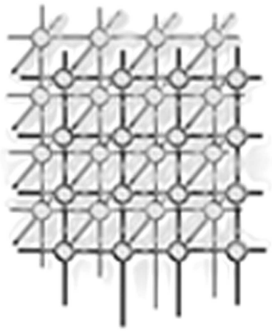
Achieving Portable and Efficient Parallel CORBA Objects

Alexandre Denis, Christian Pérez, Thierry Priol

CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2001; **3**:1–2 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

Achieving Portable and Efficient Parallel CORBA Objects



Alexandre Denis[1], Christian Pérez[2,*] and Thierry Priol[2]

[1] *IRISA/IFSIC,* [2] *IRISA/INRIA,*
Campus de Beaulieu, 35042 Rennes Cedex, France

SUMMARY

With the availability of Computational Grids, new kinds of applications are emerging. They raise the problem of how to program them on such computing systems. In this paper, we advocate a programming model based on a combination of parallel and distributed programming models. Compared to previous approaches, this work aims at bringing SPMD programming into CORBA in a portable way. For example, we want to interconnect two parallel codes by CORBA without modifying either CORBA or the parallel communication API. We show that such an approach does not entail any loss of performance compared to previous approaches that required modification to the CORBA standard. Moreover, using an ORB that is able to exploit high performance networks, we show that portable parallel CORBA objects can efficiently make use of such networks.

KEY WORDS: CORBA, Grid computing, MPI, Code coupling, High performance network

1. Introduction

With the availability of high performance networking technologies, it is nowadays feasible to couple several computing resources together to offer a new kind of computing infrastructure that is called a Computational Grid [9, 10]. A Computational Grid acts as a high performance virtual computer to users to perform various applications such as for scientific computing or for data management. This idea has already been addressed since a Computational Grid can be seen as a kind of distributed and parallel system. Some years ago, A. Tanenbaum[22] gave a definition for such system: "A distributed system is a collection of independent computers that appear to the users of the system as a single computer". Therefore, building Computational Grids raises the same design issues as for distributed systems: transparency (location of

*Correspondence to: Christian Pérez, IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France

Received November 6, 2002

resources is transparent to the user), interoperability (to hide the heterogeneity of computing and networking resources) and reliability (the system has to survive the unavailability of computing and networking resources). Computational Grids also share the same design issues as for parallel systems: performance (best use of both computing and networking resources) and scalability (efficient management of a huge number of resources).

Software infrastructures, such as Globus[9] or Legion[12], aim at providing runtime systems to allow the execution of applications on Computational Grids. However, Globus was designed mainly to allow the execution of existing parallel applications. Such approach makes sense since there are already a huge number of existing parallel applications that should benefit from Computational Grids. However, the availability of Computational Grids will give rise to new kind of applications for which parallel programming based on the use of message-passing libraries is not suitable.

Coupled simulations are an example of such new kinds of application. Complex systems that require a multi-physics approach are simulated by several parallel codes that are coupled together. Till know, code coupling is carried out thanks to the use of tools that are specialized for a given domain (CSM Flux [5] or OASIS [23] couplers for coupled climate simulation, MpCCI [2] for a larger spectrum of coupled simulation applications). These tools rely on message passing libraries (such as MPI) and they usually offer a sequential coupling. One particular process of each simulation code gathers and scatters data to the other simulation codes. MpCCI offers parallel communication between simulation codes. However, if the simulation codes are mapped on different machines, parallel communication could not be allowed by the MPI implementations (such as PACX [11] for which messages are gathered by one process on one machine before sending them to another process on the other machine in charge of scattering them to the other MPI processes). Moreover, MPI runtimes were mainly designed for a SPMD execution model instead of a MPMD (Multiple Program Multiple Data) execution model which is required for coupled simulations.

We think that message-passing programming models, even if they are hidden by code coupling tools, are not suitable for coupled simulations due to the reasons cited above. Instead, a more modern approach is required based on the use of objects or components. The idea is to encapsulate simulation codes into objects or components and let them communicate through middleware. Such middleware should provide a communication layer so that objects or components could be distributed on different computing resources within a Computational Grid. Such middleware already exists as for example CORBA, Java EJB or DCOM. However, they provide no support to encapsulate parallel codes in such a way that the coupling will be parallel. Indeed, with such middleware, it will be quite complex to let SPMD processes associated with one code communicate with the SPMD processors associated with another simulation code. Parallel coupling is seen important when there is a strong coupling of the codes (a huge data structure has to be transferred between simulation codes at every time step). It is therefore needed to combine different approaches from distributed computing and parallel computing.

This paper aims at showing how to combine parallel and distributed programming technologies. More precisely, it gives a method that combines SPMD (Single Program Multiple Data) with CORBA (Common Object Request Broker Architecture) without modification of the OMG standard.

The remainder of this paper is structured as follows. Section 2 gives an overview of different approaches to perform parallel computations with CORBA. Section 3 presents an approach that allows SPMD computation to be performed with standard CORBA. Section 4 provides some experimental results. Finally, we conclude in Section 5 by laying out the grounds for future works.

2. Parallel Computing with CORBA

Among a large set of distributed programming technologies, CORBA is probably the most promising one due to its object oriented approach and its independence from operating systems, languages and software vendors. CORBA is a specification from the OMG [16] (Object Management Group) to support distributed object-oriented applications. We have chosen CORBA for both technical and historical reasons. We think that it is a well mature technology with a large number of implementations. They are portable across a wide spectrum of computing resources (from a PC to a Supercomputer). Its provides both an object and a component models so that we can investigate the use of these two models for code coupling. There exists several open-source implementations so that it is fairly easy to have a CORBA implementation even on a supercomputer. However, CORBA is criticized due to some limitations: lack of asynchronous communication support, no complex data type in the IDL language and no standard binding for Fortran. Concerning the lack of asynchronous communication, this limitation has been overcome with the specification of the Asynchronous Method Invocation (AMI) in CORBA 2.4. IDL does not provide a complex data type but the definition of a new type could be done according with the ones used in numerical libraries for the C/C++ language that handle complex values. Finally, the lack of standard binding for Fortran is still valid although it has been shown, in the context of the Esprit PACHA project, that a mapping with Fortran-90 is feasible.

2.1. A Short Overview of CORBA

CORBA acts as a *middleware* that provides a set of services allowing the distribution of objects among a set of computing resources connected to a common network. As shown in Figure 1, CORBA architecture is made of several constituents that are described in the following paragraphs. The heart of the CORBA architecture is known as the Object Request Broker (ORB) which provides a communication infrastructure independent of the underlying network. A CORBA ORB includes the GIOP protocol that specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORB. The IIOP protocol specifies how GIOP messages are exchanged using TCP/IP connection. It is an interoperability standard so that different ORBs can interoperate. However, the ORB is not directly accessed by the client nor by the server. It is done through either the dynamic or the static interface. The dynamic interface is made of the DII (Dynamic Invocation Interface) at the client side and the DSI (Dynamic Skeleton Interface) at the server side. The DII allows dynamic creation and invocation of requests to objects whereas the DSI is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge

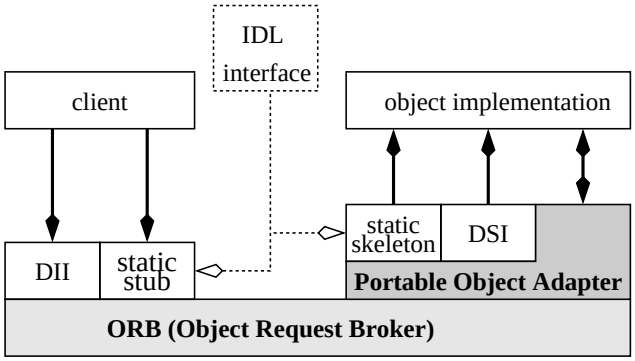


Figure 1. CORBA Architecture.

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

Figure 2. An IDL example.

of the type of the object it is implementing. Rather than using the dynamic interfaces, which are complex to use, CORBA provides a static interface thanks to the Interface Definition Language (IDL) that gives a list of allowed operations on a particular object as illustrated in Figure 2.

For a given object, an IDL file contains a list of operations (`multiply` and `skal` in Figure 2) that can be invoked remotely. It allows the definition of types (`Vector` and `Matrix`) and constants (`SIZE`) that are used in the parameter specification for each operation. An IDL compiler is in charge of generating a stub at 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. Stubs and skeletons are often tied to a CORBA implementation so they are not portable.

The last constituent of the CORBA architecture is the Portable Object Adapter (POA). It provides an interface that addresses several problems when dealing with distributed objects, such as transparent activation of objects with several policies.

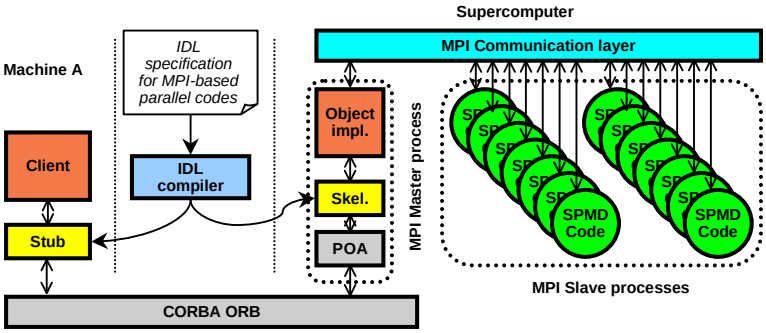


Figure 3. A master/slave approach to encapsulate MPI codes.

The first specification of CORBA appeared in end of 1991 and a first full implementation was released in 1993. As CORBA implementers gained experience since 1993, it is now a viable technology to develop high performance distributed applications. However, it is often seen as a complex technology unable to deliver high performance and thus its use for scientific applications was hampered. Latest implementations, such as OmniORB [4], are able to provide communication performance very close to message-based runtimes (such as MPI) on the same networking technology [6]. Therefore, we think that, as a distributed programming technology, CORBA can be used as a "glue" to couple several high performance simulation codes that are executed on different computing resources connected to the Internet (such as the ones in a Computational Grid). However it raises two important issues:

- the encapsulation of parallel codes into CORBA objects
- the ability of the CORBA ORB to exploit various networking technologies (such as Myrinet, SCI, Infinyband, ...) not based only on an Ethernet technology with a TCP/IP stack.

This paper addresses only the first issue. The second issue raises a more general problem that is the adaptation of various communication middleware and runtimes to a large variety of networking technologies. Recent works [6, 7] have shown that it is feasible to adapt the CORBA middleware to various networking technologies and to get really good performance.

Encapsulation of parallel codes into CORBA objects can be done with existing CORBA implementations[15]. The usual way of encapsulating such codes is to adopt a master/slave approach as shown in Figure 3. In that case, a particular process plays the role of a master that is connected to slave processes through the MPI layer. Only the master process is encapsulated into a CORBA object. Such simple solution requires some modifications to existing MPI-based codes if they did not already follow a master/slave approach. Moreover, the master may represent an important bottleneck when two MPI codes, encapsulated into CORBA objects, have to communicate with each other. The master process has to gather data from the slave processes, using MPI, and has to send them back to the callee through the ORB. The callee

```
interface diffusion {
  typedef dsequence<double,1024,(BLOCK,BLOCK)> diffusion_array;
  void diffusion(in long timestep, inout diffusion_array myarray);
};
```

Figure 4. PARDIS example.

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 overcome this problem, several attempts have already been made to extend CORBA in such a way that an object implementation can rely on a SPMD model. The first two come from research projects and the third one is from the OMG as a standardization effort. We present these attempts in Sections 2.2, 2.3 and 2.4. We do not mention research activities, such as those described in [3], that deal with the use of CORBA for parallel distributed computing. They do not tackle the problems of SPMD code encapsulation and data distribution among a collection of objects. Instead they propose to use CORBA to implement a task farming model (master-slave) instead of a message-passing library.

2.2. PARDIS: a Parallel Approach to CORBA

The PARDIS CORBA-based environment [13, 14] is one of the first attempts to allow data parallel programming within a CORBA object. PARDIS designers propose a new kind of object they call a SPMD object which is an extension of a CORBA object. SPMD objects allow the ORB to interact directly with several threads of a computation that represent a parallel application. A thread is a set of computations assign to a processing resource with its own address space. To support data distribution among different threads associated with a SPMD object, PARDIS provides a generalization of the CORBA sequence called *distributed sequence*. This new argument type requires the modification of the IDL compiler. One objective of PARDIS was to program with SPMD objects in a similar way than with standard CORBA object. Therefore, despite that the IDL syntax has been changed to add distribution extension, a SPMD object interface looks like a standard interface. Figure 5 shows a simple example of a SPMD object (object A) that is being called by a parallel application (object B) using the `diffusion` operation. The IDL interface of SPMD object A is shown in Figure 4.

In this modified IDL, the `diffusion_array` type corresponds to a distributed sequence specifying a sequence of 1024 elements of type `double` that is blockwise distributed on the client's and the server's side. As you may have noticed, this distribution specification applies to both the client's and the server's side whereas an IDL specification in the CORBA standard applies only on the server side. We think that such design choice is primitively coupled because

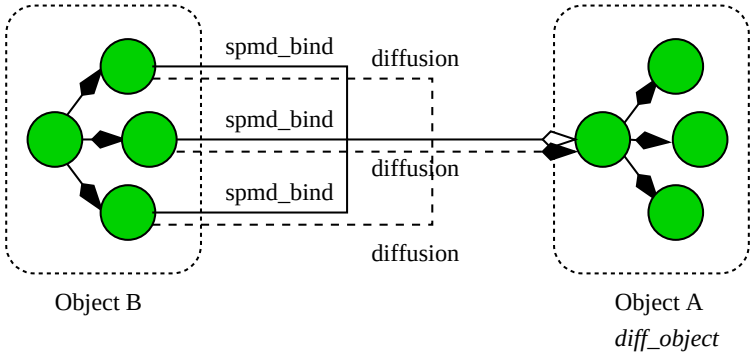


Figure 5. PARDIS approach to deal with parallel objects.

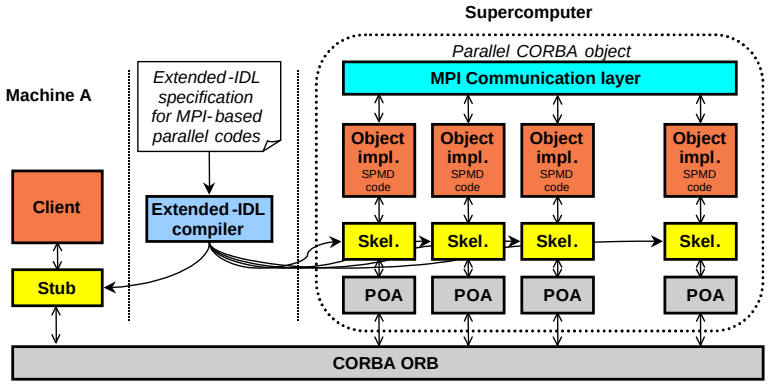


Figure 6. Parallel CORBA object.

since the IDL specification of a parallel service has to be modified accordingly to the client behavior.

Binding to a SPMD object is carried out through a specific method `spmd_bind` as shown in Figure 5. This method is a collective form of the usual `bind` method. One particular aspect of PARDIS is its ability to perform non-blocking invocations. Such asynchronous invocation mechanism is based on the returning of *future* [18] as `out` arguments of the invocation method. A *future* is a way to represent results which are not yet available. It thus allows a parallel execution of the client and the server codes.

```
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 7. Extended-IDL.

2.3. PaCO: Parallel CORBA Object

The parallel CORBA object concept (PaCO) [19, 20, 21] is another attempt for parallel programming in CORBA. A parallel CORBA object is a collection of identical standard CORBA objects as shown in Figure 6. Each CORBA object encapsulates a SPMD process of the parallel code. As our goal is to hide parallelism from the user, all the objects belonging to a collection are manipulated as a single 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 in 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. These new constructs 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 7.

All the extensions added to the IDL as well as some restrictions, like interface inheritance, are presented more in detail in [19].

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 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 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, a 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.

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 keyword `csum` indicates that the value returned by the operation is the sum of all the values given by all objects belonging to the collection.

A stub generated by the *Extended*-IDL compiler does more work than a standard stub. Indeed, it is in charge of invoking simultaneously the same operation on each object of the collection when the client is sequential. In such a case, the stub builds a request for each object belonging to the collection. If a parameter is distributed, each request contains a subset of the initial data according to the data distribution specification included in the *Extended*-IDL file associated with the collection of objects. If the client is a parallel CORBA object, the stubs are in charge of synchronizing invocations and redistributing data [21]. 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 [20].

To implement the *Extended*-IDL compiler, we modified the IDL compiler provided by MICO [1] which is a freely available[†] 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, the PaCO approach does not require any modification to the ORB. It is just a matter of stub and skeleton code generation.

2.4. Data Parallel CORBA

More recently, the OMG has adopted a specification[17] that defines the architecture for data parallel programming in CORBA. The specification addresses data parallelism as opposed to other types of parallel processing that are already possible with distributed systems, namely pipeline parallelism and functional parallelism. This specification has been produced by an active consortium of several industrial companies and a supporting organization. An implementation should be available before the end of march 2003. The proposed approach shares some similarities with the works presented in Sections 2.2 and 2.3. However, specification of data and request distributions associated with a parallel object is not performed as the expense of IDL extensions. Instead, it is included in a POA (Portable Object Adapter) policy associated with a Parallel Part Adapter (PPA) that is an extension of the POA. This approach requires a specific ORB (parallel ORB) to manage parallel objects. Figure

[†] http://www.mico.org

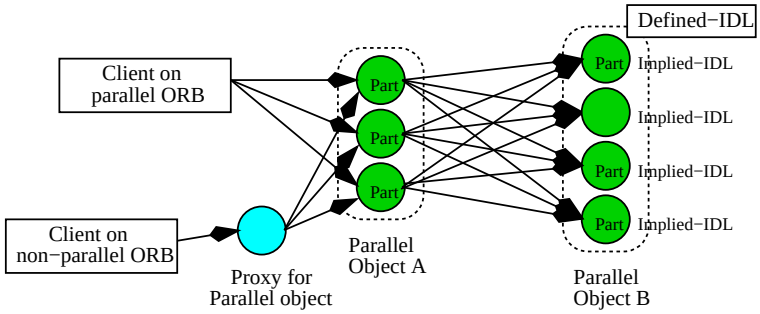


Figure 8. Data Parallel Object.

8 illustrates this approach. A parallel object (like object B) is seen as a collection of identical part objects. An IDL specification (called Defined-IDL) is associated with the parallel object and each part object is assigned with another IDL file (called Implied-IDL). This Implied-IDL is derived automatically from the Defined-IDL. As mentioned early, neither the Defined-IDL nor the Implied-IDL contain any data distribution specification. Instead the part object implementation contains such specification. Collective invocation to operations is performed by the parallel-ORB. Calling an operation to a parallel object from a standard ORB requires the use of a proxy object that aims at performing a bridge between the two different ORBs.

2.5. Discussion

In the previous three approaches, adding support for parallel processing within CORBA requires some modifications to the actual standard. These extensions concern either the IDL language (PARDIS, PaCO) or the ORB itself (Data Parallel CORBA). These approaches are unsatisfactory. Modifications of the IDL syntax require a new IDL compiler that is always dependent of a CORBA implementation. Therefore, such approach is not portable. Concerning the third approach, that is being investigated in a standardization process at the OMG, we think that the proposed approach is rather complex because it lets the implementers and the users of a parallel service handle manually distributed data. Moreover, even if the proposed approach is approved by the OMG, there are serious doubts that such extensions will be provided by numerous existing CORBA implementations.

We propose thus another approach. Our current work aims at incorporating SPMD programming within CORBA without modifying the standard. However, we constrain our work in such a way that it does not entail a loss of performance compared to those approaches that require modifications to CORBA and it has to be as transparent as possible to the users. The following Sections give a detailed description of this approach.

```
#include "Matrix.idl"

interface IExample {
    void send_data(Matrix m);
}
```

Figure 9. IDL interface of the parallel object (Matrix.idl is presented in Figure 13).

```
void f(long* A, int size, int procid, int nbproc) {

   CORBA::Object_var obj = naming_context->resolve_str("IExImpl");
   IExample iex = IExample::_narrow(obj);

   Distribution d0(Matrix::BLOCK, procid, nbproc); // Source distribution
                                                   // descriptor

   Matrix<long> data(1);         // create a Matrix of 1 dimension
   data->setBounds(0,1,size);    // bounds [1,size] for dimension 0
   data->setDistribution(0,d0); // set distribution for dimension 0
   data->setData(A);             // set data to yet allocated data (no copy)
   iex->send_data(data);         // remote method invocation
}
```

Figure 10. Motivating Example: a parallel client calls a method on a parallel server. The data are block-distributed into the client. Nothing about the server has to be known.

3. PaCO++: Portable Parallel CORBA Objects

Parallel CORBA objects are defined as a collection of identical CORBA objects. They aim at providing parallelism support to CORBA. Obviously, CORBA objects of a collection are assumed to work together. They are expected to communicate thanks to an external mechanism, like for example MPI. This work targets parallel CORBA objects on top of compliant CORBA ORBs without involving whatsoever modification of the CORBA specifications. We call such objects portable parallel CORBA objects or PaCO++ objects. Throughout this Section, we introduce PaCO++ objects with respect to a motivating example.

3.1. Motivating Example

Figure 9 presents the user level IDL interface of the motivating example presented in Figure 10. A parallel client wants to send an array `A` to a method `void send_data(Matrix m)` of the interface `IExample`. The client knows that this service is implemented by an object registered

into the naming service under the name `IExImpl`. But, the client does not know – and does not want to know – that the implementation is in fact parallel. To connect to the object, the client instantiates a local object `obj` of type `IExample` as usual in CORBA. Then, once the `Matrix` view of its local array `A` is built, the method is invoked. Given that the client is parallel, it has to declare the distribution of its data.

3.2. Achieving Portable Parallel CORBA Objects

To implement this kind of example on top of a compliant CORBA ORB, we choose to introduce a layer between the user code and the ORB, as depicted in Figure 11. This layer, called the parallel CORBA layer, embeds the complexity of connection and data distribution management. It translates user-level CORBA calls (interface *Interface1*) into private CORBA calls (interface *ManagerInterface1*). This latest interface contains operations derived from the operations defined by the user as well as some private operations. The derived operations differs from the original operations on the argument types: the type of distributed argument is changed to an intermediate *Matrix* type. The private operations provide services like the localization of all remote objects being part of the implementation of `IExample` and the retrieval of the data distribution of arguments of user-level operations.

The client and server side of the parallel CORBA layer are analog to the stub and the skeleton of ORB requests. But, while stubs and skeletons of ORB requests deal with point-to-point issues (like data marshaling), the stub and skeletons of the parallel CORBA layer handle data distribution issues, like redistribution and synchronization. Finally, the stubs and the skeletons of the parallel CORBA layer should be generated from an IDL level description of the user services.

The remaining of this section gives more information about the connection management, the operation invocation, the data distribution management, the intermediate Matrix type and interoperability.

Connection Management

The entry point of a PaCO++ object is its manager. The manager provides special functionalities for parallel-aware clients. It is the IOR (Interoperable Object Reference[‡]) of the manager which is registered in the naming service under the name of the PaCO++ object as shown in Figure 12.

The main functions of the manager for parallel aware clients are a function that returns the number of CORBA objects in the collection of the PaCO++ object and a function that returns the IORs of all these objects. These IORs may be stored in the naming service in a context whose named is related to the name of the PaCO++ object.

When a parallel client resolves a reference to a PaCO++ object from the naming service, it gets a reference to the manager. Then, the call to the `narrow` function is intercepted by the

[‡]It is an object reference that is understood by ORBs that can interoperate using the OMG defined protocols such as GIOP/IIOP

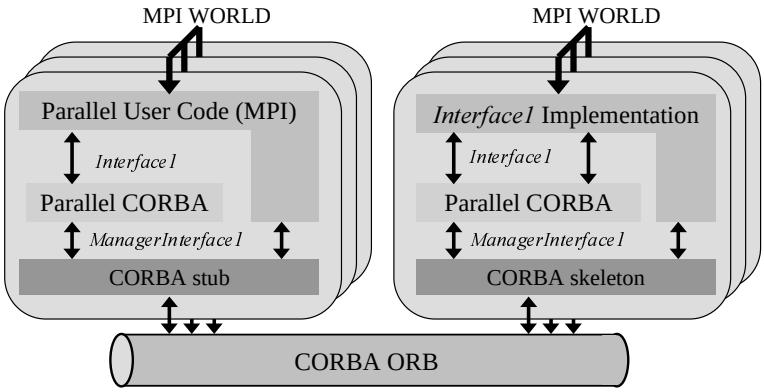


Figure 11. PaCO++ objects. The client and the server in this example are two MPI based codes only connected by CORBA. Each code runs inside its own MPI world.

PaCO++ layer in order to collect the IORs of the objects implementing the collection. All this information is stored in a local object, implementing the `ManagerIExample` interface, whose reference is returned to the user code.

For parallel unaware clients, the manager acts as a proxy. Thus, parallel-unaware clients can invoke operations on a PaCO++ object.

Operation Invocation

When the client invokes the `send_data` operation of the example, it in fact calls the corresponding operation of the `ManagerIExample` interface, locally implemented into the PaCO++ layer. This operation builds CORBA requests according to the data distributions expected by the parallel objects. Such information is available thanks to operations belonging to the `ManagerIExample` Interface. Then, it invokes CORBA requests to the `ManagerIExample` objects of the different nodes. The role of the server side operation is to wait for all data coming from a parallel client before calling the server side implementation of the `send_data` operation. Similarly, it sends back the `out` arguments to the different nodes of a parallel client.

When a client invokes an operation of a parallel object, it potentially has to send several CORBA requests. An efficient and reliable solution would be the use of the Asynchronous Message Interface that appears in CORBA 2.4. As most open source ORBs do not support this feature, we implement a temporary solution based on *oneway* requests. This solution has severe limits. First, in general it is not a satisfactory solution as such kind of requests are not reliable according to the CORBA specifications. But, some ORBs, like OmniORB 3 [4], implement reliable *oneway* requests. Second, we have to build a system to detect the termination of the request. The general scheme is that the stubs in the client code set up a CORBA interface to be notified of the termination of the parallel requests.

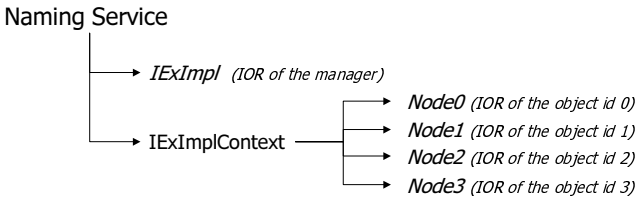


Figure 12. A PaCO++ object of name *IExImpl* registers the IOR of its manager object in the naming service under its name. It may also create a context of name *IExImplContext* which contains the IOR of all CORBA objects of the collection.

Data Distribution Management

The core of parallel objects is the data distribution management. From our experience, mainly derived from PaCO and High Performance FORTRAN[8], we believe it is important to have a high level of transparency: our choice is to separate the data distribution from the interface. The interfaces are specified into an IDL file while an auxiliary (XML) file specifies which operations are parallel and which parameters are distributed.

By decoupling the data distribution from the interface, we obtain four major benefits. A first benefit is that the CORBA IDL does not need to be modified. That is one of the major drawbacks of the previous works on PaCO objects. The second benefit is that argument data distribution is transparent to the user, as distribution does not appear in the interface. But, it is not transparent to the implementer of a service as she/he has to specify it. A third benefit is that a parallel object may dynamically change the distribution of its interface. The auxiliary XML file is not required to specify which distribution is expected for each distributed parameter. In this case, the server has to specify the distribution. A protocol is then needed between the client and the server to inform the client with the expected data distribution. This protocol needs yet to be defined. This feature implies some issues. For example, how is the client informed? A solution would be to use a listener design pattern. A second issue is: what does a parallel object do with incoming requests that have an argument with an old distribution? If all the data has correctly been received, a redistribution may be performed. However, whenever some data are missing (because of a node failure) or the parallel object does not implement the redistribution feature, a CORBA exception is returned to the client. How a node failure is supported by the server is out of scope of PaCO++ objects. The fourth benefit is the ease of the introduction of new data distribution as only clients and parallel objects that use non standard data distributions have to know about them.

Intermediate Matrix Type

Applications are expected to manipulate their own data distribution. In the general case, the data distribution in the client is different from the data distribution in the server. In order to be able to automatically redistribute the data from the client to the server, we choose

```
interface Matrix {

  struct dim_t { long size, low, high; };

  struct matrix {
    dis_t           dis;   // current distribution
    long            ndim;  // number of dimension
    sequence<dim_t> rdim;  // global view of the array
    sequence<dim_t> ddim;  // local  view of the array
    data_t          data;  // data
  };
};
```

Figure 13. IDL distributed array representation.

```
Matrix<float> data(2);          // matrix with 2 dimension

data.setBounds(0,0,size1);      // Set bounds for dimension 0
data.setBounds(1,0,size2);      // Set bounds for dimension 1
Distribution d0(Matrix::BLOCK, procid, nbproc);
Distribution d1(Matrix::SEQ);
data.setDistribution(0, d0);    // Set distribution for dimension 0
data.setDistribution(1, d1);    // Set distribution for dimension 1
data.allocateData();            // Allocate memory

for( int i0 = data.low(0); i0 < data.high(0); i0++ )
  for( int i1 = data.low(1); i1 < data.high(1); i1++ )
    data(i0, i1) = ...
```

Figure 14. C++ server side example: initialization of a 2D distributed array of floats which has a block-distributed dimension. The indexes `i0` and `i1` are global.

to require the distributed data to be mapped to a `Matrix` interface. This interface, sketched in Figure 13, provides a logical API to export/import distributed data from/to the CORBA space. This API should be easy to use for a client (like in the example shown in Figure 10) and should provide functionalities for implementers. Internally, the `Matrix` interface manages an IDL structure that contains distribution information as well as user data.

Currently, we only implement the `Matrix` interface as a C++ class whose API provides methods that manage a C++ representation of the IDL `Matrix` structure. While Figure 10 has provided a client side example, Figure 14 presents a server side example that illustrates the initialization of a 2D distributed array.

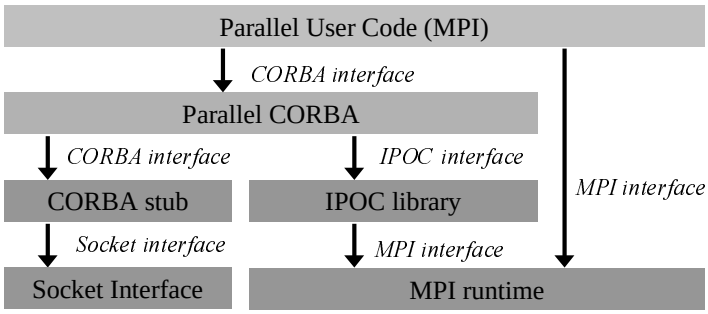


Figure 15. PaCO++ objects implementation relies on an internal communication interface to be independent of the network middleware.

Intra PaCO++ Object Communications

The PaCO++ layer requires functionalities like barriers, reductions, etc. These functions are grouped into an interface called IPOC (Internal PaCO++ Object Communication). The functions are needed to ensure the semantics of CORBA operations. For example, in the default case, the server side invocation of an operation has to be synchronous between all servers representing the PaCO++ object. A barrier is thus needed.

PaCO++ objects target to be independent from the parallel communication middleware locally available. The idea is to define a mapping of the internal PaCO++ object communication library to the different middleware. As we expect to keep its interface small, the IPOC interface should remains small and straightforward to implement on top of various communication middleware.

Figure 15 presents an example of instantiation. CORBA requests are transmitted over TCP/IP thanks to the socket interface. The IPOC module is compiled to use MPI calls.

Interoperability

In order to keep the interoperability between different ORBs, it is important not to modify the GIOP protocol. Our proposal is to achieve *portable* parallel CORBA objects by inserting a layer between the user code and the ORB runtime. Thus, the parallel layer generates standard CORBA requests. Thus, as GIOP is not modified, the interoperability is maintained.

4. Preliminary Experiments

The goal of this Section is to evaluate the performance of the PaCO++ objects on basic situations. First, we use a sequential client connected to a parallel object. Then, we connect a parallel client to a parallel object. All CORBA objects belonging to a PaCO++ object are

Table I. Performances of Mico and OmniORB ORBs for a sequential client connected to a PaCO++ object (2 objects) over Fast Ethernet.

	Version 1 - Explicit data copy			Version 2 - No explicit data copy		
	Mico	Mico patch	OmniORB	Mico	Mico patch	OmniORB
Building (ms)	267	250	284	103	2.80	2.93
Sending (ms)	1020	1003	861	986	1005	863
Total (ms)	1288	1253	1156	1090	1008	866
Sending (MB/s)	9.80	9.97	11.61	10.14	9.95	11.59
Total (MB/s)	7.76	7.98	8.65	9.17	9.92	11.55

located on different machines. In the first part of the Section, the parallelism is limited to two nodes because experiments focus on the overhead generated on a node. In the second part, though we know that aggregated performance is possible [21], we present experiments involving two clusters of eight nodes connected by a WAN, VTHD a gigabit wide-area network. We finish by showing the performance of PaCO++ object through a high performance network. All the codes have been hand-written.

4.1. Basic Experiments

We perform experiments for two versions of the PaCO++ object layer. Version 1 does explicit data copy when creating CORBA requests while Version 2 uses sequence data constructor available in the C++ mapping of sequences.

An important goal is to have *portability*. Thus, we experiment with two different ORBs: Mico 2.3.4 [1] and OmniORB 3 [4]. Since Mico 2.3.4 performs a copy when used with sequence data constructor, we remove this (unnecessary) copy by patching the unbounded sequence C++ template of Mico 2.3.4. We reference this patched Mico version as "Mico patch". We do not modify OmniORB 3 because it does not copy data in sequence data constructors. Both the ORBs and test programs have been compiled with the compiler optimization turned on. The compilers are gcc/g++ 2.95.2. The test platform is a PC cluster. The nodes are dual-processor Pentium II 450 Mhz with 256 MB memory. The network is a standard Fast Ethernet (100 Mb) and the communication protocol is TCP/IP. The operating system is Linux 2.2.13.

The experiments presented in Table I involve a sequential client transferring an array to a parallel object. The performance is presented for the PaCO++ objects with Mico 2.3.4, Mico 2.3.4 patch and OmniORB 3. The first row of the table represents the request building time, the second row is the sending time and the third row is the whole time of the operation, which is very close of the building time plus the sending time. The fourth and the fifth rows present the data bandwidth of the sending row and of the total row.

As shown in Table I, the building time leads to a huge overhead when there are data copies. The use of a sequence data constructor improves performance. But, a zero-copy sequence data constructor allows a more important decrease of the request building time (divided by 100): the bandwidth is improved by 24 % for Mico patch and by 33 % for OmniORB.

Table II. Performances of Mico and OmniORB ORBs for a parallel client (2 objects) connected to a parallel object (2 objects) over Fast Ethernet. No data redistribution.

	Version 1 - Explicit data copy			Version 2 - No explicit data copy		
	Mico	Mico patch	OmniORB	Mico	Mico patch	OmniORB
Building (ms)	129	117	141	50	0.27	0.25
Sending (ms)	547	508	432	544	518.6	431.5
Total (ms)	676	625	574	593	519.2	432.1
Sending (MB/s)	9.14	9.84	11.57	9.19	9.64	11.59
Total (MB/s)	7.39	8.00	8.71	8.43	9.63	11.57

The experiments presented in Table II are for a parallel client invoking an operation on a parallel object. We observe that a strategy based on sequence data constructor leads to better performance. The use of zero-copy data constructor leads again to better performance. The reason why the overhead is so small is we really re-use the buffer of the incoming request (forward) and so there are no creation of new sequences. The building time in Version 2 is negligible with respect to the communication time.

4.2. Comparison with PaCO Performance

With PaCO, we perform experiments similar to those of Section 4.1. We used the last available version which is based on Mico 2.3.3. We obtain 8.77 MB/s for the sequential client and 8.51 MB/s for the parallel client. When compared to Table I and Table II, one can see that performance is similar and depends mostly on the performance of the underlying ORB. Thus, PaCO++ objects are as efficient as PaCO objects.

4.3. VTHD Experiments

We have access to the VTHD network, a wide area network. It is an experimental network of 2.5 Gbit/s that, in particular, interconnects two INRIA research units, which are about one thousand kilometers apart. In a point-to-point situation using OmniORB we measure a throughput of 11 MB/s; the Ethernet 100 Mb/s card is the limiting factor. For the experiments that use an 8-node parallel client and an 8-node parallel object, we measure an aggregated bandwidth of 85.7 MB/s, which represents a point-to-point bandwidth of 10.7 MB/s. PaCO++ objects prove to efficiently aggregate bandwidth.

4.4. High Performance Network Experiments

The last series of experiment focus on intra cluster experiments: we use a cluster of 16 dual-processors machines interconnected with both a fast Ethernet network and a Myrinet-2000 networks. The processors are 1 GHz Pentium III and the operating system is Linux 2.2.18.

The same experiments described in Section 4.1 were performed: an 8-node parallel client is connected to an 8-node parallel object. Each parallel code internally performs its MPI communications through the Myrinet network.

When the CORBA communications are done through the Fast Ethernet network, we measure an aggregated bandwidth of 91 MB/s which leads to an average of 11.4 MB/s point-to-point bandwidth. These results are similar of those obtained with VTHD.

In the second experiment, both MPI and CORBA communications use the Myrinet network. Supporting CORBA and MPI, *both running simultaneously*, is not straightforward. Access conflicts for high performance networking resources or multithreading are likely to arise. PadicoTM [6, 7] is our research platform that offers a framework dealing with high performance communication and multithreading issues: it allows different middleware systems like MPI and CORBA to efficiently and cooperatively share the same network. PadicoTM-enabled implementation of CORBA (based on OmniORB) reports 240 MB/s bandwidth on Myrinet-2000 and 20 μs latency. Thanks to PadicoTM, PaCO++ object communications benefit from the high performance network. From a 8-node parallel client to an 8-node parallel object, we measured an aggregated bandwidth of 1.5 GB/s which represents an average of 187 MB/s point-to-point bandwidth.

5. Conclusion

Thanks to the continuous improvement of networks, Computational Grids are becoming more and more popular. Some Grid Architectures, like Globus, provide a parallel programming model, which does not appear well suited for certain applications, for example coupled simulations. For such applications, we advocate a programming model based on a combination of parallel and distributed programming models.

CORBA has proved to be an interesting technology. However, its lack of support for the encapsulation of parallel codes into objects makes it not suitable for scientific computations. Previous works on parallel CORBA objects [13, 19] have required modifications of CORBA specifications. In this paper, we have shown that it is feasible to define parallel CORBA objects on top of CORBA compliant ORB without modification of the IDL. As we do not modify CORBA specifications, we need to introduce a layer between the user code and the ORB to handle data distribution issues. Thanks to this layer, we can achieve data distribution transparency at the client side while allowing parallel objects to dynamically change the expected data distribution of their operation arguments. Experiments show that the overhead of this layer is very small. Efficiency relies on the zero-copy sequence data constructor and on the efficiency of the communications of the ORB. Also, contrary to a belief, the experiments show that current CORBA implementation can be very efficient.

Future work will concern the definition of interfaces related to parallel objects that we have just sketched in this paper. A second direction is to further study the issue of dynamic modification of data distribution.

Acknowledgments

We would like to thank the reviewers for their comments that helped to strengthen this article.

REFERENCES

1. Puder A. The MICO CORBA Compliant System. *Dr Dobb's Journal*, 23(11):44–51, November 1998.
2. R. Ahrem, M.G.Hackenberg, U. Karabek, P. Post, R. Redler, and J. Roggenbuck. Specification of MpCCI. Technical Report 12, GMD-SCAI, Sankt Augustin, Germany, 2001.
3. Markus Aleksy and Axel Korthaus. A CORBA-based object group service and a join service providing a transparent solution for parallel programming. In *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 123–134, Limerick, Ireland, June 2000. IEEE.
4. AT&T Laboratories Cambridge. OmniORB Home Page. http://www.omniorb.org.
5. F.O. Bryan, B.G. Kauffman, W.G. Large, and P.R. Gent. The NCAR CSM flux coupler. Technical Report TN-425+STR, National Center for Atmospheric Research, Boulder, Colorado, USA, 1996.
6. A. Denis, C. Pérez, and T. Priol. Towards high performance CORBA and MPI middlewares for grid computing. In Graig A. Lee, editor, *Proc of the 2nd International Workshop on Grid Computing*, number 2242 in LNCS, pages 14–25, Denver, Colorado, USA, November 2001. Springer-Verlag.
7. A. Denis, C. Pérez, and T. Priol. PadicoTM: An open integration framework for communication middleware and runtimes. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID2002)*, pages 144–151, Berlin, Germany, May 2002. IEEE.
8. High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, October 1996. Version 2.0.
9. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
10. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infracstructure*. Morgan Kaufmann Publishers, Inc, 1999.
11. E. Gabriel, M. Resch, T. Beisel, and Rainer Keller. Distributed computing in a heterogenous computing environment. In *EuroPVM/MPI'98*, Liverpool/UK, September 1998.
12. A. S. Grimshaw, W. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 1(40):39–45, January 1997.
13. K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *6th International Symposium on High Performance Distributed Computing (HPDC '97)*, pages 31–39, Portland, Oregon, USA, August 1997. IEEE.
14. K. Keahey and D. Gannon. Developing and Evaluating Abstractions for Distributed Supercomputing. *Cluster Computing*, 1(1):69–79, May 1998.
15. M. Li, O.F. Rana, M.S. Shields, and D.W. Walker. A wrapper generator for wrapping high performance legacy codes as Java/CORBA components. In *Supercomputing'2000*, Dallas, Texas, USA, November 2000.
16. Object Management Group. The Common Object Request Broker: Architecture and Specification (Revision 2.2), February 1998.
17. Object Management Group. Data parallel CORBA, November 2001. ptc/01-11-09.
18. W.G. O'Farrell, F.Ch. Eigler, S.D. Pullara, and G.V. Wilson. *Parallel Programming Using C++*, chapter ABC++. MIT Press, 1996.
19. T. Priol and C. René. Cobra: A CORBA-compliant Programming Environment for High-Performance Computing. In *Euro-Par'1998: Parallel Processing*, volume 1470 of *Lect. Notes in Comp. Science*, pages 1114–1122, Southampton, UK, September 1998. Springer-Verlag.
20. C. René and T. Priol. MPI code encapsulating using parallel CORBA object. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, Redondo Beach, Californie, USA, August 1999. IEEE.
21. Kamachi T., Priol T., and René C. Data distribution for parallel CORBA objects. In *Euro-Par'2000: Parallel Processing*, volume 1900 of *Lect. Notes in Comp. Science*, pages 1239–1250, Munchen, Germany, August 2000. Springer-Verlag.
22. A. Tanenbaum. *Distributed Operating System*. Prentice Hall, 1994.
23. S. Valcke, L. Terray, and A. Pacentini. *OASIS 2.4: Ocean Atmosphere Sea Ice Soil User's Guide*. CERFACS, Toulouse, France, 2000.