CODE COUPLING USING PARALLEL CORBA OBJECTS

Christophe René, Thierry Priol
IRISA/IFSIC
Rennes Cedex, France

Guillaume Alléon
Aérospatiale Matra Corporate Research Centre
Blagnac, France

Abstract This paper describes an approach for scientific code coupling using CORBA objects. Our approach is based on an extension of CORBA, called PaCO (Parallel CORBA Object), to support efficiently the encapsulation of parallel codes into distributed objects. With such extension, a parallel code can be seen as a collection of identical CORBA objects. Our extension to CORBA modifies only the Interface Definition Language (IDL) syntax by adding new language constructs. These new keywords allow the specification of several aspects associated with a collection of objects. We developed a new IDL compiler that generates stubs and skeletons to manage collections of objects transparently to the users. Parallel CORBA objects have been used within an industrial application from Aérospatiale Matra in the field of Electromagnetic simulation. The paper gives some performance results.

Keywords: CORBA, parallel CORBA object, coupled simulation, distributed numerical simulation.

1. INTRODUCTION

Scientific computing is an inescapable reality to design complex systems. By allowing virtual experiments, numerical simulation can speed-up the design phase and decrease its associated cost. It is thus an excellent approach to increase the competitiveness of the industry. For many years, the aerospace industry made an intensive use of numerical simulation. However, they are now facing an important challenge. As physical systems being more and more complex, their numerical sim-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22

ulation requires the combination of several simulation codes which are either developed in-house or purchased from software vendors. Each of these codes simulates a particular physical behavior (computation fluid dynamics, structural analysis, electromagnetic, ...). To reduce the cost of developing such simulation applications (also called *coupled applications*), aerospace companies try as much as possible to reuse, or to adapt, existing codes they already developed in previous projects. Therefore, designing a simulation application for a new industrial project is often seen as an integration of existing codes. Such integration consists in developing a framework that is in charge of calling the different simulation codes in a specific order and to let them to exchange their simulation results. An object oriented approach is thus well suited for the development of such a framework requiring that each of these simulation codes to be an object. Moreover, it is obvious to say that the execution of such numerical simulation application has to be performed in a reasonable time frame. However, since the application is made of several codes, simulation times is being increased. To keep it in a reasonable time frame, it is necessary to be able to exploit several computing resources which are available at either an intranet or at the Internet level. It is therefore mandatory to develop a coupled application in such a way that each code can be ran on a distinct computing resource. Moreover, the design of new complex systems requires the involvement of several industrial participants having their simulation tools running on their own computing resources. For confidentiality reasons, industrial actors are often reluctant to share their data or their simulation tools. For these two reasons (confidentiality and computing resources availability), it is thus necessary to perform the simulation in a distributed manner. Again, an object-oriented approach is able to fulfill these requirements. Indeed, the use of distributed objects allows transparent remote execution permitting the exploitation of geographically dispersed computing resources. However, distributed object technologies have to be adapted in the context of high-performance computing taking into consideration that a distributed object has to encapsulate a parallel code.

In this paper, we relate our experience that was aimed at coupling two instances of an electromagnetic simulation code to perform a distributed numerical simulation. The coupling is based on a extension of CORBA (Common Object Request Broker Architecture), called PaCO (Parallel CORBA Object). The remainder of this paper is structured as follows. Section 2 discusses some issues when coupling codes together. Section 3 gives a short overview of CORBA. Section 4 provides a description of the concept of parallel CORBA object. Section 5 describes the electromagnetic simulation application and some performance results. Finally, we conclude in section 6.

2. CODE COUPLING

The coupling of codes requires an underlying communication mechanism to allow the transfer of both data and control between the codes. The transfer of data corresponds to the sending of the results calculated by one of the code and the receiving by another code whereas the transfer of control is the execution of one particular function to be performed remotely in another code. Most of the attempts to couple codes together rely on the use of message-passing libraries such as MPI. However, MPI was mainly designed for parallel programming and not for distributed programming. Current MPI implementations are not interoperable so that it is not possible to exchange messages between different computing resources due to their heterogeneity. Research works have recently lead to extend existing message passing libraries, such as MPI, to be able to exchange data between heterogeneous computing resources. MPICH-G [2], PACX [1], PLUS [9] or MPI_Connect [5] are examples of such extensions. However, these extensions have some drawbacks. As for instance PLUS and PACX do not allow several processes of a parallel code to transfer data simultaneously (thus efficiently) to some other processes of a parallel code running on a remote machine. One node of each machine acts as a bridge to let processes of the two parallel codes to communicate together. Such design does not offer a scalable way to communicate between parallel machines. It thus represents a potential bottleneck when communicating between several parallel machines for which their compute nodes are connected to an Ethernet network (such as the IBM SP-2, NEC Cenju-4, and clusters of PCs or workstations). If two of these machines are connected using a multi-gigabit network, it will not be possible to exploit the whole capacity of this network since the communication rate will be bounded by the performance of the network that connect one compute node[1] to the multi-gigabit network[2].

Moreover, even if an implementation of MPI allows multiple flows of data to be sent simultaneously, we think that message-passing is not suitable to connect several parallel codes together. Indeed, message-passing was mainly designed for parallel programming and not for distributed programming. It means that it is mainly used to transfer data but not the control. As for instance, if one code would like to call a particular function into another code, this latter has to be modified in such a way that a message type is associated to this particular function. Such modification requires often a deep understanding of the code. Moreover,

```
interface MatrixOperations {
  const long SIZE = 100;
  typedef double Vector[ SIZE ];
  typedef double Matrix[ SIZE ][ SIZE ];

  void mult( in Matrix A, in Vector B,
             out Vector C );
  double skal( in Vector A );
};
```

Figure 1 Example of an IDL interface

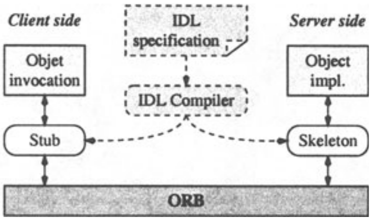

Figure 2 CORBA Architecture

entry points in a code are not really exposed to potential users that would like to include such code into their applications. When control has to be transfered between codes running on different machines, communication paradigms such as RPC or distributed objects, offer a much more attractive solution since the transfer of control is implemented by remote invocation that is as simple as calling a function or a method. We advocate an approach, like others [4], that consists in merging several communication paradigms in a coherent way. This approach is based on the use of two communication paradigms: distributed objects (CORBA) and message-passing (MPI).

3. A SHORT OVERVIEW OF CORBA

CORBA is a specification from the OMG (Object Management Group) [6] to support distributed object-oriented applications that are based on a client/server approach. An application based on CORBA can be seen as a set of independent software entities or CORBA objects. Each server object is associated with an interface that describes which operations can be remotely invoked by a client object. Object interface is specified using the Interface Definition Language (IDL) as shown in the example given in Figure 1. In this example, the interface contains two operations (*mult* and *skal*), each of them having some parameters whose types are similar to C++ ones. A keyword added just before the type specifies if the parameter is an input or an output parameter or both. IDL provides an interface inheritance so that services can be extended easily.

Figure 2 provides a simplified view of the CORBA architecture. When a client invokes an operation of a remote object, communication between the client and the server is performed through the Object Request Broker (ORB). The ORB offers a communication infrastructure independent of the underlying platform (machine and operating system). The client is connected to the ORB through a stub whereas it is done by a skeleton for the server. Stubs and skeletons are generated automatically by an IDL compiler taking as input the IDL specification of the object. Since CORBA hides the language used for the object implementation, an IDL compiler may generate stubs and skeletons for different languages (C++, Java, Smalltalk, ...). An object can thus be implemented in C++ and called by a client implemented in Java. A stub acts simply as a proxy object that behaves the same than the object implementation at the server side. Its role is to deliver requests to the server. Similarly, the skeleton is an object that accepts requests from the ORB and delivers them to the object implementation.

4. PARALLEL CORBA OBJECT

The adoption of CORBA for coupled simulation raises some difficulties mainly when dealing with the coupling of parallel codes. Indeed, the encapsulation of parallel codes into CORBA objects has been seen as the major obstacle for the use of such a technology. Therefore, the integration of MPI-based parallel codes to existing CORBA infrastructures is an important issue in many research and development projects. MPI-based parallel codes are mostly based on a SPMD execution model. With such a model, a parallel code is a set of identical processes running concurrently and exchanging data through the sending and the receiving of messages. The usual way of encapsulating such codes into CORBA objects is to adopt a master/slave approach as shown in figure 3. In that case, only one process, called the master, is encapsulated into a CORBA object. Other processes (the slaves) are connected to the master process through the MPI layer. The master may represent an important bottleneck when two MPI codes, encapsulated into CORBA objects, have to communicate with each other. Indeed, the master process has to gather data from the slave processes, using MPI, and has to send them to the callee through the ORB. The callee will then call the other CORBA object that in turn will scatter the data to its slave processes. This approach does not offer a scalable solution to the encapsulation of parallel codes. As the number of slave processes or the size of the problem (amount of data transmitted between two parallel codes) increases, it will entail a large overhead. To avoid such undesirable behavior, we advocate the use of a new kind of CORBA objects we call parallel CORBA objects (PaCO).

A parallel CORBA object is a collection of identical standard CORBA objects as shown in figure 4. Each CORBA object encapsulates a SPMD process of the parallel code. As our goal is to hide parallelism to the user, all the objects belonging to a collection are manipulated as a single

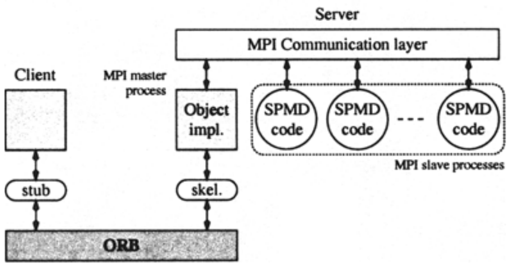

Figure 3 Master/slave approach

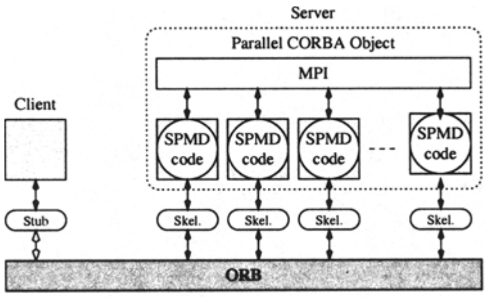

Figure 4 Parallel CORBA object

entity. In this way, a parallel CORBA object is seen as a standard object from the client point of view. Therefore, when a client invokes a remote operation to a parallel CORBA object, the associated method is executed concurrently by all objects belonging to the collection. Such parallel execution is performed under the control of the stub associated with the parallel CORBA object. Since the stub behaves differently from the one associated with a standard CORBA object, we modified the way an IDL compiler generates stubs. Such modifications were made possible by enriching the IDL language with new constructs. This new IDL language is called *Extended*-IDL. Extensions allow users to specify a collection of objects and to add data distribution attributes to operation parameters. The following paragraphs describe the *Extended*-IDL using the example shown in figure 5. All the extensions added to the IDL as well as some restrictions, like interface inheritance, are presented in more details in [7].

Mapping of objects. The number of objects in the collection, that will implement the parallel object, is specified within the two brackets after the IDL keyword `interface`. There are several ways to fix the number of objects in the collection. The expression may be an integer

```
interface[*] MatrixOperations {
  const long SIZE = 100;
  typedef double Vector[ SIZE ];
  typedef double Matrix[ SIZE ][ SIZE ];

  void mult ( in dist[ BLOCK ][ * ] Matrix A,
              in Vector B,
              out dist[ BLOCK ] Vector C );
  csum double skal( in dist[ BLOCK ] Vector A );
};
```

Figure 5 Example of an extended-IDL interface

value, an interval of integer values, a function or the "*" symbol. This latter option means that the number of objects is chosen at runtime depending on the available resources (i.e. the number of computing nodes if we assume that each object is assigned to only one node).

Data distribution. Data distribution is specified using the `dist` keyword before the type of each parameter to be distributed. In the previous example, operation `mult` has two parameters (matrix A and vector C) which are distributed. After the `dist` keyword, distribution mode for each array dimension is specified. Distribution modes are similar to the ones defined in HPF (High Performance Fortran). The "*" indicates that the corresponding array dimension is not distributed. A non distributed parameter, as vector B, is replicated among each object of the collection.

Collective operation. A collective operation is a simple way to perform computations on the values returned by the objects belonging to the collection. Collective operations are performed by the stub at the client side. Collective operations are allowed only on scalar types. Operation `skal` in the previous example, illustrates the use of this new extension. In this example, the `csum` keyword indicates that the value returned by the operation is the sum of all the values given by all objects belonging to the collection.

4.1. STUB AND SKELETON CODE GENERATION

A stub generated by the *Extended*-IDL compiler does more works than a standard stub. Indeed, it is in charge of invoking simultaneously the same operation on each object of the collection. It has also to handle data distribution. Moreover, when stubs are used within a parallel object, they are in charge of synchronizing invocations and redistributing data [3]. It is important to note that a parallel flow of data can be maintained between two parallel CORBA objects, allowing an efficient use of high-performance networks (gigabit network) that connect computing resources together. Skeleton generated by the *Extended*-IDL compiler handles distributed data. A more detailed description of the parallel CORBA object concept can be found in [10].

4.2. IMPLEMENTATION

To implement the *Extended*-IDL compiler, we modified the IDL compiler provided by MICO [8] which is a freely available[3] and fully compliant implementation of the CORBA 2.3 standard. In the current implementation, the code generated by the *Extended*-IDL compiler cannot be used in conjunction with other CORBA implementations because they use specific methods of MICO. However, we are planning to modify the stub and the skeleton code generation process in a near future to make the generated code independent of the CORBA implementation. Modifications to the MICO compiler have been done in such a way that we are able to handle new versions of MICO with little effort.

5. COMPUTATIONAL ELECTROMAGNETIC APPLICATION

For the design of its products, Aérospatiale Matra is developing Electromagnetic simulation software for more than twenty years. Despite the fact that these codes are highly optimized, a parallel run, computing the antenna interaction on a typical aircraft, may last for more than 24 hours on an high end 16 PEs IBM SP. This is partly due to the fact that the same exact method is used to study the complete object though it would have been better (in term of performance) to use an appropriate method on each part of the system whenever possible. Moreover, Aérospatiale Matra is now evolving in such a challenging world that a competitor on one project may be a sub-contractor on another one. For these different reasons, a coupling method has been developed so that each part of the system may be studied with the most suited method. Moreover, each part can be considered as a black box and therefore can be simulated by its manufacturer without revealing any details to third parties.

The chosen application aims at simulating the electromagnetic interaction between two different physical objects such as between an antenna and an aircraft. The objective was to develop a coupled application so that each object could be simulated using a distinct computing resource. The simulation of each physical object is carried out using the

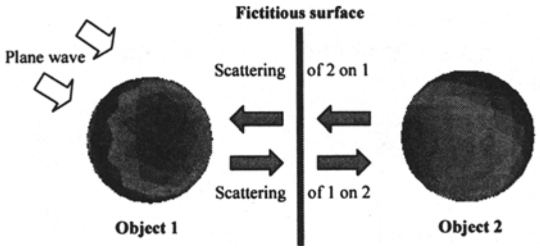

Figure 6 Electromagnetic simulation

ASERIS/BE simulation code from Aérospatiale Matra. A second objective was to make very few modifications to this simulation code so that a single code can be used for both standard simulation and coupled simulation. Figure 6 shows a simple example of an electromagnetic simulation involving two physical objects. This coupling method is based on an iterative process where the currents on an object due to the other parts are successively computed until convergence is achieved. As the different geometries may remain private (for confidentiality reasons), a fictitious surface is placed in between the different objects. For a typical simulation, the following quantities are computed: the currents due to the two objects on the fictitious surface, the interaction of the surface on each of the two objects and the currents on each of the objects due to the external illumination and the surface. The stopping criterion is based on the difference on the computed currents between two successive iterations.

The adding of a new object (the fictitious surface) requires the computation of new quantities as mentioned previously. Instead of modifying the ASERIS/BE simulation code, we took the decision to develop two new parallel codes to compute the currents on the fictitious surface (IoR code) and the interaction between the objects and the fictitious surface (PoR code). The different execution steps are illustrated in figure 7. Each physical object is simulated using a set of 5 codes: ASERIS/BE, IoR, VecSum, PoR and TestStop. These five steps are executed sequentially for each object. However, the simulation of the two objects is carried out in parallel and it is synchronized at each time step. At the beginning of the simulation, the electromagnetic radiation of one object is computed by the ASERIS/BE code (step 1). This radiation is then projected to the fictitious surface by the IoR code (step 2). The results of the two projections (one for each object) are combined by the VecSum code (step 3) in order to compute the interaction between the two objects. This is done by the PoR code (step 4). The electromagnetic radiation of the object is then recalculated by the ASERIS/BE code (step

5). The last step (6) computes the convergence criteria to determine whether a new iteration is required to reach an equilibrium.

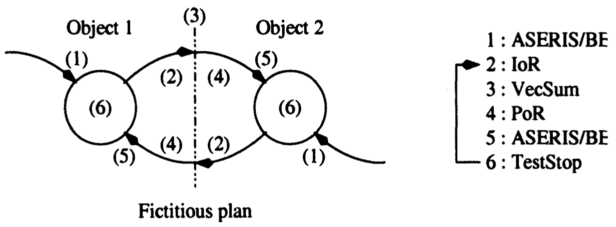

Figure 7 Execution steps

5.1. CODES COUPLING USING PARALLEL CORBA OBJECTS

The first step to design a coupled simulation application is to encapsulate the numerical codes into either CORBA objects (for the sequential code) or parallel CORBA objects (for parallel codes). It requires the specification of the interface using the *E*xtended-IDL language. We kept the interface as simple as possible since our constraint was to make little modifications to the original source codes provided by Aérospatiale Matra. The coupled application relies on the use of two CORBA services: the naming and the event services. The naming service has been modified slightly to support parallel CORBA objects. A symbolic name can be associated to a collection of object references instead of only one object reference. However the binding to a parallel CORBA object remains the same as for a standard CORBA object [10]. The event service is used to detect the termination of the coupled application.

Figure 8 gives an overview of the coupling strategy. *AS_ELFIP*, *I_o_R*, *P_o_R*, *TestStop* and *VecSum* are parallel codes encapsulated into parallel CORBA objects whereas *CtrlLoop* is a standard CORBA object. The simulation of the two physical objects can be carried out independently at each time step. Therefore, the software architecture is made of two kinds of scheduler. The master scheduler manages the overall application. It launches the two secondary schedulers by invoking an asynchronous (*oneway*) operation on each secondary scheduler (CORBA objects). Then, the master scheduler waits for events on an event channel. The receiving of an event indicates the termination of the application. Those events are sent by the two secondary schedulers when the two simulation codes reach a convergence criteria. Secondary schedulers invoke sequentially all the encapsulated simulation codes. They synchronize the computation flows through the *VecSum* parallel CORBA

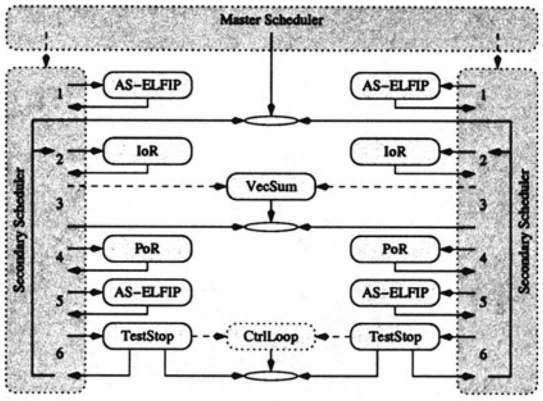

Figure 8 Architecture

object to exchange data values. The *CtrlLoop* CORBA object gathers convergence results (boolean value) from the *TestStop* parallel CORBA objects. It indicates whether the simulation process is completed by sending an event through an event channel.

Coupling simulation and visualization. One objective was to perform the simulation coupled with the visualization of the intermediate results at each time step. We developed a simple Java applet to control the execution of the application and to perform the visualization as soon as the results are available. One benefit of such approach is to let engineers to use the coupled application through their own computing system whatever the system is. Figure 9 shows the execution of the Java applet running within a Web browser. Visualization is performed thanks to the COUCHA code that takes as input the results from the ASERIS/BE code and produces a VRML file. The COUCHA code is a parallel code so that it has been encapsulated within a parallel CORBA object. The VRML file is read by the CosmoPlayer plug-ins and displayed within the Web browser window. Communication between the Java applet and the coupled application is performed through the CORBA ORB.

5.2. PERFORMANCE

We made several experiments to assess the performance of the coupling technique. These experiments were performed on a cluster of PCs (450 Mhz Pentium III processors) where each PC is connected to a Fast Ethernet network. Communication between the two sets of codes (left and right side in Figure 8) is performed using the CORBA ORB and

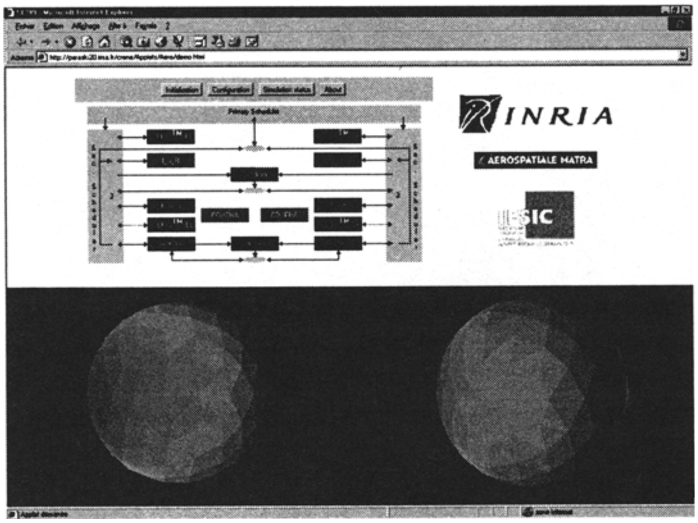

Figure 9 Monitoring the execution of the coupled application through a Web browser

a NFS file system within which files are stored. Since the two sets of codes ran in parallel, it is expected to have a speedup for the coupled application. We noticed a speedup of 1.52 when using four processors for each set of codes (a total of eight processors for the whole application) compared to the use of 4 processors to run successively the two sets of codes (at each time step, the first object is simulated followed by the second one).

6. CONCLUSION

In this paper, we give a description of our experience in the coupling of simulation codes using parallel CORBA objects. One main benefit of this technology is to allow a simple encapsulation of codes into distributed objects. Such encapsulation does not require extensive modifications to the original source codes. The main additional work was to design the primary and secondary scheduler to control the execution of the distributed simulation application. However, these schedulers were based on existing CORBA services (naming and event services) and thus were not so much time consuming to develop. It is worth mentioning that the encapsulated simulation code can be used within another coupled simulation application without modifying the existing interface.

Notes

1. The one acting as a bridge for the communication between different machines

2. Usually through a 100 Mb/s Ethernet link connected to a switch
3. http://www.mico.org

References

[1] T. Beisel, E. Gabriel, and M. Resch. An Extension to MPI for Distributed Computing on MPPs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (LNCS 1332)*, pages 75–83, 1997.

[2] Ian Foster, Jonathan Geisler, William Gropp, Nicholas Karonis, Ewing Lusk, George Thiruvathukal, and Steven Tuecke. Wide-area implementation of the Message Passing Interface. *Parallel Computing*, 24(12–13):1735–1749, November 1998.

[3] T. Kamachi, T. Priol, and C. René. Data distribution for Parallel CORBA Objects. In *Euro-Par 2000 (LNCS 1900)*, pages 1239–1249, Munich, Germany, August 2000.

[4] K. Keahey and D. Gannon. Developing and Evaluating Abstractions for Distributed Supercomputing. *Cluster Computing*, 1(1):69–79, May 1998.

[5] K. Moore, G.E. Fagg, A. Geist, and J. Dongarra. Scalable networked information processing environment (SNIPE). In *Supercomputing'97*, 1997.

[6] Object Management Group. The Common Object Request Broker: Architecture and Specification (Revision 2.3.1), October 1999.

[7] T. Priol and C. René. *Cobra*: a CORBA-compliant Programming Environment for High-Performance Computing. In *Euro-Par'98 (LNCS 1470)*, pages 1114–1122, Southampton, UK, September 1998.

[8] A. Puder. The MICO CORBA Compliant System. *Dr. Dobb's Journal*, 23(11):44–51, November 1998.

[9] A. Reinefeld, J. Gehring, and M. Brune. Communicating across parallel message-passing environments. *Journal of Systems Architecture*, 44:261–272, 1998.

[10] C. René and T. Priol. MPI Code Encapsulation using Parallel CORBA Object. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 3–10. IEEE, August 1999.

DISCUSSION

Speaker: Christoph René

Vladimir Getov : Have you considered using the Interoperable Message Passing Interface (IMPI) as an alternative environment for your project?
Christoph René : We are considering a distributed system where some of the nodes are parallel systems. Communication requirements are not the same where you have to communicate within a distributed system or within a parallel system. IMPI was designed to let MPI communication layers interoperate in a distributed system. IMPI is suitable if you would like to execute a single parallel application (MPI-based) on a distributed system. The distributed system is seen as a virtual parallel machine. In our case, we would like to support coupled applications: a set of different parallel codes connected together. We defend the idea that a MPI layer is not suitable when communicating between distinct codes in a coupled applications. In such a case, control and data have to be transfered between applications. MPI was designed to transfer data through message-passing but not the control. Moreover, in our approach we would like to have each parallel code associated with a description of its interface. IMPI does not provide an interface description language whereas CORBA IDL is a good candidate for such description.