A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components*

M. Li, O. F. Rana, and M. S. Shields
Department of Computer Science
Cardiff University, PO Box 916,
Cardiff CF24 3XF, UK
{maozhen,omer,m.s.shields}@cs.cf.ac.uk

D. W. Walker
Computer Science and Mathematics Division
Oak Ridge National Laboratory, PO Box 2008
Oak Ridge TN 37831-6367, USA
walker@msr.epm.ornl.gov

Abstract

This paper describes a Wrapper Generator for wrapping high performance legacy codes as Java/CORBA components for use in a distributed component-based problem-solving environment. Using the Wrapper Generator we have automatically wrapped an MPI-based legacy code as a single CORBA object, and implemented a problem-solving environment for molecular dynamic simulations. Performance comparisons between runs of the CORBA object and the original legacy code on a cluster of workstations and on a parallel computer are also presented.

Keywords: Problem Solving Environments, Java/CORBA Components, Wrapper Generator, Distributed Parallel Computing.

1 Introduction

A problem-solving environment (PSE) is a complete, integrated computing environment that provides all the computational facilities necessary to solve a target class of problems [4, 7]. The main motivation for developing PSEs is that they provide software tools and expert assistance to computational scientists in a user-friendly environment, allowing more rapid prototyping of ideas and higher research productivity. By relieving scientists of the burdens associated with the inessential and often arcane details of specific hardware and software systems, the PSE leaves them free to concentrate on the science. Construction of PSEs through software components is an approach that has engendered much recent interest [5, 6, 8, 14]. Our component model guarantees that components written in different programming languages can interoperate on a binary level.

*This project is funded by EPSRC grant GR/M17327.

CB-PSE [1] is a distributed component-based problem-solving environment for high performance scientific computing. In the CB-PSE, a user can visually construct scientific applications by connecting together components, which can range in granularity from simple tasks, such as matrix manipulation, to complete application programs. Each component is a CORBA [12] object and has an XML [13] interface described using a common data model, as specified in [3]. A component may contain sequential code written in Java, Fortran, or C, or it may wrap parallel code based on a SPMD implementation using MPI [17], or it may contain array-based parallelism using language extensions such as HPJava [11]. CORBA enables the seamless integration of distributed objects within one system, and is designed primarily for sequential applications. High performance computing applications are mostly parallel programs using message passing paradigms such as MPI. CORBA cannot replace the MPI communication layer due to architectural and performance constraints. To fill this gap, we add parallelism to a CORBA object by running an MPI runtime environment inside the object to manage the intra-communication within the parallel CORBA object, and using CORBA to manage the inter-communication among objects. The advantage is that users can use existing CORBA implementations (such as Visibroker, Orbacus and others) without any modification to CORBA IDL compilers, as is done in other projects with a similar objective, such as PARDIS [10] and Cobra [9].

Components in the CB-PSE can be divided into four types: (1) *compute* components are responsible for computational work such as performing a matrix multiplication or a fast Fourier transform, (2) *template* components require one or more components to be inserted into them in order to become fully functional. *Template* components can be used to embody commonly used algorithmic design patterns. This allows a developer to experiment with different numerical methods or model physics. Thus, in a partial differential equation solver a *Template* might require a pre-conditioner to be inserted. *Template* components can also be used to introduce control constructs such as loops and conditionals into a higher-level component, (3) *data manager* components are used to read and convert between the different types of data format that are held in files and internet data repositories, and (4) *user interface* components allow a user to control an application through a graphical user interface, and play a key role in supporting computational steering.

The two major parts of CB-PSE are the Visual Component Composition Environment (VCCE) and the Intelligent Resource Management System (IRMS). The VCCE is primarily used to construct applications from software components, and supports the location of components and the transfer of data between them. After an application is constructed, the VCCE generates a task graph in XML which is later passed to the IRMS. The IRMS then makes use of a resource management system, such as Intrepid [16], to run the application on distributed computing resources, subject to constraints defined by the user. Figure 1 shows the architecture of CB-PSE.

Components can be created from scratch, or automatically wrapped from legacy codes through a Wrapper Generator (WG), as described in section 3. Our work differs from the SWIG [15] project, which provides an interface compiler that connects legacy codes written in C, C++, and Objective-C with scripting languages such as Perl, Python, and

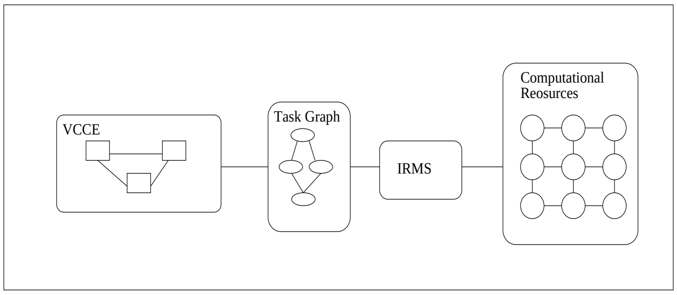

Figure 1: The CB-PSE architecture based on the component model

Tcl/Tk. Whereas SWIG focuses on manipulating legacy codes through the use of scripting languages, our WG is used to automatically wrap legacy codes as Java/CORBA components which can then be plugged together to create applications for use in a distributed computing environment.

2 A CORBA Based Component Model

A component in the CB-PSE has five parts: a Listener, a Publisher, a Body, a CORBA IDL interface, and an XML interface.

- The Listener serves as a trigger to invoke the component, based on a unique component ID and the notion of an event service. It receives requests from other components through the CORBA ORB. It is invoked if the component ID carried by a call is the same as the one that it holds. A protocol may be defined to enable communication between a Listener of one component and a Publisher of another component. The protocol includes a flag to determine if large quantities of data need to be transferred, whether computational steering is required, or if the component needs input to be obtained as a stream or a data file. In cases where there is a large amount of data to be transferred, interacting components will set up a direct socket connection between the Publisher and the Listener for faster data transfer, which will bypass the CORBA ORB. The Listener first receives data from the requesting component, and then invokes the Body of the component. The Listener is the client side implementation of the component.

- The Publisher is used to send requests (events) to other components after the present component finishes its task. It is the Publisher that sends data generated by the component of which it is a part to other components. When a user connects two components together, the CB-PSE will put the ID of the input component(s) into the Publisher of the output component. Therefore, the Publisher holds one or more IDs of the components which will be invoked next.

- The Body is the server side implementation of a component. It receives input from the Listener of the component, and it sends output to the Publisher of the component.

- A CORBA IDL interface is generated from the description of a component provided by its developer. It is used to generate stubs for the Listener and Publisher, and skeletons for the Body of a component.
- An XML interface is also generated from the description of a component provided by its developer. It is used to store information related to each component. From the XML interface, the properties of a component can be interrogated, or the interface can be used to identify components of relevance to a given application. For each computational component, for instance, there may be a performance model defined in XML for use in scheduling components for execution. The complexity of this performance model can range from results of benchmarking, to algebraic complexity metrics related to the size of the data set, and the data types managed by the component.

Figure 2 shows the component architecture used within CB-PSE. The CORBA ORB is responsible for managing communication between components. Each component selected by a user is registered with the Name Server, which helps the execution environment identify the location of this component. The Component Repository (CR) contains the XML interface of a component, and does not include any executable code (Body) associated with it. Component executables can be subsequently located based on the references provided within the XML definition (see [3] for details). There are two kinds of information in the CR, one is static information about components in XML, the other is run-time information about components which are currently running, obtained from the CORBA name service. Components can be hierarchical, and may be created from a *Template* component which identifies the use of a combination of components within a given application. Components may also be generated directly by the user, or wrapped from legacy codes through the Wrapper Generator.

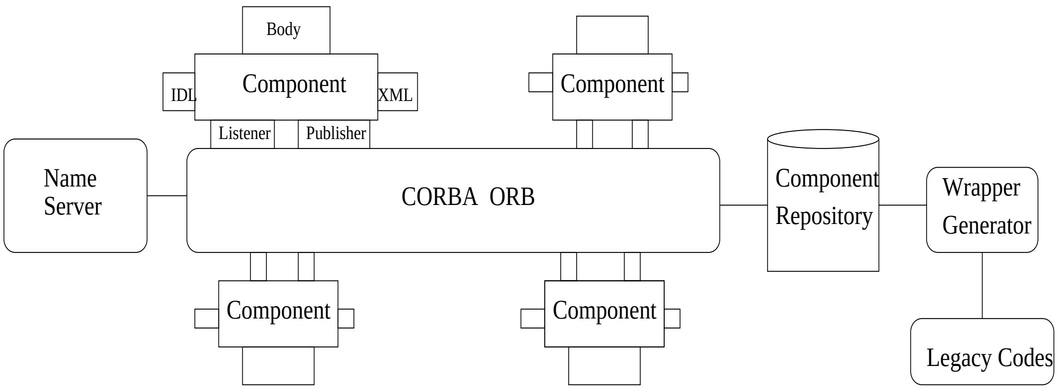

Figure 2: The CORBA-based component architecture for use in the CB-PSE

3 The Implementation of the Wrapper Generator

High performance legacy codes are pre-existing codes, mostly in C or Fortran, that possess the following features: (1) they are domain-specific; (2) they are hard to re-use in other applications; (3) they are still useful; and, (4) they are often large, complex monoliths with little structure. Wrapped as CORBA objects, these legacy codes can be re-used as components in a heterogeneous distributed computing environment. Wrapping legacy codes manually is a time consuming and error-prone task. Developers have to write all the interfaces needed to convert a legacy code as a component. For example, they need to write an IDL interface and write an implementation code for the component wrapped from the legacy code. Depending on different properties of a legacy code, the IDL and implementation code are also different. In order to automatically wrap legacy codes as components, we have implemented a Wrapper Generator (WG) on Solaris 2.7 using VisiBroker as the CORBA ORB. Making use of a CORBA ORB as the communication infrastructure, components are independent of location, language, and platform. At present, Java is chosen as the language for component wrappers generated by WG, though the wrapped component core can be C, Fortran, or Java itself. This flexibility facilitates the automatic incorporation of a wide range of existing legacy codes into the CB-PSE framework. When using WG, developers only need to specify the parameters (properties) of the legacy code they want to wrap, then submit the parameters to WG which then generates all the interfaces needed to convert the legacy code into a component. Figure 3 shows the data flow in the WG.

Developers are different from end users in that developers create components, whereas end users make use of the components to construct applications. Developers need to know some information about the legacy code, such as its location. The main constraints of a legacy code that can be wrapped as a component with the WG are: (1)The legacy code can be a sequential code or a parallel code using MPI. (2)The legacy code can be written in C, Fortran or Java. (3)The legacy code can be located anywhere within a distributed computing network. (4)The legacy code must be a binary code and can perform certain functions with some input(s)/output(s). When using the WG to wrap a single legacy code, developers need to supply the following parameters:

- Name, which is used as the name of the component generated from the legacy code.
- Component Directory, which is used to store all the files related to the component.
- The Location of the legacy code.
- The Name of the legacy code itself.
- The Name of the ORB Compiler. Since we use VisiBroker, the compiler is *idl2java*.
- The Language used by the legacy code. The WG currently supports legacy codes in C, Java, or Fortran.

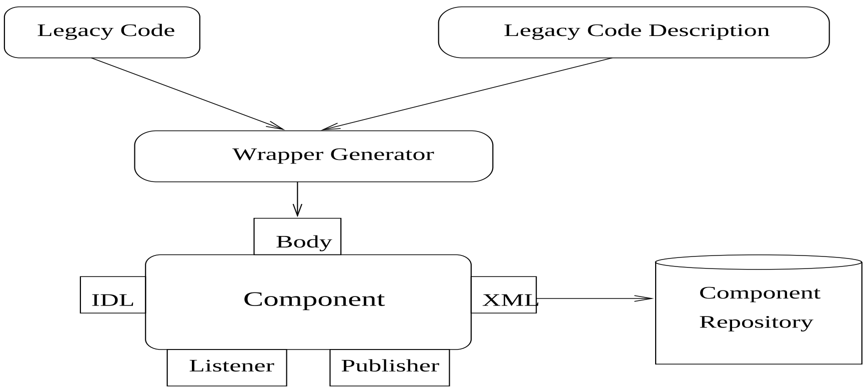

Figure 3: The WG data flow

- The Type of the legacy code. Whether the legacy code is sequential or parallel code using message passing libraries such as MPI. The component will be wrapped as a parallel CORBA object if the legacy code is a parallel code using MPI. The CB-PSE will use the MPI run time to manage the communications within the parallel object.
- The Processors used if the legacy code is a parallel code using MPI.
- The Host name where the legacy code resides.
- The Number of input parameters used for the legacy code.
- The Type of each input parameter used by the legacy code. The WG supports seven types currently, *file*, *char*, *string*, *int*, *float*, *double*, and *long*.
- The Number of outputs generated by the legacy code.
- The Type of each output parameter.
- The Data File used to store the data generated by the legacy code.

After receiving the parameters from a developer, the WG will check the validity of these parameters, such as the validity of the component directory, and whether the input types belong to the types it supports. The WG will then generate a CORBA IDL and an XML interface based on the input parameters. Using an IDL compiler, WG generates stubs and skeleton for the Listener and Publisher of the component. Based on the specified parameters, the WG knows how to generate the Listener. If there are no data input to the component, the generated Listener will automatically invoke the Body of the component once a request has been received. Otherwise, the generated Listener will first finish receiving data from another component, and then it invokes the Body of the component. Input data may either be streamed to the component, or read from a file. The Publisher is generated in a similar way to the Listener. The generation of the Body is completed by a Body template within WG, making use of the skeleton created by the IDL compiler.

The main function of the Body is to invoke the legacy code wrapped inside it. After generating all the interfaces needed to wrap the legacy code as a component, the WG stores the component in the CR in XML for future use.

4 A Molecular Dynamic Simulation Application

In this section an example of the use of the WG to automatically generate wrappers for wrapping an MPI-based legacy code as a CORBA component will be discussed. We have constructed an application for molecular dynamics (MD) simulations in the CB-PSE. There are two components in the application, one is a *User Interface*(UI) component, the other is the MD component automatically wrapped from a MD legacy code with the WG.

The legacy code used is a three-dimensional molecular dynamics code for simulating a Lennard-Jones fluid. The code has been parallelised, and makes use of the MPI message passing library for inter-processor communication. The code models short range atomic interactions by using a link-cell (geometric hashing) algorithm where all particles are hashed into a three-dimensional mesh of $N_b \times N_b \times N_b$ cells. The cell size must be no smaller than the cut-off distance (r_c) used in the short-range force evaluation so that each particle interacts only with particles in the same cell or in the neighbouring cells. The symmetry of Newton's Third Law is exploited so that atoms in only 13 (instead of 26) neighbouring cells need to be examined. The code assumes an $N_c \times N_c \times N_c$ Face Centered Cubic (FCC) periodic lattice with a total of $N = 4N_c^3$ atoms. A "shifted-force" [19] Lennard-Jones 6-12 potential ensures that the potential and its first derivative are continuous at the cut-off distance. Particle positions are updated at each time step using a simple Verlet [20] leap-frog scheme. Further details of the molecular dynamics algorithms can be found in the book *Computer Simulation of Liquids* [21].

A spatial decomposition [22] is used which distributes the cells in blocks over a three-dimensional mesh of processes so that each process is responsible for the particles in a rectangular sub-domain. Point-to-point message passing is necessary to perform two tasks in the algorithm. First, particle information in cells lying at the boundaries of a process must be communicated to one or more neighbouring processes. This is necessary because these particles must interact with particles in neighbouring processes. The standard approach of creating "ghost" cells around the boundary of each process is used, and the communication can then be performed in a series of six shift operations (one for each face of the rectangular sub-domain). The second type of point-to-point communication arises when particles migrate from the sub-domain of one process to that of another. Again this communication can be performed in a series of shift operations. In the message passing code the communication of boundary data and particle migration are combined to reduce the frequency (and hence the overhead) of message-passing.

The MD legacy code description in XML is given in figure 4, and provides an example of the 'Legacy Code Description' used in figure 3.

The legacy code "moldyn" is an MPI-based code making use of 8 processors. It has one input and six outputs (time step, total energy, kinetic energy, potential energy, pres-

```
<pse-def>
   <preface>
     <name alt="MD1" id="MD01"> MDComponent</name>
     <pse-type> Molecular Dynamics </pse-type>
     <component-directory>/home/scmlm1/wgen/Component</component-directory>
     <legacy-code>/home/scmlm1/md/moldyn</legacy-code>
     <ORB-Compiler>idl2java</ORB-Compiler>
     <processors>8</processors>
     <host-name>sapphire.cs.cf.ac.uk</host-name>
   </preface>

  <ports>
    <inports>
    <inportnum> 1 </inportnum>
    <inport id="1"> file </inport>
    </inports>

    <outports>
    <outportnum> 6 </outportnum>
    <outport id="1"> int </outport>
    <outport id="2"> float </outport>
    <outport id="3"> float </outport>
    <outport id="4"> float </outport>
    <outport id="5"> float </outport>
    <outport id="6"> float </outport>
    <href name="file:/home/scmlm1/wgen/Component/output.data" value="output" />
    </outports>
  </ports>

  <execution id="platform">
    <type> Solaris 2.7 </type>
    <type> MPI </type>
  </execution>

   <help context="instantiate">
     <href name="file:/home/pse/help/md.txt" value="NIL" />
   </help>
</pse-def>
```

Figure 4: XML description of MD legacy code.

sure, and temperature). The legacy code can run on a cluster of workstations or on a dedicated parallel machine. The IDL interface generated by the WG for the MD legacy code is given in Code Segment 1.

```
module Simulation
{
    interface MDComponent{
        void Listener(in string ComponentID, in string inputs);
        void Body(in string parameters);
        void Publisher(out string ComponentID, out string outputs);
    };
};
```

Code Segment 1: The IDL for the MD CORBA object

The UI component (which may be embedded within a web page as an Applet) provides a graphical front end to receive inputs from a user and then sends a call to the MD component through its Publisher. The Listener of the MD component is then triggered and receives messages from the UI component based on protocols used for interactions among components. After the Listener finishes its task, it then invokes the Body of the MD component which in turn starts to execute the MD legacy code using a command such as "mpirun -np 8 moldyn". When there is an output, the Body invokes the Publisher to make a callback to the UI component to display simulation results to the user. The implementation code for the MD CORBA object generated by the WG is briefly described in Code Segment 2.

```
class SimulationImpl extends _SimulationImplBase
{
    public void Listener(String ComponentID, String inputs)
    {...
        if(ComponentID=="MDSimulation")
            read parameters(inputs) from UI component
        invoke Body of the Component
    }
    public void Body(String parameters)
    {...
        execute "mpirun -np 8 moldyn parameters output.dat"
        send data(output.dat) to the Publisher of the component
        invoke the Publisher
    }
    public void Publisher(String ComponentID, String outputs)
    {...
        read data(output.dat) from the Body of the component
        ComponentID="UIComponent"
        invoke UI component with outputs
    }
```

```
}
```

Code Segment 2: The implementation code for the MD CORBA object

5 Performance Comparisons

It is important to minimise performance overheads when using wrapped legacy codes as components for use in the CB-PSE. In order to measure and compare performance between the wrapped MD CORBA object and the legacy code itself we carried out a number of experiments running the MD CORBA object and the legacy code on a cluster of workstations and on a dedicated parallel machine (named *cspace*). The cluster runs Solaris2.7 and MPICH1.2.0, connected over an intranet (with shared file space) with 10Mb/s Ethernet. The parallel machine is a Sun E6500 with thirty 336MHz Ultra Sparc II processors, running Solaris2.7 and using MPI libraries from Sun Microsystems. We increased the number of molecules from 2048 to 256,000, using 8 workstations in the cluster and 8 processors in cspace. We use the Visibroker ORB from Inprise on both cspace and the workstation cluster. The results are illustrated in figure 5. We find that there is almost no loss of performance between the CORBA component wrapped from the native code, and the native code itself. These results are observed on both the cluster of workstations, and on the dedicated parallel machine. We do however see a significant difference between execution speeds on cspace and the network of workstations when using CORBA.

Our results suggest that parallel CORBA objects wrapped from parallel legacy codes using MPI can still maintain high performance. We attribute this to improved implementation of data management within the Visibroker ORB, and the use of a shared file space in our experiments. We also reached a similar conclusion in a previous study where we have compared the performance of an MD code wrapped as a single CORBA object, versus the performance of the MD code divided into a collection of co-operating CORBA objects [2]. Making use of CORBA to manage the inter-communications among objects and a MPI runtime to manage the intra-communications within a parallel object is a feasible approach for distributed parallel computing.

6 Summary

High performance legacy codes mostly written in C or Fortran making use of message passing paradigms such as MPI are very useful computational resources. Our WG has been used to automatically wrap such legacy codes as Java/CORBA components for use in the CB-PSE, a distributed component-based problem-solving environment. In general, the legacy code can be a whole application or a subroutine. Since components in the CB-PSE are CORBA objects, they are independent of location, language, and platform. The wrapped high performance components running on a cluster of workstations or on a parallel machine can be invoked by a user from a thin desktop computer. The intra-communication within a parallel component is managed through an MPI runtime and the

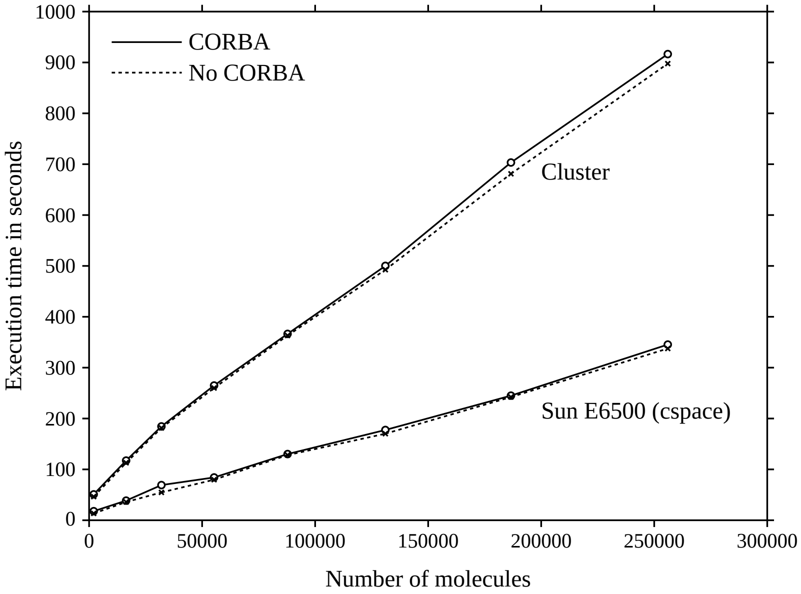

Figure 5: Performance comparisons running the wrapped CORBA object and the legacy code itself on a cluster of workstations and on a dedicated parallel machine

inter-communication among different components is managed by a CORBA ORB. Our approach does not make any proprietary extensions to the CORBA IDL or ORB, as in other similar projects, and therefore can be used with a range of commercial and research ORBs, such as Visibroker, Orbacus, and TAO [18].

The WG provides a feasible way to automatically wrap high performance legacy codes as Java/CORBA components, however, it is rather simple and there is still work that needs to be done to improve it. We plan to extend the data types supported by the WG (such as structured data types) and to explore ways to use the WG to generate wrappers for wrapping legacy codes as components at the source level.

References

[1] Maozhen Li, Omer F. Rana, and David W. Walker. CB-PSE: A Component-Based Problem Solving Environment. In Proceedings of the First International Conference on Information Reuse and Integration, published by the International Society of Computers and their Applications, held in Atlanta, USA, pp. 7-10, November 1999.

[2] Maozhen Li, Omer F. Rana, and David W. Walker. Wrapping MPI-Based Legacy Codes as Java/CORBA Components. to appear in a special issue on Java in high performance computing by the Journal of Future Generation Computer Systems(FGCS), 2000.

[3] O. F. Rana, M. Li, D. W. Walker, and M. Shields. An XML Based Component Model for Generating Scientific Applications and Performing Large Scale Simulations in a Meta-computing Environment. Proceedings of Generative Component Based Software Engineering, Erfert, Germany, September 1999.

[4] E. Gallopoulos, E. N. Houstis, and J. R. Rice. Computer as Thinker/Doer :Problem-Solving Environments for Computational Science. IEEE Computational Science and Engineering, Vol. 1, No. 2, pp. 11-23, 1994.

[5] Common Component Architecture Forum. See web page on CCA at `http://www.acl.lanl.gov/cca-forum`.

[6] Katarzyna Keahey, Peter Beckman, and James Ahrens. Component Architecture for High-Performance Applications, First NASA Workshop on Performance-Engineered Information Systems, September 28-29, 1998.

[7] J. R. Rice and R. F. Boisvert. From Scientific Software Libraries to Problem-Solving Environments. IEEE Computational Science and Engineering, Vol. 3, No. 3, pp. 44-53, 1996.

[8] Y. N.Lakshman, Bruce Char, and Jeremy Johnson. Software components using symbolic computation for problem solving environments. In proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation, August 13 - 15, 1998, Rostock Germany.

[9] T. Priol and C. Ren. Cobra: A CORBA-compliant Programming Environment for High-Performance Computing. In Proceedings. of Euro-Par'98, number 1470 in LNCS, Springer Verlag, pp. 1114–1122, Southampton, UK, September 1998.

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

[11] B. Carpenter, Y.-J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with HPJava. In *Java for High Performance Scientific and Engineering Computing, Simulation, and Modelling*, Syracuse, New York, 1996.

[12] `http://www.omg.org`.

[13] `http://www.w3.org/XML`.

[14] C. Szyperski, Component Software:Beyond Object-Oriented Programming. Addison-Wesley,1997.

[15] David M. Beazley and Peter S. Lomdahl. Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations. In Proceedings of SuperComputing'96.

[16] N. Floros, A. J. G. Hey, K. E. Meacham, J. Papay and M. Surridge. Predictive resource management for meta-applications. Future Generation Computer Systems, Vol. 15, pp 723-734, 1999.

[17] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org`, 1995.

[18] Douglas C. Schmidt, David L. Levine, and Chris Cleeland. Architectures and Patterns for High-performance, Real-time CORBA Object Request Brokers. Advances in Computers, Academic Press, Ed., Marvin Zelkowitz.

[19] J. G. Powles, W. A. B. Evans, and N. Quirke, Non-destructive Molecular Dynamics Simulation of the Chemical Potential of a Fluid, Mol. Phys, Vol. 46, pp. 1347–1370, 1982.

[20] L. Verlet, Computer Experiments on Classical Fluids I. Thermodynamical Properties of Lennard-Jones Molecules, Phys. Rev., Vol. 159, pp. 98–103, 1967.

[21] M. P. Allen and D. Tildesley, "Computer Simulation of Liquids", Claredon Press, Oxford, 1987.

[22] S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, J. Comput. Phys., Vol. 117, pp. 1–19, March 1995.