi-dimensional POOMA field.

  Binding allows us to compromise between generality of the CDS descriptions and the efficiency needed by a particular implementation. For example, a grid fragment defined in CDS and exchanged between components should not assume any particular implementation of the grid; on the other hand knowing that the grid is implemented in contiguous memory could substantially speed up interactions with it. Pushing this information into binding allows us to satisfy both requirements.

  Since providing exact and efficient bindings to every implementation framework is difficult, Ligature assumes two-levels of binding: a generic language binding using some standard representation, and a specific binding that can be formulated by more advanced users with more stringent requirements. A generic binding will usually force the user to make a copy between the standard representation and structures which are actually used, but will be immediately available; the specific binding will require the user to design a mapping to a particular implementation but will allow very efficient direct interaction. Once designed, a binding can be generated in all future interactions with a given framework. We are currently working on developing a compiler automatically generating the binding code, and also on a generic compiler builder. Furthermore, we plan to use annotated application code to generate interface definitions.

- *Repository*. Repositories contain information about components needed to enable the programmer to put together an efficient multi-component application. We distinguish two kinds of repositories: CDS Repositories and Run-Time Repositories. CDS repositories contain descriptions of component interfaces, that is sufficient information to enable the programmer to design and put together a correct multi-component application. Run-Time Repositories contain information necessary to run and refine the performance of the multi-component application, that is they contain information on how to locate and activate components, their resource requirements, security information, scalability and other performance characteristics. This will be discussed in detail in the next section.

- *Building Mechanism*. The Building Mechanism enables the programmer to combine different components into a multi-component application in a convenient and flexible fashion. It is composed of a *Building Interface* and *Helper Modules*.

  The Building Interface allows the programmer to communicate to the system which components should be composed into an application and how; it also interfaces elements of the system described above in order to provide the programmer with information about components. Such interface could be provided by a scripting language or a Graphical User Interface (GUI). An example of the GUI approach is shown in [6] where the programmer simply draws line connections between components represented as boxes. Scripting has been used in PAWS where the application programmer writes a script which selects and combines specific components, once components are combined the script starts them running and can also control their execution. Using both mechanisms interchangeably means that a GUI session can be saved as a script or a script input can serve as an initial state in a GUI session; in either case the programmer is able to save a library of multi-component applications for future use.

  Helper Modules aid the programmer in the composition phase. For example a syntax checking module can find out if all the necessary inputs to participating components have been provided and if they match the outputs from which they are derived. Of particular importance are the scheduling module (establishes where and when components can be scheduled to execute) and the performance evaluation/prediction module (tries to evaluate or predict performance given a certain machine/time configuration). In cooperation, and allowing for some experimentation, these modules allow the programmer to choose between the best configurations of components.

3.2. The Mechanism of Component Interaction

There are two modes in which the programmer can use Ligature: contribute components for future use and sharing,

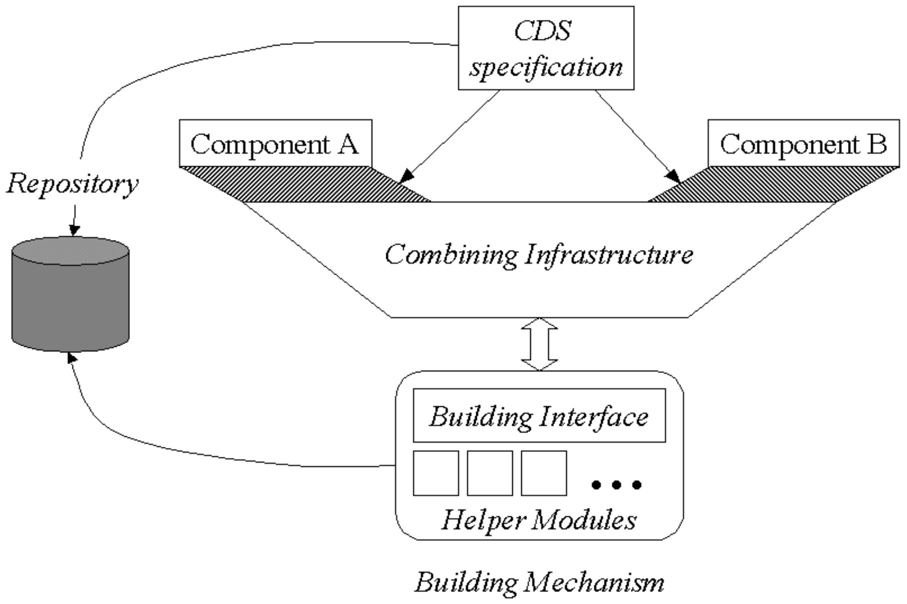

Figure 1. The figure shows how the elements of Ligature work together: the building mechanism allows the programmer to access component information in the repository; a finished scenario is then run using the combining infrastructure which interfaces components through implementation-specific bindings (shaded areas in the picture). Bindings are generated based on a CDS specification which is also used in the repository.

and use components to compose multi-component applications.

When contributing components the programmer first implements a component, then specifies its interface, or has that interface automatically generated based on an annotated version of the implementation. Automatic generation can also be used to provide binding, which uses the information contained in the component interface to enable interaction between the component implementation and the combining infrastructure. The component, or rather its interface, location and activation information are then registered with the repositories and become ready for use. Once a component is implemented and registered with a repository it can be reused in many applications.

In order to use components the programmer browses the component repository to choose components suitable for his or her application. Then he or she compose those components using the combining mechanism. The reusable infrastructure is responsible for locating, if necessary downloading, compiling and activating these components. The programmer then connects component inputs and outputs to form a multi-component application. When this process is finished the programmer can start and control the application. The process of application building can then be repeated and refined using information described in the runtime repository; the components can be exchanged, as the programmer becomes familiar with the requirements and shortcomings of the application.

4. Mechanisms for Building Performance-Sensitive Multi-Component Applications

An important and difficult aspect of developing a component architecture is ensuring high performance of multi-component applications. There are two distinct problems that need to be addressed here: designing an efficient implementation of the architecture, and providing enough information to the application programmer to develop multi-component applications efficiently. The first problem can be approached by the development of sufficiently detailed set of abstractions (for example including the data distribution information and thereby enabling parallel transfer), providing mechanisms exploiting implementation-specific features of components (for example through binding), and providing an efficient implementation of the combining infrastructure. We are trying to address these issues through our design of the architecture as described in the previous section. The second problem, which will be addressed in this section, requires recognizing and defining performance requirements of multi-component applications. The traditional approach is not sufficient here, since in a traditional ”monolithic” application all of its parts are designed to work specifically within that application. The purpose of a component architecture is to compose already existing self-contained components, not to modify them. It is therefore essential that we formulate a ”performance interfaces” of components and ways of integrating those interfaces which parallel their functional interfaces as described by the architecture.

We recognize that performance of components can be integrated on multiple levels; at present our focus is to explore ways of integrating component performance on two levels: application level, and run-time system level.

- *Application-Level*. Due to component interdependencies in a multi-component application optimal performance of individual components does not guarantee optimal performance of the entire application. An illustration of this problem is provided in [10] which describes a scheduling problem for an image processing pipeline. Let A and B be components in the pipeline such that optimal data distribution for A is different from the optimal distribution for B. Since the components have to interact with each other in the pipeline, the following question arises: what is more efficient, to use optimal data distribution for each component and pay the cost associated with data translation between components, or to use unoptimal distribution for at least one of the components if that eliminates the need for translation. This issue is made even more challenging if we consider that different components can be run on different resources. Initial research in this area includes work by Brewer [3] using statistical models of profiling data to choose an optimal algorithm from a collection of algorithms for a given architecture. Another example might involve components with different scalability characteristics.

  We address this problem by attaching to each component interface its performance data and presenting it together with other run-time information; this enables the user to choose the best possible fit from available resources. In order to do that, the architecture performs the following actions:

  – *Performance data harvesting*. This process works by instrumenting the component binding code, and at programmer’s request also component code, with performance measurement calls. After each execution the component uses the run-time repository interface embedded in the combining infrastructure to report results together with information about the environment to the run-time repository. We use the TAU system [13] to measure performance; a collaboration between LANL and the University of Oregon is currently working on automating the instrumentation process and making it a part of binding code generation; in this way gathering performance data requires no special effort from the user. The performance data is harvested for components as well as component framework parts; in the image processing problem it could for example be used for estimating the time the combining infrastructure would use to redistribute data.

  – *Performance prediction and composition*. The data harvested over multiple component executions can be used in predicting performance of new compositions of components. Given such predictions, the combining mechanism will not only present the user with a set of choices matching interactive requirements of components, but also several scenarios of the expected performance of those choices. The initial implementation will be confined to data present in the database; later we plan to experiment with extrapolating performance models from datasets, and scaling them based on different environmental parameters.

- *Run-Time-Level*. Just as adjusting the performance of one component application can influence the whole, the combining framework and run-time systems underlying the particular components can also influence each other’s performance. For example consider a multi-threaded application designating one thread to handle communication with the combining infrastructure; on one hand this thread competes for resources with the application and can therefore slow it down, on the other through enabling asynchronous communication it can prevent other components from stalling, and in this way contribute to increasing the overall performance of the multi-component application.

  In order to preserve generality and at the same time allow programmers to experiment with performance tuning Ligature supports two styles of implementing the combining infrastructure: a basic implementation containing no interface to the component run-time system, and therefore incapable of interacting with it, and another which specifies such interface. The first approach can support interactions with components of sequential and SPMD; it guarantees interaction with a wide range of components, but at the same does not allow the programmer to exploit the features of run-time system. Providing a run-time system interface on the other hand allows the combining infrastructure to to interact with the run-time system of the component in implementing communication. Providing this interface will allow the programmer to experiment with different configurations and answer questions about the relationship of communication and computation resources in the context of a particular component. Again, this performance data will be harvested, and given to the user in the form of recommendations. Implementing a run-time system interface per run-time system implementation can be an arduous task, however we are hoping to limit the number of necessary implementations by defining classes of run-time systems based on the functionality required by the combining infrastructure for implementation of interesting interactions.

5. Summary and Status

This paper describes an architecture for component programming capable of integrating high-performance components. We identified several requirements that such architecture should fulfill, and presented a design which answers these requirements. We outlined our implementation of the component description syntax understandable to all components, independent of their implementation which allows us to combine heterogeneous components. By specifying a binding to a particular implementation we ensure that the genericity of the description syntax won't prevent us from using implementation-specific features necessary for high-performance. Furthermore, the existence of such common syntax allows us to dynamically integrate application components. Automating the generation of binding code with performance instrumentation will enable us to facilitate integrating components into the framework and harvesting performance data. We further described how we plan to use this data contained in Ligature repositories to allow the programmer to develop efficient component applications.

Ligature is currently in progress. More implementation and simulation work will have to be done to gain better insight into details of the proposed architecture and ways to develop the suggested performance mechanisms.

References

[1] S. Atlas, S. Banerjee, J. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A High Performance Distributed Simulation Environment for Scientific Applications. In *Supercomputing '95 Proceedings*, December 1995.

[2] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation (to appear)*, July 1998.

[3] E. A. Brewer. High-Level Optimization via Automated Statistical Modeling. In *5th ACP SIGPLAN Symposium on Principles and Proctice of Parallel Programming (PPoPP)*, July 1995.

[4] I. Foster and C. Kesselman, editors. *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 1999.

[5] A. Gokhale and D. Schmidt. Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In *Proceedings of the 17th International Conference on Distributed Systems*, May 1997.

[6] C. R. Johnson and S. G. Parker. A Computational Steering Model Applied to Problems in Medicine. In *Supercomputing '94 Proceedings*, pages 540–549, 1994.

[7] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[8] K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation*, pages 31–39, August 1997.

[9] K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-Level Parallel Distributed Computation. In *Supercomputing '97 Proceedings*, November 1997.

[10] G. C. Lee, Y.-F. Wang, and T. Yang. Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory. *Journal of Parallel and Distributed Computing*, 45(1):29–45, 1997.

[11] OMG. *The Common Object Request Broker: Architecture and Specification. Revision 2.0*. OMG Document, June 1995.

[12] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[13] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.