Data distribution for Parallel CORBA Objects

Tsunehiko Kamachi[1], Thierry Priol[2], and Christophe René[2]

[1] C&C Media Research Laboratories, NEC Corporation, 4-1-1 Miyazaki, Miyamae-ku, Kawasaki, Kanagawa 216-8555 Japan
[2] IRISA/INRIA, Campus de Beaulieu - 35042 Rennes Cedex, France

Abstract. The design of application for Computational Grids relies partly on communication paradigms. In most of the Grid experiments, message-passing has been the main paradigm either to let several processes from a single parallel application to exchange data or to allow several applications to communicate between each others. In this article, we advocate the use of a modern approach for programming a Grid. It is based on the use of distributed objects, namely parallel CORBA objects. We focus our attention on the handling of distributed data within parallel CORBA objects. We show some performance results that were obtained using a NEC Cenju-4 parallel machine connected to a PC cluster.

1 Introduction

With the availability of high-performance networking technologies, it is nowadays feasible to couple several computing resources together to offer a new kind of computing infrastructure that is called a *Computational Grid* [4]. Such system can be made of a set of heterogeneous computing resources that are interconnected together through multi-gigabit networks. Software infrastructures, such as Globus [3] or Legion [6], provide a set of basic services to support the execution of distributed and parallel programs. One of the problem that arises immediately is how to program such a computational Grid and what is the most suitable communication model for Grid-enabled applications ? It is very tempting to extend existing message-passing libraries so that they can be used for distributed programming. We believe that this approach cannot be seen as a viable solution for the future of Grid Computing. Instead, we advocate an approach that allows the combination of communication paradigms for parallel and distributed programming. This approach, called PaCO, is based on an extension to a well known and mature distributed object technology, namely CORBA.

2 Communication within a Computational Grid

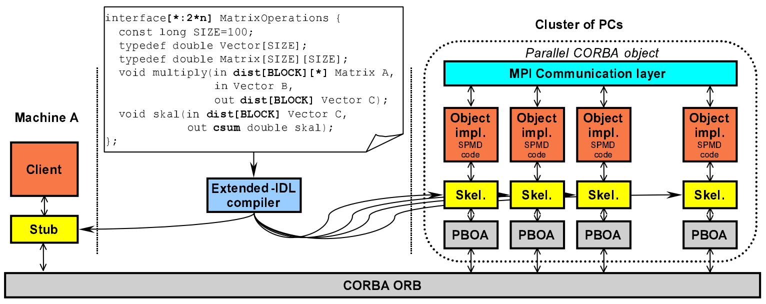
There exist two main approaches to communicate within a computational grid. The first approach is to allow the execution of a parallel code over heterogeneous machines taking benefit of the available computing resources. Research works have recently lead to extend existing message passing libraries to be able to exchange data between heterogeneous computing resources, such as MPICH-G [5], PACX [1] or PLUS [11]. A parallel code, based on one of these communication libraries, can be executed on a Grid with some minor modifications. We think that such approach is relevant since the purpose of these Grid-enabled communication libraries is to allow parallel programming at a larger scale. Such libraries can also be used to connect several parallel codes together to perform coupled simulations. It constitutes the second approach. The objective is to solve new kind of problems that were not affordable due to the lack of computing resources. The aggregating of computing resources may allow the simulation, in a shorter time frame, of complex manufactured products for which different physical behaviors have to be taken into account (structural mechanics, computational fluid dynamics, electromagnetism, noise analysis, etc...). Moreover, distributed execution of simulation codes are nowadays imposed by the way the industrial companies work together to design manufactured products. It requires that each company, participating to the design of a manufactured product, to contribute to the simulation of the whole product by providing access to its own simulation tools. However, a company is often reluctant to give both its simulations tools and the necessary simulation data to other companies (that may act as competitors later on). Therefore, there is a strong need to have part of the simulation of the whole product performed on their own computing resources to avoid the exchange of confidential data (i.e. the model of the object to be simulated). Thus, there is a clear need to have a mechanism to let simulation codes to communicate together. However, such mechanism requires that it is capable of both transferring data and control efficiently between codes.

We think that message-passing is not suitable to connect several parallel codes together. Indeed, message-passing was mainly designed for parallel programming and not for distributed programming; it is mainly to transfer data but not the control. As for instance, if one code would like to call a particular function into another code, this latter has to be modified in such a way that a message type is associated to this particular function. Such modification requires a deep understanding of the code. Moreover, entry points in a code are not really exposed to potential users that would like to include such code into its application. Communication paradigms, such as RPC or distributed objects, offer a much more attractive solution since the transfer of control is implemented by remote invocation that is as simple as calling a function or a method. However, they are not suitable for parallel programming due to their higher communication cost. It is thus clearly difficult to have a single communication paradigm for the programming of computational grids. We advocate an approach, like others [8], that consists in merging several communication paradigms in a coherent way so that they fit the requirements mentioned previously.

The remainder of this paper is structured as follows. Section 2 discusses communication issues for Computational Grids. Section 3 gives an overview of the parallel CORBA object concept. Section 4 describes data redistribution within a parallel CORBA object. Section 5 provides some experimental results. Finally, we conclude in section 6 by laying the grounds for future enhancement.

3 Overview of parallel CORBA object

CORBA is a specification from the OMG (Object Management Group) to support distributed object-oriented applications. CORBA acts as a *middleware* that provides a set of services allowing the distribution of objects among a set of computing resources connected to a common network. Transparent remote method invocations are handled by an Object Request Broker (ORB) which provides a communication infrastructure independent of the underlying network. An object interface is specified using the Interface Definition Language (IDL). An IDL file contains a list of operations for a given object that can be invoked remotely. An IDL compiler is in charge of generating a *stub* for the client side and a *skeleton* at the server side. A stub is simply a proxy object that behaves as the object implementation at the server side. Its role is to deliver requests to the server. Similarly, the skeleton is an object that accepts requests from the ORB and delivers them to the object implementation. The concept of parallel CORBA



```
interface[*:2*n] MatrixOperations {
  const long SIZE=100;
  typedef double Vector[SIZE];
  typedef double Matrix[SIZE][SIZE];
  void multiply(in dist[BLOCK][*] Matrix A,
                in Vector B,
                out dist[BLOCK] Vector C);
  void skal(in dist[BLOCK] Vector C,
              out csum double skal);
};
```

Fig. 1. Encapsulation of MPI-based parallel codes into CORBA objects.

object[1] is simply a collection of identical CORBA objects as shown in figure 1. It aims at encapsulating a MPI code into CORBA objects so that a MPI code can be fully integrated into a CORBA-based application. Our goal is to hide as much as possible of the problems that appear when dealing with coarse-grain parallelism on a distributed memory parallel architecture like a cluster of PCs. However, this is done without entailing a lost of performance when communicating with the MPI code. First of all, the calling of an operation by a client will result in the execution of the associated method by all objects belonging to the collection at the server side. Execution of parallel objects is based on the SPMD execution model. This parallel activation is done transparently by our system. Data distribution between the objects belonging to a collection is entirely handled by the system. However, to let the system to carry out parallel execution and data distribution between the objects of the collection, some specifications have to be added to the component interface. A parallel object interface is thus described by an extended version of IDL, called Extended–IDL as shown in figure 1. It is a set of new keywords (in bold in the figure), added to the IDL syntax[2], to specify the number of objects in the collection, the shape of the virtual node array where objects of the collection will be mapped on, the data distribution modes associated with parameters and the collective operations applied to parameters of scalar types.

[1] we will use parallel object from now on

[2] A more complete description of these extensions is given in [10, 12]

4 Data redistribution in a parallel object

Application programmers have to specify, for each operation, parameters that have to be distributed among the collection and how they are distributed. Since they can define data distribution for a parameter in each parallel object differently, parameter values need to be redistributed. This problem is made difficult due to various configurations at both the client and the server side (data distribution modes, distributed dimension, object collection size or its virtual shape). It is thus necessary to provide a data redistribution mechanism, as part of the operation invocation, to facilitate the coupling of several parallel objects. Furthermore, while users are responsible for data distribution management within a parallel object, data redistribution should have to be handled by a runtime system in order to hide all the operations associated with the invocation of operations from or to a parallel object.

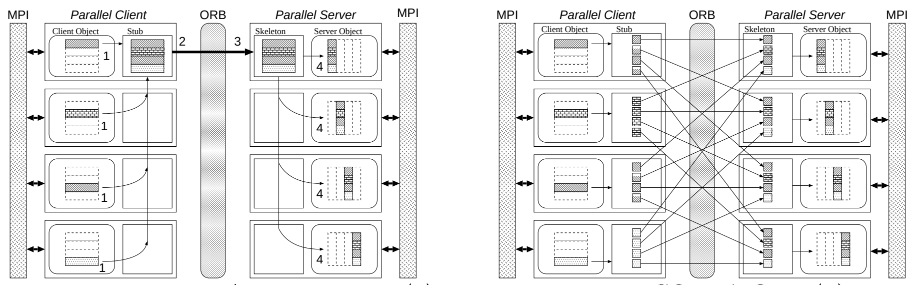


Fig. 2. Master/slave approach (a) Through the CORBA ORB (b)

4.1 Design considerations

The most obvious way to perform data redistribution is to use gather and scatter operations at the client and the server sides. Figure 2-a illustrates such a technique for an operation invocation with a `in` parameter array. Four steps are required: first, one of the stubs gathers distributed data from the client objects using the MPI communication layer (1). Then, it invokes one of the server objects and sends gathered data to it through the CORBA ORB (2). During the third step, the skeleton of the activated server object receives the data from the ORB (3). Finally, the skeleton activates the remaining server objects and scatters data to them using MPI (4). Although this technique is simple to implement, it has some severe drawbacks. The gathering and scattering of data values associated with distributed parameters do not offer a good scalability when the number of objects increases. Data transfer between two collections is serialized through only one object. Furthermore, the gathering of data by one stub is memory consuming.

To avoid this problem, we need to incorporate both a parallel invocation of operations and a data redistribution strategy into parallel objects. One possible approach is to let client objects to send data values of the distributed parameters to the server objects through the ORB as shown in figure 2-b. In this approach, each client splits its own data according to the data distribution at the server side, and sends them to the relevant server objects directly. On the server side, each object receives pieces of data which it should own from several client objects. This approach suffers from a higher number of ORB requests compared to the master/slave approach. Since the ORB is usually much more slower than message-passing layers, we have to keep the number of requests as low as possible. Taken into account these remarks, we propose the following technique. Data

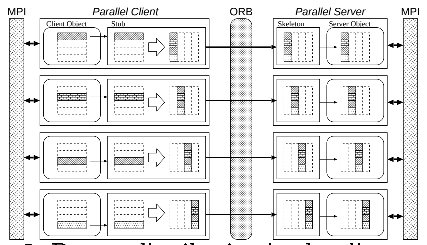


Fig. 3. Data redistribution in the client stub.

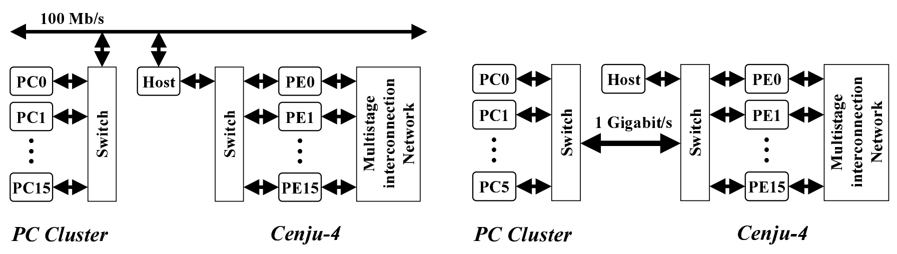
redistribution is performed by the client side or the server side before or after the sending of the request associated with the operation invocation. Figure 3 shows the case of data redistribution at the client side. Accordingly, all communications needed for the redistribution are carried out over the high-speed network of the parallel machines on which the parallel object, acting as the client, is running. The reason why we incorporated data redistribution into both the client and server sides is to obtain the maximum performance from the available communication resources. This allows us to select the most suitable place to perform data redistribution depending on the performance of the network at the client or the server side. More precisely, when data redistribution is performed at the client side, it is handled by the stubs that are aware of the data distribution mode of both the client and the server. First, all the stubs exchange their own data using the MPI communication layer to prepare data meeting the distribution mode at the server object, then each stub sends the redistributed data to the relevant server object through the ORB. When the data redistribution is performed by the server, the client has to provide to the server extra information associated with the distribution of the parameter along with the parameter data values. Such information includes, for each distributed parameter, the distribution mode and the distributed dimension; the virtual shape of the objects is also part of this information. When the skeleton receives such information, it redistributes the parameter data values to adapt the client data mapping to the server one.

4.2 Implementation

The generation of the stub code is rather complex due to various possibilities at the client side. As for instance, a client can be either a standard CORBA object or a parallel object. In the later case, data that have to be sent when invoking an operation on a parallel object are distributed among the objects of the client collection. Therefore, one possibility is to perform the data redistribution by the stub, using a data redistribution library, so that it fits the one at the server side. Another modification to the stub code concerns the invocation mechanism. Since the number of objects of the collection at the client side does not often coincides with the one at the server side, we added a mechanism that associates an object of the client collection to one of the object of the server collection. When there is only one object at the client side, the stub generates a request for each object of the server collection. Similarly, when there is only one object in the server collection, one object of the client collection is associated with this single object. As data redistribution has been done by the stub, the skeleton performs roughly the same work as a standard skeleton. It is worth mentioning that in this situation, skeletons do not need to communicate between each other within the server collection.

Another possibility is to perform data redistribution by the skeleton instead of the stub. In that case, the modification of the stub code generation is very simple. Each object of the client collection sends the data it owns to another object of the server collection. Before sending this data, the stub includes, for each parameter, data distribution information at the client side. It can then call a data redistribution library providing both the client and the server data distribution mode. Once redistribution is performed, the skeleton invokes the implementation method as a standard skeleton. If the parameter has an `inout` or `out` attribute, the skeleton builds the reply where it puts distributed data values following data distribution information sent by the client. It uses again the data redistribution library. This approach has the drawback of adding extra information (data distribution information) to the data sent by the client to the server. Moreover, such technique cannot be used when a sequential object has to invoke a method implemented by a parallel object. In such case, the client has to set up a request for each object of the server collection and thus has to distribute the data to each object of the collection.

To avoid implementing a new data redistribution library, we decided to adapt our stub and skeleton code generation process in such a way that we can exploit existing libraries. These libraries were developed for the High Performance Fortran (HPF) compilation system such as the NEC HPF/SC[7] and the GMD Adaptor system[2]. These two systems support all patterns of data redistribution in the scope of the HPF-1.1 specification. Since our extension for describing data distribution can be seen as a subset of HPF-1.1, all data redistribution patterns are covered. However, there are some limitations in our current implementation due to the difference of execution model between HPF and a parallel object. These data redistribution libraries are intended to be used within stand alone parallel program in which the number of processes is constant. However, we want to use these libraries to reorganize data when two parallel objects communicate. Such parallel programs may run on different number of processors and therefore we may have to redistribute data between two parallel objects that run on different number of processors. Moreover, such libraries were intended to be used with *Fortran* programs whereas we are using *C++*. Therefore, using these libraries requires extra memory copy operations to map *C++* arrays to *Fortran arrays.*



(a) Using a 100 Mb/s Ethernet network (b) Using a 1Gb/s Ethernet network

Fig. 4. Experimental environment.

5 Experimental results

We performed several experiments using two platforms (Figure 4); namely, the NEC distributed-memory parallel computer Cenju-4[9] and a PC cluster. The Cenju-4 has 16 PEs (processing elements) connected via a multistage interconnection network as well as a 100 Mb/s Ethernet network. Each PE consists of a 200MHz VR10000 RISC microprocessor. The PC cluster is a set of PCs connected to a 100 Mb/s Ethernet network. Each PC is equipped with two 450MHz Pentium III processors running Linux. ORB communications between the PEs of the Cenju-4 and the PCs of the PC cluster can go either through the Cenju-4 host machine using a 100 Mb/s network or through a 1Gb/s network. For the experiment, we used the DALIB redistribution library [2]. The MPI communication layer was implemented on the multistage interconnection network on the Cenju-4 and on the fast Ethernet network on the PC cluster. In order to illustrate the effectiveness of our approach, we experimented a simple code; a parallel client issues a single invocation operation to a parallel server with a two dimensional distributed array parameter (long) with a `in` attribute. Distribution of the matrix at the client side follows a `[BLOCK][*]` distribution mode whereas at the server side the matrix to be distributed using a `[*][BLOCK]` distribution mode.

5.1 Comparison with the master/slave approach

In this experiment, we ran both a parallel client and a parallel server on the PC cluster to compare the performance of the master/slave approach with that of our approach. Results presented in Figure 5 make it evident that our approach provides a scalable solution as compared with the master/slave approach. The other point we can observe is that a distinct difference between the performance of the client side redistribution and that of the server side redistribution cannot be seen. This result tells us the overhead of sending extra data in case of the server side redistribution is insignificant with our experimental environment.

5.2 Redistribution at the client versus the server

We measured the performance of a single operation invocation similar to what have be done to compare the master/slave approach with the parallel object one. However, this time we mapped the client onto the PC cluster and the server onto the Cenju-4. Data redistribution is performed either by the stub (using the MPI layer with the Ethernet network of the PC Cluster) or by the skeleton (using the MPI layer with the multistage network of the Cenju-4 parallel system). Results are presented in Figures 6. They were obtained by using either the 100 Mbit/s or the 1 Gbit/s Ethernet network. The time associated with the ORB communication, the memory copy and the data redistribution in the invocation are measured separately. ORB communication time corresponds to

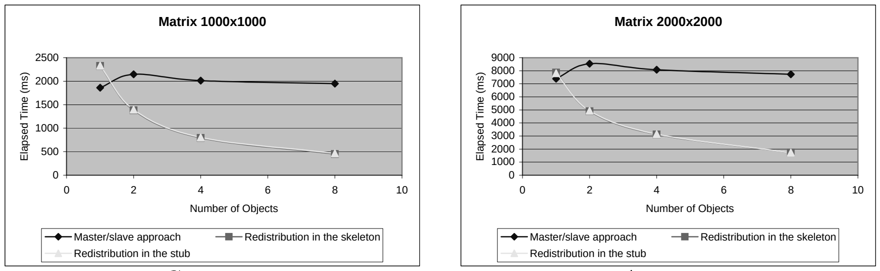


Fig. 5. Comparison our approach with the master/slave approach.

the invocation time without redistribution and memory copy. We cannot see a significant difference in the two test cases (data redistribution within the stub or the skeleton). However, as compared with the results measured within the PC cluster, the ORB communication time between the Cenju-4 and the PC cluster is slow. In addition, when using the 100 Mbit/s Ethernet network, it does not provide a good speedup when number of objects increases. This is because, as shown in Figure 4, all the ORB communications between the PEs in the Cenju-4 and the PCs in the PC cluster have to go through the Cenju-4 host computer. If the PEs are connected to the network directly (using the 1 Gbit/s Ethernet network), the performance and the speedup ratio is improved.

To handle both distributed data and the information about its distribution mode, we introduced a new data structure, called `darray` [12], based on the CORBA `sequence` data structure. Unlike an array, data in the `darray` is stored non-continuously into the memory if the `darray` realizes an array which has more than two dimensions in the same way as the `sequence`. Therefore, two memory copy operations are required in the redistribution process, that is, copying data from `darray` structure to a *Fortran* array before redistribution and copying redistributed data from the *Fortran* array to `darray` structure after redistribution. Moreover, since the difference of memory mapping scheme between *C++* and *Fortran* arrays forces this memory copy by element, its overhead increases. The results clearly show that this memory copy causes serious overhead. In addition, we see from the figure that the parallel machines provide quite different result-s due mainly to the performance of the processors and the memory hierarchy that equippe each computing node. Consequently, as the Cenju-4 suffered from overhead of the memory copy, the PC cluster achieves better performance of the invocation in total, even if the Cenju-4 provides good performance for the data redistribution.

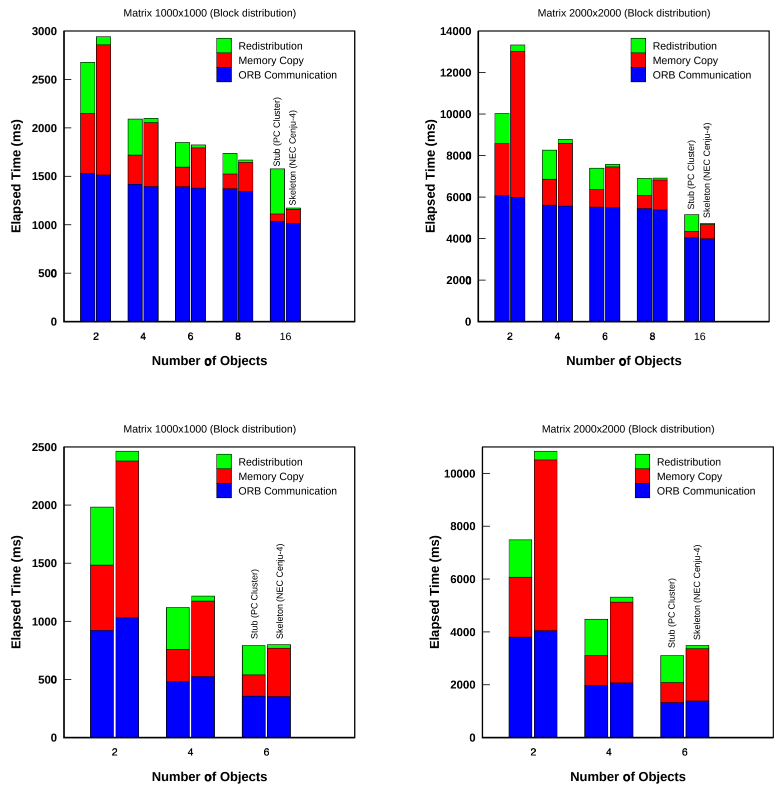


Fig. 6. Comparison of communication costs (top: 100 Mbit/s, bottom: 1 Gbit/s).

Redistribution time is the communication time for exchanging data to perform the redistribution using the MPI interface. Experiments shows that the time of redistribution on the Cenju-4 is up to 9 times faster than that on the PC cluster. However, as compared with the overhead of the memory copy, this performance difference gives less impact to the total time of the invocation.

6 Conclusion and future works

This paper discusses the implementation of data redistribution within a parallel CORBA object. We implemented a capability of performing data redistribution within parallel object in both a stub and a skeleton. This allows programmers to obtain the maximum performance under their distributed computing environment. In our current implementation, the selection is performed at compile time by specifying Extended-IDL compiler options. This means that programmers are responsible for deciding which side should perform redistribution. Although it is important to provide means to control data redistribution to the programmers, it is usually difficult for them to know which side provides better performance. This is because they have to take into account a lot of factors related to characteristics of data redistribution and their underlying computing environment (communication network, memory hierarchy and processor). In addition, the fact that such factors can vary at run-time due to the network contention makes this problem much harder. In order to relieve programmers from the pains of making such decision, as well as to provide the maximum performance automatically, we are developing a run-time service system to manage static and dynamic system information during the execution of parallel objects. This information will be used do decide at runtime the best place to carry out data redistribution.

Acknowledgments We would like to thank Satoshi Goto and Toshiyuki Nakata for their continuous valuable advice. This work was carrying out within the INRIA-NEC collaboration framework under contract 099C1850031308065.

References

1. T. Beisel, E. Gabriel, and M. Resch. An extension to MPI for distributed computing on MPPs. *Lecture Notes in Computer Science*, 1332, 1997.
2. T. Brandes and F. Zimmermann. Adaptor — A transformation tool for HPF programs. In *Programming environments for massively parallel distributed systems: working conference of the IFIP WG10.3*, pages 91–96, April 1994.
3. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
4. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infracstructure.* Morgan Kaufmann Publishers, Inc, 1998.
5. Ian Foster, Jonathan Geisler, William Gropp, Nicholas Karonis, Ewing Lusk, George Thiruvathukal, and Steven Tuecke. Wide-area implementation of the Message Passing Interface. *Parallel Computing*, 24(12–13):1735–1749, November 1998.
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. T. Kamachi, K. Kusano, K. Suehiro, Y. Seo, M. Tamura, and S. Sakon. Generating realignment-based communication for HPF programs. In *10th International Parallel Processing Symposium*, 1996.
8. K. Keahey and D. Gannon. Developing and Evaluating Abstractions for Distributed Supercomputing. *Cluster Computing*, 1(1):69–79, May 1998.
9. T. Nakata, Y. Kanoh, K. Tatsukawa, S. Yanagida, N. Nishi, and H. Takayama. Architecture and software environment of parallel computer cenju-4. *NEC Research & Development*, 39(4):385–390, October 1998.
10. T. Priol and C. René. Cobra: A CORBA-compliant Programming Environment for High-Performance Computing. In *Euro-Par'98*, pages 1114–1122, September 1998.
11. A. Reinefeld, J. Gehring, and M. Brune. Communicating across parallel message-passing environments. *Journal of Systems Architecture*, 44:261–272, 1998.
12. C. René and T. Priol. MPI code encapsulating using parallel CORBA object. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, August 1999.