Portable Parallel CORBA Objects: an Approach to Combine Parallel and Distributed Programming for Grid Computing

Alexandre Denis, Christian Pérez, Thierry Priol

To cite this version:

Alexandre Denis, Christian Pérez, Thierry Priol. Portable Parallel CORBA Objects: an Approach to Combine Parallel and Distributed Programming for Grid Computing. 7th International Euro-Par Conference (EuroPar 01), 2001, Manchester/UK, United Kingdom. Springer, 2150, pp.835-844, 2001, Lecture Notes in Computer Science. <inria-00000131>

HAL Id: inria-00000131

https://hal.inria.fr/inria-00000131

Submitted on 24 Jun 2005

Portable Parallel CORBA Objects: an Approach to Combine Parallel and Distributed Programming for Grid Computing

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

[1] IRISA/IFSIC,
[2] IRISA/INRIA,
Campus de Beaulieu – 35042 Rennes Cedex, France
Contact: {Alexandre.Denis, Christian.Perez, Thierry.Priol}@irisa.fr.

Abstract With the availability of Computational Grids, new kinds of applications that will soon emerge will raise the problem of how to program them on such computing systems. In this paper, we advocate a programming model that is based on a combination of parallel and distributed programming models. Compared to previous approaches, this work aims at bringing SPMD programming into CORBA. For example, we want to interconnect two MPI codes by CORBA without modifying MPI or CORBA. We show that such an approach does not entail any loss of performance compared to previous approaches that required modification to the CORBA standard.

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 [3, 4]. 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[14] 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 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). It also shares 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[3] or Legion[5], aim at providing runtime systems to allow the execution of applications on Computational Grids. However, Globus was mainly designed to allow the execution of parallel applications. Such approach makes senses 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. It aims at coupling several parallel codes to simulate complex systems that require a multi-physics approach. Therefore, one important question arises when using a grid system: what is the most appropriate approach to program a Computational Grid, or said differently, what programming models have to be provided to Grid application designers ? On that matter, there is no consensus mainly due to the wide nature of applications that could benefit from Computational Grids. Since such systems are a combination of parallel and distributed systems, it is very tempting to extend programming models that were associated to parallel systems (message passing libraries, shared memory) so that they can be used for distributed programming. Similarly, programming models for distributed systems (remote procedure call, distributed objects) can be adapted to program parallel systems. Neither of these two approaches can be seen as viable solutions for the future of Grid Computing. It is thus important to try to combine the two different worlds into a single coherent one. Such a programming model will have to give an answer to the design issues already mentioned: transparency, interoperability, reliability, scalability and performance.

This paper aims at showing how two 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.

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 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 and languages. CORBA is a specification from the OMG (Object Management Group) to support distributed object-oriented applications. 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. Transparent 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) that gives a list of allowed operations on a particular object. 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. However, CORBA lacks of supporting efficiently the encapsulation of parallel codes. 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 PARDIS CORBA-based environment [7, 8] 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 SPMD object which is an extension of a CORBA object. To support data distribution among different threads associated with a SPMD objects, PARDIS provides a generalization of the CORBA sequence called *distributed sequence*. This new argument type requires the modification of the IDL compiler. PARDIS provides a mechanism to invoke operations on objects asynchronously based on the *future* concept. A *future* is the basic mechanism to get the results of services activated asynchronously.

The PaCO CORBA-based environment [11, 13, 6] is another attempt for parallel programming in CORBA. We introduced the concept of parallel CORBA object as a collection of identical CORBA objects. It aims at encapsulating a MPI code into CORBA objects so that a MPI code can be fully integrated into a CORBA-based application. Execution of parallel CORBA objects is based on the SPMD execution model. Data distribution between the objects belonging to a collection is entirely handled by the system. However, to let the system carry out parallel execution and data distribution between the objects of the collection, some specifications have been added to the object interface. A parallel object interface is thus described by an extended version of IDL, called Extended–IDL. It is a set of new keywords, added to the IDL syntax[1], to specify the number of objects in the collection, the shape of the virtual node array where objects of the collection will be mapped, the data distribution modes associated with parameters and the collective operations applied to parameters of scalar types.

More recently, the OMG has issued an RFP[10] (Request For Proposal) that solicits proposals to extend CORBA functionality to conveniently and efficiently support parallel processing applications. A response[9] was submitted by a consortium of several industrial companies and a supporting organization. The proposed approach shares some similarities with previous works ([7, 11]). However, specification of behaviors of parallel objects (data and request distributions) is not performed thanks to 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. 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.

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 or the ORB itself. There are serious doubts that such extensions will be provided by numerous existing CORBA implementations. Our current work is aiming at incorporating SPMD programming

[1] A more complete description of these extensions is given in [11, 13]

```
    #include "Matrix.idl"

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

Figure 1. IDL interface of the parallel object

```
void f(long* A, int size) {
   IExample obj("Servant");
   Matrix<long> data(1);      // create a Matrix of 1 dimension
   data->setBounds(0,1,size); // bounds [1,size[ for dimension 0
   data->setData(A);          // initialize data pointer (no data copy)
   obj->send_data(data);      // remote invocation
}
```

Figure 2. Motivating Example: a sequential client calls a parallel method.

within CORBA without modifying the standard. It does not entail a lost of performance compared to those approaches that require modifications to CORBA and it has to be easy to use.

3 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. Throughout this section, we discuss with respect to a motivating example.

3.1 Motivating Example

Figure 1 presents the user level IDL interface of the motivating example presented in Figure 2. A sequential 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 of named `Servant`. 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` with the name of the remote object as argument. Then, once the Matrix view of its local array `A` is built, the method is invoked.

3.2 Achieving Portable Parallel CORBA Objects

To implement this kind of example on top of a compliant CORBA ORB, we need to introduce a layer between the user code and the ORB, as depicted in

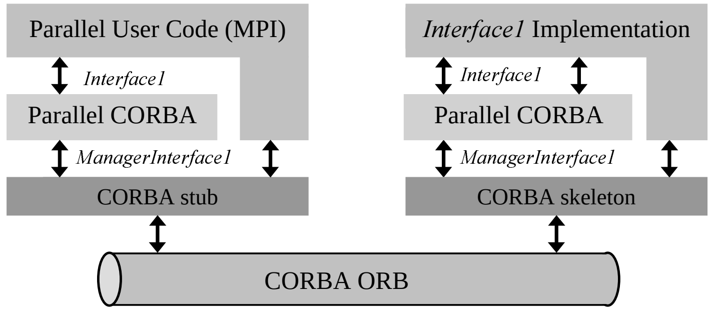

Figure 3. Portable parallel CORBA objects

Figure 3. This layer embeds the complexity of connection and data distribution management. Its main role is to map an user-level interface – IExample in the example – to an IDL interface, that is called ManagerIExample. This latest interface contains the methods define by the user as well as private methods. The private methods 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 methods.

The client and server side methods of the parallel CORBA object layer are analog to the stub and the skeleton of ORB requests. But, while stubs and skeletons of ORB requests deal with peer-to-peer issues (like data marshaling), the stub and skeletons of the parallel CORBA object layer concentrate on data distribution issues. Finally, the stubs and the skeletons of the parallel CORBA layer should be generated from an IDL level description of the user services. However, they are currently hand-written.

The rest of this section reviews different aspects of the internals.

Connection Management. A parallel object is defined by a name (string). This name in fact represents a context in the Naming Service that contains two kind of entries: the IOR of the service manager and all the IOR of the objects that belongs to the parallel objects, as illustrated in Figure 4. The constructor of IExample retrieves information like the number of objects thanks to the Manager object. Then, it can collect their respective IOR from the Naming Service.

Method Invocation. When the client invokes the send_data method, it in fact calls the corresponding method of the ManagerIExample interface, locally implemented into the parallel CORBA layer. This method builds CORBA requests according to the data distributions expected by the parallel objects. Such information is available thanks to methods belonging to the ManagerIExample Interface. Then, it sends the CORBA requests to the ManagerIExample objects. The role of the server side method is to gather data coming from different clients (when the client is parallel) before calling the server side implementation of the send_data method. Similarly, it scatters the out arguments.

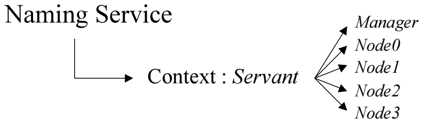

Figure 4. A parallel CORBA object of name *Servant* registers all its CORBA objects in the naming service under the context of same name. This context also contains the IOR of the manager and the IOR of all objects.

When a client invokes a method 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 we are not aware of open source ORB that supports this feature, we implement a temporary solution based on *oneway* requests. This solution has severe limits. First, it is not a reliable solution as such kind of requests are not reliable according to the CORBA specifications. But, as we used TCP to transport CORBA requests, all *oneway* requests are delivered. Second, we have to build a system to detect the termination of the request.

Data Distribution Management. The core of parallel objects is the data distribution management. From our experience, mainly derived from PaCO and High Performance FORTRAN[2], we believe its important to have a high level of transparency: our choice is to separate the data distribution from the interface. By decoupling the data distribution from the interface, we obtain four major benefits. A first benefit is there is no need to modify the CORBA IDL. The second benefit is that argument data distribution is transparent to the user, as distribution does not appear in the interface. A third benefit is that a parallel object can dynamically change the distribution pattern it is awaiting. This may happen for example if some objects are removed (due to node failure for example) or some objects are added. This feature implies some interesting 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 is missing (node failure) or the parallel object does not implement the redistribution feature, a CORBA exception is returned to the client. The fourth benefit is the ease of the introduction of new data distribution patterns as only clients and parallel objects that use non standard data distributions have to know about these.

Intermediate Matrix Type. Applications are expected to be written with their own data distribution scheme. So, we face the problem of embedding user data into a standard IDL representation so as to provide interoperability. We achieve data distribution interoperability thanks to a `Matrix` interface, sketched in Figure 5. It provides a logical API to manipulate an internal IDL representation

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

of data distributions. This API should be straightforward for client (like in the example of Figure 2) and should provide functionalities for implementers. Internally, the `Matrix` interface manages an IDL structure that contains distribution information as well user data. That's this structure which is sent through the ORB.

Currently, we only implement the `Matrix` interface as a C++ class whose API provides methods that manages a C++ representation of the IDL `Matrix` structure. While Figure 2 has provided a client side example, Figure 6 presents a server side example that illustrates the initialization of a 2D distributed array.

4 Preliminary Experiments

The goal of this section if to evaluate the performance of the portable parallel CORBA objects on basic situations: a client connected to a parallel object. 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 parallel CORBA object are located on different machines. For most experiments, we limit the parallelism to two nodes. We concentrate on the overhead generated on a node as we know that aggregated performance is possible [6]. However, we finish this section by presenting experiments involving two clusters of height nodes connected by VTHD, a gigabit wide-area network.

	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

Table 1. Performances of Mico and OmniORB ORBs for a sequential client connected to a parallel CORBA object (2 objects) over Fast Ethernet.

4.1 Basic Experiments

We perform experiments for two version of the portable parallel CORBA object layer. Version 1 does explicit data copy when creating CORBA requests while Version 2 uses sequence data constructor.

An important goal is to have *portability*. So, we experiment two different ORBs: Mico 2.3.4 [12] and OmniORB 3 [1]. As 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 as it does not copy data in sequence data constructors. The ORBs have been compiled for speed as well as the test programs. 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 1 are for a sequential client transferring an array to a parallel object. The performances are presented for the portable parallel CORBA objects with Mico 2.3.4, Mico 2.3.4 patch and OmniORB 3. The first row of the table represents the building time (computing part), the second row the sending time and the third row 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 the total row.

As shown in Table 1, the building time leads to a huge overhead when there are data copies. The use of sequence data constructor improve performances. But, the use of a zero-copy sequence data constructor allow a more important decrease of the building time (divided by 100). The consequence is an bandwidth improvement of 24 % for Mico patch and of 33 % for OmniORB.

The experiments presented in Table 2 are for a parallel client invoking a method 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 performances. 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.

	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

Table 2. 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.

4.2 VTHD Experiments

Very recently, we had access to the VTHD network. It's an experimental network of 2.5 Gb/s that in particular interconnects two INRIA laboratories, which are about one thousand kilometers apart. In a peer-to-peer situation using OmniORB we measure a throughput of 11 MB/s; the Ethernet 100 Mb/s card being the limiting factor. For experiments with 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. Portable CORBA parallel objects prove to efficiently aggregate bandwidth.

4.3 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 1 and Table 2, one can see that performances are similar and depend mostly on the performance of the underlying ORB. So, the portable parallel CORBA objects are as efficient as parallel CORBA objects of PaCO.

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, as it does not handle parallelism, there is a clear need of parallel CORBA objects when interconnecting for example two MPI parallel codes. Previous works on parallel CORBA objects [7, 11] 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 method arguments. Experiments show that the overhead of this layer is very small. Efficiency relies on the no-copy sequence data constructor and on the efficiency of the communications of the ORB. Also, contrary to a belief, the numbers show that current CORBA implementation can be very efficient on Ethernet networks.

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 issue with dynamic modification of data distribution. Note that distributions are always decided by the server side application.

References

[1] AT&T Laboratories Cambridge. OmniORB Home Page. http://www.omniorb.og.

[2] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, October 1996. Version 2.0.

[3] 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.

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

[5] 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.

[6] T. Kamachi, T. Priol, and C. René. Data distribution for parallel corba objects. In *EuroPar'00 conference*, August 2000.

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

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

[9] Mercury Computer Systems, Inc. and Objective Interface Systems, Inc. and MPI Software Technology, Inc. and Los Alamos National Laboratory. Data Parallel CORBA - Initial Submission, August 2000.

[10] Object Management Group. Request For Proposal: Data Parallel Application Support for CORBA, March 2000.

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

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

[13] 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, August 1999.

[14] A. Tanenbaum. *Distributed Operating System.* Prentice Hall, 1994.