MPI Code Encapsulation Using Parallel CORBA Object

Christophe René, Thierry Priol

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MPI Code Encapsulation using Parallel CORBA Object

Christophe René,Thierry Priol

N ° 3648

mars 1999

THÈME 1

ISSN 0249-6399

MPI Code Encapsulation using Parallel CORBA Object

Christophe René,Thierry Priol

Thème 1 — Réseaux et systèmes
Projet Caps

Rapport de recherche n3648 — mars 1999 — 10 pages

Abstract: This paper describes a technique that allows a MPI code to be encapsulated into a component. Our goal is to provide a software infrastructure to build distributed simulation applications for computational grids[3]. Our technique is based on an extension to the Common Object Request Broker Architecture (CORBA) from the OMG (Object Management Group). The proposed extensions do not modified the CORBA core infrastructure (the Object Request Broker) so that it can fully co-exist with existing CORBA applications. A MPI code is seen as a new kind of CORBA object that hides most of the cumbersome problems when dealing with parallelism.

Key-words: CORBA,MPI,HPCN,METACOMPUTING,PSE

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Encapsulation de codes MPI au sein d'objets CORBA parallèles

Résumé : Ce papier décrit une technique qui permet d'encapsuler un code MPI au sein d'un composant. Notre objectif est de construire une infrastructure logicielle pour les application de simulation distribuée sur des architecture de type "grid"[3]. La technique proposée est fondée sur une extension de l'architecture CORBA de l'OMG. Ces extensions ne remettent pas en cause l'infrastructure principale de CORBA (l'ORB) ce qui permet de co-exister avec des applications déjà écrites avec CORBA. Un code MPI, encapsulé au sein d'un composant, est vu comme un nouveau type d'objet CORBA qui permet de masquer les problèmes d'implémentation inhérent au parallélisme.

Mots-clé : CORBA,MPI,HPCN,METACOMPUTING,PSE

1 Introduction

Nowadays, the design of complex manufactured products, such as cars or aircrafts, is based on numerical simulation techniques. As the performance of computers is increasing very fast, it is now possible to envisage, in a short term range, a complete simulation of such products including multi-physics aspects such as structural mechanics, computational fluid dynamics, electromagnetism, noise reduction, etc. It is expected that the next-generation aircraft would involve several thousands of codes [12] among which some of them will have to be executed by high-performance computers. However, there is a lack of support to help the integration of simulation codes into existing software infrastructure. High-performance simulation codes are often designed as standalone applications taking as input data from files and producing results into new files. This fact can be explained by the poor integration of existing parallel programming environments, such as MPI or PVM, into modern software infrastructures.

Increasing complexity of simulation applications, as well as the need to couple several codes together, will no doubt lead to the use of programming paradigms based on the assembling of software components. Such approach will allow application programmers to build such applications using a set of *off-the-shelf* software components. Distributed objects, such as provided by ActiveX (Microsoft), Java (Sun) and CORBA (OMG), are technologies of choice for this purpose since they provide both a technique to encapsulate codes and to distribute them across several computers either at an intranet or at an internet level. Such distribution is often mandatory by most of the simulation applications since they require specific resources such as the access to database machines, high-performance graphics hardware for visualization or high-performance computers for the most computational intensive part of the application.

Among distributed object technologies, CORBA is probably the most promising one since it has been designed to handle a wide variety of object systems. CORBA is independent of the underlying hardware architecture and programming languages. CORBA can thus be seen as the glue technology to let components to talk together whatever languages are to implement the components (C++, Java, Ada95, F90[1], ...). However, CORBA is not able to manage all the aspects of heterogeneity. A CORBA object is always associated with a single process that implements the object using an appropriate language. When dealing with parallel codes, object implementation consists of a set of processes which are distributed among different processors of a supercomputer or different nodes of a cluster. This paper aims at showing how to extend CORBA to handle coarse-grain parallelism within a component using an SPMD (Single Program Multiple Data) execution model. More precisely, the proposed technique is able to encapsulate a MPI code into a new kind of CORBA object we called parallel CORBA object.

The remainder of this paper is structured as follows. Section 2 presents the parallel CORBA object concept. Section 3 gives our strategy to encapsulate MPI codes into a parallel CORBA object. Section 4 describes extensions made to the CORBA naming service

[1] IDL to F90 mapping has been done within the Esprit R&D PACHA project funded by the EC

to take into account parallel CORBA objects. Section 5 lists some related works. Finally, we conclude in section 6 by presenting some future works.

2 Parallel CORBA Object

CORBA is a specification from the OMG (Object Management Group) [9] to support distributed object-oriented applications. An application based on CORBA can be seen as a collection of independent software components or CORBA objects. Remote method invocations are handled by an Object Request Broker (ORB) which provides a communication infrastructure independent of the underlying network. An object interface is specified using the Interface Definition Language (IDL). An IDL file contains a list of operations for a given object that can be remotely invoked. An IDL compiler is in charge of generating a stub for the client side and a skeleton at the server side. Stubs and skeletons aim at connecting a client of a particular object to its implementation through the ORB.

The concept of parallel CORBA object[2] is simply a collection of identical CORBA objects as shown in figure 1.a. These objects can be manipulated as a single entity by our system. Our aim was to hide as much as possible most of the problems when dealing with coarse-grain parallelism on a distributed memory parallel architecture like clusters. First of all, the calling of a operation by a client is performed by all objects belonging to the collection using a SPMD execution model. This parallel activation is done transparently by our system. Data distribution between the objects belonging to a collection is entirely handled by the system. However, to let the system to carry out parallel execution and data distribution among the objects of the collection, some specifications have to be added to the component interface. A parallel object interface is described by an extended version of IDL, called Extended–IDL. It is a set of new keywords, added to the IDL syntax, to specify both the number of objects within the collection and the data distribution. The following example is an Extended–IDL specification of a service implementing matrix and vector operations.

```
interface[*:2*n] MatrixOperations {
  const long SIZE = 100;
  typedef double Vector[ SIZE ];
  typedef double Matrix[ SIZE ][ SIZE ];
  void multiply(in dist[ BLOCK ][*] Matrix A,
                in Vector B,
                out dist[ BLOCK ] Vector C );
  void skal(in dist[ BLOCK ] Vector C, out csum double skal);
  };
```

Two brackets have been added after the keyword `interface` within which several parameters can be given to specify the number of objects of the collection that will implement

[2] we will use parallel object from now

the parallel object and the shape of the virtual node array where objects of the collection will be mapped on. There are several ways to fix the number of objects in the collection. The expression may be an integer 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 networks nodes if we assume that each object is assigned to only one node). The shape of the virtual node array can be added just after when multi-dimensional arrays have to be distributed among the objects of the collection. Data distribution is specified using the `dist` keyword before the type of each parameter to be distributed. In the previous example, operation `multiply` 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 by 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. Operation `skal` illustrates another extension that allows collective operations to be performed on scalar parameters. 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. These extensions, as well as some restrictions about interface inheritance, are presented in more details in [10].

3 Adding MPI support to CORBA

Reason is twofold to add MPI support to CORBA. First and foremost, our main goal was to allow the encapsulation of MPI codes into parallel objects. We assume that such codes rely on a SPMD execution model so that the MPI code process corresponds to the object implementation. It is therefore necessary to offer a message-passing interface for the object. The second reason is the need to use a communication layer to distribute or redistribute data between the objects of a collection. This is illustrated by figure 1.b that shows a parallel object calling a standard object. Stubs, generated by our Extended-IDL compiler, have to both exchange data and to synchronize themselves. When invoking an operation using distributed data as a parameter, data has to be gathered from the objects of the collection prior to calling an operation. Moreover, only one object of the collection has to be selected to invoke the operation to the CORBA object at the server side. Remaining objects of the collection have to wait till the termination of the invocation operation. Similar problems arise when a parallel object invokes an operation to another parallel object. If there is a distributed parameter, data has to be redistributed according to both the data distribution specification of the caller and the one from the callee. Such operation is handled by both the skeleton of the caller that is aware on how data is distributed and the stub of the callee.

Both CORBA and MPI implementations offer remote process activation services when executing a MPI code or calling a distributed object. Such activation services often depend on CORBA and MPI implementations. In our case, we selected MICO[11] and MPICH[4]. Both of these environments exist on a large number of platforms. In CORBA, server activation is handled through a Basic Object Adapter (BOA) that is in charge of loading and

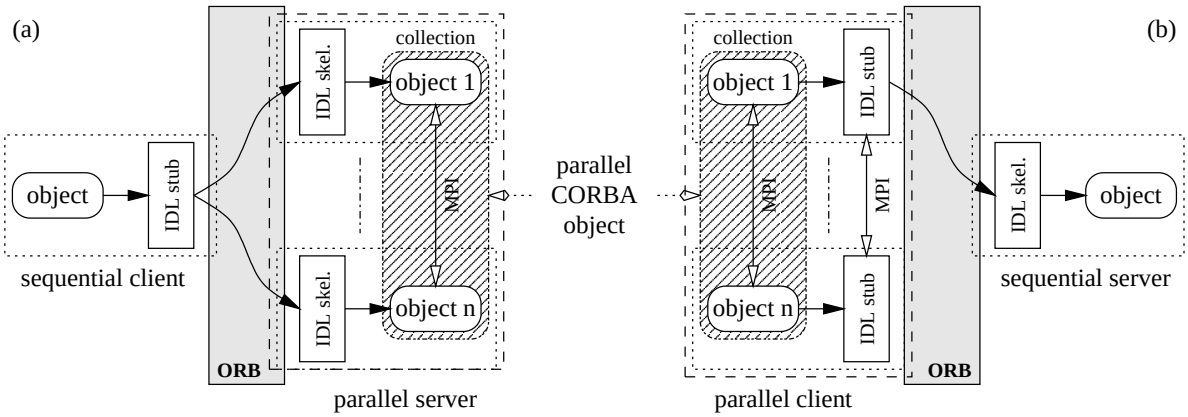

Figure 1: Parallel CORBA Object Concept

executing a process that implements an object when it is called by a client. Information concerning the implementation of a given service is obtained by the BOA through the Implementation Repository. In our case, activation of a parallel object cannot be performed by the standard BOA. The use of the MPI runtime is mandatory. We proposed to integrate the MPI runtime into an extension of the BOA we called the Parallel BOA (PBOA). Using such PBOA, parallel object activation requires two steps. At the beginning, one object of the collection is activated. This object starts by creating a MPI communication group which is used later when objects of the collection need to exchange data or have to synchronize themselves. Then, this object activates the other objects of the collection giving them the communication group they have to join. The nodes where the objects have to be mapped on are given by *Cobra*[1] that is a set of CORBA services for resource allocation.

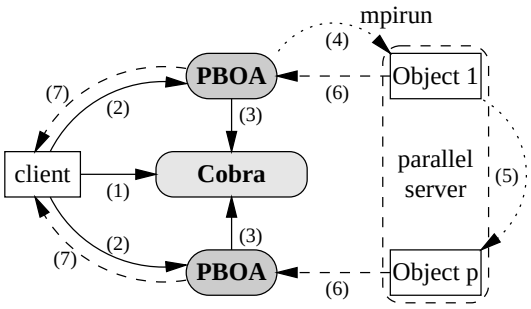

Figure 2: Parallel server remote activation

Figure 2 shows the different steps that have to be performed when calling a parallel object. First of all, the client allocates a group of nodes for its computation using *Cobra*(1)

by invoking a specific operation. Then, it asks (2) each PBOA (a UNIX process running on each node of the allocated group) to activate the service. Each PBOA sends a request (3) to the Cobra service to know which PBOA is in charge of executing the *mpirun* provided by MPICH (4). This command will start the execution of the MPI process on each node of the allocated group (5). When the objects are ready, they return their reference to the PBOA (6). Each reference is then forwarded to the client (7). All these different steps, when activating a parallel object, are hidden to the client. The following example illustrates the binding to a parallel object at the client side.

```
// Create a group of nbnode nodes using Cobra
cobra->mkvpm( myvpm_name, nbnode, NORES );
// Get information about the newly create node group
pap_get_info_vpm( &myvpm_struct, myvpm_name );
// Create a sequence that contains a list of node names
Sequence< String_var > addrs = VPMtoAddr( myvpm_struct );
// Obtain a reference to the parallel CORBA object
parobj = MatrixOperations::_bind( addrs );
// Invoke an operation
parobj->multiply(A, B, C);
```

4 Extending the CORBA Naming Service

The Naming Service[8] belongs to the CORBA Common Object Services (COS). It allows a user to manipulate objects through symbolic names instead of using object references. A symbolic name is associated with only one object reference. To handle parallel objects the same way, it is necessary to tie a symbolic name to a collection of object references. These references are needed by the stub when an operation is called to a parallel object. We do not implement a separate naming service for parallel objects, instead we proposed some extension to the existing naming service. We decided to keep the context mechanism defined in the standard naming service. This mechanism allows to organize symbolic name into a hierarchy. The following IDL specification presents new types and new operations added to the standard specification in order to support the symbolic naming of parallel objects.

```
module CosNaming {
  /* ... */
  interface NamingContext {
    /* ... */
    typedef sequence<Object> ObjectCollection;
    void join_collection( in Name n, in Object obj );
    void leave_collection( in Name n, in Object obj );
    ObjectCollection resolve_collection( in Name n );
  };
};
```

Operations `join_collection`, `leave_collection` and `resolve_collection` play respectively the same role as the `bind`, `unbind` and `resolve` operations defined by the standard naming service.

Each object that is joining a collection invokes the `join_collection` method. This method starts by checking whether a collection of references with the same name has already been created. Then, the reference of the object, wishing to join the collection, is added. When an object wants to leave the collection, it calls the `leave_collection` method. It removes the object reference from the collection. It is also in charge of freeing the name associated with the collection, when it becomes empty. The following example illustrates the registration process at the server side.

```
MatOp_Impl* obj = new MatOp_Impl();
NamingService->join_collection( "Matrix", obj );
/* ... */
NamingService->leave_collection( "Matrix", obj );
```

At the client side, the calling of the standard `resolve_collection` method returns a sequence of object. An invocation of the `_narrow` method allows to get a reference to the parallel object as illustrated by the following example.

```
objs = NamingService->resolve_collection( "Matrix" );
svr = MatrixOperations::_narrow( objs );
svr->multiply( A, B, C );
```

5 Related works

There are very few projects aiming at support coarse-grain parallelism with CORBA. PARDIS environment[5, 6] is the most advanced project in this direction. Like us, PARDIS designers propose a new kind of object they called SPMD object which is an extension of a CORBA object. 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 objects, PARDIS provides a generalization of the CORBA sequence called *distributed sequence*. Like our IDL extensions, this new argument type requires the modification of the IDL compiler. One of the key feature of PARDIS is no doubt its ability to support concurrency between objects. PARDIS provides a mechanism to invoke operations on objects asynchronously that is based on the *future* concept. A *future* is the basic mechanism to get the results of services that has been activated asynchronously. In our approach, asynchronous invocation 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. Contrary to our work, PARDIS does not provide a specific service for resource allocation to carry out the execution of SPMD objects.

Some of the problems we encountered, such as parallel invocation, are quite similar with those that have been investigated to make a CORBA object fault tolerant. The basic

principle to handle a failure is to have a group of identical object mapped onto different machines across a network. This idea has been investigated in [2, 7]. Two approaches have been proposed. In [2], a group communication is provided as a service on top of the ORB whereas in [7], the proposed solution is part of the ORB itself.

6 Conclusion

In this paper, we have described a technique able to encapsulate MPI-based codes into CORBA parallel objects. To this aim, we proposed some extensions to the IDL syntax to specify the number of objects in the collection and data distribution. A software prototype, made of an Extended-IDL compiler plus the necessary support for parallel objects, has been developed using MICO. We are currently experimenting parallel objects in various applications (signal processing [1] or virtual reality that will be described in the final version of this paper). These two applications illustrate the use of our technology using a client/server approach. We are also investigating the use of the concept of parallel objects to couple simulation codes together when dealing with multi-physics simulation. Our goal is to show that component programming is a suitable approach for scientific code coupling. Our research effort is embedded in the Jaco3 Esprit project [3] which aims at building a Java and CORBA based Collaborative Environment for coupled simulation. Parallel CORBA object will be the key technology to couple high performance simulation codes together.

References

[1] P. Beaugendre, T. Priol, G. Alleon, and D. Delavaux. A Client/Server Approach for HPC Applications within a Networking Environment. In *HPCN'98*, pages 518–525, Amsterdam, The Netherlands, April 1998.

[2] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. Thése, École Polytechnique Fédérale de Lausanne, 1998.

[3] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc, 1998.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

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

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

[3] A more detailed description of this project is available at `http://www.arttic.com/projects/JACO3`

[7] S. Maffeis. The Object Group Design Pattern. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Toronto, Canada, June 1996.

[8] Object Management Group. CORBAServices: Common Object Services Specification, November 1997.

[9] Object Management Group. The Common Object Request Broker: Architecture and Specification (Revision 2.2), February 1998.

[10] T. Priol and C. René. COBRA: A CORBA-compliant Programming Environment for High-Performance Computing. In *Euro-Par'98*, pages 1114–1122, Southampton, UK, September 1998.

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

[12] J.R. Rice and R.F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Computational Science & Engineering*, 3, 1996.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399