Wrapping MPI-Based Legacy Codes as Java/CORBA components

Maozhen Li, Omer F. Rana and David W. Walker

Department of Computer Science
Cardiff University
P.O.Box 916, CF24 3XF
Cardiff, UK
Email: Li.Maozhen@cs.cf.ac.uk
Telephone: +44 2920 876925
Fax: +44 2920 874598

Abstract

Techniques for wrapping an MPI-based molecular dynamics (MD) simulation code as Java/CORBA components, for use within a distributed component based Problem Solving Environment (CB-PSE), is presented. A legacy code for simulating a Lennard-Jones fluid is first wrapped as a single CORBA object, followed by division of the code into computational sub-units, where each sub-unit is wrapped as a CORBA object containing MPI calls, and run on a cluster of workstations – enabling different MPI implementations to inter-operate. Using a Java implementation, users can submit simulation tasks through a Web based interface, without needing to know implementation details of the legacy code, or the exact interaction between sub-units within the code. We provide performance comparisons of wrapping the entire MD code as a single object, verses wrapping sub-units within it, and offer a simple performance model to explain our findings.

Wrapping MPI-Based Legacy Codes as Java/CORBA components*

Maozhen Li, Omer F. Rana and David W. Walker
Department of Computer Science,
Cardiff University, POBox 916,
Cardiff CF24 3XF, UK
Li.Maozhen@cs.cf.ac.uk

Abstract

Techniques for wrapping an MPI-based molecular dynamics (MD) simulation code as Java/CORBA components, for use within a distributed component based Problem Solving Environment (CB-PSE), is presented. A legacy code for simulating a Lennard-Jones fluid is first wrapped as a single CORBA object, followed by division of the code into computational sub-units, where each sub-unit is wrapped as a CORBA object containing MPI calls, and run on a cluster of workstations – enabling different MPI implementations to inter-operate. Using a Java implementation, users can submit simulation tasks through a Web based interface, without needing to know implementation details of the legacy code, or the exact interaction between sub-units within the code. We provide performance comparisons of wrapping the entire MD code as a single object, verses wrapping sub-units within it, and offer a simple performance model to explain our findings.

Keywords: PSE, Java/CORBA, wrapping legacy applications, distributed parallel computing

1 Introduction

A Problem Solving Environment (PSE)[1, 3, 5, 12]is a complete, integrated computing environment for composing, compiling, and running applications in a specific domain. A PSE may also incorporate many features of an expert system, and provide extensive assistance to users in formulating problems, running the problem on an appropriate platform, and viewing and analyzing results[2]. In addition, a PSE may have access to virtual libraries, sophisticated execution control systems, and visualization environments[6]. The main motivation for developing PSEs is that they provide software tools and expert assistance to the computational scientist in a user-friendly environment, allowing more rapid prototyping of ideas and leading to a higher research productivity[7, 9]. By relieving the scientist of the burdens associated with the inessential and often arcane details of specific hardware and software systems the PSE leaves the scientist free to concentrate on the science.

A PSE may be domain specific, with specialised support for particular types of applications such as molecular dynamics or finite elements, or it may be more general and provide a more common set of functions, such as in Matlab and Mathematica. A generic component based distributed PSE (CB-PSE) is described here, which provides infrastructure which is problem independent, but must support knowledge and libraries which are problem specific. Using CB-PSE, a user can visually construct a scientific application by connecting together software components, some or all of which may have been developed independently. The components may be sequential codes written in Java, Fortran, or C, or they may be parallel codes that make use of message passing libraries such as MPI, or they may exploit array-based parallelism through language extensions such as HPJava [4]. Complete legacy codes, in Fortran or C for instance, can also be wrapped as components for use in the CB-PSE.

*This project is funded by EPSRC grant GR/M17327

Most high performance applications composed of many distributed, heterogeneous components[13] have previously been developed in an ad hoc fashion. Developers of such applications explicitly make use of low level message passing calls, based on MPI or PVM libraries. Such software is generally fine tuned for a particular type of platform, with virtually no emphasis on long term cross-application integration or maintainability. Hence, although such communication libraries were developed for parallel programming, they do not offer the necessary support for designing components which can be re-used within other applications. Moreover, a component interface is difficult to specify using either PVM or MPI, and even if a specification of a component interface is available, it may conflict with other components which have been developed in other contexts.

Solutions already exist to decrease the design complexity of such applications – distributed object-oriented technology being one of these solutions. A complex application can be seen as a collection of objects, which represent components running on different machines and interacting through remote method invocations. Emerging standards such as CORBA are aimed at the design of applications using independent components as CORBA objects, which are independent of programming languages, location and platform. CORBA provides inter-operability between different components by specifying their interfaces in a meta-language, the CORBA Interface Definition Language(IDL), which is translated into the language of interacting components by a compiler. In addition, due to its platform-independent execution model, CORBA provides support for networking, multi-threading and mobile code. Java has become a language of choice for applications executing in heterogeneous environments utilizing distributed objects, and an ideal companion to CORBA. The ability to re-use existing code while making use of newer technologies like Java, makes CORBA a good choice for developers interested in extending existing applications, especially high performance codes containing low level message passing libraries such as MPI or PVM to an Intranet or the Internet.

In the CB-PSE, we use Java and CORBA to wrap high performance legacy codes as components. Each component is a CORBA object with a well-defined interface. Wrapping applications is a method of encapsulation that provides clients with pre-defined interfaces for accessing server applications, or components. The principal advantage is that, behind the interface, the client need not know the exact implementation of a component. Wrapping can be accomplished at multiple levels of granularity: around data, individual modules, sub-systems, or the entire system. The SWIG [8] project provides an interface compiler that connects legacy codes written in C, C++, and Objective-C with scripting languages including Perl, Python, and Tcl/Tk. It works by taking the declarations commonly found in C/C++ header files and using them to generate the glue code (wrappers) that scripting languages need to access the underlying C/C++ code. However, SWIG provides a proprietary solution, requiring the users to download and install additional software. Our approach is based on an industry standard specification and based on CORBA IDL, hence, we can cater for a wider variety of operating environments and languages. Although SWIG is a useful alternative, we believe that our emphasis on code re-usability is better addressed by the use of CORBA IDL. In this paper, we first focus on wrapping an MPI-based molecular dynamics (MD) simulation code as one CORBA object, and then as different CORBA objects in order to improve re-usability. Performance in these two situations is compared, and a simple performance model is presented. With a wrapped component, it is also possible to keep the interface static, and dynamically vary component instances based on where the component is to be run. To the client, the component appears to be platform independent, and interface definitions do not need to be modified when a new machine is added to the host pool, if the component is modified, or if a task is migrated.

The remainder of this paper is organized as follows: section 2 presents an overview of research work on extensions to the CORBA architecture, to facilitate parallel computing, and section 3 gives a software architecture of the CB-PSE. Section 4 describes the wrapping of a legacy code as one CORBA object, and subsequently as different CORBA objects. Section 5 compares the performance of different levels of wrapping and Section 6 concludes the paper.

2 Previous Work on Parallel CORBA objects

CORBA was not originally intended to support parallelism within an object. However, some CORBA implementations provide support for multi-threading which enables a programmer to make more effective use of several simultaneous processors sharing a physical memory within a single computer, such as in the TAO ORB [11]. However, the sharing of a single physical memory does not allow a large number of processors, since it could create memory contention. There are very few projects aimed at supporting coarse-grain parallelism in CORBA. The PARDIS[14] environment and Cobra[15] being the most advanced projects in this direction, both of which extend the CORBA specification and add parallelism to CORBA objects based on a SPMD (Single Program Multiple Data) execution model. The SPMD model is now widely used for programming distributed memory parallel computers. PARDIS designers propose a new kind of object they call SPMD object and Cobra designers provide a kind of parallel CORBA objects. SPMD objects represent parallel applications which roughly adhere to the SPMD style of computation. To support data distribution among different threads associated with a SPMD object, PARDIS provides a generalization of the CORBA `sequence` called a `distributed sequence`. However, this new argument type requires a modification to the standard IDL compiler. In the Cobra system, a parallel object belongs to a collection of objects and its interface must be defined in a particular system, for instance:

```
interface[*]
 Test{...}
```

where the object `Test` is a parallel object and belongs to a collection of objects. `*` means that the number of objects belonging to the collection is not specified in the interface, and defines a polymorphic type.

The PARDIS system provides a mechanism to invoke operations on objects asynchronously, that is based on the concept of a `future` message. In the Cobra system, asynchronous invocations of services will be handled by an extension of the 'Asynchronous Method Invocation' (AMI) which is a core part of the CORBA messaging specification to appear in the next release(3.0) of CORBA.

Our approach is quite different from these two for adding parallelism to a CORBA object. In wrapping the MPI-based legacy code as one or more components (CORBA objects), we do not extend CORBA specification and modify any IDL compilers. Hence, any CORBA system may be employed for using our proposed technique, whereas with PARDIS and Cobra additional software must be downloaded and installed. We combine the MPI runtime with the CORBA environment and use MPI runtime to manage the intra-communications of components, and the CORBA ORB to manage inter-communications of components. Each object is composed of two parts: one is the wrapper which can be invoked by a client, the other is the executive unit of the object. In addition, when starting each CORBA object, we can put the parallel CORBA object to run on a less loaded cluster of workstations based on the resource available information collected by a resource management system (IRMS below).

3 The Software architecture of the CB-PSE

The main components of the CB-PSE are the Visual Program Composition Environment (VPCE), the Intelligent Resource Management System (IRMS), and CORBA Wrappers for legacy codes. Figure 1 illustrates the architecture of the CB-PSE. The functions of each component in the CB-PSE are given below.

3.1 VPCE

The VPCE enables a user to develop an application by combining a diverse range of components that may come from different vendors. In the VPCE, a user can visually construct a specific application by plugging together portable software components, which is represented as a dataflow

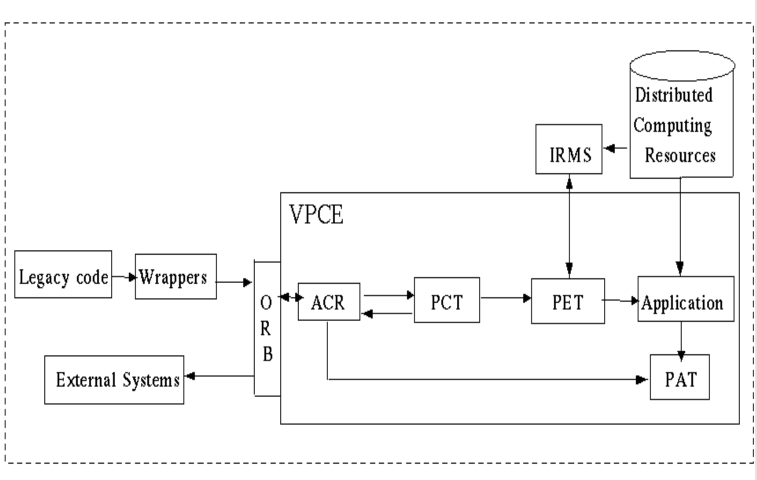

Figure 1: *The CB-PSE Architecture*

graph. The components may be sequential codes written in C, Fortran, or Java, or they may be parallel implementations that make use of MPI or PVM libraries. Legacy codes, in Fortran for instance, can be wrapped as CORBA objects. The VPCE is itself a CORBA object, and can communicate with other applications using the ORB interface, as illustrated.

The main component of the VPCE is the Program Composition Tool (PCT), which provides a visual tool that enables a user to build and edit applications by plugging together components, by inserting application components into pre-defined templates, or by replacing application components in existing programs. The PCT allows a user to connect two components only if their interfaces are compatible, this being defined as either being of the same data type, having the same internal data distribution, or being of identical cardinality. The PCT also enables a user to create new application components and place them in the Application Component Repository (ACR). The Program Execution Tool(PET) builds the task graph generated for every application defined using the PCT, and passes the graph to the IRMS. The VPCE also includes a Program Analysis Tool (PAT) for displaying the hierarchical structure of an application component, describing the purpose and interface of each of its components, as well as of itself. Each component has a well defined interface, and associated constraints on its I/O behaviour, identifying which components may be interconnected during composition. When creating a new application component or placing it in the ACR, the user is prompted to optionally supply this information, such as particular constraints on its execution environment, the number and types of input and output ports, etc. Components in the VPCE are CORBA-compliant objects, and may be implemented in Fortran, C or Java, and held at any location that can be described as a URL. Each component can be described by its type, interface, and constraints, and optionally also has a performance model and help file associated with it. The component interface is defined using CORBA IDL.

3.2 CB-Wrappers

CB-Wrappers in the CB-PSE enable programs written in programming languages other than Java to be converted into CORBA objects. The services provided by CB-Wrappers are the same as those implemented in the legacy code. Either the complete legacy application may be wrapped as one object, or the application may be divided into smaller objects, provided the structure of the application is known, and constraints on the interfaces of components of a sub-divided applications can be derived. The use of hierarchy in the VPCE facilitates this sub-division.

3.3 IRMS

The IRMS parses every task graph obtained from the PET, and extracts tasks for scheduling and allocation onto a virtual machine. Depending on the available resources, and a performance model of components, it assigns tasks to different computational resources. Through the IRMS, an application built in the VPCE may be scheduled onto single or multiple processor machines. The IRMS provides local scheduling and allocation for tasks obtained from PET, and negotiates with local scheduling systems such as LSF and Codine, where available, to build a possible schedule. The scheduler is therefore assumed to be obtained from a third party, provided the IRMS can track task execution on the platform onto which each task has been mapped.

4 Wrapping an MD Code as CORBA objects

The code used in the PSE 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 three-dimensional $N_b \times N_b \times N_b$ cells. The cell size is at least 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 neighboring cells. The symmetry of Newton's Third Law is exploited so that atoms in only 13 (instead of 26) neighboring 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" [16] 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 [17] leap-frog scheme. Further details of the molecular dynamics algorithms can be found in the book *Computer Simulation of Liquids* [18].

A spatial decomposition [19] 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 neighboring processes. This is necessary because these particles must interact with particles in neighboring 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.

4.1 Wrapping the MD code as one CORBA object

In the CB-PSE, a user can not only create new components from a pre-defined template, but may also re-use components wrapped from legacy codes. At first, we wrap the MD code as one CORBA object. A user constructs an application by putting together a front-end input component which is GUI-based, a wrapper for the executive MD unit and then the MD unit itself. The CB-Wrapper implementation is illustrated in Figure 2.

The entire legacy code in the server node is wrapped as one CORBA object. The CB-Wrapper is responsible for communication with the legacy code through a shared file, and provides services to the client. Since the operations in the CORBA object are pre-defined , a client invokes the wrapper with IDL stub on the client side and IDL skeleton on the server side.

When invoked from the Client, the MD CORBA object initiates two processes: one to start the MPI runtime environment, and then start the MD executive code on a cluster of distributed computational nodes, the other process to wait for the output of the MD code and generate a call

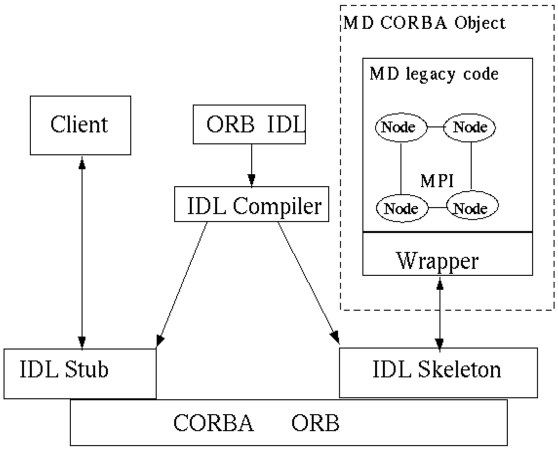

Figure 2: *Wrapping MD code into one CORBA object*

back to the Client to display the results. The operations defined through CORBA IDL are given in Code Segment 1.

From the IDL definitions in Code Segment 1, the service provided by the wrapper is `startSimulation()` which has two input parameters. One is a proxy to the client called `InvokeClient`, which is used by the server to invoke operations on the client. The method supported by the client proxy is `displaySimulation()` which has six parameters, and is used to display simulation results.

Hence, by embedding the MPI runtime into a CORBA environment, we can add parallelism to CORBA objects. Using a combination of CORBA and MPI, a user can submit a parallel simulation task and wait for the results without knowing about the exact implementation of the simulation.

```
module Simulation
{
    interface Client{
        void displaySimulation(in unsigned long a,in float f1,in float f2, in float f3,in float f4,in float f5);
    };
    interface Wrapper{
        void startSimulation(in Client obj, in string SimulationParameter);
    };
};
```

Code Segment 1: The MD CORBA object IDL

4.2 Wrapping the MD code as different CORBA objects

In order to improve the extent of software re-usability, we divide the MD code into different parts and wrap each part as a CORBA object. We have wrapped the MD code into four objects, an `Initialization` object, a `Moveout` object, a `Force` object and an `Output` object. Each object is a parallel CORBA object, and uses MPI. The function of each object is as follows:

- The `Initialization` object calculates the starting position(X, Y, Z) and velocities in each direction for every molecule used.
- The `Moveout` object performs the movement of molecules after each time step, including the communications in east, west, south, north, up and down directions. This object also handles the "ghost" regions around each process, for supporting molecule migration at each time step.

- The `Force` object calculates forces among molecules,
- The `Output` object generates the simulation results at each time step.

In addition, we add a `Controller` Object to perform the loop for the four objects during simulation. There is one CORBA IDL definition for each object, the IDL for Moveout object is given in Code Segment 2.

```
interface Moveout{
    typedef sequence <float, NMAX > rx ;
    typedef sequence < float, NMAX > ry;
    typedef sequence < float, NMAX > rz;
    typedef sequence < float, NMAX > vx;
    typedef sequence < float, NMAX > vy;
    typedef sequence < float, NMAX > vz;
    typedef sequence < long, NMAX > head;
    typedef sequence < long, NMAX > list;
    typedef sequence < long, NMAX > map;

    import-data(in float rx, in float ry, in float rz, in float vx, in float vy, in float vz, in long head, in long list, in long map);
    void start-moveout();
    export-data(out float rx, out float ry, out float out, out float vx, out float vy, out float vz, out long head, out long list, out long map);
    };
```

Code Segment 2: The Moveout object IDL

where `rx,ry,rz` correspond to the position of each molecule, `vx,vy,vz` the velocities in each direction, `head, list` represent two arrays used for molecule migrations, `map` represents the array used to store neighboring molecule positions for each molecule, `NMAX` corresponds to the maximum number of molecules used in the MD code, `import-data()` is a method to receive data from the `Controller` object, the `start-moveout()` method starts the executive legacy moveout code, and the `export-data()` method sends data to the `Controller` object. The software architecture when wrapping the MD code into different CORBA objects is illustrated in Figure 3.

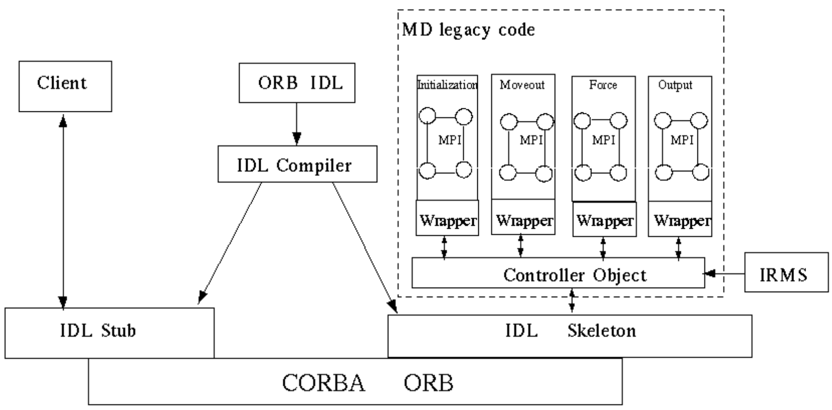

Figure 3: *Wrapping MD code into one CORBA object*

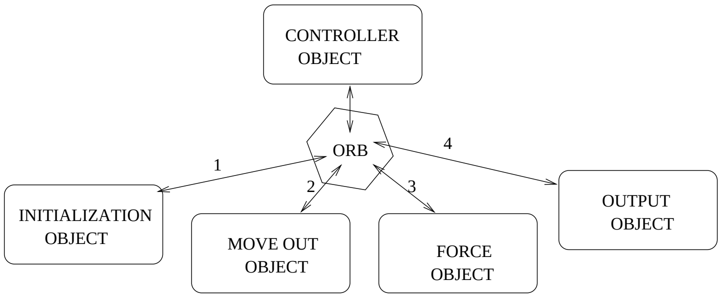

Figure 4: *Data flow between objects in the MD application*

Each object includes two parts, one is the wrapper, the other is the executive legacy code for each part of the MD application. When a user submits a simulation task, the `Client` first invokes the `Controller` object on the server side, followed by the `Controller` object performing the simulation loop. The `Controller` object invokes the `Initialization` object once, and invokes the other three objects many times. When there is output, the `Controller` object makes a call back to the `Client` to display simulation results to a user. The `Initialization`, `Moveout`, `Force` and `Output` objects are all MPI-based parallel CORBA objects, they can run on different clusters of workstations and they receive input data from the `Controller` object and send output to the `Controller` object. In addition, using functions from the IRMS, the `Controller` object can invoke each of the other four objects to run on less loaded computational nodes to improve performance. Using CORBA, different MPI implementations can also talk with each other, as each of the different objects contains its own MPI runtime.

All communications within objects are handled by the MPI runtime. Communication between objects is through the CORBA ORB, and managed by the `Controller` object, as illustrated in Figure 4. Each object must register itself with the `Controller` object on start-up, and can only communicate with it, hence all other data transfers are managed by the `Controller`.

5 Performance comparisons

In order to measure and compare performance, we carried out a number of experiments using the ORB in JDK1.2, an MPI implementation from Argonne National Laboratory, on the Solaris2.7 and WindowsNT environments. Each experiment involved measuring the response time, by changing the number of molecules in the fluid, while keeping the number of workstations in the cluster fixed to 8. The results are illustrated in figure 5.

It is important to minimise performance overheads when wrapping legacy codes for use in a distributed environment. The performance in wrapping the MD code as one CORBA object is quite different from that of wrapping the MD code as several different CORBA objects. If T represents the total time in each time step for a user to receive one simulation result, then in the first situation,

$$T = T_{cs} + T_{md} + T_{sc}, \tag{1}$$

where T_{cs} is the time to invoke the CORBA object from the client, T_{md} is the communication cost within the MD CORBA object, and T_{sc} is the invocation cost from the MD CORBA object to the client.

In the second situation,

$$T = T_{cs} + (T_i + T_m + T_f + T_o + T_{ic} + T_{mc} + T_{fc} + T_{oc}) + T_{sc}, \tag{2}$$

where T_i is the communication cost within the Initialization object, T_m is the communication cost within the `Moveout` object, T_f is the communication cost within the `Force` object, T_o is the communication cost within the `Output` object, T_{ic} is the communication cost between the `Initialization` and the `Controller` objects, T_{mc} is the communication cost between the `Moveout` and `Controller` objects, T_{fc} is the communication cost between the `Force` the `Controller` objects, and T_{oc} is the communication cost between the `Output` and the `Controller` objects.

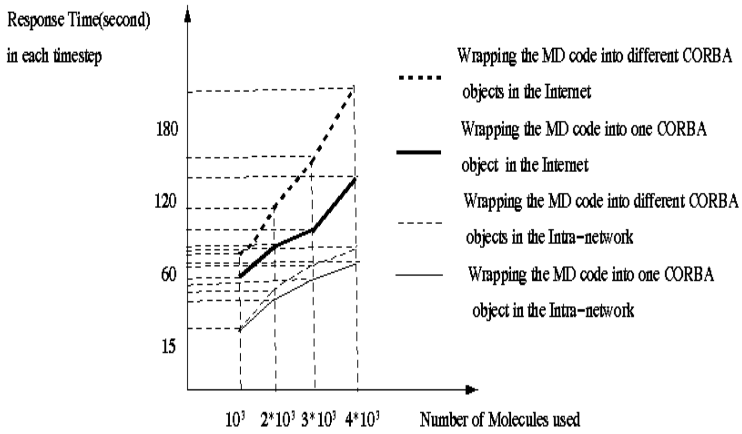

Figure 5: *Performance comparison between wrapping the MD code as a single CORBA object and several different CORBA objects*

We treat each of the four MPI-based CORBA objects as a daemon. There is some cost for each object to initialise the MPI runtime. After an object is running, the cost mainly arises from the communications within each object and between objects. If we run the client and the MPI-based CORBA objects on an Intranet and let them share the same file system, the costs in the two situations are almost the same, namely

$$T_{md} \approx T_i + T_m + T_f + T_o + T_{ic} + T_{mc} + T_{fc} + T_{oc}. \tag{3}$$

If we run them on the Internet, the cost of communication among different objects in the second situation is much higher than that of the first situation, especially when the number of molecules is large;

$$T_{md} < T_i + T_m + T_f + T_o + T_{ic} + T_{mc} + T_{fc} + T_{oc}. \tag{4}$$

The performance results in these two situations are illustrated in Figure 5, and are presented for comparison on the same graph. The top curve on the graph, shown with the broken line, corresponds to communications over a distributed file system, and suggests a 150% decrease in performance compared to a similar set up over a shared file system. We find a similar reduction in performance when wrapping the MD code as a single object. When wrapping the MD code as separate CORBA objects, the communication costs between objects are significant, and with a shared file system represent a performance degradation of 23% with respect to wrapping the entire MD code as one object. When using the Internet with a distributed file system, the degradation in performance when wrapping the MD code as multiple objects is 33%, compared to wrapping the entire MD code as one object. These results correspond to 4000 molecules in the fluid. Our results therefore show that communication delays between CORBA objects with large amounts of data are significant, and using CORBA for communication between processes, in place of the MPI

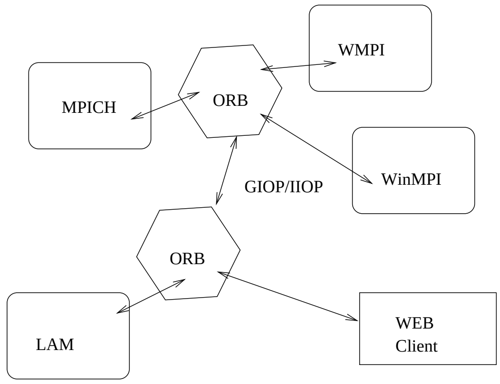

Figure 6: *Inter-operability between MPI implementations through CORBA*

runtime is not recommended. Using CORBA in place of MPI to achieve parallelism, with standard CORBA ORBs can have significant performance degradation, unless the CORBA ORB is modified to support multi-threading or has specialised support for parallel processes. However, what we do recommend is the use of MPI clusters connected via CORBA, which will also enable multiple MPI runtimes to work together, as illustrated in Figure 6. This is particularly important when integrating applications over the Internet, each of which may contain its own MPI implementation internally. In this way, we can re-use each part of the MD code and improve the extent of software re-usability.

6 Conclusion and Future Work

Wrapping pre-existing computational resources as reusable CORBA components in a heterogeneous distributed computing environment is playing an important role in the development of distributed PSEs. By embedding the MPI runtime into the CORBA environment, we have added parallelism to CORBA objects without modifying the CORBA IDL compiler. We have wrapped a legacy MD code into a single CORBA object, and as several different CORBA objects. The performance in these two situations is almost the same when the client and the MPI-based CORBA objects are on the the Intranet sharing file space, but quite different on the Internet. Further work needs to be done to manage data communication efficiently. In addition, we plan to use XML to define the components wrapped from legacy codes and store them as component templates in the CB-PSE.

References

[1] 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

[2] Vijay Menon and Anne E. Trefethen, "MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing", SuperComputing97, 1997.

[3] E. Gallopoulos, E. N. Houstis and J. R. Rice, "Workshop on Problem-Solving Environment:Findings and Recommendations", ACM Computing Surveys, Vol. 27, No. 2, pp 277-279, June 1995

[4] 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.

[5] 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.

[6] H. Casanova and J. J. Dongarra, "NetSolve: A Network-Enabled Server for Solving Computational Science Problems", Int. J. Supercomputing Appl. Vol. 11, No. 3, pp. 212-223, 1997.

[7] K. M. Decker and B. J. N. Wylie, “Software Tools for Scalable Multilevel Application Engineering”, Int. J. Supercomputing Appl. Vol. 11, No. 3, pp. 236-250, 1997.

[8] David M. Beazley and Peter S. Lomdahl, “Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations”, SuperComputing96.

[9] G. Spezzano, D. Talia, S. Di Gregorio, “A Parallel Cellular Tool for Interactive Modeling and Simulation”, IEEE Computational Science and Engineering, Vol. 3, No. 3, pp. 33-43, 1996.

[10] S. Weerawarana. Proceedings of the Second Annual Object-Oriented Numerics Conference, Sun river, Oregon, April 1994.

[11] Douglas C. Schmidt, “Evaluating Architectures for Multithreaded Object Request Brokers”, Communications of the ACM, Vol:41, No. 10, October 1998.

[12] R. Bramley and D. Gannon. See Web page on PSEWare at `http://www.extreame.indiana.edu/pseware`.

[13] Alan W. Brown, Kurt C. Wallnau, ” The Current State of CBSE”, IEEE Software Sep. 1998.

[14] K.Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In Supercomputing’97. ACM/IEEE, November 1997.

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

[16] 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. 13476–1370, 1982.

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

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

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