Cobra: A CORBA-compliant Programming Environment for High-Performance Computing

Thierry Priol and Christophe René

IRISA -Campus de Beaulieu - 35042 Rennes, France

Abstract. In this paper, we introduce a new concept that we call a parallel CORBA object. It is the basis of the *Cobra* runtime system that is compliant to the CORBA specification. *Cobra* is being developed within the PACHA Esprit project. It aims at helping the design of high-performance applications using independent software components through the use of distributed objects. It provides the benefits of distributed and parallel programming using a combination of two standards: CORBA and MPI. To support CORBA parallel objects, we propose to extend the IDL language to support object and data distribution. In this paper, we discuss the concept of CORBA parallel object.

1 Introduction

Thanks to the rapid increase of performance of nowadays computers, it can be now envisaged to couple several high-intensive numerical codes to simulate more accurately complex physical phenomena. Due to both the increased complexity of these numerical codes and their future developments, a tight coupling of these codes cannot be envisaged. A loosely coupling approach based on the use of several components offers a much more attractive solution. With such approach, each of these components implements a particular processing (pre-processing of data, mathematical solver, post-processing of data). Moreover, several solvers are required to increase the accuracy of simulation. For example, fluid-structure or thermal-structure interactions occur in many field of engineering. Other components can be devoted to pre-processing (data format conversion) or post-processing of data (visualisation). Each of these components requires specific resources (computing power, graphic interface, specific I/O devices). A component, which requires a huge amount of computing power, can be parallelised so that it will be seen as a collection of processes to be ran on a set of network nodes. Processes within a component have to exchange data and have to synchronise. Therefore, communication has to be performed at different levels: between components and within a component. However, requirements for communication between components or within a component are not the same. Within a component, since performance is critical, low level message-passing is required whereas between components, although performance is still required, modularity/interoperability and reusability are necessary to develop cost effective applications using generic components.

However, till now, low level message-passing libraries, such as MPI or PVM, are used to couple codes. It is obvious to say that this approach does not contribute to the design of applications using independent software components. Such communication libraries were developed for parallel programming so that they do not offer the necessary support for designing components which can be reused by other applications. Solutions already exist to decrease the design complexity of applications. Distributed object-oriented technology is one of these solutions. A complex application can be seen as a collection of objects, which represent the components, running on different machines and interacting together using remote object invocations. Existing standard such as CORBA (Common Object Request Broker Architecture) aims at helping the design of applications using independent software components through the use of CORBA objects [1]. CORBA is a distributed software platform which supports distributed object computing. However, exploitation of parallelism within such object is restricted in a sense that it is limited to a single node within a network. Therefore, both parallel and distributed programming environments have their own limitations which do not allow, alone, the design of high performance applications using a set of reusable software components.

This paper aims at introducing a new approach that takes advantage of both parallel and distributed programming systems. It aims at helping programmers to design high performance applications based on the assembling of generic software components. This environment relies on CORBA with extensions to support parallelism across several network nodes within a distributed system. Our contribution concerns extensions to support a new kind of object we called a parallel CORBA object (or parallel object) as well as the integration of message-passing paradigms, mainly MPI, within a parallel object. These extensions exploit as much as possible the functionality offered by CORBA and requires few modifications to existing CORBA implementations. The paper is organised as follows. Section 2 gives a short introduction to CORBA. Section 3 describes our extensions to the CORBA specification to support parallelism within an object. Section 4 introduces briefly the *Cobra* runtime system for the execution of parallel objects. Section 5 describes some related works that share some similarities with our own work. Finally, section 6 draws some conclusions and perspectives.

2 An overview of CORBA

CORBA is a specification from the OMG (Object Management Group) [5] to support distributed object oriented applications. Such applications can be seen as a collection of independent software components or CORBA objects. Objects have an interface that is used to describe operations that can be remotely invoked. Object interface is specified using the Interface Definition Language (IDL). The following example shows a simple IDL interface:

[1] For the remaining of the paper, we will use simply object to name a CORBA object

```
interface myservice {
  void put(in double a);
  double myop(inout long i, out long j);
};
```

An interface contains a list of operations. Operations may have parameters whose types are similar to C++ ones. A keyword added just before the type specifies whether the parameter is an input or an output parameter or both. IDL provides an interface inheritance mechanism so that services can be extended easily. Figure 1 provides a simplified view of the CORBA architecture.

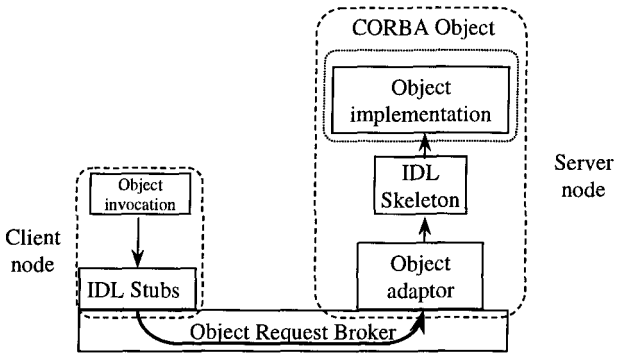

Fig. 1. CORBA system architecture

In this figure, an object located at the client side is bound to an implementation of an object located at the server side. When a client invokes an operation, communication between the client and the server is performed through the Object Request Broker (ORB) thanks to the IDL stub (client side) and the IDL skeleton (server side). Stub and skeleton are generated by an IDL compiler taking as input the IDL specification of the object. A CORBA compliant system offers several services for the execution of distributed object-oriented applications. For instance, it provides object registration and activation.

3 Parallel CORBA object

CORBA was not originally intended to support parallelism within an object. However, some CORBA implementations provide a multi-threading support for the implementation of objects. Such support is able to exploit simultaneously several processors sharing a physical memory within a single computer. Such level of parallelism does not require modification of the CORBA specification since it concerns only the object implementation at the server side. Instead of having one thread assigned to an operation, it can be implemented using several threads. However, the sharing of a single physical memory does not allow a large

number of processors since it could create memory contention. One objective of our work was to exploit several dozen of nodes available on a network to carry out a parallel execution of an object. To reach this objective, we introduce the concept of parallel CORBA object.

3.1 Execution model

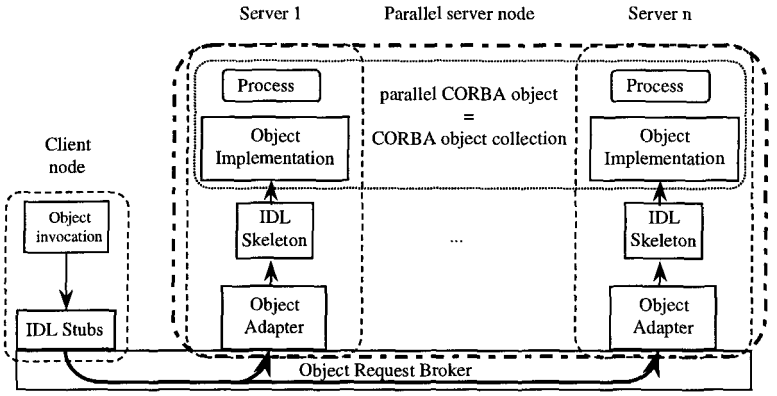

Fig. 2. Parallel CORBA object service execution model.

The concept of parallel object relies on a SPMD (Single Program Multiple Data) execution model which is now widely used for programming distributed memory parallel computers. A parallel object is a collection of identical objects having their own data so that it complies with the SPMD execution model. Figure 2 illustrates the concept of parallel object. From the client side, there is no difference when calling a parallel object comparing to a standard object. Parallelism is thus hidden to the user. When a call to an operation is performed by a client, such operation is executed by all CORBA objects belonging to the collection. Such parallel execution is handled by the stub that is generated by an Extended-IDL compiler, which is a modified version of the standard IDL compiler.

3.2 Extended-IDL

As for a standard object, a parallel object is associated with an interface that specifies which operations are available. However, this interface is described using an IDL we extended to support parallelism. Extensions to the standard IDL aim at both specifying that an interface corresponds to a parallel object and at distributing parameter values among the collection of objects. Extended-IDL is the name of these extensions.

Specifying the degree of parallelism The first IDL extension corresponds to the specification of the number of objects of the collection that will implement the parallel object. Modifications to the IDL language consist in adding two brackets to the IDL *interface* keyword. A parameter can be added within the two brackets to specify the number of objects belonging to the collection. Such parameter can be a "*", that means that the number of objects belonging to the collection is not specified in the interface. The following example illustrates the proposed extension.

```
interface[*] ComputeFEM {
  typedef double dmat[100][100];
  void initFEM(in dmat mat, in double p);
  void doFEM(in long niter, out double err);
};
```

In this example, the number of objects will be fixed at runtime depending on the available resources (i.e. the number of network nodes if we assume that each object of the collection is assigned to only one node). The implementation of a parallel object may require a given number of objects in the collection to be able to run correctly. Such number may be inserted within the two brackets of the proposed extension. The following example gives an example of a parallel object service which is made of 4 objects.

```
interface[4] ComputeFEM {
  ...
};
```

Instead of giving a fixed number of objects in the collection, a function may be added to specify a valid number of objects in the collection. The following example illustrates such possibility. In that case, the number of objects in the collection may be only a power of 2.

```
interface[n^2] ComputeFEM {
  ...
};
```

It is the responsibility of the runtime system, in our case *Cobra*, to check whether the number of network nodes has been allocated according to the specification of the parallel object. IDL allows a new interface to inherit from an existing one. Parallel interface can do the same but with some restrictions. A parallel interface can inherit only from an existing parallel interface. Inheritance from standard interface is forbidden. Moreover, inheritance is allowed only for parallel interfaces that could be implemented by a collection of objects for which the number of objects coincides. The following example illustrates this restriction.

```
interface[*] MatrixComponent {
  ...
};
interface[n^2] ComputeFEM : MatrixComponent {
  ...
```

In this example, interface *ComputeFEM* derives from interface *MatrixComponent*. The new interface has to be implemented using a collection having a power of 2 objects. In the following example, the Extended-IDL compiler will generate an error when compiling because inheritance is not valid :

```
interface[3] MatrixComponent {
  ...
};
interface[n^2] ComputeFEM : MatrixComponent {
  ...
};
```

Specifying data distribution Our second extension to the IDL language concerns data distribution. The execution of a method on a client side will provoke the execution of the method on every objects of the collection. Since, each object of the collection has it own separate address space, we must envisage how to distribute parameter values for each operation. Attributes and types of operation parameters act on the data distribution. Proposed extension of IDL for data distribution is allowed only for parameters of operations defined in a parallel interface. When a standard IDL type is associated with a parameter with an **in** mode, each object of the collection will receive the same value. When a parameter of an operation has either an **out** or a **inout** mode, as a result of the execution of the operation, stub generated by the Extended-IDL compiler will get a value from one of the objects of the collection.

The IDL language provides multidimensional fixed-size arrays which contains elements of the same type. The size along each dimension has to be specified in the definition. We provide some extensions to allow the distribution of arrays among the objects of a collection. Data distribution specifications apply for both **in**, **out** and **inout** mode. They are similar to the ones already defined by HPF (High Performance Fortran). The following example gives a brief overview of the proposed extension.

```
interface[*] MatrixComponent {
  typedef double dmat[100][100];
  typedef double dvec[100];
  void matrix_vector_mult(in dist[BLOCK][*] dmat, in dvec v,
                          out dist[CYCLIC] dvec u);
};
```

This extension consists in the adding of a new keyword (**dist**) which specifies how an array is distributed among the objects of the collection. For example, the 2D array *mat* is distributed by block of rows. Stubs generated by the Extended IDL compiler do not perform the same work when the parameter is an input or an output parameter. With an input parameter, the stub must scatter the distributed array so that each object of the collection received a subset of the whole array. With an output parameter, the stub must do the reverse operation. Indeed, each object of a collection contains a subset of the array. Therefore, the stub is in charge of gathering data from objects belonging to the collection.

Such gathering may include a redistribution of data if the client is itself a parallel object. In the previous example, the number of objects in the collection is not specified in the interface. Therefore, the number of elements assigned to a particular object can be known only at runtime. It is why a distributed array of a given IDL type is mapped to an unbounded sequence of this IDL type. Unbounded sequence offers the advantage that its length is set up at runtime. We propose to extend the sequence structure to store information related to the distribution.

4 A runtime system to support parallel CORBA objects

The *Cobra* runtime system [2] aims at providing resource allocation for the execution of parallel objects. It is targeted to a network of PCs connected together using SCI [6]. Resource allocation consists in providing network nodes and shared virtual memory regions for the execution of parallel objects. Resource allocation services are performed by the resource management service (RmProcess) of *Cobra*. It is used when a parallel service must be bind to a client. We propose to extend the *_bind* method provided by most of the CORBA implementations. Binding to a parallel object differs from the standard binding method. Indeed, a reference to a virtual parallel machine (**vpm**) is given as an argument of the *bind* method instead of a single machine. The **vpm** reference is obtained through the *Cobra* resource allocator. The following example illustrates how to use a parallel object service within the *Cobra* runtime:

```
...
// Obtain a reference from the RmProcess service
cobra = RmProcess::_bind("cobra.irisa.fr");
// Create a VPM
cobra->mkvpm(vpmname, NumberOfNodes, NORES);
// Get a reference to the allocated vpm
pap_get_info_vpm(&vpm, vpmname);
// Obtain a reference from the parallel object service: MatrixComponent
cs = MatrixComponent::_bind ( &vpm );
...
// Invoke an operation provided by MatrixComponent service
cs->matrix_vector_mult( a, b, &c);
...
```

The *bind* method may be called either by a single object, or by all objects belonging to a collection if the client is itself a parallel object.

5 Related works

Several projects deal with environments for high-performance computing combining the benefits of distributed and parallel programming. The RCS [1], NetSolve [3] and Ninf [8] projects provide an easy way to access linear algebra method libraries which run on remote supercomputer. Each method is described

by a specific interface description language. Clients invoke methods thanks to specific functions. Arguments of these functions specify method name and method arguments. These projects propose some mechanisms to manage load balancing on different supercomputers. One drawback of these environments is the difficulty for the user to add new functions in the libraries. Moreover, they are not compliant to relevant standard such as CORBA. The Legion [4] project aims at creating a world wide environment for high-performance computing. A lot of principles of CORBA (such as heterogeneity management and object location) are provided by the Legion run-time, although Legion is not CORBA-compliant. It manipulates parallel objects to obtain high-performance. All these features are in common with our *Cobra* run-time. However, Legion provides others services such as load balancing on different hosts, fault-tolerance and security which are not present in *Cobra*. The PARDIS [7] project proposes a solution very close to our approach because it extends the CORBA object model to a parallel object model. A new IDL type is added: *dsequence* (for distributed sequence). It is a generalisation of the CORBA sequence. This new sequence describes data type, data size, and how data must be distributed among objects. In PARDIS, distribution of objects is let to the programmers. It is the main difference with *Cobra* for which a resource allocator is provided. Moreover in *Cobra*, extended-IDL allows to describe object parallel services in more details.

6 Conclusion and perspectives

This paper introduced the parallel CORBA object concept. It is a collection of standard CORBA objects. Its interface is described using an Extended-IDL to manage data distribution among the objects of the collection. *Cobra* is being implemented using Orbix from Iona Tech. It has already been tested for building a signal processing application using a client/server approach [2]. For such application, the most computing part of the application is encapsulated within a parallel CORBA object while the graphical interface is a Java applet. This applet, acting as a client, is connected to the server through the CORBA ORB. Current works are now focusing on the experiment of the coupling of numerical codes. Particular attention will be paid on the performance of the ORB which seems to be the most critical part of the software environment to get the requested performance. It is planned, within the PACHA project, to implement an ORB that fully exploits the performance of the SCI clustering technology while ensuring compatibility with existing ORB through standard protocols such as TCP/IP.

References

1. P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. In *HPCN Europe '96*, volume 1067 of *LNCS*, pages 662–667, 1996.
2. P. Beaugendre, T. Priol, G. Alleon, and D. Delavaux. A client/server approach for hpc applications within a networking environment. In *HPCN'98*, pages 518–525, April 1998.

3. H. Casanova and J. Dongara. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.
4. 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.
5. Object Management Group. The common object request broker: Architecture and specification 2.1, August 1997.
6. Dolphin Interconnect. Clustar interconnect technology. White Paper, 1998.
7. K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-level Parallel Distributed Computation. In *Proceedings of Supercomputing '97*, November 1997.
8. M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure. In *HPCN Europe '97*, volume 1225 of *LNCS*, pages 491–502, 1997.