Programming SCI Clusters Using Parallel CORBA Objects

Thierry Priol, Christophe René, Guillaume Alléon

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Programming SCI Clusters using Parallel CORBA Objects

Thierry Priol, Christophe René, Guillaume Alléon[1]

N ° 3649

mars 1999

THÈME 1

[1] Aérospatiale Joint Research Centre, 12 rue Pasteur, BP 76, 92152 Suresnes Cedex, France.
email:Guillaume.Alleon@siege.aerospatiale.fr

ISSN 0249-6399

Programming SCI Clusters using Parallel CORBA Objects

Thierry Priol, Christophe René, Guillaume Alléon[†]

Thème 1 — Réseaux et systèmes
Projet Caps

Rapport de recherche n3649 — mars 1999 — 17 pages

Abstract: This paper presents a software infrastructure for component programming on SCI-based Clusters. This environment is based on the concept of Parallel CORBA object that is an extension to the Common Object Request Broker Architecture (CORBA) from the OMG (Object Management Group). The proposed extensions do not modified the CORBA core infrastructure (the Object Request Broker) so that it can fully co-exist with existing CORBA applications. A signal processing application is presented to illustrate the use of this software infrastructure.

Key-words: CORBA,HPCN,METACOMPUTING,PSE

(Résumé : tsvp)

[†] Aérospatiale Joint Research Centre, 12 rue Pasteur, BP 76, 92152 Suresnes Cedex, France.
email:Guillaume.Alleon@siege.aerospatiale.fr

Unit´e de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
T´el´ephone : 02 99 84 71 00 - International : +33 2 99 84 71 00
T´el´ecopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Programmation de grappes SCI avec des objets CORBA parallèles

Résumé : Ce papier décrit une infrastructure logicielle pour la programmation par composants sur une grappe SCI. Cet environnement est fondé sur le concept d'objet CORBA parallèle qui est une extension de l'architecture CORBA de l'OMG. Ces extensions ne remettent pas en cause l'infrastructure principale de CORBA (l'ORB) ce qui permet de coexister avec des applications déjà écrites avec CORBA. Une application de traitement du signal est présentée pour illustrer l'utilisation de cet environnement.

Mots-clé : CORBA,HPCN,METACOMPUTING,PSE

1 Introduction

This chapter introduces a programming environment for SCI clusters that takes advantage of both parallel and distributed programming paradigms. It aims at helping programmers to design high performance applications based on the assembling of generic software components. This environment is based on CORBA (Common Object Request Broker Architecture), with our own extensions to support parallelism across several cluster nodes within a distributed system. Our contribution concerns extensions to support a new kind of object, which we call 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 require few modifications to an available CORBA implementation. This paper reports on these extensions and the description of a runtime system, called *Cobra*, which provides resource allocation services for the execution of parallel objects.

The chapter is organized as follows. Section 2 discusses some issues related to parallel and distributed programming. Section 3 gives a short introduction to CORBA. Section 4 describes the concept of parallel CORBA objects. Section 5 introduces the *Cobra* runtime system for the execution of parallel objects. Section 6 presents a case study based on a signal processing application from Aerospatiale. Section 7 describes some related work which has some similarities with our work. Finally, section 8 draws some conclusions and outlines perspectives of this work.

2 Parallel vs. Distributed Programming

Thanks to the rapid performance increase of today's computers, it can be now envisaged to couple several computationally 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 loose coupling approach based on the use of several components offers a much more attractive solution. One can envisage to couple fluid and structure components or thermal and structure components. Other components can be devoted to pre-processing (data format conversion) or post-processing of data (visualization). Each of these components requires specific resources (computing power, graphics, specific I/O devices). A component which requires a huge amount of computing power can be parallelized so that it will be seen as a collection of processes to be run on a set of cluster nodes. Processes within a component have to exchange data and have to synchronize. Therefore, communication has to be performed at different levels: between components and within a component. However, the requirements for communication between or within a component are quite different. 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, until now, most programmers who are faced with the design of high-performance applications use low-level message-passing libraries such as MPI or PVM. Such libraries can be used for both coupling components and for handling communication among processes of a parallel component. 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; 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 such applications. Distributed object-oriented technology is one of them. A complex application can be seen as a collection of objects which represent the components, running on different machines and interacting using remote object invocations. Emerging standards, such as CORBA, support the design of applications using independent software components through the use of CORBA objects. For the rest of the chapter, we will use the term object to name a CORBA object. CORBA is a distributed software platform which supports distributed object computing. However, exploitation of parallelism within such an object is restricted in a sense that it is limited to a single node within a cluster. CORBA implementations such as Orbix from Iona Technologies [8], allow the design of multi-threaded objects that can exploit several processors within a single SMP (Symmetric Multi-Processing) node. Such an SMP node cannot offer the large number of processors which is required for handling scientific applications in a reasonable time frame. However, the required number of processors is available at the cluster level where several dozens of machines are connected. Nevertheless, application designers have to deal "manually" with a large number of objects that have to be mapped onto different nodes of a cluster, and to distribute computations and data among these objects.

Therefore, either parallel and distributed programming environments have their limitations which do not allow the design of high performance applications using a set of reusable software components. The remaining part of this chapter introduces a programming environment which combines the advantages of both parallel and distributed programming.

3 An Overview of CORBA

CORBA is a specification from the OMG (Object Management Group) [7] to support distributed object-oriented applications. An application based on CORBA can be seen as a collection of independent software components or CORBA objects. Remote method invocations are handled by an Object Request Broker (ORB) which provides a communication infrastructure independent of the underlying network. Within the ORB, several protocols exist to handle specific network technologies. The most important protocol is the IIOP (Internet Inter-ORB Protocol) which is used to support Ethernet-based networks. However, IIOP was designing for interoperability and thus offers limited performance. Fortunately, CORBA designers have provided the ESIOP (Environment-Specific Inter-ORB Protocol) which can handle other network technologies (SCI, for instance). An object interface is specified using the Interface Definition Language (IDL). An IDL file contains a list of operations

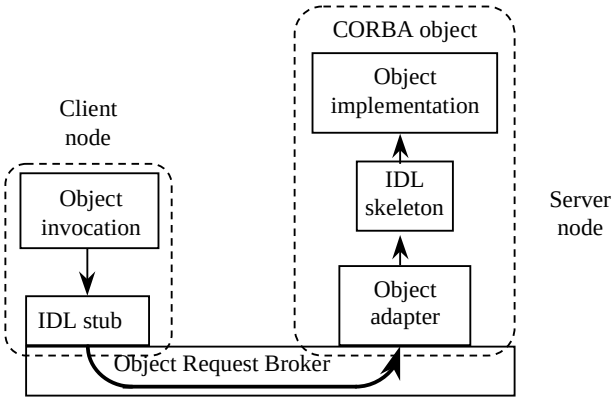

Figure 1: CORBA system architecture

for a given object that can be remotely invoked. Figure 1 provides a simplified view of the CORBA 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 of the object, communication between the client and the server is performed through the ORB thanks to the IDL stub (client side) and the IDL skeleton (server side). The stub and the skeleton are generated by an IDL compiler taking as input the IDL specification of the object. Since CORBA is independent of the language used for the object implementation, an IDL compiler may generate stubs for different languages (e.g., Java, C++, Smalltalk). An object can thus be implemented in C++ and called by a client implemented in Java. The following example shows a simple IDL interface:

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

An interface corresponds to an object class and an operation to an object method. In this interface example, there are two operations associated with the interface. Each operation has a single parameter. An operation parameter is assigned a type which is similar to a C++ type (e.g., a scalar, an array). A keyword added just before the type specifies if the parameter is an input or an output parameter or both. IDL types are mapped to the language to be used at the server and the client side. IDL provides an interface inheritance mechanism so that services can be extended easily.

A CORBA-compliant system offers several services for the execution of distributed object-oriented applications. It provides object registration and activation through the use of repositories. Object registration consists of specifying a process that implements the object so that when an operation is called, the process is executed.

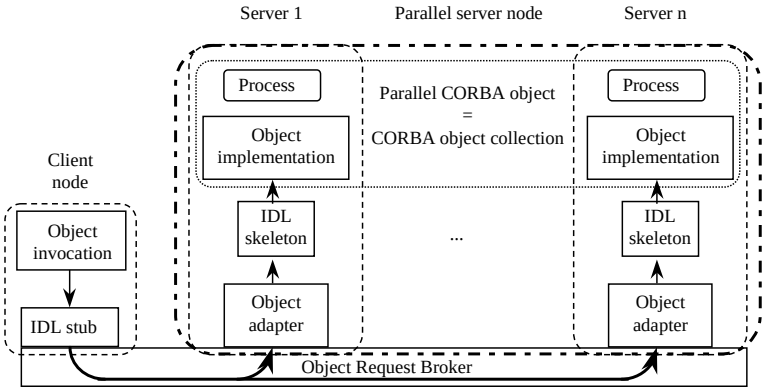

Figure 2: Parallel CORBA object service execution model

4 Parallel CORBA Objects

CORBA was not originally intended to support parallelism within an object. However, available CORBA implementations provide a multi-threading support for the implementation of objects. Such support is able to exploit several processors sharing a physical memory within a single computer. This level of parallelism does not require any modification to 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 these could create bus and memory contention. One objective of our work is to exploit the several dozens of nodes available within a cluster to carry out a parallel execution of an object. To reach this objective, we introduce the concept of parallel CORBA object.

4.1 Execution Model

The concept of parallel objects relies on an SPMD (Single Program Multiple Data) execution model. A parallel object is a collection of identical objects having their own data, in compliance with the SPMD execution model. Figure 2 illustrates the concept of parallel objects. From the client side, there is no difference in calling a parallel object to calling a standard object. Parallelism is thus hidden to the user. When a call to an operation is performed by a client, the operation is executed by all objects belonging to the collection. The parallel execution is handled by the stub that was generated by an Extended-IDL compiler which is a modified version of the standard IDL compiler.

4.2 Extended-IDL

Like a standard object, a parallel object is associated with an interface which specifies the operations 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 of 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. This parameter can be a "*", which means that the number of objects belonging to the collection is not specified in the interface. An integer value or a function which determines the number of objects, is also valid. The following example illustrates the proposed extension:

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

In the previous example, the number of objects will be fixed at runtime depending on the available resources (i.e., the number of cluster nodes if we assume that each object of the collection is assigned to a single node). The implementation of a parallel CORBA object may require a given number of objects in the collection to be able to run correctly. The following example gives an example of a parallel object service which comprises four objects:

```
interface[4] ComputeFEM {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err);
};
```

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 this possibility. In this case, the number of objects in the collection may be only a power of two:

```
interface[n^2] ComputeFEM {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err);
};
```

IDL allows a new interface to inherit from an existing one. Parallel interfaces can do the same but with some restrictions. A parallel interface can inherit only from an existing parallel interface. Inheritance from a standard interface is prohibited. Moreover, inheritance

is allowed only for parallel interfaces that can be implemented by a collection of objects with a corresponding number of objects. The following examples illustrate this restriction:

```
interface[*] MatrixComponent {
  void matrix_vector_mult(in double mat[100][100], in double v[100],
                          out double u[100]);
  void matrix_transpose(in double A[100][100],
                        out double B[100][100]);
};
interface[n^2] ComputeFEM : MatrixComponent {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err)
};
```

In this example, interface *ComputeFEM* inherits from interface *MatrixComponent*. The new interface has to be implemented using a collection having a square number of objects. In the following example, the inheritance is not valid:

```
interface[3] MatrixComponent {
  void matrix_vector_mult(in double mat[100][100],
                          in double v[100], out double u[100]);
  void matrix_transpose(in double A[100][100],
                        out double B[100][100]);
};

interface[n^2] ComputeFEM : MatrixComponent {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err)
};
```

The Extended-IDL compiler will generate an error when compiling this specification. The intersection of the valid range of values of each inherited parallel interface and the new parallel interface must not be empty; otherwise the inheritance is not valid.

Specifying data distribution

Our second extension to the IDL language to support parallel objects concerns data distribution. Remember that the execution of a method on the client side will provoke the execution of the method on every object of the collection on the server side. Since each object of the collection performs a part of the work and has its own separate address space, we must envisage how to distribute the parameter values for each operation. We add new keywords to specify how to distribute the parameter values among the objects of the collection. The following paragraphs will explain how the data can be distributed for both **in**, **out** and **inout** modes depending on their type. A data-distribution extension of an IDL specification is allowed only for parameters of operations defined in a parallel interface.

The IDL language provides multidimensional fixed-size arrays which contain elements of the same type. Types can be either basic or constructed types. 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 **in**, **out** and **inout** modes. These data distribution specifications follow those already defined by HPF (High Performance Fortran). This design choice permits to map Extended-IDL to the HPF language in the future. It will be thus possible to implement a parallel object using HPF. Such a mapping could be based on the IDL to Fortran90 compiler which is being designed and implemented within the Esprit PACHA project. The following example gives a brief overview of the proposed extension:

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

This extension consists of adding the new keyword **dist**, which specifies how an array is distributed among the objects of the collection. The 2D array *mat* is distributed by blocks of rows. Since the parameter is assigned an **in** mode, each object of the collection will receive a block of rows instead of the whole array. A distributed array of a given IDL type is mapped to an unbounded sequence of this IDL type which has been extended to store information related to the distribution. An unbounded sequence offers the advantage that its length is determined at runtime. Scattering of distributed arrays among the objects of a collection is performed by the stub generated by the Extended-IDL compiler. If the client is itself a parallel object, the stub is in charge of gathering data from client objects before sending them to the server objects with the correct distribution. 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. Parameter *v* does not have a data distribution specification so that each object receives the whole array. The last parameter *u* has an **out** mode assigned to it. Each object of the collection will send back to the client a part of the array. The code generated by the Extended-IDL compiler is in charge of gathering the data from the objects of the collection and to give them back to the client which invoked the operation. Gathering may include a redistribution of data if the client is itself a parallel object. As a matter of fact, distribution of variable *u* may not be identical at the client and the server side. At the client side, information related to the distribution is stored in the corresponding unbounded sequence structure. This information is accessed by the stub of the parallel object to redistribute data if necessary.

4.3 Implementation of Parallel CORBA Objects

As we have shown in the previous paragraphs, the code generated by the Extended-IDL compiler is in charge of managing the communication between a client that issues a request

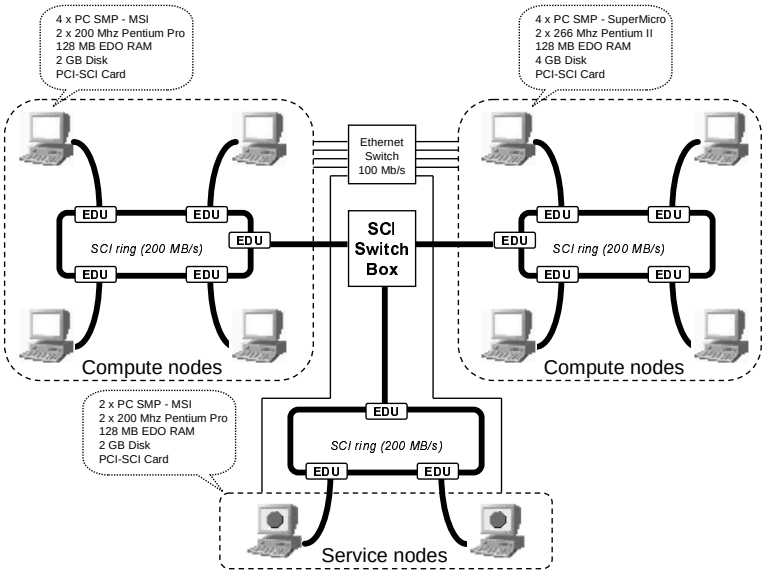

Figure 3: The PACHA multiprocessors

and a parallel object. Implementation relies on a new stub to be generated by the Extended-IDL compiler. After binding the client object to the parallel object, the client is able to send method invocations to the parallel object service. Modification of the ORB is not required since the CORBA specification provides a mechanism to issue multiple requests within a single call to the ORB. When an operation is invoked, a request is constructed containing the object references of all objects belonging to the collection as well as the name of the operation to be invoked. This request is then sent through the ORB which in turn will issue a request to each object to execute the operation.

5 The *Cobra* Runtime System

The *Cobra* runtime system provides resource allocation for the execution of parallel objects on SCI-based clusters. This runtime system is being developed with the Esprit PACHA R&D project. The project aims at building a parallel scalable computer system for high performance applications. This system includes both the development of hardware components and runtime systems. Emerging standards both in software, namely CORBA, and in hardware, namely SCI, are exploited to investigate the design and implementation of a full-featured CORBA-compliant software with minimum overhead. The *Cobra* runtime system is targeted to the PACHA multiprocessors, as shown in Figure 3. It is based on the clustering of PC systems using the PCI-SCI technology from Dolphin Interconnect Solutions. The

machine is a set of three SCI ringlets connected together through an SCI switch. The first SCI ringlet contains two service nodes which act as the front-end of the PACHA machine. These two nodes run the *Cobra* runtime system for resource allocation. The two other SCI ringlets connect compute nodes. These compute nodes are allocated to users on demand by the *Cobra* runtime system for the execution of parallel objects. Resource allocation consists of providing cluster nodes and shared virtual memory regions for the execution of parallel objects.

5.1 *Cobra* Services

Cobra provides the concept of a virtual parallel machine which is associated with the execution of a parallel object. Allocating a virtual parallel machine consists of choosing a set of cluster nodes where objects of the collection will be mapped to for execution. Selection of nodes is performed statically since the PACHA multiprocessors act as a computational server. There are no other applications running simultaneously with parallel objects. Dynamic strategies could be added in the future to allow sharing of the machine by several user applications. *Cobra* offers services for the allocation of virtual shared memory regions. Such regions can be accessed simultaneously by several components of the application. We think that the coupling of software components will need the exchange of unstructured data which cannot be passed efficiently between components through the ORB due to the cost of marshalling and demarshalling data. The two resource allocation services are implemented using CORBA objects so that they are available from any machine within the cluster. Therefore, a client running somewhere in the network is able to allocate resources through CORBA, and once the resources have been allocated, a client can bind a parallel object to the virtual parallel machine which has been created.

Cobra provides basic services for parallel programming. Execution of objects belonging to a collection associated with a parallel object requires some basic functions such as identification and communication between objects. *Cobra* provides an application programming interface for these objects. This interfaces contains a set of C and Fortran77 functions for parallel programming such as synchronisation between objects (barrier, lock), low-level message-passing and shared memory region management.

5.2 *Cobra* Software Architecture

Cobra has been designed for the PACHA multiprocessors and thus benefits from the SCI technology. *Cobra* can allocate both physical nodes and shared memory for the execution of parallel objects. Figure 4 shows the overall architecture of *Cobra*. *Cobra* is a set of three standard CORBA objects which run on the service nodes of the PACHA multiprocessors. Implementation of these services is carried out using either MICO [14], a freely available CORBA implementation from the University of Frankfurt, or ORBSCI, which is a CORBA implementation being implemented by Spacebel within the PACHA project.

The *AdminProcess* object provides services for the administration of the multiprocessors. The *RmProcess* gives a set of services for resource allocation, while *AppProcess* supports

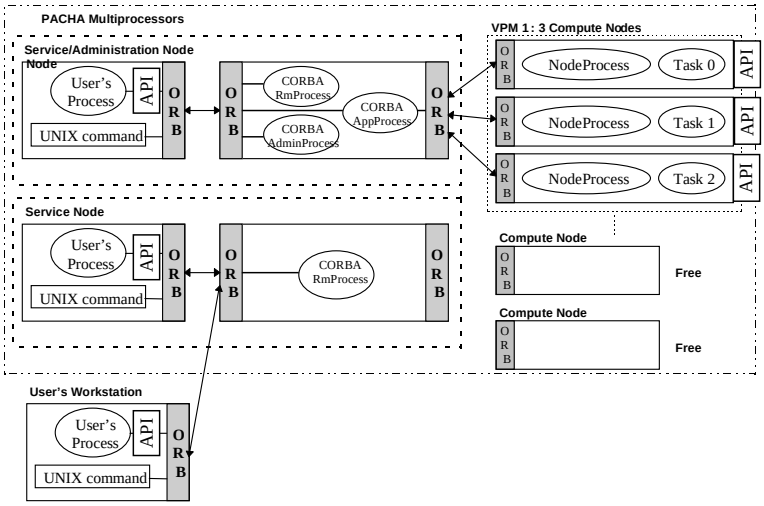

Figure 4: The *Cobra* runtime system architecture

the execution of standalone applications. Since several service nodes are allowed by the runtime system, resource allocation tables are mapped onto SCI shared memory regions so that each service node is able to access the allocation tables. Running on the compute nodes, the *NodeProcess* object provides services to the *RmProcess* object for the execution of objects belonging to a collection. Accesses to these services are performed using either UNIX commands, specific APIs, or simply by using the IDL specification.

Administration service

The *AdminProcess* service administrates the PACHA multiprocessors. It is mapped on a specific node of the machine which is called the administration node that acts also as a service node. There is always one such node in the PACHA multiprocessors. This service provides basic support for adding and removing nodes or changing the node state. For instance, at any time a compute node can be changed to a service node to let more users have access to the PACHA multiprocessors. To protect the system, the *AdminProcesss* service can be executed only by a user who has the administrator privilege. A list of users is maintained by the runtime system indicating if a user is an administrator or a standard user. For each node of the machine, the runtime system maintains a list of resources associated with that node (e.g., access to a fast network, frame buffer, number of processors). This information is used later when a set of nodes is allocated to a particular parallel object which may require specific resources for execution.

Resource management service

The *RmProcess* services provides resource management for standard users. This service is run on each service node of the PACHA multiprocessors. It manages resources such as compute nodes and shared memory regions provided by SCI. To let service nodes manage their own set of users concurrently and to avoid contention when the number of users increases, allocation tables are shared between service nodes. This sharing is performed using several SCI shared memory regions. Accesses to these tables, by both the *AdminProcess* and *RmProcess* services, are performed using critical sections to avoid any incoherent state.

The *Cobra* runtime system provides the concepts of a Virtual Parallel Machine (VPM) and Shared Memory Regions (SMR). A VPM is a set of compute nodes of the PACHA multiprocessors allocated by a user on demand. It is identified in the system by a name. A VPM is used for the execution of a parallel object. When executing a parallel object on a VPM, the runtime system creates as many objects as there are processors in the compute nodes of a VPM.

The second kind of resource managed by *Cobra* are shared memory regions. An SMR is identified by an unique name in the system. Shared memory regions can be used in several ways. They permit objects which are executed on different nodes of a VPM to share data. An SMR is also a way to exchange information between parallel objects running on different VPMs either simultaneously or sequentially. Exporting or importing an SMR between VPMs is granted depending on access rights specified during the creation of the SMR. Data stored in an SMR is persistent. Shared memory regions can thus be seen as a data repository which can be used to avoid huge data transfers between objects. Concerning the implementation, SMRs managed by *Cobra* correspond either to SCI shared memory regions or Shared Virtual Memory (SVM) regions [13]. SVM is implemented using SCI as a communication layer, and it provides better performance since it offers page migration and replication. Coherence is enforced by a strong consistency protocol.

Application service

Although the *Cobra* runtime system was mainly designed to support execution of parallel objects, we provide a specific service, called *AppProcess*, for supporting stand-alone applications (which are not CORBA compliant). This service allows the loading and the monitoring of parallel application (either SPMD or MPMD) onto a VPM using a set of UNIX commands.

Application programming interface

The *Cobra* runtime system provides several application programming interfaces for both the C and Fortran languages. The first API supports the client side. It provides a function for each operation of the IDL specification of the *AdminProcess*, *RmProcess* and *AppProcess* services. At the server side, an API is provided for parallel programming. This API contains C and Fortran functions for node identification, low-level message passing, shared memory management and synchronization such as locks and barriers.

6 A Case Study: the IDAHO Application

The IDAHO application is being developed by the Aérospatiale Joint Research Centre for electromagnetic experimentation and simulation purposes. The IDAHO application is a set of tools to help the engineers in processing the data coming from experiments performed in an anechoic chamber. The electro-magnetic illumination is generated by a transmitter-receiver driven by the operator. The reduced model is placed on a rotating column in the anechoic chamber. For each angle of rotation (from 0 to 360 degrees) and each frequency (typically from 2 to 8 GHz), a complex number representing the reflected field is stored. This experimentation procedure can last up to 90 hours and generate up to 1 GByte of data.

The IDAHO application has been designed to adopt a client/server approach to let engineers process data remotely from their workstations. The most compute-intensive part of the application has been encapsulated in a parallel object which contains several operations. Among them, the *normalization* operation computes the correction on the measured values to balance the noise effects of the anechoic chamber, using the measurements of the empty chamber and the measurements of a reference. We can then represent in 2D the reflected field for each rotation angle and each frequency. This representation is commonly named hologram. The *windowing normalization* operation is very similar to the previous one, except that it includes a window multiplication in the computation. This is used to focus the analysis around the measured object. The *ISAR* image computation is an operation to compute a 2D ISAR image, by making a 2D FFT on the data. With an ISAR image, we are able to locate on the object the reflecting points. The transverse response computation aims at calculating the 2D transverse response, by making a 1D FFT on one dimension of the measured data. With a transverse response, it is possible to follow a reflecting point while the object is rotating in the anechoic chamber. Each of these operations has been parallelized using the *Cobra* parallel application programming interface. The most complex task is the parallelization of the matrix transpose needed by the 2D FFT. The complexity is due to the limited size of shared memory regions provided by the SCI driver (512 kByte). Preliminary results have shown a speedup of three for the matrix transpose when running on a four-processor VPM.

Visualization is performed using a graphical user interface which has been implemented in Java to be run on any machine in the network. A version of this interface has been designed and implemented as a Java applet. This applet is stored on a service node of the PACHA multiprocessor which acts as a Web server. Therefore, the IDAHO application can be executed from any machine in the network having a Web browser supporting Java. Figure 5 shows the client/server overall architecture of the IDAHO application. In the first step (1), the applet connects to the *Cobra* services to allocate a VPM, then it connects to a proxy object which acts as a bridge between the applet and the parallel object. The proxy object has to be executed on the service node from which the applet was downloaded. The proxy object implements the same operations as the parallel object. The proxy object adds communication overhead but is required as a result of the security rules of the Java virtual machine. Once data has been sent to the parallel object (2), parallel execution starts (3)

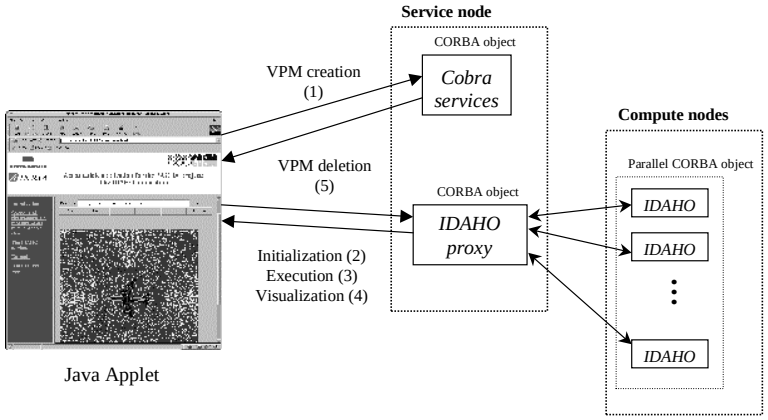

Figure 5: The IDAHO client/server application

and an image is sent back to the applet (4) as a result of the execution. The final step (5) consists of releasing the VPM which was previously allocated.

7 Related Work

Several projects deal with environments for high-performance computing combining the benefits of distributed and parallel programming.

The RCS [1], NetSolve [4] and Ninf [15] projects provide an easy way to access linear algebra method libraries which run on remote supercomputers. Each method is described by a specific interface description language. Interface descriptions have been made for all methods of standard libraries (such as BLAS and LAPACK). Specific functions are provided for invoking methods of these libraries. 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 to the libraries. Moreover, they are not compliant to relevant standards such as CORBA.

The Legion [5, 6] 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 runtime system, although Legion is not CORBA-compliant. It manipulates parallel objects to obtain high performance. All these features are in common with our *Cobra* runtime system. However, Legion provides other services such as load balancing on different hosts, fault tolerance and security which are not present in *Cobra*. Furthermore, the Legion communication layer manages different networking technologies such as Ethernet and ATM.

The PARDIS [10, 11, 12] project proposes a solution very close to our project 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 generalization of the CORBA sequence. This new sequence describes data type, data size, and how data must be distributed among objects. The PARDIS IDL compiler creates a new bind function, *spmd_bind*, which is added to the client's stub. Concurrent threads may use this function to make a collective request to a server. Therefore, this server has to reply only to one request. For each operation listed in the interface description, the PARDIS IDL compiler adds a non-blocking method. Non-blocking functions may be executed concurrently even if they are called in a sequential order. **out** arguments of these functions are returned in *futures*. This idea results from the work on parallel C++. In PARDIS, distribution of objects is up to the programmer. This is the main difference from *Cobra*, for which a resource allocator is provided. Moreover, in *Cobra*, Extended-IDL allows to describe parallel services in more detail, such as the number of objects associated with a parallel object.

8 Conclusion and Perspectives

This chapter described the *Cobra* runtime system which provides a software environment for building high-performance applications using software components. *Cobra* is a set of CORBA services for the execution of CORBA parallel objects. A parallel object is a collection of standard CORBA objects. Its interface is described using an extension of IDL to manage data distribution among the objects of the collection. Current work now focuses on coupling numerical codes. Particular attention will be paid to the performance of the ORB which seems to be the most critical part of the software environment to get the desired performance. We are currently designing an efficient ORB for MICO, based on the Virtual Interface (VI) architecture.

References

[1] P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. In *Proceedings of HPCN Europe '96*, volume 1067 of *LNCS*, pp. 662–667, Springer Verlag, 1996.

[2] P. Beaugendre, T. Priol, G. Alleon, and D. Delavaux. A Client/Server Approach for HPC Applications within a Networking Environment. In *Proceedings of HPCN Europe '98*, volume 1401 of *LNCS*, pp. 518–525, Springer Verlag, 1998.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[4] 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.

[5] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, University of Virginia, 1994.

[6] 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.

[7] Object Management Group. The Common Object Request Broker: Architecture and Specification 2.1. August 1997.

[8] C. Horn. The Orbix Architecture. Technical Report, IONA Technologies, August 1993.

[9] Dolphin Interconnect Solutions. CluStar Interconnect Technology. White Paper, 1998.

[10] K. Keahey. A Model of Interaction for Parallel Objects in a Heterogeneous Distributed Environment. Technical Report IUCS TR 467, Indiana University, September 1996.

[11] K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. Technical Report IUCS TR 475, Indiana University, February 1997.

[12] K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-level Parallel Distributed Computation. In *Proceedings of Supercomputing '97*, November 1997.

[13] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[14] A. Puder. The MICO CORBA Compliant System. *Dr. Dobb's Journal*, 291:44–57, November 1998.

[15] 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 *Proceedings of HPCN Europe '97*, volume 1225 of *LNCS*, pages 491–502, Springer Verlag, 1997.

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

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