Engineering High Performance Legacy Codes as CORBA Components for Problem Solving Environments

M. Li *

Dept. of Electronic and Computer Engineering, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK

D. W. Walker, O. F. Rana and Y. Huang

Dept. of Computer Science, Cardiff University, P.O.Box 916, CF24 3XF, UK

P. T. Williams and R. C. Ward

Computational Sciences and Engineering Division, Oak Ridge National Laboratory, P.O.Box 2008, Oak Ridge, TN 37831-6359, USA

Abstract

This paper describes techniques used to leverage high performance legacy codes as CORBA components to a distributed problem solving environment. It first briefly introduces the software architecture adopted by the environment. Then it presents a CORBA oriented wrapper generator (COWG) which can be used to automatically wrap high performance legacy codes as CORBA components. Two legacy codes have been wrapped with COWG. One is an MPI-based molecular dynamic simulation (MDS) code, the other is a finite element based computational fluid dynamics (CFD) code for simulating incompressible Navier-Stokes flows. Performance comparisons between runs of the MDS CORBA component and the original MDS legacy code on a cluster of workstations and on a parallel computer are also presented. Wrapped as CORBA components, these legacy codes can be reused in a distributed computing environment. The first case shows that high performance can be maintained with the wrapped MDS component. The second case shows that a Web user can submit a task to the wrapped CFD component through a Web page without knowing the exact implementation of the component. In this way, a user's desktop computing environment can be extended to a high performance computing environment using a cluster of workstations or a parallel computer.

Key words: Problem Solving Environments, High Performance Legacy Codes, CORBA Components, Wrapper Generator, Parallel and Distributed 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 [5,13]. 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 [6,10,15,25]. Our CORBA [23] compliant component model guarantees that components written in different programming languages can interoperate with each other.

The remainder of the paper is structured as follows. Section 2 briefly introduces the software architecture of the PSE. Section 3 presents COWG, a CORBA Oriented Wrapper Generator which can be used to automatically wrap high performance legacy codes in C or Fortran as CORBA components for reuse in the PSE. Section 4 describes case study one in which a MDS code is wrapped as a CORBA component with COWG and performance comparisons between runs of the MDS CORBA component and the original legacy code on a cluster of workstations and on a parallel computer are also presented. Section 5 describes case study two, using COWG to wrap a CFD code as a CORBA component. A Web user can submit a task through a Web page to invoke the CFD component without knowing the exact implementation of the component. Section 6 presents some related work on leveraging legacy codes to a distributed computing environment. Section 7 concludes the paper and gives future work on the wrapper generator.

2 The Software Architecture of the PSE

There are two major parts in the PSE, 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. In the VCCE, a user can visually construct scientific applications by plugging together components, which can range in granularity from simple tasks, such as matrix manipulation, to complete application programs. Each component is a CORBA component and has an XML [24] interface de-

* Maozhen.Li@brunel.ac.uk

scribed using a common data model, as specified in [4]. A component may contain sequential code written in Java, Fortran, or C, or it may be a parallel code based on a SPMD implementation using MPI [29], or it may contain array-based parallelism using language extensions such as HPJava [22].

After an application is constructed, the VCCE generates a task graph described in XML which is later passed to the IRMS. The IRMS then makes use of a resource management system, such as Intrepid [28] or Globus [31], to run the application on distributed computing resources, subject to constraints defined by the user. Figure 1 shows the constitution of the PSE.

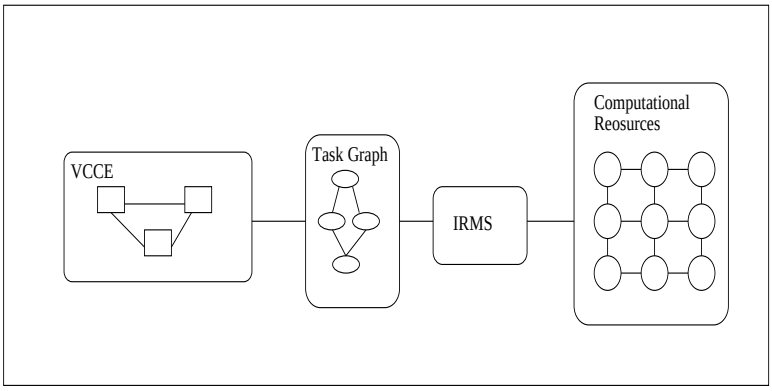


Fig. 1. The constitution of the PSE.

Walker [2] gives a detailed description on the software architecture of the PSE.

There are four common types of components. The main task of *compute components* is to do computational work such as performing a matrix multiplication or a fast Fourier transform. *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. For example, for a simple `for` loop the initial and final values of the loop control variable, and its increment, must be specified, together with a component that represents the body of the loop. For an `if-then-else` construct a Boolean expression, and a component for each branch of the conditional must be given. A template component can be viewed as a compute component in which one or more of the inputs is a component. *Data management components* are used to read and convert between the different types of data format that are held in files and Internet data repositories. The intention of this type of component is to be able to (1) sample data from various archives, especially if there is a large quantity of data to be handled, (2) convert between different data formats,(3) support specialized transports

between components if large quantities of data needs to be migrated across a network,(4) undertake data normalization, and perhaps, also generate SQL or similar queries to read data from structured databases. The fourth type of common component is the *user interface components* which allow a user to control an application through a graphical user interface, and plays a key role in computational steering.

Components can be created from scratch, or automatically wrapped from legacy codes through COWG, as described in section 3.3.

3 Engineering High Performance Legacy Codes as CORBA Components

High performance legacy codes are pre-existing codes, mostly in C or Fortran, that possess the following features:

- They are domain-specific.
- They are not reusable.
- They are still useful.
- They are large, complex monoliths.

One of the important research issue in the PSE is to exploit ways to leverage these legacy codes to a distributed computing environment and make them be pluggable and reusable CORBA components. These components can be easily assembled together to construct applications for solving domain problems.

3.1 The Strategies to Leverage Legacy Codes to a Distributed PSE

According to [27], there are three strategies for leveraging legacy codes to a distributed computing environment. One strategy is to start from scratch and redevelop all of the legacy codes with the distributed object concepts. This approach frees the developers from any consideration of the existing codes. But, every function must be re-implemented and tested in a new language and in a new environment, which is expensive and time consuming.

Another strategy is a re-engineering approach. Engineers convert the legacy codes to distributed objects appropriately. This approach is a promising method since it is not necessary to re-implement functions whose functionalities are the same as that of the legacy codes. However, code conversion is not easy. Few tools and methods are available.

The third strategy is to wrap legacy codes as distributed components and to

invoke them from a distributed computing environment. Wrapping is a method of encapsulation that provides clients with well-known interfaces for accessing wrapped components. The principal advantage is that behind the interface, the client need not know the exact implementation Wrapping can be accomplished at multiple levels: around data, individual modules, subsystems, or the entire system. After being wrapped as CORBA components, these legacy codes can be reused as components in a heterogeneous distributed computing environment. However, wrapping legacy codes manually is a time consuming and error-prone task. For example, to wrap a legacy code as a CORBA component, developers have to write all the interfaces needed such as an IDL interface, a CORBA implementation code for the legacy code. In addition, they also need to write interfaces for interactions between the component and other components. The interfaces and implementation codes are different for different legacy codes. Therefore, a wrapper generator is necessary and critical to reuse high performance legacy codes as CORBA components in a distributed computing environment. In addition, a component model is needed to manage the interactions and data flows between components wrapped from legacy codes.

3.2 A CORBA Compliant Component Model

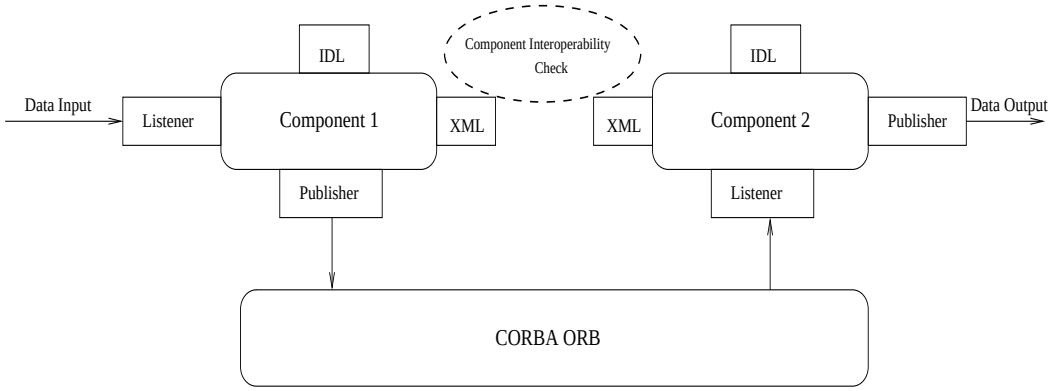


Fig. 2. A CORBA compliant component model.

A component model is a framework for assembling applications from components. The framework defines a set of rules that specify the precise execution environment provided to each component and the rules of behavior and special design features that components must have in order to be considered true components. A component model is the software environment that provides mechanisms to instantiate components, compose them, verify them, and use them to build applications. In this section, a CORBA based component model is described. The model defines the interaction and data flow between multiple components in the PSE.

A component in the 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 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 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 exe-

cutables can be subsequently located based on the references provided within the XML definition (see [4] 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.

The CORBA Component Model (CCM) [19] from CORBA 3 is used to standardize the component development cycle using CORBA as its middleware infrastructure. However, the CCM standard and implementations are as immature today as the underlying CORBA standard and ORBs were three to four years ago. Moreover, the CCM vendor community is largely focusing on the requirements of e-commerce, workflow, report generation, and other general-purpose business applications. The middleware requirements for these applications generally focus on functional interoperability, with little emphasis on assurance of or control over mission-critical QoS aspects. As a result, it is not feasible to use off-the-shelf CCM implementations for high-performance and realtime systems [20]. The CCM requires a significant number of new classes and interfaces to support its specified features. These requirements may cause problems for high-performance and real-time applications due to unnecessary time and space overhead incurred when components are collocated within the same process or machine. Our component model can have a better performance by directly binding the publisher of a component to the listener of another component. The publisher and listener are used as CORBA component ports as described in the CCM. However, they are supported by component methods inside of a component instead of using extra classes outside of a component.

3.3 A CORBA Oriented Wrapper Generator

In order to automatically wrap legacy codes as components, we have implemented COWG, a CORBA oriented wrapper generator [18] on Solaris 2.7 using VisiBroker [37] 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 COWG, 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 PSE framework. When using COWG, developers only need to specify the parameters (properties) of the legacy code they want to wrap, then submit the parameters to COWG

which then generates all the interfaces needed to convert the legacy code into a component. The interfaces include a CORBA IDL interface, an XML definition, an implementation code (Body), a Listener and a Publisher. The Listener and Publisher are used to interact with other components. Figure 3 shows the data flow in COWG.

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 input(s)/output(s). However, they do not need to know the exact implementation of the legacy code. The main constraints for a legacy code to be wrapped as a component with COWG 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 COWG to wrap a single legacy code, developers need to supply the following parameters:

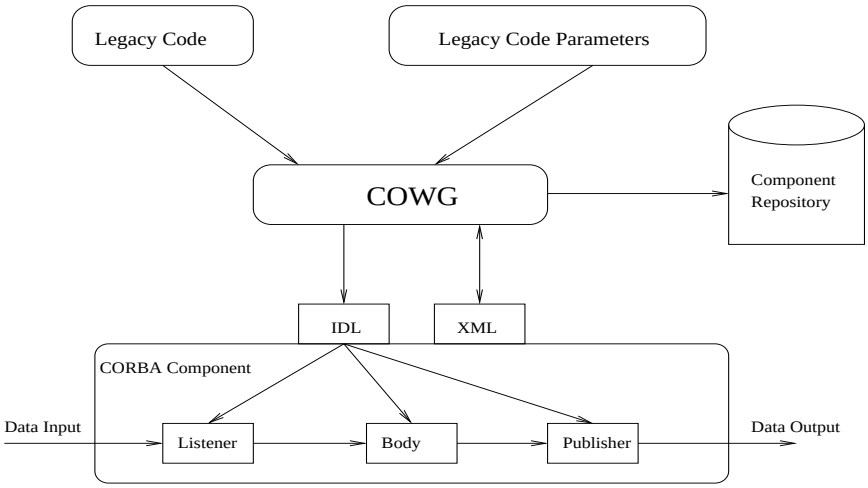


Fig. 3. The data flow in COWG.

- The Name of the component generated from the legacy code.
- The Name of the legacy code including the host name and directory in which the legacy code resides.
- The Name of the ORB Compiler. Since we use VisiBroker, the compiler is *idl2java*.
- The Language used by the legacy code. COWG currently supports legacy codes in C, Java, or Fortran.
- The Type of the legacy code. The legacy code could be a sequential or parallel code using message passing libraries such as MPI. A parallel CORBA component can be wrapped from a parallel legacy code using MPI. COWG makes use of an MPI run time to manage the intra-communications of multiprocessors within the parallel component.
- The preferred Processors used if the legacy code is a parallel code using

MPI to achieve high performance.

- The Number of input parameters used for the legacy code.
- The Type of each input parameter used by the legacy code. COWG 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, same to the Type of input parameter.
- The Data File used to store the data generated by the legacy code.

Input parameters are organized in an input file, and output parameters are put in an output file. Figure 4 gives the graphic user interface (GUI) of the wrapper generator.

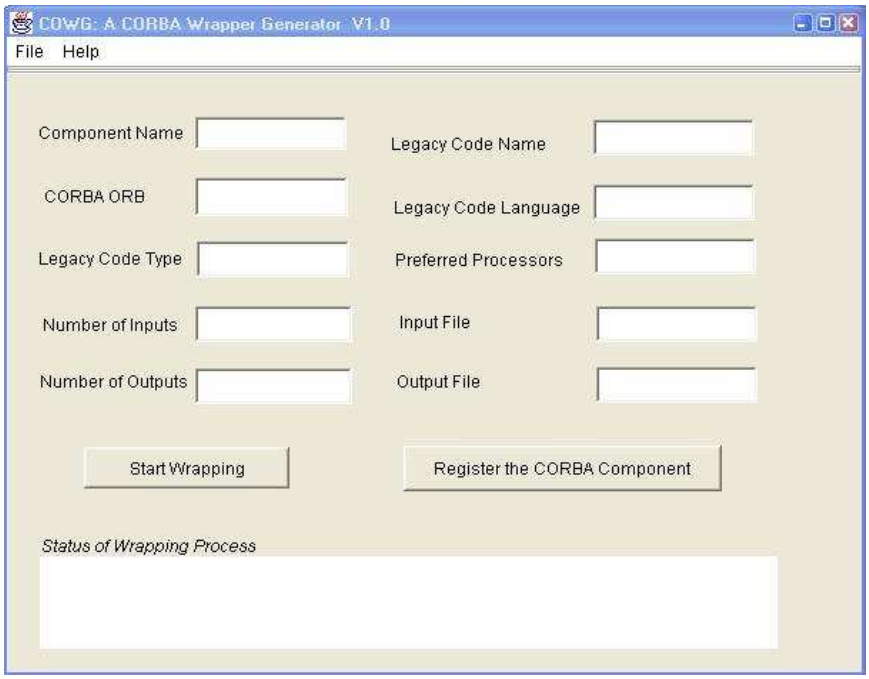


Fig. 4. The COWG GUI.

After receiving the parameters from a developer, COWG 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. COWG will then generate a CORBA IDL and an XML interface based on the input parameters. Using an IDL compiler, COWG generates stubs and skeleton for the Listener and Publisher of the component. Based on the specified parameters, COWG 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 COWG, 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, COWG stores the component in the component repository in XML for future use.

4 Case Study 1: A Molecular Dynamic Simulation (MDS) Application

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

4.1 The MDS Legacy Code

The legacy code used is a three-dimensional molecular dynamics code for simulating a Lennard-Jones fluid. The code has been parallelized, 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" [32] 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 [33] leap-frog scheme. Further details of the molecular dynamics algorithms can be found in the book *Computer Simulation of Liquids* [34].

A spatial decomposition [35] 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

```
<pse−component>
  <preface>
    <name alt="MDS" id="MDS">MDSComponent</name>
    <component−des>Molecular Dynamics</component−des>
    <component−directory>/home/Component</component−directory>
    <legacy−code>/home/MDS/moldyn</legacy−code>
    <ORB−Compiler>idl2java</ORB−Compiler>
    <language>C</language>
    <processors>8</processors>
  </preface>
  <port>
    <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/Component/output.dat" value="output"/>
    </outports>
  </ports>

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

  <help context="instantiate">
      <href name="file://home/pse/help/mds.txt" value="NIL"/>
  </help>
</pse−component>
```

Fig. 5. The XML description of the MDS legacy code.

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 MDS legacy code description in XML is given in figure 5, and provides an example of the 'Legacy Code Description' used in figure 3.

The legacy code called *"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, pressure, and temperature). The legacy code can run on a cluster of workstations or on a dedicated parallel machine. The IDL interface generated by COWG for the MDS legacy code is given in figure 6.

```
module MDS
{
    interface MDSComponent
    {
        void Listener(in string ComponentID, in string InputFile);
        void Body(string parameters);
        void Publisher(out string ComponentID, out string OutputFile);
        };
    };
```

Fig. 6. The IDL interface of the MDS CORBA component.

The UI component provides a graphical front end to receive inputs from a user and then sends a call to the MDS component through its Publisher. The Listener of the MDS 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 MDS component which in turn starts to execute the MDS 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 MDS CORBA component generated by COWG is briefly described in figure 7.

4.2 Performance Comparisons

It is important to minimize performance overheads when using wrapped legacy codes as components for use in the PSE. In order to measure and compare performance between the wrapped MDS CORBA component and the legacy code itself we carried out a number of experiments running the MDS CORBA component and the legacy code on a cluster of workstations and on a dedicated

```
class MDSComponent extends _MDSComponentImplBase
{
    public void Listener(String ComponentID, String inputs)
    {
        ......
        if (ComponentID=="MDS")
            read inputs from an InputUIComponent;
        invoke the body of the MDSComponent;
        }
    public void Body(String inputs)
    {
        ......
        execute "mpirun –np 8 inputs output.dat";
        invoke the Publisher of the MDSComponent with output.dat;
        }
     public void Publisher(String ComponentID, String outputs)
     {
        ......
        ComponentID="OutputUIComponent";
        invoke the OutputUIComponent with outputs;
        }
    }
```

Fig. 7. The implementation segment of the MDS CORBA component.

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 8. 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 components 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 MDS code wrapped as a single CORBA component, versus the performance

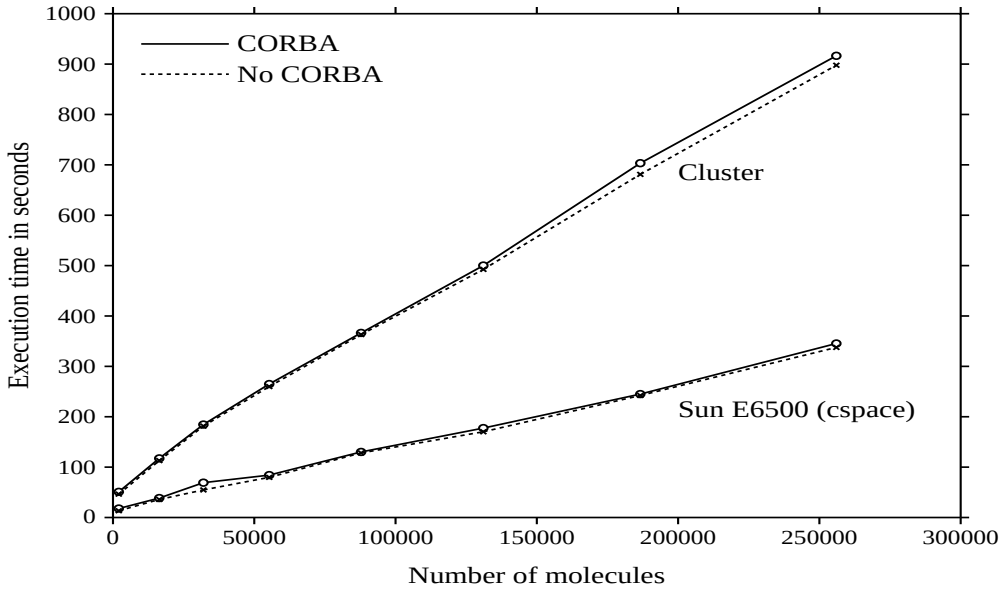


Fig. 8. Performance comparisons running the wrapped MDS CORBA component and the MDS legacy code itself on a cluster of workstations and on a dedicated parallel machine.

of the MDS code divided into a collection of co-operating CORBA components [1]. Making use of CORBA to manage the inter-communications among components and a MPI runtime to manage the intra-communications within a parallel object is a feasible approach for distributed parallel computing.

5 Case Study 2: A Computational Fluid Dynamics (CFD) Application

The CFD code called PHI3D [3] written in Fortran is a finite element based computational fluid dynamics code for simulating incompressible Navier-Stokes flows, and is being used to model flow in the lung and upper respiratory system. The theoretical basis for the new continuity constraint method consists of a finite-element spatial semi-discretization of a Galerkin weak statement, equal-order interpolation for all state-variables, a q-implicit time-integration scheme, and a quasi-Newton iterative procedure extended by a Taylor Weak Statement (TWS) formulation for dispersion error control and stabilization.Using a finite-element methodology, complex geometries can be easily simulated with PHI3D using unstructured grids. A time-accurate integration scheme allows the simulation of both transient and steady-state flows. Verification and validation studies have been completed for 3-dimensional laminar problem classes including heavily separated and buoyancy-driven flow fields. Available finite-elements include linear 8-node hexahedra, 6-node prisms, and 4-node tetrahedra, allowing the application of structured, unstructured, and hybrid mesh

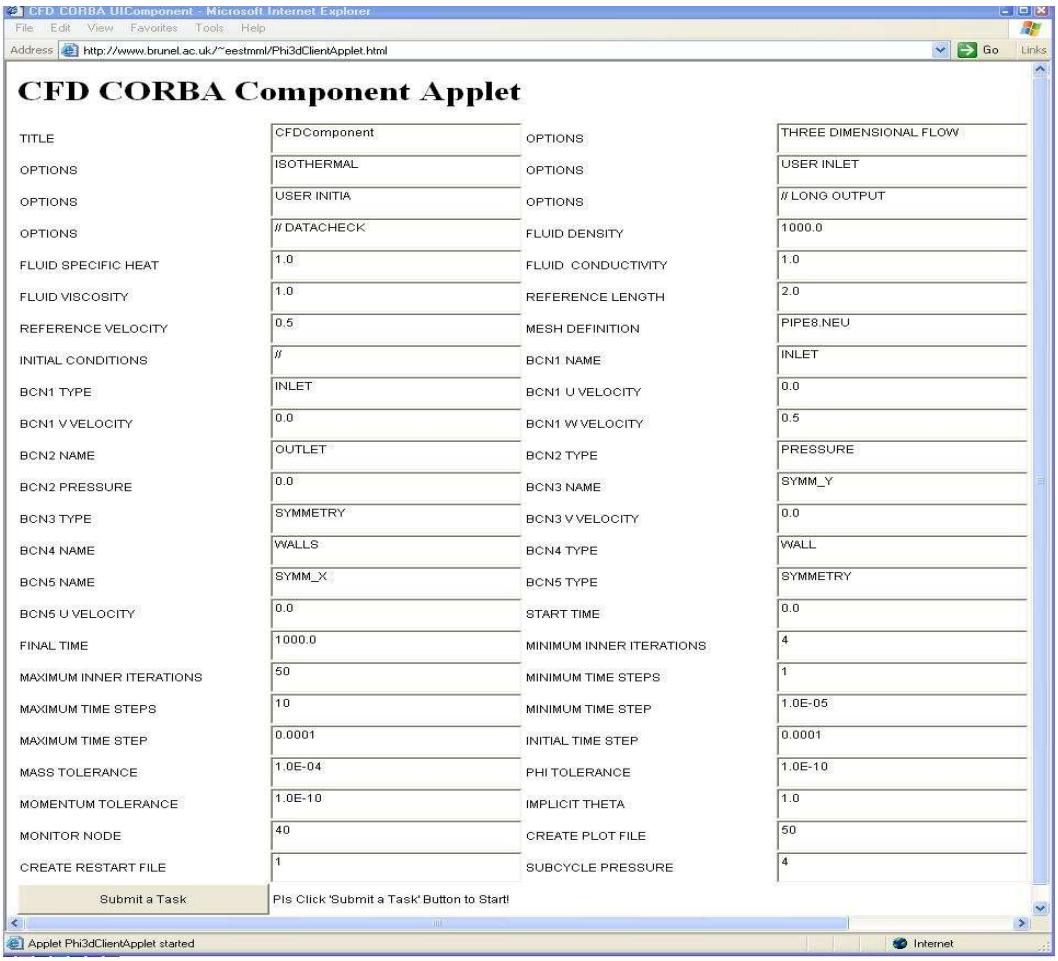


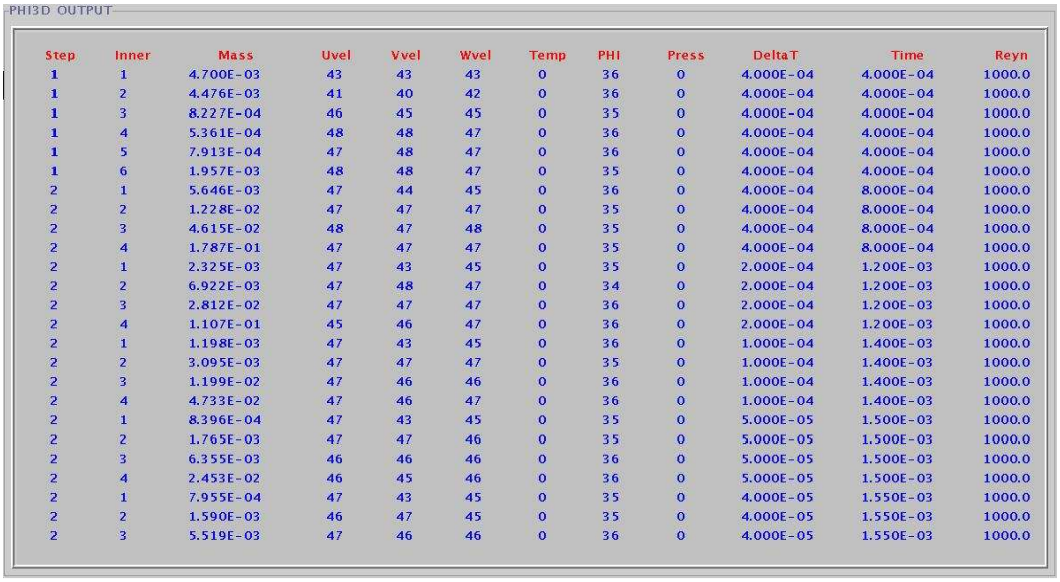
Fig. 9. A task submission Web page for the CFD CORBA component.

configurations to simulate complex geometries.

The CFD code has been wrapped as another CORBA component with COWG and can be invoked in a Web based computing environment. The wrapped CFD component can be assembled with different UI components in the PSE to provide different views for the output. Figure 9 shows the Web page by which a Web user submits a task to the CFD component. Figure 10 shows the output of the CFD component invoked by the Web user. In this way, users can share the legacy code to request services in a distributed computing environment.

The legacy code "PHI3D.x" is a Fortran code making use of 1 processor. It has one input and twelve outputs (steps, inner steps, Mass, Uvel, Vvel, Wvel, Temp, PHI, Press, DeltaT, Time, Reyn). The IDL interface generated by COWG for the CFD legacy code is given in figure 11.

The UI component is an Applet embedded within a Web page provides a


PHI3D OUTPUT

Step	Inner	Mass	Uvel	Vvel	Wvel	Temp	PHI	Press	DeltaT	Time	Reyn
1	1	4.700E-03	43	43	43	0	36	0	4.000E-04	4.000E-04	1000.0
1	2	4.476E-03	41	40	42	0	36	0	4.000E-04	4.000E-04	1000.0
1	3	8.227E-04	46	45	45	0	35	0	4.000E-04	4.000E-04	1000.0
1	4	5.361E-04	48	48	47	0	36	0	4.000E-04	4.000E-04	1000.0
1	5	7.913E-04	47	48	47	0	36	0	4.000E-04	4.000E-04	1000.0
1	6	1.957E-03	48	48	47	0	35	0	4.000E-04	4.000E-04	1000.0
2	1	5.646E-03	47	44	45	0	36	0	4.000E-04	8.000E-04	1000.0
2	2	1.228E-02	47	47	47	0	35	0	4.000E-04	8.000E-04	1000.0
2	3	4.615E-02	48	47	48	0	35	0	4.000E-04	8.000E-04	1000.0
2	4	1.787E-01	47	47	47	0	35	0	4.000E-04	8.000E-04	1000.0
2	1	2.325E-03	47	43	45	0	35	0	2.000E-04	1.200E-03	1000.0
2	2	6.922E-03	47	48	47	0	34	0	2.000E-04	1.200E-03	1000.0
2	3	2.812E-02	47	47	47	0	36	0	2.000E-04	1.200E-03	1000.0
2	4	1.107E-01	45	46	47	0	36	0	2.000E-04	1.200E-03	1000.0
2	1	1.198E-03	47	43	45	0	36	0	1.000E-04	1.400E-03	1000.0
2	2	3.095E-03	47	47	47	0	35	0	1.000E-04	1.400E-03	1000.0
2	3	1.199E-02	47	46	46	0	36	0	1.000E-04	1.400E-03	1000.0
2	4	4.733E-02	47	46	47	0	36	0	1.000E-04	1.400E-03	1000.0
2	1	8.396E-04	47	43	45	0	35	0	5.000E-05	1.500E-03	1000.0
2	2	1.765E-03	47	47	46	0	35	0	5.000E-05	1.500E-03	1000.0
2	3	6.355E-03	46	46	46	0	36	0	5.000E-05	1.500E-03	1000.0
2	4	2.453E-02	46	45	46	0	36	0	5.000E-05	1.500E-03	1000.0
2	1	7.955E-04	47	43	45	0	35	0	4.000E-05	1.550E-03	1000.0
2	2	1.590E-03	46	47	45	0	35	0	4.000E-05	1.550E-03	1000.0
2	3	5.519E-03	47	46	46	0	36	0	4.000E-05	1.550E-03	1000.0

Fig. 10. A snapshot of the CFD CORBA component output on the Web.

graphical front end to receive inputs from a user and then sends a call to the CFD component through its Publisher. The Listener of the CFD 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 CFD component which in turn starts to execute the CFD legacy code using a command such as "PHI3D.x". 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 CFD CORBA component generated by COWG is briefly described in figure 12.

6 Related Work

There is some prior work that addresses issues for generating wrappers with semi-automatic generation or meta-wrapper style to leverage legacy codes to a distributed environment. Vidal [8] suggests wrappers and mediators to access data from heterogeneous database or legacy servers. Ashish [14] suggests that information mediator for obtaining information from multiple Web sources. Sounder [9] provides wrappers for securely integrating legacy systems into a distributed environment. However, these works are about how to migrate legacy information systems to a distributed environment mainly on obtaining information from multiple data sources. A wrapper is provided for each data source. COWG is about automatically leveraging high performance legacy codes making use of parallel libraries such as MPI as CORBA components to

a parallel and distributed computing environment.

```
module CFD
{
    interface CFDComponent
    {
        void Listener(in string ComponentID, in string InputFile);
        void Body(string parameters);
        void Publisher(out string ComponentID, out string OutputFile);
        };
    };
```

Fig. 11. The IDL interface of the CFD CORBA component.

Kim [16] provides a wrapping technique that enables various legacy systems to be reused on CORBA based distributed environments without any changes to them. An automatic wrapper generation method based on extensible wrapping template classes is presented for wrapping legacy codes. The legacy codes in the work are sequential codes. COWG differs from Kim's work in two ways. First, the legacy codes in COWG can be sequential codes or parallel codes using MPI. Second, COWG is a software tool, not just a template class. Therefore, users do not need to write any codes to use COWG. They only need to specify the parameters related to a legacy code when wrapping the legacy code as a CORBA component with COWG.

SWIG [26] is a software tool that provides an interface compiler that connects high legacy codes written in C, C++, and Objective-C with scripting languages such as Perl, Python, and Tcl/Tk. Legacy codes can be sequential codes or parallel codes using MPI. Whereas SWIG focuses on manipulating legacy codes through the use of scripting languages, COWG is used to automatically wrap legacy codes as CORBA components (IDLs, component implementations, component dataflow) which can then be plugged together to create applications for reuse in a distributed problem solving environment.

COWG also provides a method to extend CORBA for parallel computing. 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. When wrapping an MPI based high performance legacy code as a parallel CORBA component, COWG makes use of an MPI runtime to manage the intra-communication of multiple processors within the parallel component. The inter-communication among different components is managed by CORBA. The advantage is that users can use existing CORBA implementations (such as Visibroker, Orba-

```
class CFDComponent extends _CFDComponentImplBase
{
    public void Listener(String ComponentID, String inputs)
    {
        ......
        if (ComponentID=="CFD")
            read inputs from an InputUIComponent;
        invoke the body of the CFDComponent;
        }
    public void Body(String inputs)
    {
        ......
        execute "PHI3D.x inputs output.dat";
        invoke the Publisher of the CFDComponent with output.dat;
        }
     public void Publisher(String ComponentID, String outputs)
     {
        ......
        ComponentID="OutputUIComponent";
        invoke the OutputUIComponent with outputs;
        }
    }
```

Fig. 12. The implementation segment of the CFD CORBA component.

cus [38] and others) without any modification to CORBA IDL compilers, as is done in other projects with a similar objective, such as PARDIS [21] and Cobra [17].

There are also some projects on parallel CORBA objects without any modifications to a CORBA IDL compiler [11,12]. However, they use a multi-threaded mechanism to provide a CORBA Group Communication Service through which to implement parallel CORBA objects. Our work differs from the approach in that we combine CORBA and MPI to provide parallel CORBA objects without modifications to a CORBA IDL compiler. In addition, the point-to-point communication between two processors in MPI is more efficient than that of using the multi-threaded mechanism in which the communication is done through an intermediate node.

7 Conclusions and Future Work

High performance legacy codes mostly written in C or Fortran making use of message passing paradigms such as MPI are very useful computational

resources. COWG has been used to automatically wrap such legacy codes as CORBA components for use in the PSE, a distributed component-based problem-solving environment. In general, a legacy code can be a whole application or a subroutine provided that they meet the requirements of constrains for wrapping a legacy code as a component with COWG. Since components in the PSE are CORBA component, 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 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 [30].

COWG provides a feasible way to automatically wrap high performance legacy codes as CORBA components, however, it is rather simple and there is still work that needs to be done to improve it.

- The wrapper generator currently uses a shared file system in an intranet environment to locate a legacy code. We plan to extend the wrapper generator to the Internet environment in which a legacy code with a host name on the Internet, or a legacy code in a virtual library such as Netlib [7] can be wrapped as a CORBA component and registered with a component repository in the PSE. The component repository itself works like Netlib, but the Visual Component Composition Environment (VCCE) in the PSE provides an integrated environment to access the CORBA components in the repository.
- Web Services [36] are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Since Web Services are simple and based on standard Web technologies, they have the advantage of widespread academic and industrial support, which implies greater uptake, ease-of-use, and true ubiquity. The work is undergoing to migrate COWG to a Web Services oriented wrapper generator in which the XML description of a legacy code will be used to generate a WSDL interface, the wrapped CORBA code will be used as the core part of a Web Services object, a UDDI repository will be used to manage the registration and discovery of components in the PSE.
- The Grid [39] appears to be a promising computing platform for solving large-scale resource intensive problems. The Grid couples a wide variety of geographically distributed resources such as PCs, workstations and clusters, storage systems, data sources, databases and special purpose scientific instruments and presents them as a unified integrated resource. The PSE can be a part of a Grid environment running on top of the Grid to provide high level services. The CORBA CoG Kit [40] combines CORBA with the Grid

to make Grid services accessible through the CORBA programming interface. By wrapping Globus Grid services such as MDS, GRAM, GASS, and GSI as CORBA objects, the CORBA CoG provides CORBA users the ability to transparently access these back end Grid services from their CORBA applications. The CORBA CoG can be integrated with the PSE in which an application built from CORBA components wrapped from the wrapper generator can use the CORBA CoG to access Grid services.

References

[1] Maozhen Li, Omer F. Rana and David W. Walker, Wrapping MPI-Based Legacy Codes as Java/CORBA Components, Future Generation Computer Systems(FGCS), Vol.18, Issue 2, pp. 213-223, October 2001.

[2] D. W. Walker, M. Li, O. F. Rana, M. S. Shields and Y.Huang, The Software Architecture of a Distributed Problem Solving Environment, Concurrency:Practice and Experience, Vol.12, No.15, pp.1455-1480, December 2000.

[3] P. T. Williams and A. J. Baker, Incompressible Computational Fluid Dynamics andthe Continuity Constraint Method for the 3D Navier-Stokes Equations, Numerical Heat Transfer, Part B Fundamentals 29, PP. 137-273, 1996.

[4] 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. in "Proc. of Generative Component Based Software Engineering", Erfert, Germany, September 1999.

[5] 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.

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

[7] http://www.netlib.org

[8] M.E.Vidal, L.Raschid and J.R.Gruser, A Meta-Wrapper for Scaling up to Multiple Autonomous Distributed Information Sources, in "Proc. of 3rd Int'l Conf.on Cooperative Information Systems", pp. 148-157.

[9] T. Souder and S. Mancoridis, A Tool for Securely Integrating Legacy Systems into a Distributed Environment, in "Proc. of sixth Working Conf. on Reverse Engineerin", pp.47-55, 1999.

[10] 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.

[11] Richard Cruceau, Felicia Ionescu, Doina Profeta, Parallel Programming with CORBA Group Communication Service, in Proc. of Int. Workshop on Trend and Recent Achievements in Information Technology, May 2002, Cluj Napoca, Romania.

[12] Markus Aleksy, and Martin Schader, A CORBA-Based Object Group Service and a Join Service Providing a Transparent Solution for Parallel Programming, in Proc. of International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000), June 2000, Limerick, Ireland.

[13] 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.

[14] N.Ashish and C.A.Knoblock, Semi-automatic Wrapper Generation for Internet Information Sources, in "Proc. of 2nd Int'l Conf. on Cooperative Information System", pp.160-169, 1997.

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

[16] H. S. Kim and J. Bieman, Migrating legacy systems to CORBA based distributed environments through an automatic wrapper generation technique. in "Proc. Joint meeting of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000) and the 6th International Conference on Information Systems Analysis and Synthesis", ISAS'2000.

[17] T. Priol and C. Ren, Cobra: A CORBA-compliant Programming Environment for High-Performance Computing. in "Proc. of Euro-Par'98", pp. 1114-1122, 1998.

[18] M.Li, O.F.Rana, M.S.Shield and D.Walker, A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components, in "Proc. of the ACM/IEEE SuperComputing'00", Dellas, USA, Nov. 2000.

[19] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan, An Overview of the CORBA Component Model, in Component-Based Software Engineering (G. Heineman and B. Councill, eds.), Reading, Massachusetts: Addison-Wesley, 2000.

[20] N. Wang, D. C. Schmidt, and D. Levine, Optimizing the CORBA Component Model for High-performance and Real-time Applications, in 'Work-in-Progress' session at the Middleware 2000 Conference, ACM/IFIP, Apr. 2000.

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

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

[23] http://www.omg.org.

[24] http://www.w3.org/XML.

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

[26] David M. Beazley and Peter S. Lomdahl, Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations. in "Proc. of SuperComputing'96", 1996.

[27] H.M.Sneed and R.Majnar, A Case Study in Software Wrapping, in "Proc. of ICSM'98", pp.86-93, 1998.

[28] 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.

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

[30] Douglas C. Schmidt, David L. Levine, and Chris Cleeland, Computer Communications, vol. 21, pp. 294-324, April 1998.

[31] I. Foster and C. Kesselman, The Globus Project: A Status Report, in "Proc. of IPPS/SPDP'98 Heterogeneous Computing Worksho", pp4-18, 1998.

[32] 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.

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

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

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

[36] Luis F. G. Sarmenta, Bayanihan Computing .NET: Grid Computing with XML Web Services, in "Proc. of CCGrid 2002", Berlin, Germany, May 2002.

[37] http://www.borland.com/visibroker

[38] http://www.ooc.com/ob

[39] I. Foster and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc., San Francisco, USA, 1998.

[40] Snigdha Verma, Manish Parashar, Jarek Gawor and Gregor von Laszewski, Design and Implementation of a CORBA Commodity Grid Kit, in "Proc. of GRID2001", Denver, Colorado, USA, November 2001.