Efficient Support of MPI-based Parallel Codes within a CORBA-based Software Infrastructure

Thierry Priol, Christophe René

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Support of MPI-based Parallel Codes within a CORBA-based Software Infrastructure

Thierry Priol, Christophe René

No˚3750

Août 1999

THÈME 1

ISSN 0249-6399

Efficient Support of MPI-based Parallel Codes within a CORBA-based Software Infrastructure

Thierry Priol, Christophe René

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n ° 3750 — Août 1999 — 15 pages

Abstract: This document (orbos/99-07-10) aims at giving a response to the Aggregated Computing RFI (orbos/99-01-04) published by the OMG. It describes a technical solution to incorporate efficiently parallel codes into an existing CORBA based software infrastructure.

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

(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

Support efficace de codes parallèles MPI au sein d'une infrastructure logicielle CORBA

Résumé : Ce document (orbos/99-07-10) est une réponse au Aggregated Computing RFI (orbos/99-01-04) publié par l'OMG. Il décrit une solution technique pour incorporer efficacement des codes parallèles au sein d'une infrastructure logicielle fondée sur CORBA.

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

1 Overview

This document (orbos/99-07-10) aims at giving a response to the Aggregated Computing RFI (orbos/99-01-04) published by the OMG. It describes a technical solution to incorporate efficiently parallel codes into an existing CORBA based software infrastructure. This solution is based on either an extension of the IDL language or pragmas to support the orchestration of a group of processing subsystems to work together on a large problem by distributing both computation and data. It does not require the modification of the ORB. The proposed solution aims at combining both CORBA and MPI runtimes[1]. The merging of the two technologies allows to get the required level of performance and interoperability. Questions about this response should be directed to:

- Dr. Thierry Priol IRISA/INRIA
  Campus de Beaulieu
  35042 Rennes Cedex
  France
  phone: +33 2 99 84 72 10
  fax: +33 2 99 84 25 28
  email: Thierry.Priol@irisa.fr

This document addresses the following items of the Aggregated Computing RFI:

- 4.2.2 Parallel Processing
- 4.2.3 Other Issues (CORBA services and capabilities that are problematic)
- 4.2.4 Other Standardization Activities (MPI Forum)

This document describes the results of research works that have been already published in scientific conferences. The following references contains technical information about our work:

1. P. Beaugendre, T. Priol, G. Alléon, and D. Delavaux. A Client/Server Approach for HPC Applications within a Networking Environment. In Proc. of HPCN'98, number 1401 in LNCS, Springer Verlag, pages 518–525, Amsterdam, Pays-Bas, April 1998. available at http://www.irisa.fr/paris/biblio/Fichiers-PS/Thierry-Priol/hpcn98.ps.gz
2. T. Priol and C. René. 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. available at http://www.irisa.fr/paris/biblio/Fichiers-PS/Thierry-Priol/europar98.ps.gz

[1] MPI stands for Message Passing Interface; it is a de facto standard for parallel programming.

3. C. René and T. Priol. MPI Code Encapsulating using Parallel CORBA Object. In Proc. of of the Eighth IEEE International Symposium on High Performance Distributed Computing, August 1999. available at http://www.irisa.fr/paris/biblio/Fichiers-PS/Thierry-Priol/hpdc8.ps.gz

4. T. Priol, C. René, and G. Alléon. SCI-based Cluster Computing, chapter Programming SCI Clusters using Parallel CORBA Objects, to appear, Springer LNCS State of the Art Surveys. Springler Verlag, 1999. available at http://www.irisa.fr/paris/biblio/Fichiers-PS/Thierry-Priol/lncs99.ps.gz

2 Parallel processing with CORBA

Nowadays, the design of complex manufactured products, such as cars or aircrafts, is based on numerical simulation techniques. With the performance of computers rapidly increasing, it is possible to foresee in the near future comprehensive simulations of these designs that encompass multi-disciplinary aspects (structural mechanics, computational fluid dynamics, noise analysis, etc). Numerical simulation of these different aspects will require the aggregation of several computing resources, that are available on a network, to keep simulation times within a reasonable bound. A distributed software infrastructure will therefore be required to make a cluster of computing resources act like a single large virtual computer. Such software infrastructure will have to be based on an appropriate middleware to hide the distribution of resources and to provide services for the execution of applications. CORBA is a technology of choice for this purpose since it provides both a technique to encapsulate codes and to distribute them across several computing resources. Moreover, CORBA is being adopted by the engineering industry that is right now the largest user community for numerical simulation[2].

However, CORBA does not provide an efficient way to encapsulate high performance numerical simulation applications that are used by the engineering industry. Most of these applications are parallel codes running on supercomputers. Such codes require the use of specific runtime systems for their execution. The most well know runtime is the Message Passing Interface (MPI) which has been described in a previous response to the Aggregated Computing RFI (Document orbos/99-04-12). The coupling of MPI-based simulation codes cannot be done efficiently through the CORBA ORB. This document aims at proposing some minor extensions to the CORBA architecture in order to support efficiently the encapsulation of MPI-based codes into CORBA objects.

This work does not aim at extending CORBA for parallel programming so that it can replace MPI. Instead, it aims at showing how to combine efficiently these two standards in order to be able to develop high-performance distributed software platforms.

[2]The CATIA CAD tool offers a CORBA interface in its last release.

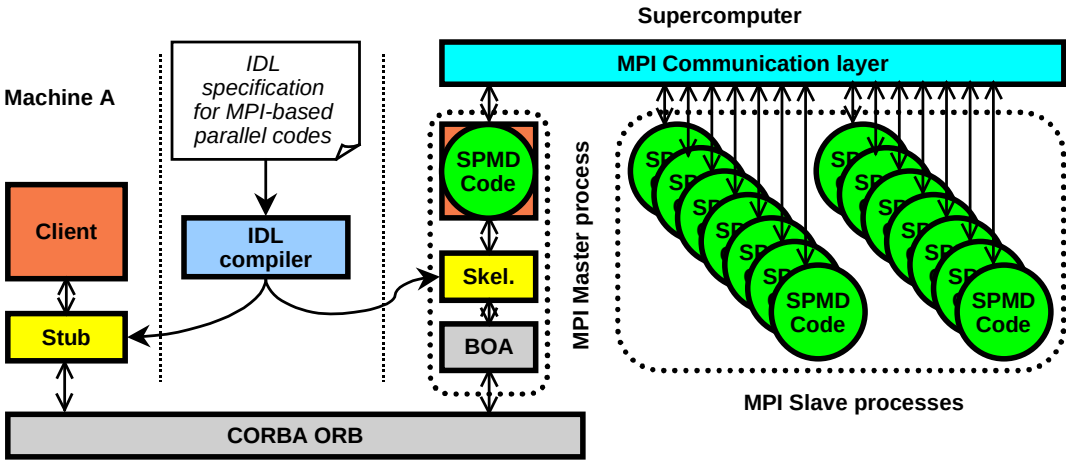


Figure 1: A master/slave approach to encapsulate MPI codes

3 Encapsulating MPI-based parallel codes into CORBA objects

The integration of MPI-based parallel codes to existing CORBA infrastructures is an important issue in many research and development projects[3]. In most of these projects, the usual way of encapsulating parallel codes is to adopt a master/slave approach as shown in Figure 1.

A MPI-based parallel code can be seen as a set of identical processes called SPMD[4] process. The code that corresponds to such process is called a SPMD code. Each processor of a parallel system runs one SPMD process. The master/slave approach consists in selecting one process (usually the one having a logical process identifier equal to 0) to play the role of a master that is connected to slave processes through the MPI communication layer. Only the master process is encapsulated into a CORBA object. Such simple solution requires some modifications to existing MPI codes if they did not already follow a master/slave approach. Moreover, the master may represent an important bottleneck when two MPI codes have to communicate with each other. To illustrate this problem, a simple example is shown in Figure 2. A client is scheduling the execution of two MPI-based parallel codes. Data, generated by the first code, are gathered by the client and are sent to the second code, which in turn scatters the data to the MPI slave processes, performs the computation and gathers the results to send them back to the client. This approach does not offer a scalable solution

[3] In the framework of the Esprit programme funded by the European Union (http://www.arttic.com/projects/JACO3, http://www.6s.org) or from other initiatives funded by European countries (http://www.sistec.dlr.de/en/projects/tent/WebTent.html)

[4] Single Program Multiple Data (SPMD) is an execution model largely used to program parallel systems.

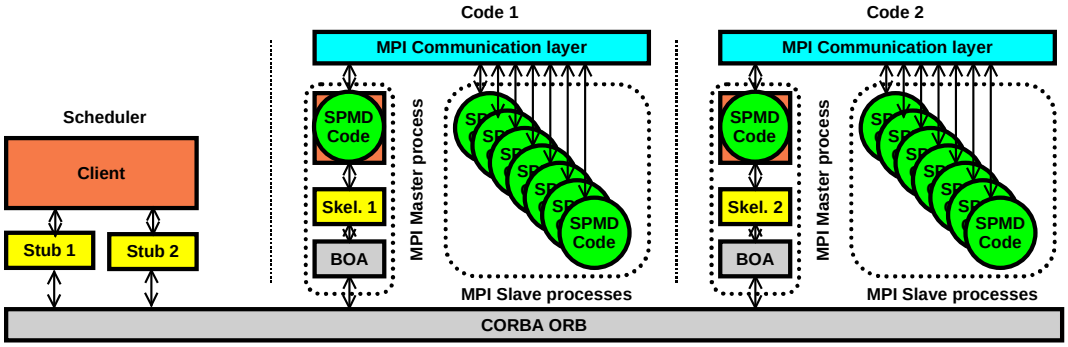


Figure 2: Coupling of two MPI-based parallel codes through the CORBA ORB.

to the encapsulation of parallel codes. As the number of SPMD processes or the size of the problem (amount of data transmitted between two parallel codes) increases, it will entail a large communication overhead.

To avoid the gathering and the scattering of data when calling a CORBA object encapsulating a MPI code, it is necessary to offer a technique in which all the SPMD processes will be encapsulated into CORBA objects. With such an approach, all the SPMD processes will be connected to the ORB allowing to perform communication between objects belonging to two different collections through the ORB. Such solution is mainly driven by performance reasons. However, it is necessary not to expose parallelism to a client. This means that the client has to see only one entity instead of all the CORBA objects encapsulating SPMD processes. As for instance, an operation call, issued by a client, has to be performed by all CORBA objects that represent the MPI code. This calling has to be done transparently by the client. Moreover, if a client needs to distribute data among CORBA objects, it has to be done as transparently as possible. These two opposite requirements (performance and transparency) lead to the introduction of a new concept, called parallel CORBA object, that provides a trade-off between performance and transparency. The next section gives a description of this concept.

4 Parallel CORBA object

The concept of parallel CORBA object[5] is simply a collection of identical CORBA objects as shown in Figure 3. A CORBA object encapsulates a SPMD process. It thus complies with the first requirement since during the execution of a parallel code, every process (i.e. an CORBA object) is connected to the ORB. It complies also with the second requirement (transparency) since a parallel CORBA object is used as a standard CORBA object. All the CORBA objects belonging to a collection can be manipulated as a single entity by the

[5]Parallel object is used from now on instead of parallel CORBA object

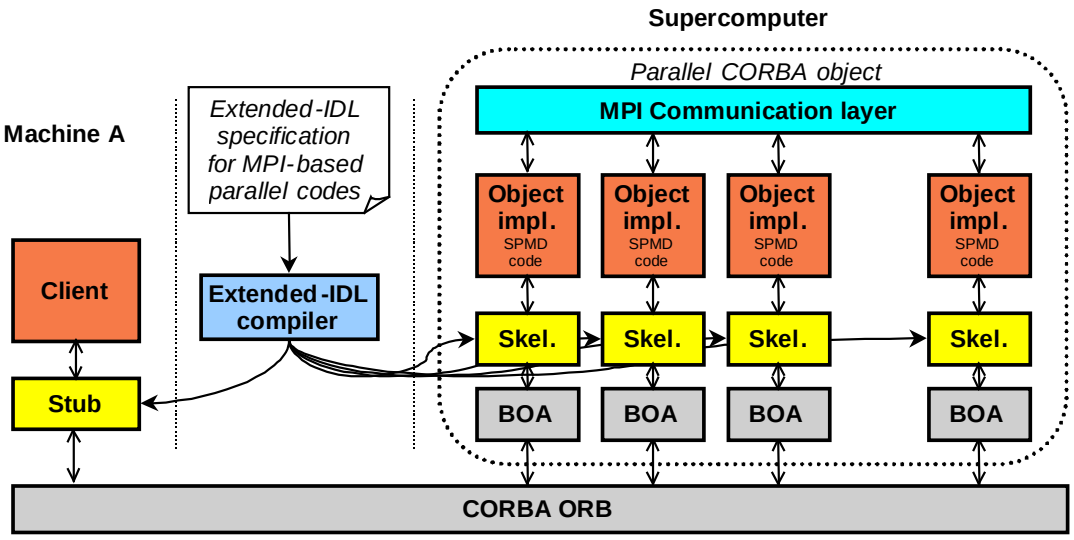


Figure 3: Parallel CORBA Object.

system. First of all, the calling of an operation by a client will result in the execution of the associated method by all objects belonging to the collection at the server side (following an SPMD execution model). This parallel activation is done transparently and data distribution between the objects belonging to a collection is entirely handled by the system. Such level of transparency is achieved through the adding of new information to the component interface represented by an IDL specification. With such information, new stubs and skeletons can be generated to hide as much as possible the collection of CORBA objects to the client.

```
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 val);
};
```

Figure 4: Version with IDL extension

A parallel object interface is described by an extended version of the IDL, called Extended-IDL. It is a set of new keywords or pragmas to specify both the number of objects within

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

Figure 5: Version with pragmas.

the collection and the data distribution. Figure 4 shows an Extended-IDL specification of a very simple example.

In Figure 4, extensions to the IDL language appear in bold face. The first extension consist in specifying the number of objects in the collection and the virtual shape of these objects. This extension appears just after the interface keyword. The second extension (dist, csum) is related to the data distribution associated with parameters of operations. Such extensions appear as new IDL keywords however pragmas can be used instead as shown in Figure 5. In that case, parallel CORBA object does not require the modification of the IDL syntax.

The following subsections give a short description about how to map objects and how to distribute data among the objects belonging to the collection using the example given in Figure 4/5. All the extensions added to the IDL, as well as some restrictions about interface inheritance, are presented in more details in [2].

4.1 Mapping of objects

The number of objects in the collection, that will implement the parallel object, and the shape of the virtual node array where objects of the collection will be mapped on, are 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 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 network nodes if one assume that each object is assigned to only one node). The shape of the virtual node array is an optional parameter that describes how objects must be organised. A virtual node array is used during data distribution of multidimensional arrays. It indicates the number of objects to be distributed on a given dimension (of an array)

for which a distribution is specified. When this parameter is omitted and when there are several dimensions in arrays given as parameters of an operation, the Extended-IDL compiler computes a shape of the virtual node array that matches the number of objects used. In Figure 4/5, it is required that the first dimension of the virtual node array is a multiple of 2. Only the operation multiply uses a multi-dimensional array in the list of its parameters. It requires that the first dimension of the matrix must be distributed. Therefore, objects will be organised as a vector.

4.2 Data distribution

Data distribution is specified using the dist keyword before the type of each parameter to be distributed. In Figure 4, 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. The "*" indicates that the corresponding array dimension is not distributed. Distribution mode can be either BLOCK or CYCLIC. A non distributed parameter, as vector B, is replicated among each object of the collection. Distribution modes are similar to the ones defined by HPF (High Performance Fortran).

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.

5 Stub and skeleton code generation associated with the Extended-IDL

Generation of stub and skeleton codes for parallel objects allow to manage the collection of objects as a single entity thus complying with the transparency requirement. These codes, generated by a new IDL compiler, are slightly different from stub and skeleton codes associated with standard CORBA object. They both handle the distribution of data among the collection of objects and the simultaneous invocation of an operation on every object of the collection.

To handle distributed data, a new data type has been created. This new type, called distributed array (darray), is based on the CORBA sequence able to store various information about data distribution. The use of sequence is required to support collection of objects for which the Extended-IDL interface does not provide a fixed number of objects. In such a case, data that has to be sent to each object of the collection depends on the number of objects belonging to the collection. To keep the size of the generated stub as small as possible, all the parameters of an operation belonging to a parallel object, that are of array types, are mapped to distributed array even if they are not distributed parameters.

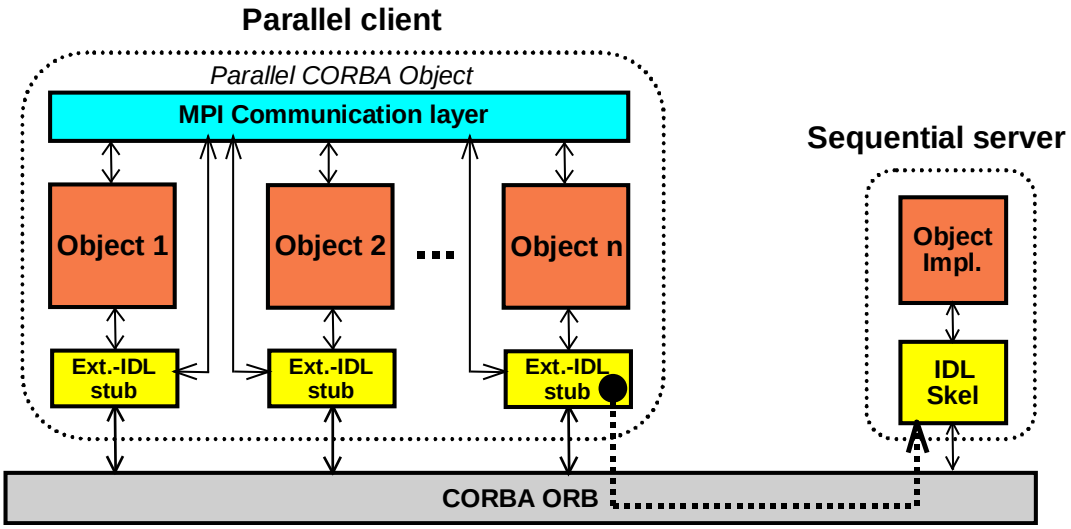


Figure 6: Invocation from a Parallel CORBA object to a standard CORBA object.

The concept of parallel object is not restricted to a sequential client able to bind to a parallel object. It is necessary to allow both the caller and the callee to be parallel or sequential. Stub and skeleton codes have to handle various situations.

The first situation is when the client is sequential and the server is a parallel object. Such case is illustrated in Figure 3. In this situation, the stub generated by the Extended-IDL compiler has to build and send simultaneously a request to each object of the collection. It has also to scatter data among the objects belonging to the collection following the data distribution specification. Moreover, as explained in the previous section, it can perform some computation when a collective operation is specified for a parameter.

Figure 6 illustrates a second situation where the client is itself a parallel object and the callee is a standard CORBA object. In such case, the stub has to perform more complex operations before calling the object. If parameters of an operation are distributed, the stub has to gather data from objects belonging to the collection prior to invoking the operation. Moreover, only one object of the collection has to be selected to invoke the operation to the CORBA object at the server side. The stub is responsible for selecting one object of the collection to perform the invocation. The other objects of the collection have to wait until the completion 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 of the callee. Depending on the number of objects at the client side and at the server side, the stub has to send one or several requests or wait for the completion of the invocation operation. Synchronisation and data distribution or redistribution are performed using the MPI message passing layer for both performance and portability reasons.

```
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);
 ;
;
```

Figure 7: Naming service operations to handle parallel object.

In the previous paragraphs, modifications to the stub code generation process were mainly described. The skeleton has also to be modified to support parallel objects. The code generation process consists in storing information about the data distribution specification for each distributed parameter. This information is necessary to ensure data redistribution when invoking an operation from a parallel object to either a sequential or a parallel object. Information stored in the skeleton of the caller will be given to the stub associated to the callee.

6 Extending the Naming Service to support Parallel CORBA object

The Naming Service 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 bind 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. To solve this problem, two possibilities are offered. The first one consists in designing a separate naming service for parallel object whereas the second one is based on the extension of the existing naming services. These two approaches have their own advantages and drawbacks. The proposed solution is based on the extension of the existing naming service but can be the basis of the design of a new naming service. Figure 7 gives the IDL specification that presents new types and operations added to the standard specification in order to support the symbolic naming of parallel objects.

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. The tt Name type corresponds to a data structure that contains several information such as the symbolic name associated with the parallel object.

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

Figure 8: Registration of a parallel object at the server side.

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

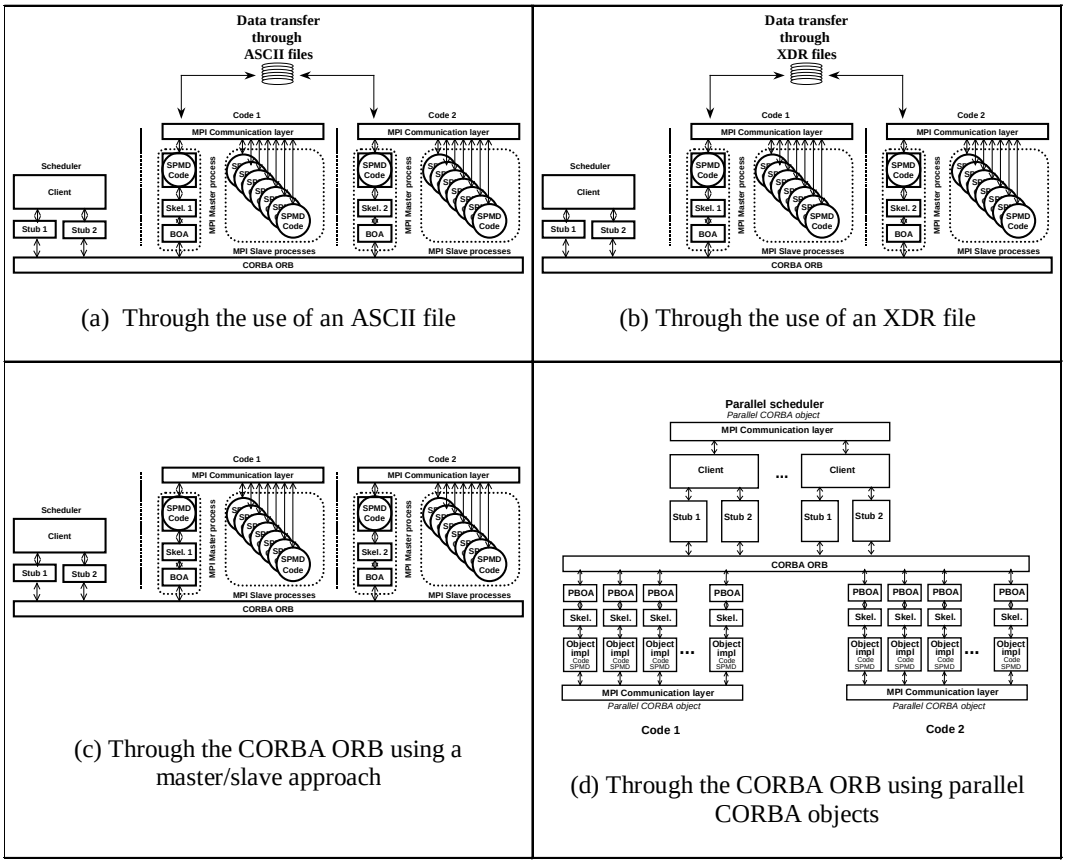
Figure 9: Binding to a parallel object using a symbolic name.

An object 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 joining the collection is added. When an object wants to leave the collection, it calls the leave_collection method, which removes the object reference from the collection. This method is also in charge of freeing the name associated with the collection when it becomes empty. The example, given in Figure 8, illustrates the registration process at the server side.

At the client side, the calling of the standard `resolve_collection` method returns a sequence of objects. An invocation of the `_narrow` method gets a reference to the parallel object as illustrated by the example shown in Figure 9.

7 Performance

To show the benefits of parallel CORBA objects comparing to other standard approaches, we performed an experiment that consists in exchanging a matrix between two MPI-based parallel codes encapsulated into either standard CORBA objects or parallel CORBA objects. Figure 10 shows four different strategies to transfer data between the two encapsulated parallel codes. The first strategy (Figure 10-a) consists in sending data through an ASCII file stored on a NFS file server so that the two CORBA objects located on different machines can have access to it. Since data is encoded in ASCII, interoperability between the two machines is achieved. The second strategy (Figure 10-b) is similar to the previous one but differs by the encoding technique which is based on XDR. In that case, the file is a binary one but interoperability is still achieved. The third strategy consists in sending data entirely through the ORB (Figure 10-c). Finally, the fourth strategy is based on the use of parallel CORBA objects (Figure 10-d). Data are distributed among the collection of objects (scheduler, code 1 and code 2). In the four test cases, data correspond to a two dimensional matrix that is distributed by block among the SMPD processes of both code 1 and code 2.



(a) Through the use of an ASCII file

(b) Through the use of an XDR file

(c) Through the CORBA ORB using a master/slave approach

(d) Through the CORBA ORB using parallel CORBA objects

Figure 10: Four strategies to transfer data between CORBA objects.

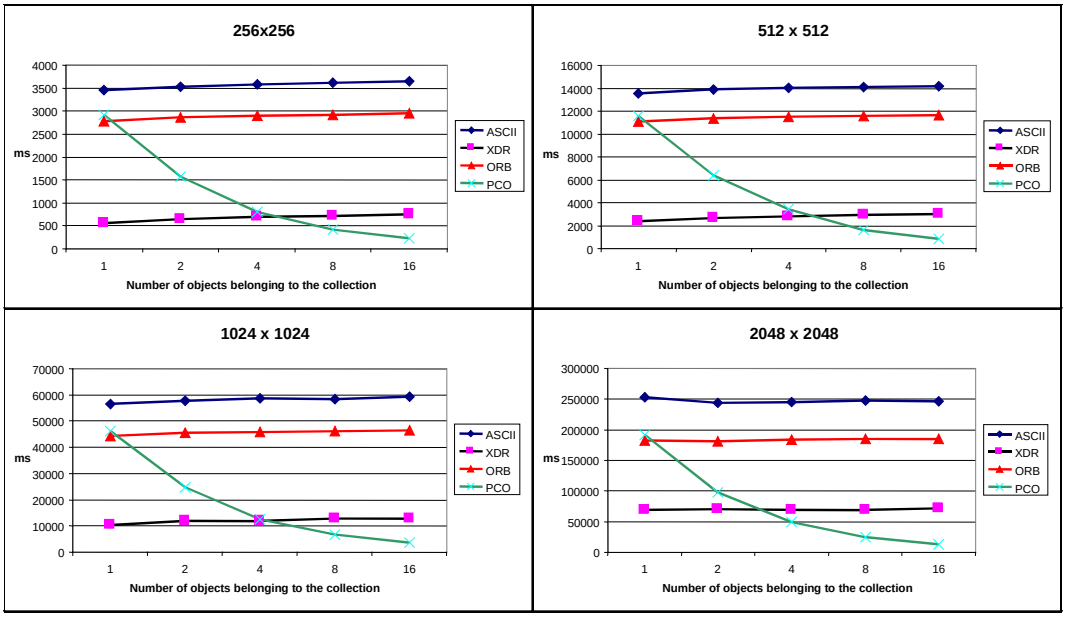


Figure 11: Elapsed times for the four strategies.

Figure 11 gives the performance results obtained with the four approaches. The vertical axis gives the elapsed time (in ms) corresponding to the execution of the client (scheduler) that has to perform successively a call to code 1 and to code 2. The horizontal axis gives the number of SPMD processes (for the first three approaches) or the number of CORBA objects belonging to the collection (for the fourth approach). This experiment was performed using a cluster of 16 Pentium-II based PCs, running Linux, interconnected by Fast Ethernet. CORBA objects were mapped onto PCs in such way that there is no local communication between the scheduler (client) and the encapsulated codes. To evaluate the performance of the fourth approach (Figure 10-d), objects of the scheduler were mapped onto PCs starting with logical number from 0 to 15 (object 0 mapped onto PC 0, 1 onto 1, etc.) whereas the objects, associated with code 1 and code 2, were mapped onto PCs starting with logical number from 15 to 0 (object 0 mapped onto PC 15, 1 onto 14, etc.).

The CORBA implementation for this experiment was MICO from Frankfurt University and the MPI communication layer is MPICH from Argonne National Laboratory. Four matrix sizes were used to analyse the impact of the problem size. From this experiment, the following conclusions were drawn. When facing with sequential code (one object belonging to the collection), the best strategy to exchange data between the two CORBA objects is to use a distributed file system in which the data are stored into a file using a XDR encoding scheme. The marshalling and de-marshalling of data associated with the communication with the ORB add a huge overhead.

The conclusion is different when dealing with parallel codes. As the number of objects belonging to the collection increases, the approach based on parallel CORBA objects provides much better performances when the number of objects in the collection is greater than a certain number. This number depends on the problem size. For this benchmark, when the number of objects is greater than 4, performance of the approach based on parallel CORBA objects outperforms the technique based on the use of a XDR encoded file.

8 Conclusion

The parallel CORBA object concept has been designed to efficiently encapsulate MPI-based parallel codes so that they can be used in existing CORBA-based software infrastructure. Preliminary performance evaluation shows that a better performance is achieved when using parallel CORBA objects instead of other usual techniques. An implementation of such concept is being developed at INRIA-Rennes within the PARIS project. It is being used in the design of a CORBA-based software environment for coupled simulations within the JACO3 Esprit R&D projects[6]. However, our approach has some drawbacks. It requires the modification of the stub and skeleton code generation process. This raises the problem of interfacing stubs and skeletons to the ORB. Our approach requires to target the stub and skeleton code generation process to a particular CORBA implementation (in our case MICO). This is due to the proprietary stub-to-ORB interface provided by the CORBA implementers. Our approach should be more generic if there was a standard open interface to the ORB specified by the OMG.

[6] Information related to this project can be obtained at http://www.arttic.com/projects/JACO3.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité 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