A Parallel CORBA Component Model for Numerical Code Coupling*

Christian Pérez, Thierry Priol, André Ribes

IRISA/INRIA, PARIS research group

Campus de Beaulieu - 35042 Rennes Cedex, France

{Christian.Perez,Thierry.Priol,Andre.Ribes}@irisa.fr

May 27, 2003

Abstract

The fast growth of high bandwidth wide area networks has allowed the building of computational grids, which are constituted of PC clusters and/or parallel machines. Computational grids enable the design of new numerical simulation applications. For example, it is now feasible to couple several scientific codes to obtain a multi-physic application. In order to handle the complexity of such applications, software component technology appears very appealing. However, most current software component models do not provide any support to transparently and efficiently embed parallel codes into components. This paper describes a first study of GridCCM, an extension to the CORBA Component Model to support parallel components. The feasibility of the model is evaluated thanks to its implementation on top of two CCM prototypes. Preliminary performance results are very good: there is no noticeable overhead and the bandwidth is efficiently aggregated.

1 Introduction

The fast growth of high bandwidth wide area networks (WAN) allows to build computing infrastructures, known as computational grids [13]. Such infrastructures allow the interconnection of PC clusters and/or parallel machines with high bandwidth WAN. For example, the bandwidth of the French VTHD WAN [4] is 2.5 Gb/s and the US TeraGrid project [3] targets 40 Gb/s bandwidth.

Thanks to the performance of grid infrastructure, new kinds of applications are enabled in the scientific numerical simulation field. In particular, it is now feasible to couple scientific codes, each code simulating a particular aspect of a physical phenomenon. Hence, it is possible to perform more realistic simulations of the behavior of a satellite by incorporating all aspects of a simulation: dynamic, thermal, optic and structural mechanic. Each of these aspects is simulated by a dedicated code, which is usually a parallel code, and is executed on a set of nodes of a grid. Because of the complexity of the phenomena, these codes are developed by independent specialist teams. So, one may expect to have to couple codes written in different languages (C, C++, FORTRAN, etc.) and depending on different communication paradigms (MPI, PVM, etc).

The evolution of scientific computing requires a programming model, including technologies, coming from both parallel and distributed computing. Parallelism is needed to efficiently exploit PC clusters or parallel machines. Distributed computing technologies are needed to control the execution and to handle communications between codes running in grids, that is to say a in distributed and heterogeneous environment.

The complexity of targeted applications is very high with respect to design issue but also with respect to deployment issue. So, it appears that it is required to consider adequate programming models. Software component [29] appears as a very appealing solution: a code coupling application can be seen as an assembly of components; each component embeds a simulation code. However, most software component models such as Enterprise JAVA Bean [21], COM+ [26], Web Services [8] or CORBA Component Model [23] only support sequential components. With these models, a component is associated to a unique address

*This work was supported by the Incentive Concerted Action "GRID" (ACI GRID) of the French Ministry of Research.

space: the address space of the process where the component is created. Communication between (not collocated) components consists in transferring data from one address space to another one using some communication mechanisms such as ports.

Embedding parallel codes in sequential components raises some problems. Usually, parallel codes are executed by processes. Each process owns its private address space and uses a message passing paradigm like MPI to exchange data with other processes (SPAS model). A first solution consists in allowing only one process to handle the component ports to talk with other components. This solution leads first to a bottleneck in the communications between two parallel components and second it leads to a modification of the parallel code to introduce this master/slave pattern: the node handling the port is a kind of master, the other nodes are the slaves. A second solution would be to require that all communications have to be done through the software component model. This does not seem to be a good solution: first, modifying existing codes is a huge work. Second, parallel oriented communication paradigms like MPI are much more tailored to parallelism while the component communication paradigm is more oriented toward distributed computing.

A better solution would be to let parallel components choose their internal communication paradigm and to allow all processes belonging to a parallel component to participate to inter-parallel component communications. Hence, a data transfer from the address spaces of the source parallel component to the address spaces of the destination component can generate several communication flows. It should support data redistribution as the source and destination data distributions may be different.

The only specification that we are aware of with respect to high-performance components is the work done by the Common Component Architecture Forum [6] (CCA). The CCA Forum objectives are *"to define a minimal set of standard interfaces that a high-performance component framework has to provide to components, and can expect from them, in order to allow disparate components to be composed together to build a running application"*. The model, currently being developed, does not intentionally contain an accurate definition of a CCA component. It only defines a set of APIs. It is a low level specification: only point-to-point communications are defined. There is also a notion of collective ports, that allow a component to broadcast data to all the components connected to this port. While the CCA's goal is to define a model which specifically targets

high performance computing, we aim to adapt an existing standard to high performance computing.

There are several works about parallel objects like PARDIS [18] or PaCO [27]. The OMG has started a normalization procedure to add data parallel features to CORBA. The current specification [22] requires modifications to the ORB (the CORBA core). Another work, PaCO++ [10], is an evolution of PaCO that targets efficient and portable parallel CORBA objects.

The contribution of this paper is to study the feasibility of defining and implementing parallel components within the CORBA Component Model (CCM). We choose to work with CCM because it inherently supports the heterogeneity of processors, operating systems and languages; it is an open standard and there are several implementations being developed with an Open Source license. Moreover, the model provides a deployment model. CORBA seems to have some limitations but it appears possible to overcome most of them. For example, we have shown that high performance CORBA communication can be achieved [9]. The type issues related to the IDL can be solved by defining and using domain specific types or value-types to handle complex numbers or graph types.

Our objective is to obtain both parallelism transparency and high performance. Transparency is required because a parallel component needs to look like a standard component at the assembly phase and at the deployment phase. The effective communications between parallel components need to be hidden to the application designer. To obtain high performance, we propose to apply a technique that allows all processes of a parallel component to be involved in inter-component communications. Thus, inter-parallel component communications remain efficient when increasing the number of nodes of a parallel component.

The remaining of this paper is divided as follows. Section 2 presents a brief discussion about objects and components. The CORBA component model is presented in Section 3. GridCCM, our parallel extension to the CORBA component model, is introduced in Section 4. Some preliminary experimental results are reported in Section 5. Section 6 concludes the paper.

2 Software component programming model

One major issue when writing an application is the selection of an adequate programming model. Several programming models have emerged; each of them increasing the level of abstraction. The object oriented programming model is currently the most successful.

Software component is expected to bring an improvement to software technology similar to the substantial advantages object-oriented programming has provided over structured programming. However, while object-oriented programming targets application design, component software technology emphasizes component composition and deployment. Before introducing the software component technology, the object-oriented programming model is briefly reviewed.

2.1 Object-oriented programming

Object is a powerful abstraction mechanism. It has demonstrated its benefits, in particular in application design, in a very large number of applications.

Most recent programming languages such as C++, JAVA, Python or Caml are object-oriented. The next version of FORTRAN, i.e. FORTRAN 2000, has clearly adopted an object oriented approach.

However, objects have failed some of their objectives. Code reuse and maintenance are not satisfactory mainly because object dependencies are not very explicit. For example, it is very difficult to find object dependencies in a large application involving hundreds of objects using inheritance and delegation. Experience has shown that it is better to use an approach only based on delegation [15] that allows objects to be "composed".

For distributed applications, objects do not intrinsically provide any support for deployment. For example, neither JAVA RMI [16] nor CORBA 2, two distributed object-oriented environments, provide a way to remotely install, create or update objects.

The situation is even worse for distributed applications with many distributed objects to be deployed on *several* machines. The application needs to explicitly interconnect the objects. A change in the application architecture requires modifications of the code.

Despite its benefits, object oriented programming lacks some important features for distributed applications. Software components aim at providing them.

2.2 Software component

Software component technology [29] has been emerging for some years [7] even though its underlying intuition is not very recent [20]. Among all the definitions of software components, here is Szyperski's definition [29]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

First, the main component operation is the composition with other components. This composition is done through well-defined interfaces: components need to agree on the contract related to their connected interfaces. This agreement brings a lot of advantages: the abstraction of the service is made explicit. In particular, interfaces are strongly typed so that checks such as type conformance can be performed at connection time.

Second, a component inherently embeds deployment capabilities. All the information like the binary code (i.e. the code of the component), the dependencies in terms of processors, operating systems and libraries are embedded into the component. A component may also embed binary codes for different processors or operating systems. The adequate version of the binary is automatically selected by the deployment tool. So, deployment in a heterogeneous environment is made easier.

Building an application based on components emphasizes programming by *assembly*, i.e. manufacturing, rather than by *development*. The goals are to focus expertise on domain fields, to improve software quality and to decrease the time to market thanks to reuse of existing codes.

Components exhibit advantages over objects. Applications are naturally more modular as each component represents a functionality. Code reuse and code maintainability are eased as component are well-defined and independent. Last, components provide mechanisms to be deployed and to be connected in a distributed infrastructure. Thus, they appear very well suited for grid computing.

3 CORBA Component Model

This section begins with an overview of the CORBA architecture. Then, the CORBA component model (CCM) is presented. It ends with a discussion on the relationship between CCM and grid environments.

3.1 CORBA overview

CORBA [24] (*Common Object Request Broker Architecture*) is a technical standard which describes a framework to create distributed client-server applications. CORBA is based on an Object Request Broker (*ORB*) which transports communications between clients and servers. Communications are based on the method invocation paradigm like JAVA RMI. CORBA manages the heterogeneity of the languages, computers and networks. For interoperability purposes, the CORBA model defines a protocol for inter-ORB communications. The servers interfaces are defined with a neutral language named IDL (*Interface Definition Language*).

To make an application, the designer has to describe the server's interface in IDL. The compilation of this description generates stubs (client side) and skeletons (server side). The stub's role is to intercept client invocation and to transmit it to the server through the ORB. The skeleton's role is to receive the client's invocations and to push them to the service implementation. Stubs and skeletons may be generated in different languages. For example, a stub can be in JAVA whereas a corresponding skeleton is in C++. The IDL language mapping is defined for many languages like C, C++, JAVA, Ada, Python but not for FORTRAN though it is feasible as shown within the European Esprit PACHA project.

Client invocations are received at the server side by an adapter (the *Portable Object Adapter*) that delivers requests to the adequate object implementation. Figure 1 presents an overview of the CORBA's architecture.

Stubs and skeletons are not required by CORBA. Dynamic invocations are possible on the client side (Dynamic Invocation Interface or DII) or on the server side (Dynamic Skeleton Interface or DSI). Hence, it is possible to dynamically create, discover or use interfaces.

3.2 CORBA Component Model

The CORBA Component Model [23] (CCM) is part of the lastest CORBA [24] specifications (version 3). The CCM specifications allow the deployment of components into a distributed environment, that is to say that an application can deploy interconnected components on different heterogeneous servers in one operation.

CCM abstract model

A CORBA component is represented by a set of ports described in the CORBA 3 version of the OMG Interface Definition Language (IDL), which extends the OMG IDL of the version 2 of CORBA. There are five kinds of ports as shown in Figure 2.

Facets are named connection points that provide services available as interfaces.

Receptacles are named connection points to be connected to a facet. They describe the component's ability to use a reference supplied by some external agent.

Event sources are named connection points that emit typed events to one or more interested event customers, or to an event channel.

Event sinks are named connection points into which events of a specified type may be pushed.

Attributes are named values exposed through accessor (read) and mutator (write) operations. Attributes are primarily intended to be used for component configuration, although they may be used in a variety of other ways.

Facets and receptacles allow a synchronous communication model based on the remote method invocation paradigm to be expressed. An asynchronous communication model based on the transfer on some data is implemented by the event sources and sinks.

A component is managed by an entity named *home*. A home provides factory and finder operations to create and/or to find a component instance. For example, a home exposes a **create** operation that locally creates a component instance.

CCM execution model

CCM uses a container based programming model. Containers provide the run-time execution environment for CORBA components. A container is a framework for integrating transactions, security, events, and persistence into a component's behavior at runtime. Containers provide a standard set of services to a component, enabling the same component to be hosted by different container implementations. All component instances are created and managed at runtime by its container.

CCM deployment model

The deployment model of CCM is fully dynamic: a component can be dynamically connected to and disconnected from another component. For example, Figure 3 describes how a component **ServerComp** can be connected to a component **ClientComp** through the facet **FacetExample**: a reference is obtained from the facet and then it is given to a receptacle. Moreover, the model supports the deployment of a static application. In this case, the assembly phase has produced a description of the initial state of the application. Thus, a deployment tool can deploy the components of the application according to the description. It is worthwhile to remark that it is just the initial state of the application: the application can change it by modifying its connections and/or by adding/removing components.

The deployment model relies on the functionality of some fabrics to create component servers which are hosting environments of component instances. The issues of determining the machines where to create the component servers and actually how to create them are out of the scope of the CCM specifications.

3.3 CORBA Component Model and grid environments

This section aims at clarifying the relationships between CCM and a grid environment. The subject is large enough to deserve a dedicated paper. We restrict ourselves to sketch some general relationships.

The key point is that CCM and a grid environment such as OGSA do not have the same objectives. CCM is a programming model that describes how to program an application and more precisely its life cycle. A grid environment deals with resource management. The current understanding of grids [14, 17] assumes that a grid environment has in particular to provide some mechanisms to discover, schedule and

allocate resources within a security framework. These aspects are related to resource management not to the definition of an application programming model. Also, it seems to us that an application should not be restricted to run only in a grid environment. For example, it should be possible to execute it within a parallel machine without having to deal with grid environment issues.

As briefly described in Section 3.2, the specifications of CCM rely on some external mechanisms to discover computing resources, and to create processes (the component servers). Thus, the objectives of CCM and the goals of a grid environment are clearly complementary. CCM is a programming model which allows the user to write his application a priori *without involving resource management issues*. These issues should be taken into account by an integration of the deployment model of CCM and of a grid environment. A grid-aware CCM deployment tool appears to be a natural client for a grid environment in at least two steps. The first step is the discovery of resources that satisfy the requirements of the components. The second step is the actual creation of the component servers into the allocated machines.

Last, grid environments need to define their own communication models to achieve interoperability. For example, the OGSA platform is built on top of Grid Services and its associated communication protocol. However, there is *a priori* no relationship between them and the communications utilized by the application such as CORBA or MPI. So, all communications models should cohabit.

4 A Parallel CORBA Component Model: GridCCM

The CORBA component model is an interesting component model mainly because its heterogeneity support, its container model and its deployment model. However, it does not provide an efficient support to embed parallel codes into components. This section presents GridCCM, a parallel extension to the CORBA Component Model. The GridCCM's goal is to study the concept of parallel components.

4.1 Definition of parallel components

Our objective is to efficiently encapsulate parallel codes into CORBA components with as few modifications to parallel codes as possible. Similarly, we target to extend CCM but not to modify the model. The choice

stems from several considerations. First, we aim at defining a *portable* extension to CCM so that it can be added to any implementation of CCM. Second, the parallelism of a component appears to be an implementation issue of the component. So, we do not need to modify the OMG Interface Definition Language which is independent from implementation issues.

We currently restrict ourselves to only supporting the *Single Component Multiple Data* (SCMD) execution model. SCMD defines an execution model for parallel components similar to the *Single Program Multiple Data* execution model for parallel programs. This choice stems mainly from the consideration that most parallel codes we target are actually SPMD.

In an SPMD code, each process executes the same program but on different data. Each process exchanges data with other via a communication library like MPI [28] (*Message Passing Interface*). Figure 4 shows the GridCCM vision of a parallel component in the CORBA framework. The SPMD code continues to be able to use MPI for its intra-component inter-process communications, but it uses CORBA to communicate with other components. In order to avoid bottlenecks, all processes of a parallel component can participate to inter-component communications. This scheme has been successfully demonstrated with parallel CORBA object [10]: an aggregated bandwidth of 103 MB/s (820 Mb/s) has been obtained on VTHD [4], a French high-bandwidth WAN, between two 11-node CORBA parallel objects. An aggregated bandwidth of 1.5 GB/s has been obtained on a Myrinet 2000 network between two 8-node CORBA parallel objects.

GridCCM's objective is to extend CCM to allow an efficient encapsulation of SPMD codes. For example, GridCCM has to be able to aggregate bandwidth when two parallel components exchange data. As data may need to be redistributed during communications, the GridCCM model has to support it as transparently as possible. The client should only have to describe how its data are locally distributed and the data should be automatically redistributed accordingly to the server's preferences. A GridCCM component should appear as close as possible to a *standard* component. For example, a standard component should be able to connect itself to a parallel component without noticing the latter is a *parallel* component.

To achieve these objectives, GridCCM defines the notion of parallel components.

Definition: *a parallel component is a collection of identical sequential components. It executes in parallel all or some parts of its services*.

The designer of a parallel component needs to describe the parallelism of the component in an auxiliary XML file. This file contains a description of the parallel operations of the component, the distributed arguments of these operations and the expected distribution of the arguments. An example is given in Section 4.4.

4.2 Parallelism support in GridCCM

As previously presented, GridCCM *adds* new functionalities in order to handle parallel components to CCM. But, it does not require to modify the CCM specifications. The parallelism is supported thanks to the addition of a software layer, hereafter called the GridCCM layer, between the client's code and the stub's code as illustrated in Figure 5. This scheme has been successfully used with PaCO++ [10] for a similar problem: the management of parallel CORBA objects.

The role of the GridCCM layer is to allow a transparent management of the parallelism. An invocation to a parallel operation of a parallel component is intercepted by the GridCCM layer at the client side. The layer sends the distributed data from the client nodes to the server nodes. The corresponding GridCCM layer at the server-side waits to receive all the data before invoking the user's code implementing the CORBA operation. The argument data redistribution is actually performed either on the client side, or on the server side, or during the communication between the client and the server. The decision depends on several constraints like feasibility (mainly memory requirements) and efficiency (client network performance versus server network performance).

The parallel management layer is generated by a compiler specific to GridCCM, as illustrated in Figure 6. This compiler uses two files: the IDL description of the component and the XML description of the component's parallelism. In order to have a transparent layer, an IDL description is generated during the generation of the component from the user's IDL. The GridCCM layer internally uses the operations defined in this IDL description to actually invoke an operation on the server. The original IDL interface is used between the user code and the GridCCM layer on the client and server sides.

In the new IDL interface, the user arguments declared as distributed have been replaced by their equivalent distributed data types. Because of this transformation, some constraints may exist about the types

that can be distributed. The current implementation requires the user type to be an IDL `sequence` type, that is to say a variable-length 1D array. So, one dimension distribution can automatically be applied. This scheme can easily be extended for multidimensional arrays: a 2D array can be mapped to a sequence of sequences, etc. It is worthwhile to note that the IDL types do not allow a direct mapping of "scientific" types like multidimensional arrays or complex numbers. However, CORBA is just a communication technology. Higher level environments, like code coupling environments such as HLA [5] or MpCCI [2], should define these "scientific" types. As the expressiveness of the CORBA type system is roughly identical to this of the C language, it should not be an issue.

In order to have a complete parallelism support, GridCCM has to support parallel exceptions [19]. This task is also devoted to the GridCCM layer though it is still under investigation.

4.3 Home and Component proxies

A GridCCM objective is to allow a parallel component to be seen as a standard component. As a parallel component is a collection of sequential components, every node involved in the execution of a parallel component holds an instance of a sequential component and its corresponding home. These components and homes are not directly exposed to other components. GridCCM introduces two internal entities, named the *HomeManager* and the *ComponentManager*, that are proxies respectively for the homes of the component instance' and for the component instances themselves.

An application that needs to create a parallel component interacts according to the CCM standard with the *HomeManager*. The *HomeManager* is configured during the parallel component deployment phase. The references to the homes are collected via a specific interface to the *HomeManager*.

Similarly, when a client gets a reference to a parallel component, it actually receives a reference to the *ComponentManager*. When two parallel components are connected, the *ComponentManager*s of both parallel components transparently exchange information so as to configure the GridCCM's layers.

Figure 7 shows an example of connection between the parallel components A and B. First, the deployment tool connects A with B using the standard CCM protocol (see Figure 3). Second, A's *ComponentManager* retrieves information from B's *ComponentManager*. For example, component B returns the references of all

B's nodes to component A. Third, A's *ComponentManager* configures the layers of all A's nodes. Fourth, when component A invokes a parallel operation of B, all A's nodes may participate to the communication with B's nodes.

4.4 Example

This section illustrates the definition and the use of a parallel component through an example.

A component, named *A*, provides a port of name *myPort* which contains an operation that computes the average of a vector. Figure 8 shows the IDL description of the component and the IDL description of the *Average* interface. The facet and the interface are described without any OMG IDL modification. The facet of type *Average* is declared with the keyword *provides*. It has one operation *compute* which takes a vector and returns a value.

The component implementer needs to write an XML file that describes the parallelism of component *A*. This file, shown in figure 9, is not presented in XML for the sake of clarity. An important remark is that this file is not seen by the client. In this example, the operation *compute* is declared parallel and its first argument is block distributed.

Standard clients normally use the *myPort* facet. However, a parallel client has to specify how the data to be sent are locally distributed thanks to a client-side API. Figure 10 shows an example of this API whose specification is not yet stable mainly because we are still working to allow a redistribution library [1] to be plugged into the GridCCM model. In the example of Figure 10, the parallel client gets a reference of the facet *myPort*. Then, it configures its local GridCCM layer with the method *init_compute* before invoking the *compute* operation.

It is worthwhile to note that a GridCCM component offers two different views as depicted in Figure 11. Externally, a parallel component looks like a standard CCM component but it also offers an extended interface to GridCCM-aware components. Internally, it exhibits a parallel oriented interface that is designed to be used by the parallel component implementer.

5 Preliminary experiments

This section presents some preliminary experiments that evaluate the feasibility and the scalability of GridCCM. These experiments also show that the model is generic with respect to two different CCM implementations. Before presenting the experimental protocol and reporting some performance measurements, we need to introduce PadicoTM.

5.1 PadicoTM

GridCCM requires several middleware systems at the same time, typically CORBA and MPI. They should be able to efficiently share the resources (network, processor, etc.) without conflicts and without competing with each other. These issues becomes very important if both middleware systems want to use the same network interface, like Myrinet or TCP/IP. *A priori*, nothing guarantees that a CORBA implementation can coexist with a MPI implementation. There are several cases that lead to an application crash as explained in [11].

In order to be sure to have a correct and efficient cohabitation of several middleware systems, we have designed PadicoTM [11]. PadicoTM is an open integration framework for communication middleware and runtimes. It allows several middleware systems, such as CORBA, MPI, or SOAP, to be used at the same time. It provides an efficient and transparent access to all available networks with the appropriate method.

PadicoTM enables us to simultaneously utilize CORBA and MPI and to associate both the CORBA and the MPI communications to the network that we chose. All experiments involving code not written in JAVA have been done using PadicoTM.

5.2 Experimental protocol

The experimental setup, shown in Figure 12, contains two parallel components (*Customer* and *Server*), a GridCCM-aware sequential component and a standard sequential CORBA client. First, the standard CORBA client creates a vector and sends it to the sequential component. Second, the vector is sent to *Customer* with a transparent distribution in *Customer*'s nodes using a bloc distribution. Third, *Customer* invokes a *Server*'s

service which takes the distributed vector as an in-out argument. The *Server* method implementation contains only an MPI barrier. Then, the vector is sent back to *Customer*. In this example, the two parallel components use MPI but they are running in distinct MPI environments (i.e. two different MPI_COMM_WORLD). *Customer* uses MPI for barriers and reductions. *Server* only uses MPI for barriers.

There is no modification in the sequential component code to call a *Customer*'s parallel service. The parallel service is transparent for the sequential code. With regard to the connection, the deployment application complies with the CCM specifications. The *ComponentManager*s perform their role correctly: the parallel component nodes are transparently connected according to the CCM specifications.

5.3 Performance measurements

We have implemented a preliminary prototype of GridCCM on top of two *OpenSource* CCM implementations. The first prototype is derived from OpenCCM [30]. OpenCCM is made by the research laboratory LIFL (*Laboratoire d'Infomatique Fondamentale de Lille*). It is written in JAVA. The second prototype has been derived from MicoCCM [25]. MicoCCM is an implementation based on the Mico ORB. It is written in C++.

The test platform is a Linux cluster of 16 dual-pentium III, 1 Ghz with 512 MB of memory interconnected with both a Myrinet-2000 network and a switched Fast Ethernet network. For OpenCCM, the JAVA Runtime Environment is the SUN JDK 1.3 and ORBacus 4.0.5 is the ORB we used.

The measurements have been made in the first parallel component (*Customer*). After an MPI barrier to synchronize all *Customer*'s nodes, the start time is measured. *Customer* performs 1000 calls to *Server*. Then, the end time is taken. The MPI communications always use the Myrinet network.

Figure 13 reports the latency for the MicoCCM/Myrinet configuration and the aggregated bandwidth between the two parallel components for several configurations. First, the measured latency is the sum of the plain MICO CORBA invocation and of the MPI barrier. The variation in the latency reflects the variation of the time taken by the MPI barrier operation in function of the number of nodes. Thus, the GridCCM layer, without data redistribution, does not add a significant overhead.

Both C++ and JAVA implementations of GridCCM efficiently aggregate the bandwidth. As expected, the C++ implementation is more efficient than the JAVA implementation and the C++ version's bandwidths

are higher for the Myrinet network than for the Ethernet network. In all case, the aggregated bandwidth is linear with the number of nodes: GridCCM offers a scalable mechanism to aggregate bandwidth.

In the Myrinet-2000 experiments, an aggregated bandwidth of 280 MB/s is reached between two 8-node parallel components. This number is less than the 1.5 GB/s aggregated bandwidth that we achieved between two 8-node parallel CORBA objects on the same hardware configuration [12]. As explained in [9], the MICO implementation always performs a data copy during a CORBA communication while omniORB, another CORBA Open Source implementation, achieves a zero copy communication when the source and destination node have the same memory data representation (endian-ness, word length).

The same behavior is observed for the latency: while MICO on top of Myrinet has a latency of 62 μs, omniORB achieves 20 μs. High performance for CORBA communications on top of a high performance network such as Myrinet requires an efficient CORBA implementation such as omniORB. If it is available for CORBA objects, it is not yet available for CORBA components.

6 Conclusion

Computational grids allow new kinds of applications to be developed. For example, code coupling applications can benefit a lot from grids because grids provide very huge computing, networking and storage resources. The distributed and heterogeneous nature of grids raises many problems to application designer. Software component technology appears to be a very promising technology to handle such problems. However, software component models do not offer a transparent and efficient support to embed parallel codes into components.

This paper studies an extension of the CORBA Component Model in order to introduce the concept of parallel components. This study is mainly focused on the CCM abstract model and on the execution model. The proposed parallel CCM model, called GridCCM, has been successfully added on top of two existing CCM implementations. With both prototypes, the performance results have shown that the model is able to aggregate bandwidth without introducing any noticeable overhead.

The next step of our work is to finalize the GridCCM model, in particular with respect to data redis-

tribution and to parallel component exceptions. The GridCCM model does not aim at defining any data distributed type. However, it should be able to handle them thanks to some data redistribution libraries. Another important ongoing work is the integration of grid environments within the CCM deployment model. Hence, to deploy an application into a grid, the CCM deployment model has to interact with grid environments. With respect to security issues, a parallel component should behave as a standard CORBA component. Then, the issue will be the handling of the interactions between the CORBA security infrastructure and a grid environment security infrastructure. Supporting system services like persistence or transaction is another issue that needs to be further investigated. Moreover, the prototype needs to be finalized. In particular, the current GridCCM layer is mainly hand-written. A dedicated compiler will soon be operational. Last, we plan to test the model with real applications. One of them will be an EADS code coupling application which involves five MPI codes.

References

[1] The DARPA data reorganization effort. http://www.data-re.org.

[2] MpCCI - mesh-based parallel code coupling interface. http://www.mpcci.org.

[3] The TeraGrid project. http://www.teragrid.org.

[4] The VTHD project. http:/www.vthd.org.

[5] IEEE standard for modeling and simulation (M&S) high level architecture (HLA)—federate interface specification. IEEE Standard 1516, September 2000.

[6] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, Redondo Beach, CA, August 1999.

[7] L. Barroca, J. Hall, and P. Hall. *Software Architectures: Advances and Applications*, chapter An Introduction and History of Software Architectures, Components, and Reuse. Springer Verlag, 1999.

[8] E. Cerami. *Web Services Essentials*. O'Reilly & Associates, 1st edition, February 2002.

[9] 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.

[10] A. Denis, C. Pérez, and T. Priol. Portable parallel CORBA objects: an approach to combine parallel and distributed programming for grid computing. In *Proc. of the 7th Intl. Euro-Par'01 conf.*, pages 835–844, Manchester, UK, 2001. Springer.

[11] A. Denis, C. Pérez, and T. Priol. PadicoTM: An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 19:575–585, 2003.

[12] A. Denis, C. Pérez, T. Priol, and A. Ribes. Padico: A component-based software infrastructure for grid computing. In *17th International Parallel and Distributed Processing Symposium (IPDPS2003)*, Nice, France, April 2003. IEEE Computer Society.

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

[14] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[16] W. Grosso. *Java RMI*. O'Reilly & Associates, 2001.

[17] W. Johnston and J. Brooke. Core grid functions: A minimal architecture for grids, July 2002. Working Draft, Version 3.1.

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

[19] H. Klaudel and F. Pommereau. A concurrent semantics of static exceptions in a parallel programming language. In J.-M. Colom and M. Koutny, editors, *Applications and theory of Petri nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 204–223. Springer, 2001.

[20] M. D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO.

[21] Sun Microsystems. Enterprise JavaBeans Specification, August 2001.

[22] OMG. Data parallel CORBA. Technical report, 2001. Document orbos/01-10-19.

[23] OMG. Corba component model. Technical report, jun 2002. Document – formal/02-06-65.

[24] OMG. Common object request broker architecture (CORBA/IIOP). Technical report, nov 2003. Document – formal/02-11-03.

[25] F. Pilhofer. The MICO CORBA component project. http://www.fpx.de/MicoCCM.

[26] D. S. Platt. *Understanding COM+*. Microsoft Press, 1999.

[27] C. René and T. Priol. MPI code encapsulating using parallel CORBA object. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, Redondo Beach, CA, August 1999.

[28] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The complete reference*. MIT Press, 1995.

[29] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[30] M. Vadet, P. Merle, R. Marvie, and J.-M. Geib. The OpenCCM platform. http://corbaweb.lifl.fr/OpenCCM/.

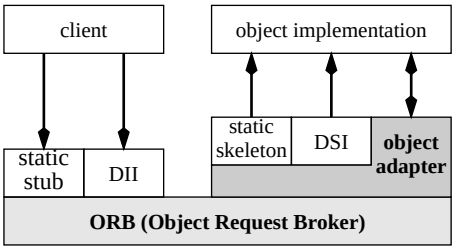

Figure 1: CORBA architecture.

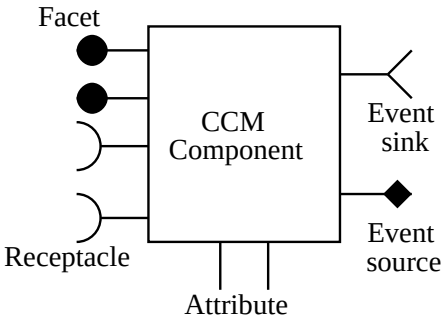

Figure 2: A CCM component.

```
foo ref = ServerComp->provide_FacetExample();
ClientComp->connect_FacetExample(ref);
```

Figure 3: Example of code to connect two components.

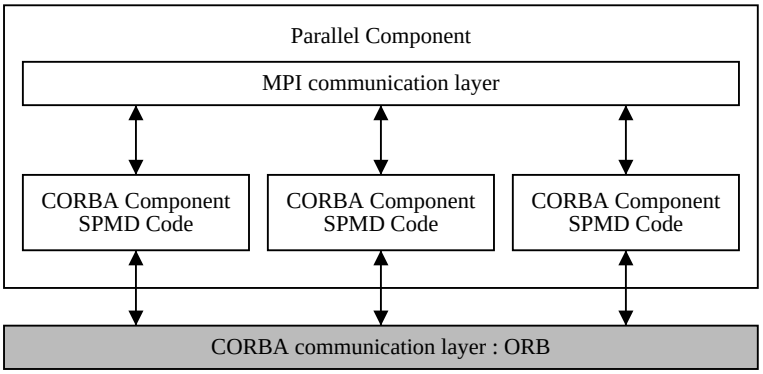

Figure 4: Parallel component concept.

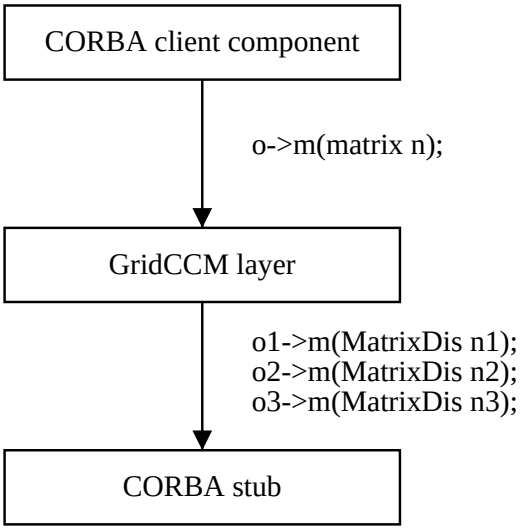

Figure 5: The user invokes a method on a remote component. The GridCCM layer actually uses an internal version of this method on the different instances of the parallel component.

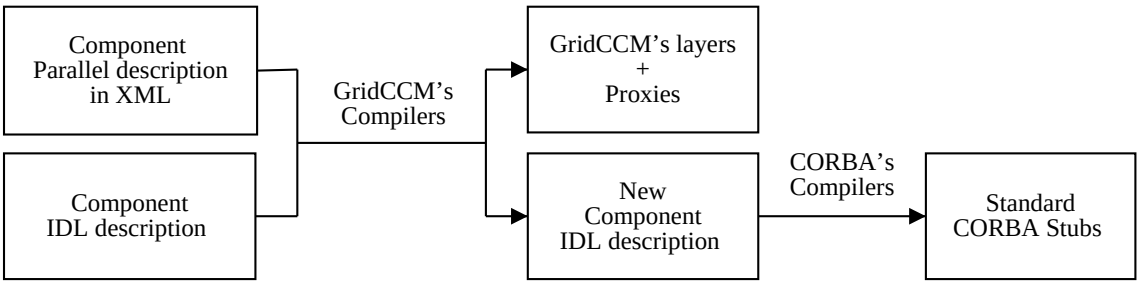

Figure 6: The different compilation phases to generate a parallel component.

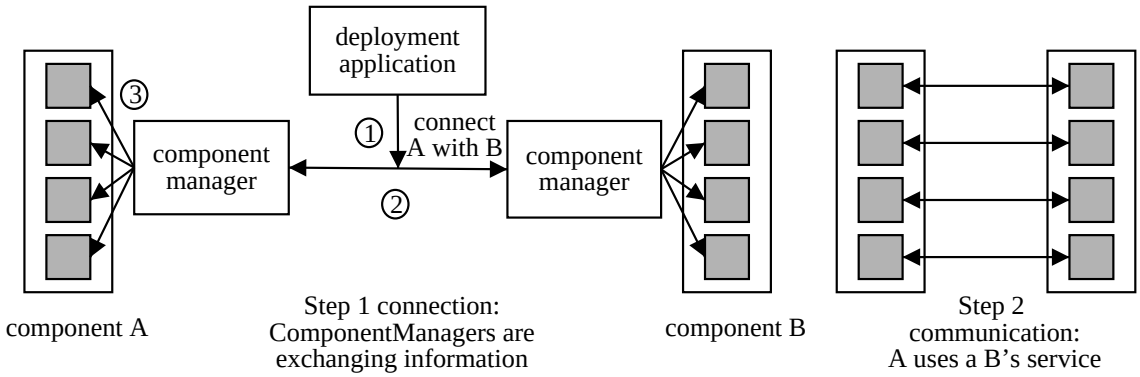

Figure 7: An example of connection and communications between two parallel components.

```
// Interface Average definition
typedef sequence<long> Vector;
interface Average {
  long compute(in Vector v);
};
// Component A definition
component A {
  provides Average myPort;
};
```

Figure 8: A component IDL definition.

```
// Parallelism definition
Component: A
Port     : myPort
Port Type: Average
Operation: compute
Argument1: bloc
Result   : noReduction
```

Figure 9: Parallelism description of a facet.

```
 // Get a reference to the facet myPort
Average avg = aComponent.get_connection_myPort();
 // Configure it with a GridCCM API
avg.init_compute('bloc',0); // 1st arg is bloc distributed
 // "Standard" Ccm call
avg.compute(MyMatrixPart);
```

Figure 10: A client initializing and using a parallel operation.

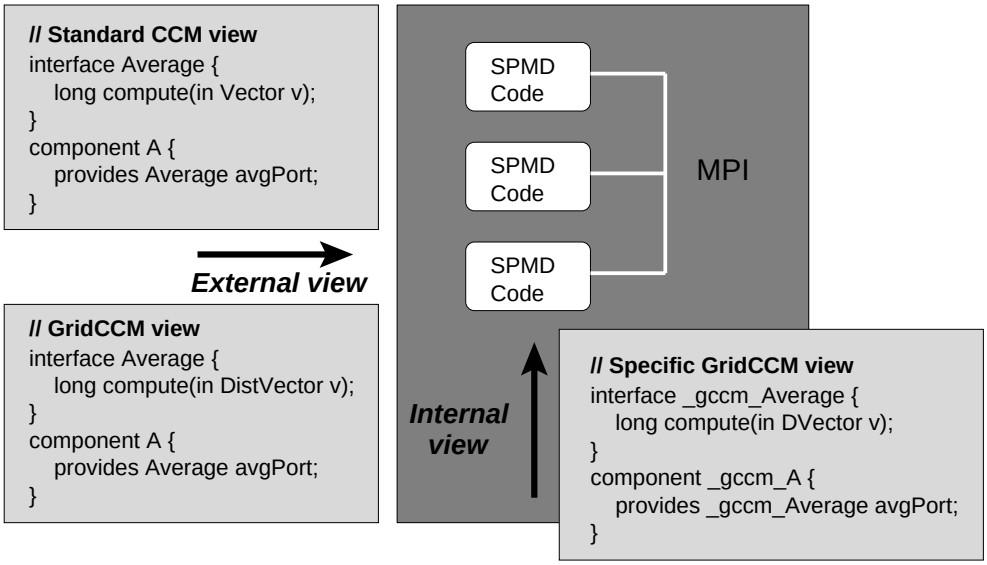

Figure 11: The two kinds of views of a GridCCM component.

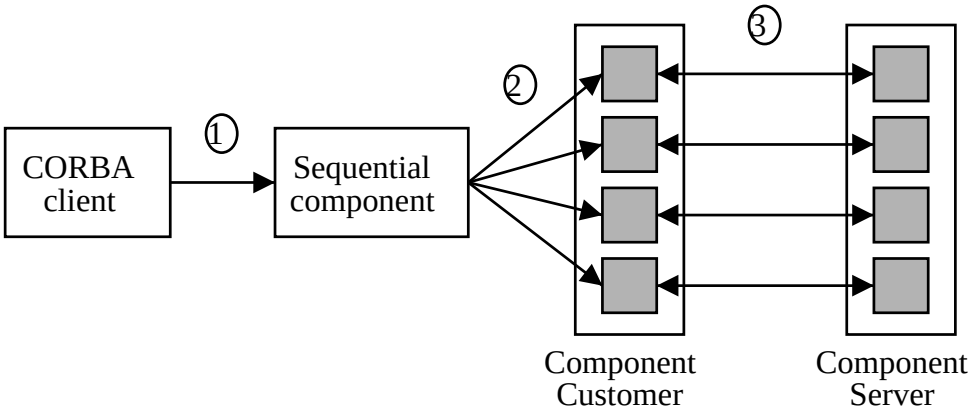

Figure 12: Experimental protocol.

Parallel component node number	Latency in μs	Aggregated bandwidth in MB/s		
	C++/Myrinet	C++/Myrinet	C++/Eth	JAVA/Eth
1 to 1	62	43	9.8	8.3
2 to 2	93	76	19.6	16.6
4 to 4	123	144	39.2	33.2
8 to 8	148	280	78.4	66.4

Figure 13: Latency and bandwidth between the parallel component *Customer* and the parallel component *Server*. The two parallel components are deployed over the same number of nodes.