INTERLACE: An Interoperation and Linking Architecture for Computational Engines

Matthew J. Sottile and Allen D. Malony

Department of Computer and Information Science, University of Oregon, Eugene, Oregon 97403-1202

Abstract. To aid in building high-performance computational environments, INTERLACE offers a framework for linking reusable computational engines in a heterogeneous distributed system. The INTERLACE model provides clients with access to computational servers which interface with "wrapped" computational engines. The wrappers implement mechanisms to translate client requests to engine actions and to move data across the server interface. These mechanisms are programmable, allowing engines of different type to be integrated. The framework takes advantage of the HPC++ runtime system to access servers through distributed object operations. The INTERLACE framework has been demonstrated by building a distributed computational environment with MatLab engines.

1 Introduction

Scientific research that involves the analysis of large data sets, the simulation of complex phenomena, or the execution of detailed computational experiments often requires scientists to develop software that runs on high-performance parallel and distributed computer systems. Unfortunately, these systems have been difficult for a scientist to program and use. The software that is developed tends to be specialized and can be hard to integrate with other tools that are commonly found in the scientist's environment. The reuse and linking of software with components from other projects suffers from the lack of a common interface and interoperation model; concerns that are usually not at the forefront of the scientist's priorities. When data sets, simulations, and experiments are of small scale, scientists will use more accessible computational programming packages, such as MatLab [4], IDL [7], and Mathematica [8]. The relative simplicity of the programming system, the consistent means to access and link code modules, and the wide availability and integration of modules of scientific interest makes these packages both productive and desirable.

However, when high-performance or distributed operation is required, the support to work with these packages, either in association with specialized software components or in replicated instances in multi-computer environments, is minimal. Scientists and other users of high-performance environments could benefit from a framework in which they can both easily create programs utilizing reusable components and scale their applications to larger systems. Where the function of those components can conveniently be served by standard tools, the framework would ideally support their easy integration and interoperation.

In this paper, we propose a framework of this type based on the more general abstractions of components, or tools as *computational engines*, their interoperation as interconnected *computational servers*, and their control as *computational scripts*. A computational server is a program that allows clients access to functionality and data provided by the computational engines linked to it. Servers are assumed to be persistent entities, available at any time to receive requests without needing to create a new server process for each. Any computational engine that creates data and is expected to share that data also must be persistent in order to maintain the data during the application's operation. In order to utilize the services of the computational engines, their functionality must be accessible to clients within the environment. In this way, the functional behavior of the server and its computational engines is dynamically determined. This is accomplished by a user program or script that accesses the interface provided by the framework. This interface is intended to hide the complexity of the underlying parallel and distributed platform, offering the user a view of a consistent set of tools for computational problem solving. What computational tools exist in the problem solving environment is determined by how the computational engines are implemented and configured. Because the framework supports a computational engine as an abstract component, any tool can be provided as long as its server interface can be created. In particular, it is important to link in both tools that are special purpose (e.g., a parallel simulation code) and tools that are derived from commercial off-the-shelf packages (e.g., MatLab-based signal analysis). The common interface and linking mechanisms of the framework makes this possible.

In the remainder of the paper, we first describe the INTERLACE framework architecture in Section §2. The implementation of the computational server and computational engine interface is discussed in Section §3. The interconnection middleware and INTERLACE nameservices are described in Section §4. At present, we have built an INTERLACE system that provides for the use of multiple MatLab tools. Our experiences with this system are reported in Section §5 as well as our plans for extending the COTS toolset to include IDL and Mathematica. The INTERLACE research is in the company of work that attempts to provide scientists with high-level support for accessing computational resources. In Section §6, we contrast INTERLACE with this related research. Future work and conclusions are given in Sections §7 and §8, respectively.

2 Architecture

A system view of the INTERLACE framework architecture is shown in Figure 1. The shaded portions of the diagram indicate INTERLACE software components. To create an environment in which computational engines are available to clients through a homogeneous interface, each engine must be wrapped and linked with a computational server. These wrappers provide the support by the server between the abstract INTERLACE environment and the specific interface of each engine. Client applications which utilize these computational servers and their engines do so by issuing messages to the wrappers, which are then translated and forwarded to the engine for processing. Users then can write scripts which are interpreted by a client in order to make the appropriate server requests on their behalf. The INTERLACE framework architecture provides a server API and description of expected server interaction behaviors, both of which are combined and implemented to create the individual computational servers.

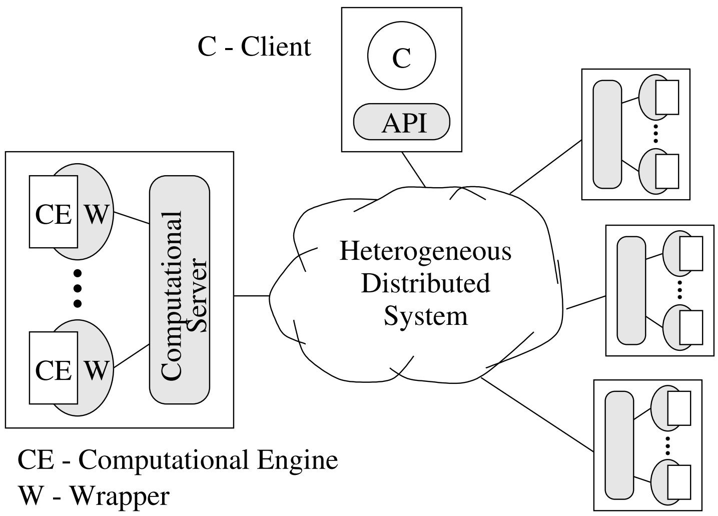

Fig. 1. INTERLACE system architecture.

The API defines how a server appears within the programmable environment by providing a small set of functions accessible to client applications. These functions facilitate data movement, manipulation, and function invocation within the engine. Since the calls are performed using messages, the developer of the wrapper is free to implement an interpreter appropriate for their purpose. This results in a range of possible server designs, from simple interpreters which pass the messages directly to the engine (as in the case for MatLab), to very complex interpreters. A complex interpreter could have the ability to convert a single message into a predefined set of discrete calls to the engine. For example, a user may wish to use a computational server such as MatLab to solve a series of linear equations using basic linear algebra operations. In this case, a single message could represent steps which build appropriate matrices, perform row reduction, and return the resulting vector.

In order to establish rules regarding the behavior of server entities within the environment, the architecture description also provides abstract protocols for data transactions and command invocation. These protocols assist clients in dealing with issues such as data synchronization, server deadlock, and cross-server synchronization. Since INTERLACE allows client applications to utilize multiple servers in parallel, these protocols must be included in the architecture specification for properly constructing computational servers.

This abstract model naturally breaks down into two distinct subcomponents for implementation - a wrapper that controls the computational engine, and an interconnection scheme providing the data and control transfer mechanisms. The following sections describe the multi-threaded wrapper design currently used with MatLab, and the interconnection facilities which INTERLACE provides to extend the existing low-level functionality of the HPC++ runtime system [1, 6].

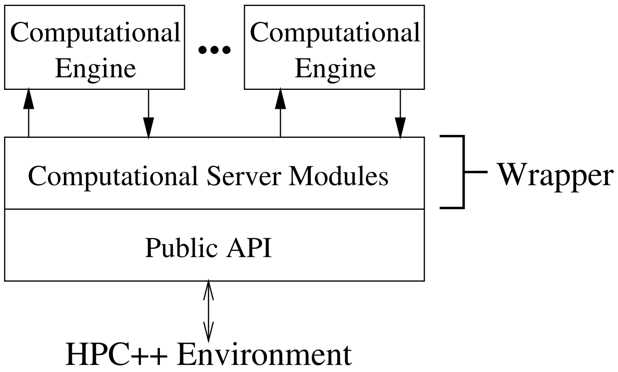

Fig. 2. Single computational server with multiple engines.

3 Wrapper Creation and Engine Interfacing

3.1 Wrapper Design

A tool, or computational engine, that is to be used within an INTERLACE system must be "wrapped" in order to be available to a computational server. This wrapper provides a translation layer between the interconnection mechanisms and the engine itself. (See Figure 2.) Since client applications utilize the server by calling functions from the given API, the wrapper must provide implementations of these functions for each specific engine. The wrapper itself is composed of four distinct modules, only one of which is accessible by clients. (See Figure 3.) The module known as the *stub* is in the server and provides registered functions within the HPC++ programming environment for clients to invoke. The stub then makes calls to three hidden modules based on the operation requested by the client. For data movement and manipulation, the stub calls the *node registry*, which serves as a data storage and registration facility for the entire server. Calls which require the engine to perform computations are passed on to the *engine* *control thread*. This module is one of two threads which directly communicate with the engine. The engine control thread manages all calls to the engine, while the *probe control thread* acts as a gateway for the engine itself to gain access to resources in the environment. Functions within the probe control thread are not directly accessed by the stub as is explained below.

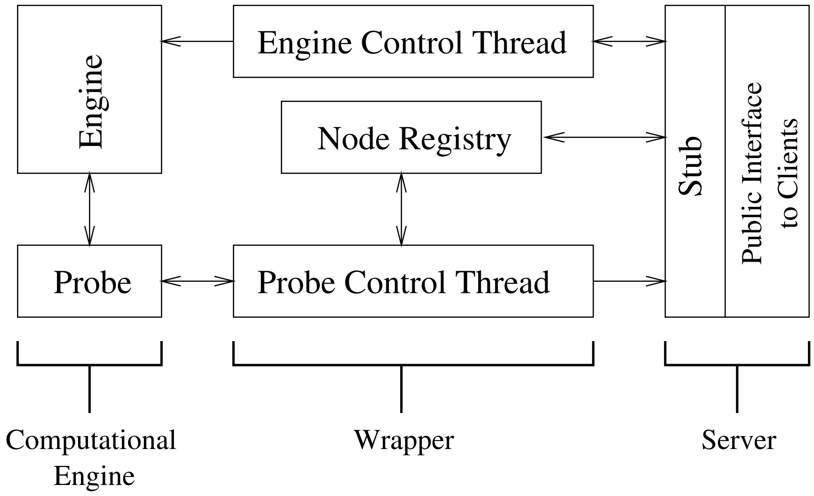

Fig. 3. Detail of server, wrapper, and engine modules.

The most important portions of the wrapper are the two threads which coordinate all data flow and control transfer between the engine and the environment. The engine control thread typically provides only two functions beyond the required inherited thread operations - a simple method to push a command request into a queue, and a method to execute all commands within the queue. Commands are string formatted messages specified by the author of the wrapper for a particular computational engine. When the command queue is processed and operations are given to the engine, the individual messages pass through an interpretation phase before being given to the engine. This allows a single message to represent a set of engine operations, and also relieves clients of the responsibility to know the exact syntax of each engine available in the environment. This message and interpreter-based command system allows INTERLACE based environments to abstract operations involving heterogeneous engines into a common, generic language.

3.2 Methods for Interfacing the Wrapper and Engine

A computational server requires that the wrapper and engine are provided with a bi-directional method for exchanging data and execution control. When dealing with engines similar to MatLab, the wrapper developer is provided with an API for creating simple C or C++ programs callable from within the engine. These programs are given some means for extracting or inserting data within the memory space of the engine itself. In combination with the engine command interface used by the engine control thread, they provide the required functions for data manipulation. These callable programs are referred to within INTERLACE as *probes*, and are monitored and controlled by the probe control thread.

The probe and probe control thread are the most complex portions of the wrapper. This is due to severe limitations on what requirements can be imposed upon the probes for communication with the HPC++-based probe control thread. Many programs such as MatLab may have compatibility problems with threading libraries used by HPC++, therefore one cannot assume that the probe can be written using these services. Alternative methods for communication and control synchronization must be utilized such as operating system level interprocess communications (IPC) or TCP/IP sockets. These interfaces are far more likely to be compatible with the engine programming interface and can be easily used within an HPC++ based program. Furthermore, such low-level facilities are also available to programmers using a variety of languages, whereas HPC++ interfaces would currently be available only to C++ and Java programmers.

Since the probe provides an interface for the engine to gain access to the external environment, the probe must have a sufficiently flexible command set. Currently, the probe may implement six commands related to data manipulation, command invocation, cross-server synchronization, and engine execution interruption. Data manipulation is accomplished through registration of data with the node registry (or centralized nameservice, if available) so that data can be published, but not actually moved until required. This movement is accomplished by providing operations for registering, requesting, and pushing data between servers. Function invocation is a single simple command which allows a message to be sent to another server for processing. Synchronization operations are sufficient in situations where barrier-based methods are appropriate, and the probe allows the creation and use of these structures. Finally, the probe is given the ability to gracefully terminate the entire server.

Execution flow for the server is managed at the engine control thread for incoming commands, and at the probe control thread for commands issued to other servers. Since other servers may execute simultaneously, there as a possibility of incoming work arriving at a server from another while it is busy. This is solved by the stub not blocking while the engine is executing, allowing incoming commands to be received and pushed into the command queue. When the engine is ready again, the queue is processed. Since simultaneous execution of servers is allowed, it is possible for two servers to deadlock. However, commands issued by a server do not require it to block until the command is complete. If the server does not block when issuing to another server, it will continue to process its queue and eventually fulfill the request of the other busy server.

4 Server Interconnection Mechanisms

As stated earlier, HPC++ provides basic data transport and method invocation services. In order to maintain the layer of environment abstraction required by INTERLACE, there must be additional services providing functionality beyond that offered by HPC++. Among these are the implementation of a centralized *nameservice* to maintain a dynamic list of available servers, data entities, and available computational services. INTERLACE attempts to avoid "hard coding" information about other servers within a wrapper by dynamically providing such information at runtime. The entire nameserver is divided into four portions which provide specific types of information mappings.

Depending on the type of information to be located, the nameserver maintains four sets of mappings: HPC++ *context* to name, data name to location, computational operation to location, and HPC++ function registration ID to function name. The first mapping is the most important and allows client and server developers to ignore many issues dealing with locating other servers by explicit name to context associations. Authors can simply assume a name will be associated with a particular host/port identifier, which will be determined at runtime when the context is created and registered with a known identifier. The data and function name maps provide a method for addressing server functionality and data elements as named, untyped objects. This is particularly important in the case of function addressing since it hides details related to the HPC++ remote member function invocation(RMFI) services which perform the actual function call.

As a result of these maps, it is almost possible to have a computational environment in which servers are dynamically inserted and removed. The final nameserver mapping makes this behavior possible by giving entities the ability to dynamically determine information required for participation with other HPC++ services, such as class and function identifiers. HPC++ requires that functions and classes be given a numeric identifier which is used by clients to identify and call the particular service with RMFI. Since a particular server may be invoked in multiple instances within the same environment, these identifiers must be unique between each instance to prevent conflicts. An example of such use of a multiply-instantiated server would be in the case of a large parallel or distributed environment in which a problem involves the execution of a single program with each node processing different data sets. In this case, the functions would conflict unless their identifying parameters were determined at runtime.

5 Application of INTERLACE with MatLab

The prototype implementation utilizing INTERLACE demonstrates how high performance applications can be composed of multiple MatLab engines. Since this was the initial target, much of the design for the current specification was influenced by MatLab behavior and its interface. However, the MatLab interface is similar to that of many other tools, such as IDL and Mathematica. The wrapper uses libraries provided with MatLab for gaining access to the engine command interface and data manipulation services. The engine control thread is able to invoke an instance of the engine and provide it with string-based MatLab commands for processing. Though this interface provides methods for moving data between the control thread and engine, the probe interface was chosen for this task to keep all data transactions in a single module of the wrapper.

The probe was constructed in C as a MatLab *MEX* file. These files are compiled C programs which are invoked by the engine as MatLab commands. Since the probe implemented in this case was done using C, the chosen facilities for communicating with the probe control thread were basic operating system level interprocess communication functions. Signaling was accomplished using semaphores and data transfer relied upon shared memory. The probe was built such that a single MEX program could be invoked to perform all probe functions. It accepts two parameters, the first being a character string containing the command to process, and the second parameter containing the MatLab variable which the command may use.

The implemented commands within the MEX probe included simple functions for requesting and publishing data elements, issuing commands to other servers, synchronizing with other processes using barriers, and a simple action to end the execution of the computational server. By providing these commands, a user can easily utilize the INTERLACE environment by instrumenting normal MatLab scripts with probe commands.

We initially targeted a computational environment where MatLab servers would provide for individual MatLab commands to be sent directly to the engine. This resulted in a very simple interpreter in the server stub which simply passes the command message to the engine control thread for queueing. In effect, the MatLab programming interface is extended by INTERLACE to the client. That does not mean that the commands are simple MatLab commands. On the contrary, the MatLab engines can be customized with modules that offer high-level operations to the client.

6 Related Work

The concept of providing scientists convenient access to computational software resources distributed across a network and possibly running on parallel machines is one that has been investigated in a variety of ways. One approach is to start from a familiar scientific programming environment and add support for allowing multiple computational engines derived from the environment to run across machines and interoperate. The MultiMatLab [3] system provides facilities for building "parallel" MatLab [4] programs that can execute on a distributed or parallel platform. The Message Passing Interface (MPI) [5] is used as the interoperation glue to distribute data and work between MatLab processes. The advantage is that the full MatLab computing environment is available at all nodes. The disadvantage of MultiMatLab is that the computational engines can only be based on MatLab.

If computational engines of various types are to be supported, the problem becomes one of designing a framework in which the engines are easily interfaced and their functionality accessed. The Virtual Notebook environment (ViNE) [2] offers such a framework built from web protocol and browser technologies. Computational engines (ViNE *tools*) are specified through a notebook interface, associating tool location, input and output formats, and tool interaction control with a global tool name within the environment. Tools that are command line driven and work with files are most easily integrated. The ViNE experiment component then provides users with the ability to create and execute distributed computational experiments from a data flow description of tool execution. ViNE succeeds in providing the user a high-level logical interface for accessing tools while handling all distributed computing issues behind the scenes. The shortcomings of the approach are the primitive nature of tool interoperation. Although MatLab derived tools can be configured into a ViNE system, none of the tools are persistent, needing to be restarted with each invocation,, and their control occurs from a central point. Other collaborative tool environments include SciVis [10] from the Northeast Parallel Architectures Center at Syracuse, and PUNCH [11] at Purdue.

The NetSolve [12] model approaches the problem of user access to computational tools by defining API's for function execution. Client interfaces to NetSolve are supported for C, Fortran, MatLab, and Mathematica, encoding function name, input and output objects, and status for synchronous and asynchronous function requests. NetSolve agent components handle requests by locating computational servers that offer the function and passing along the request to the server for execution. The client-server function invocation model is well understood making it easy to extend server libraries and retarget the interface to different client types. The simplicity of the model also accounts for NetSolve's limitations. Computational services are not persistent in NetSolve, requiring additional support in the framework for global or, at least, peer-to-peer data naming. Function names are encoded in the API making it cumbersome to extend the system with new computational functionality. Finally, NetSolve's operational model makes it difficult to create a true parallel computation. Only by managing multiple asynchronous function requests can a client take advantage of parallelism. Some of these problems are being address in future NetSolve research.

The Harness project [14] at Oak Ridge National Laboratories is attempting to address some of these aforementioned problems. Harness builds on the concept of the Distributed Virtual Machine that was pioneered by the PVM [13] research. It seeks to support a generalized plug-in interface that allows users to dynamically customize, extend, and adapt the environment, and multiple parallel applications to attach, interact, and detach from the system. The goals of Harness are much larger than those of INTERLACE, but many of the fundamental issues of providing an extensible framework for configuring and integrating computational engines are the same.

7 Future Work

The current implementation of an INTERLACE based system involves only the connection of MatLab servers and a C++-based control client. As a next step, the wrapping of IDL as a computational server will provide a more interesting example showing heterogeneous servers coexisting within a common programmable environment. Other potential targets for wrapping as computational engines include Mathematica, GNUPlot [9], and other analysis tools commonly used in scientific and mathematical research. It is also likely that investigation of new methods for easily generating tool wrappers will provide useful for rapidly building analysis environments. Currently, the developer of the wrapper must build virtually from scratch, using only a skeletal set of source files defining the required API common between all computational servers.

8 Conclusion

Fulfilling the need for a framework for connecting computational engines in a scalable, reusable manner is the central goal of INTERLACE. By building upon HPC++, INTERLACE is usable within a variety of high-performance environments. The abstraction of functionality and data offered by computational engines to the homogeneous INTERLACE API and interconnection scheme provides many advantages. It allows dynamic environments in which computational engines may be instantiated as required, while hiding implementation details required to support such behavior. Since the computational server framework provided by INTERLACE is an extendible skeleton, the application of this framework to specific tools can be fine-tuned to the user's needs. As a result, INTERLACE is a powerful tool for scientists and others who wish to build a reusable and scalable environment for doing high-performance analysis and research.

References

1. E. Johnson and D. Gannon, HPC++: Experiments with the parallel standard template library. *Technical Report TR-96-51* Indiana University, Department of Computer Science, December 1996.
2. J. Skidmore, et al., A Prototype Notebook-Based Environment for Computational Tools, *Proceedings of SC98*, November 1998.
3. V. Menon and A.E. Trefethen, MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing, *Proceedings of SC97*, November 1997.
4. The Math Works, *MATLAB: High Performance Numeric Computation and Visualization Software Reference Guide* Natick, MA. 1992.
5. Message Passing Interface Forum, *A Message-Passing Interface Standard*, May 1994.
6. P. Beckman and D. Gannon, Tulip: A portable run-time system for object-parallel systems. *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

7. Research Systems, IDL.
`http://www.rsinc.com/`.
8. Wolfram Research, Mathematica.
`http://www.wolfram.com/`.
9. GNU's Not Unix, GNUplot.
`http://www.gnu.org/software/gnuplot/gnuplot.html`.
10. B. Ki and S. Klasky, Scivis, *Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, February 1998.
11. N. Kapadia and J. Fortes, On the Design of a Demand-Based Network-Computing System: The Purdue University Network-Computing Hubs, *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.
12. H. Casanova and J. Dongarra NetSolve: A Network Server for Solving Computational Science Problems, *Proceedings of the Supercomputing Conference*, 1996.
13. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck, and V. Sunderam, *PVM: Parallel Virtual Machine User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.
14. J. Dongarra, A. Geist, J. Kohl, P. Papalopoulos, an V. Sunderam, HARNESS: Heterogeneous Adaptable Reconfigurable Networked Systems, Proceedings of HDPC'98, February 1998.