A Model of Interaction for Parallel Objects in A Heterogeneous Distributed Environment

Katarzyna Keahey
keahey@cs.indiana.edu
Indiana University
215 Lindley Hall
Bloomington, IN 47401

September 23, 1996

Abstract:

This paper describes a model of interaction developed as a part of an environment which enables heterogeneous parallel objects to interoperate in a distributed domain. It describes the interactive needs of parallel objects in such an environment and proposes a solution which would satisfy them within the object model as used by CORBA. The model of interaction has been prototyped in PARDIS, a PARallel DIStributed environment based on the CORBA design principles. PARDIS is built on top of the Nexus run-time system and has been used to demonstrate how the model of interaction can be used in scenarios involving parallel objects implemented using the POOMA framework.

1 Introduction

This paper presents the rationale and design of a model of interaction which, although based on a client-server *relationship* between interacting objects, offers them the flexibility of peer-to-peer *interaction*. This work is a part of an effort aimed at designing an environment which would enable parallel objects implemented according to different programming paradigms (such as : pC++, POOMA, CC++, ABC++ [GBB+93, RHea96, CCK92, OEPW96] etc.) to interact in a distributed domain. This design relies on the idea of interoperability through a common object architecture model introduced by the Common Object Request Broker Architecture (CORBA) [OMG95b] from the Object Management Group (OMG). In CORBA the client-server relationship which arises from the object model is carried out in the client-server model of interaction; in parallel processing this model of interaction is too restrictive. Designing a suitable model of interaction was therefore the first difficulty which needed to be overcome in adapting the CORBA design to the needs of parallel processing. The object model presented here could be also useful for non-parallel objects with similar interactive requirements.

The abstraction and mechanisms used by the object model have been prototyped in PARDIS, an experimental system aiming to provide interoperability in a PARallel DIStributed environment. The current, initial implementation of PARDIS is built on top of the Nexus [FKOT94] run-time system. The work on the implementation of PARDIS is currently in progress.

Although the main focus of this paper is to describe the rationale and design of the model of interaction, it also provides an illustration of how this model can be implemented and used with parallel servers. The presented example scenario shows interaction of a parallel object implemented in the POOMA framework [RHea96] with non-parallel objects running on remote and heterogeneous platforms.

2 Motivation and Design Guidelines

The design of CORBA is based on two concepts: the OMG object model and client-server interaction. The object model provides a set of abstractions which serve as a gateway between remote objects implemented in different languages. CORBA IDL expresses these abstractions; object *interfaces* defined in this language are used to generate code interfacing between the implementation of an object and the communication libraries provided by the CORBA system. Code generated in order to provide access to the object is called a *skeleton*; code generated in the language of the client to work with a particular type of server is called a *stub* (note the client-server relationship implied by the object model). All the information needed by the objects to interact is summarized in the interface, which makes them self-contained, independent units. This feature is further supported by the one-sided nature of the client-server interaction, which makes the objects independent of any particular scenario, and thus ensures their reusability.

The idea of using an object model as a common denominator between different programming paradigms is a very useful one, and has been adopted as the central concept of our design. Research is currently underway on how to best adapt it to the needs of parallel processing [GK96]. However, the client-server model of interaction is too restrictive to fulfill the needs of parallel processing. The purpose of the research described here, is to provide a model of interaction compatible with the client-server relationship arising from the object model, but at the same time satisfying the needs of peer-to-peer interaction between parallel objects. Furthermore, this model of interaction has to be reflected in the object model, both in order to preserve the reusability of objects, and to make its implementation independent of the features provided by a particular system, as they may influence its portability and efficiency.

The design was produced using the following guidelines:

- **Concurrency:** wherever possible, the programmer should be able to exploit concurrency between interacting objects.
- **Design Efficiency:** the design should open opportunities for efficient solutions: for example, opportunities for overlapping computation and communication should be provided.
- **Programmer Convenience:** the programmer should have access to convenient abstractions, flexible scenarios and non-restrictive programming style.

3 Model of Interaction in a Parallel Distributed Environment

In a typical client-server scenario, the client issues a request to the server and blocks awaiting the reply. The server blocks waiting for requests from clients. Once the server begins processing a request it cannot be interrupted; it can accept and output information using modes of communication outside of the client-server model (message-passing for example), or it can spawn another thread to process those requests concurrently, but there is no explicit way of accepting one request during the processing of another *within* the client-server model. It is this limitation, in situations where two processes have to be both client and server to each other, that make the client-server model unsuitable for peer-to-peer interaction of parallel objects. The purpose of the following sections is to examine the interactive needs of parallel objects and propose a solution which will allow the flexibility of peer-to-peer interaction while preserving consistency with the client-server relationship arising from the object model.

3.1 Interactive Requirements of Parallel Objects

In the interest of efficiency, it is imperative that the client should compute rather then block waiting for the reply to its request; likewise, it is unlikely that a parallel server will ever block, it may however need to process requests while it computes. Many parallel computations (especially data-parallel computations[NBB+96, RHea96]) generate output not only at the end, but also *during* computation. These *partial results* can represent for example scientific data describing the evolution of a system or visualization data. This output should be available for processing by other objects while the computation which produced it is still in progress, thus enabling pipelining and satisfying the concurrency guideline.

This feature of a returning partial results during computation is not only an efficiency issue. The computations of two parallel objects may actually depend on access to each other's partial results. To illustrate this consider the example in figure 1. Here, the progress of computation of each of the parallel objects depends on the other's partial results. Therefore, each of the objects needs to be able to request partial results from the other during its computation. This requirement could be circumvented by forcing the client which started the computation to request it on a per-update basis; this approach would be both inefficient and burdensome (and thus contradicts the Design Efficiency and Programmer Convenience guidelines) and was therefore rejected.

In addition to the need for reflecting this "continuous return" nature of parallel computation, there are other requirements of less fundamental importance. In large distributed scenarios it may be convenient to make the partial output of parallel objects persistent and subsequently use it to simulate those objects for purposed of debugging and demonstration [NBB+96]. Also, the requirements of modern problem-solving environments incorporating interactive and visualization components call for providing facilities for computational steering [JP94].

Based on these considerations, the following requirements for a model of parallel object interaction were formulated:

1. Mechanisms should be provided that will enable the client to compute while waiting for results

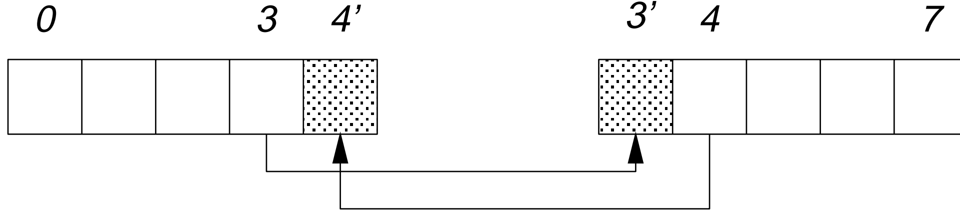

Figure 1: In this computation the value of each vector cell is repeatedly updated based on the values of its neighbors. It has been implemented as the interaction of two parallel objects where each is responsible for some part of the vector. After the computation is started by an external client, at every update the objects need to exchange information about the values of cells adjacent to the extreme end of their part of the vector.

from server.

2. Partial results produced during the computation of a server should be made available as they are produced.

3. Facilities should be provided to make these results persistent so they could be used in simulations.

4. Opportunities for computational steering should be offered.

It is important that these postulates be satisfied *within* the model of parallel object interaction, as they should be independent of the availability of particular implementation or architectural features (such as availability of threads); instead the diversity of those features should be exploited under a common interface.

3.2 Sketch of a Solution

Satisfactory solution to the first requirement is already present in some parallel systems (see section 4.2) and will be adopted here. This section will therefore focus on the remaining three requirements. Making the partial results available during the computation can happen in two ways: either the clients request them, or the computing object sends them to all interested clients. These two approaches are termed *demand-driven* and *data-driven* respectively.

The data-driven approach has the advantage that the data can be delivered to the remote client as soon as it is produced, saving the client time necessary to make the request and await the reply. Also, if the demand-driven approach were adopted, the client might have to synchronize with the server to ensure that the partial results it needs are still available, and thus potentially slow down the server [NBB+96]. On the other hand, data-driven approach requires buffering and by design provides communication in one direction only [1]. Also, only demand-driven interaction allows a truly asynchronous mode of communication, which would satisfy the needs of computational steering.

[1] The revisions to the object model reflecting the nature of data-parallel computation will allow the client to provide input to some degree.

The third requirement can be best satisfied within the data-driven model; accepting both interactive approaches is therefore necessary to fulfill all of the requirements efficiently.

Data-driven and demand-driven interaction is embodied in the object model by *registered* and *priority* requests. An object can offer three kinds of requests:

- **standard requests** which have the usual semantics of a client-server request
- **priority requests** which can interrupt the server (if the server allows it) during the processing of a non-priority request
- **registered requests** which will be processed whenever the server is ready to output data associated with them and send them to interested clients

It should be noted that neither priority nor registered requests are functionally a new thing. Requests for remote execution, employed by many run-time systems [vEGS92, FKOT94, BG96] are a low-level form of a priority request. Including them in the object model is a natural consequence of extending the design domain to heterogeneous architectures and programming styles. Likewise, registered request have their counterpart in channels (see [OMG95a]). Treating them as requests in the interface of an object, in addition to improving clarity of the declaration, gives them all the advantages of stubs and skeletons and allows them to use types associated with a particular object, especially the exceptions.

4 PARDIS — Design and Implementation

PARDIS is a distributed system aiming to provide interoperability between distributed, heterogeneous, data-parallel objects. The present implementation supports the basic CORBA interactive scenario, that is interoperability between non-distributed clients and servers, as well as data-parallel servers, and incorporates the extensions to the client-server model of interaction described in this paper.

The present implementation of PARDIS is layered on top of the Nexus [FKOT94] run-time system. PARDIS supports interaction of objects running on widely distributed heterogeneous architectures, as well as activation of remote objects registered with PARDIS database. So far, PARDIS has been tested with objects written in C++; interoperability with data-parallel servers has been tested in simulations with POOMA [RHea96] applications.

4.1 Mapping to the Object Model

The current implementation is based on the OMG object model with extensions reflecting changes to the model of interaction. Stubs and skeletons for non-parallel objects have been adapted based on the C++ mapping through inheritance [OMG95b]. In the mapping for data-parallel objects, SPMD model of execution was assumed. In this case it was necessary to convey the information about invocation, as well as its arguments, to all processes of the parallel computation. There

are two ways in which this problem can be approached: the client can invoke the request on each process of the parallel computation (fig 2.1), or it can invoke the request on a selected process, and have that process notify all the other processes (fig 2.2).

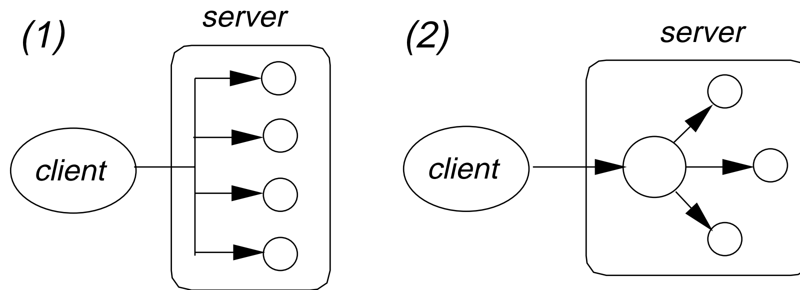

Figure 2: Request invocation on a parallel object

The latter model was adopted in PARDIS as more general and potentially more efficient; individual invocations of the processes will often have to be serialized, whereas in the second case facilities accessible on a given architecture, such as shared memory, or high-bandwidth network can be used to speed up the process of node notification.

Making this choice means that in addition to a skeleton mapping, PARDIS now needs an interface to the run-time system underlying the implementation of the server. Experiences with designing mapping for the POOMA framework showed that the necessary functionality can be obtained with a minimal interface; the functions required were concerned mainly with method invocation and argument marshaling. Further, the impact of different run-time system designs needs to be considered. To date, two general designs were considered: *message-passing based* systems (such as MPI or PVM [GLS94, GBD+94]) and *remote execution requests* (RER) based systems [vEGS92, FKOT94, BG96].

4.2 Futures

In order to enable the client to compute while waiting for results from requests processed by remote servers, PARDIS allows non-blocking invocations. Two alternative stubs are generated for every operation declared in the interface: a blocking stub — operation invoked using this stub will block till all the results are returned, and a non-blocking stub, which will cause the invocation to return immediately with *futures* of its results. Futures were first introduced in Multilisp [Hal85] to identify the results of computation which were still in progress, and they were recently revived in ABC++ [OEPW96]. ABC++ defines an excellent representation of futures in C++, which has been adopted in PARDIS.

A future can be defined as a pair composed of a value, and a semaphore which guards the value; if the semaphore is raised, the value can be accessed (we will say that the future has been *realized*), otherwise it cannot. Systems using futures provide an interface to them which causes every attempt to read an unrealized future to block. Further, ABC++ allows the programmer to probe the future in order to ascertain if it has been realized.

4.2.1 Multiple Results as Futures

To the best of my knowledge, so far futures have been used to represent a *single* return value only. CORBA allows the programmer to return multiple results through `out` and `inout` parameters in addition to return values, and there is no reason why the client shouldn't be able to receive more then one result as a future. PARDIS addresses the problem by making all the results returned by a non-blocking call futures.

Two choices for semantics are possible in this case:

- all the futures returned by a request are associated with the same semaphore, and are returned by the server once the processing of the request is finished
- each future has a different semaphore associated with it and can be realized as soon as the processing associated with *that future* is finished. This calls for an interface capable of signaling the readiness to realize a future on the server's side, and for keeping track of futures realized explicitly, so that all others can be realized at the end of request.

The latter solution has been adopted in PARDIS, as it is more general, gives the programmer additional control and subsumes the first one since all results not returned explicitly will be returned at the end of request by default. The function `return_result` (associated with a client in a given scope) is used to explicitly return `inout` and `out` arguments. There is one exception: the return value of an operation can only be returned at the end of the operation (using the keyword `return`). This exception was made as it was felt that doing otherwise would be confusing to the programmer and does not bring substantial functionality loss. After `return_result` has been used, no modifications of the returned result will appear on the client's side. Note that all the results produced by a request can be returned before the processing associated with this request is finished. This feature makes this mechanism similar to `rtf` in Mentat [Gri93].

4.3 Priority Requests

4.3.1 Syntax and Semantics

The declaration of a priority request in IDL is marked by a `priority` attribute. Since a priority request may be associated with the invocation of a specific non-priority request, an optional keyword `requires` can be added to the declaration, to mark when this request can be processed. An example IDL declaration can look as follows:

```
priority my_array get_data(in coordinates x) requires(diffusion);
```

where `coordinates` and `my_array` are types, and `diffusion` is the name of a non-priority method of the same object.

The primary characteristic of a priority request is that it can interrupt the processing of a non-priority request. During its processing, the priority request can change the state of the server, which could affect correctness of the interrupted request. Also, the programmer of the server may not wish the server to be interrupted in certain areas for reasons of efficiency (such as overwriting the cache for example). The programmer of the server is therefore given control over where those interrupts should occur. PARDIS provides two options for exercising this control. A pair of

```
enable_priority_requests(<list of requests>);
disable_priority_requests(<list of requests>);
```

can be used to define areas in the program where processing of certain priority requests can occur. Alternatively,

```
process_priority_requests(<list of requests>);
```

can be invoked to enforce processing of priority requests at a given point. The choice between the two depends on characteristics of a particular application and its implementation.

The implementor of the server is of course free to use a synchronization mechanism provided by the run-time system of the server rather then rely on this interface. Priority processing can also be disabled by a flag set at the time of activation of the server; in this case the client invoking a priority request will be returned a system exception. Otherwise, from the client's perspective there is no difference between invoking a priority request and a standard request.

4.3.2 Implementing Priority Requests

There are two issues that arise in the implementation of priority requests in the adopted model of invocation on parallel objects. One is how to interrupt the computation of a non-parallel object (or one node of the parallel object). The other is how to notify all processes of the data-parallel object of this interruption.

At least three techniques have been used in parallel processing to deal with the first problem: a system interrupt, threads, and polling [vEGS92, FKOT94, BG96]. Of these three only the first two provide truly asynchronous modes of interaction and a practical solution for implementing the enable/disable pair; polling demands and offers a high degree of control on priority processing at the expense of response time. At present only this option is implemented in PARDIS.

Approaching the second issue depends on the run-time system underlying the implementation of the parallel server. Given the model of invocation adopted in section 4.1, systems based on message-passing are limited to invoking `process_priority_requests` as they cannot process asynchronous signals. In these systems `process_priority_requests` is implemented as a broadcast of the information about the invoked methods with their arguments. RER based run-time systems can implement all the calls. Both enabling and disabling priority requests requires a barrier; it is however probable that in some cases an existing barrier could be used. Wherever priority processing is enabled, an interrupt can be transferred as an asynchronous broadcast. `process_priority_requests` can be implemented either as a call enabling priority processing followed by a call disabling it, or directly by a broadcast.

Preliminary enquiry into efficiency-flexibility tradeoffs between the implementation of `process_priority_requests` in message-passing based and RER based systems confirmed the expectation of higher efficiency in the message-passing systems. Similar problems were investigated in [BG96]. The exact performance prediction here will depend on the communication model supported by a given architecture and features of a particular application.

4.4 Registered Requests

4.4.1 Syntax and Semantics

Registered requests are marked by the attribute `registered` and, like priority requests, they can optionally be bound to another request of the object whose partial output they return. The same request can have both registered and priority attributes. An example declaration might look like this:

```
registered my_array get_data() [requires(diffusion)];
```

Registered requests take no arguments; they only return results. Whenever the server is ready to do processing associated with producing output from a register request, it calls

```
process_registered_request(<list of requests>);
```

The output is then sent to all the clients that registered interest in the processed requests; if no clients are subscribed for a request, the request is not invoked.

A client can subscribe (register interest) or unsubscribe to a particular request by invoking:

```
s_ptr->register_request(<list of requests>);
s_ptr->unregister_request(<list of requests>);
```

where `s_ptr` is a pointer to the server obtained by the client; and `register_request and unregister_request` are provided as standard stub methods on any object that declares registered requests.

In order to obtain results returned by a registered request the client has to explicitly invoke it; this invocation is always non-blocking and returns a future. Instead of calling the server, the call is redirected to a buffer where the output from the server is collected. If the server has generated output sufficient to satisfy the request, the future is filled from that buffer (or a pointer to the buffer is returned). If not, memory is allocated for the future, and the call returns like any non-blocking call. When the server sends the results, memory allocated for futures is filled (bypassing the buffer). The future returned from a registered request behaves like any other future; in particular it can be probed, and the access to an unrealized future will block.

A call to a registered method can return an exception associated with the server implementing the method. This is particularly useful if the data provider has to be suspended; it can then inform the clients of the state of the system in a intelligible form. A system client making the output of a registered request persistent can be specified as an activation option to the server.

4.4.2 Implementing Registered Requests

The calls to `register_request` and `unregister_request` are functionally priority requests which can be invoked at any time in the program with the restriction that a change in registration status cannot appear during the processing of a registered request. In servers implemented on top of RER based run-time systems, a remote procedure call is made to change the registration status; in message-passing based systems, a bit array representing the registration status of requests is broadcast at every invocation of `process_registered_request`. At the invocation of `process_registered_request` each request whose registration status bit is set will be processed and the output sent to all clients registered for that request.

On the client's side the main implementational concern is in the implementation of a hierarchical buffering system and buffer-overflow strategies. This work is still in progress as it partially depends on modifications to the object model and interaction with a parallel client.

4.5 Example

This section shows how PARDIS can be used to build distributed scenarios of interacting objects involving data-parallel components. We will look at a very simple example involving two objects: a data-parallel object performing diffusion on a two-dimensional grid, and a visualizer monitoring its progress. The computation of both is initiated by an external client which also interacts with the data-parallel server by modifying the grid values during the computation (see figure 3).

The IDL interfaces of the diffusion server and the visualizer are as follows:

```
interface diffusion_server {
   double do_diffusion(in long iterations);
   priority void modify_grid(in long x, in long y);
   registered sequence<double> grid_values();
};

interface visualizer {
   long visualize();
};
```

After binding to the servers, the client first starts the computation of the visualizer and the diffusion server, using stubs based on interfaces defined above. Both invocations are non-blocking and return futures: the visualizer returns status, and the diffusion object — sum over all grid values. The visualizer, a client of the diffusion object, registers interest in the `grid_values` request; subsequently, whenever this registered request is processed, its output will be forwarded to the visualizer. Meanwhile, a user interacting with the client can modify the grid values of the diffusion object while the computation is in progress: for every modification the priority request `modify_grid` is called. After the modifications are over, the client attempts to realize the futures: if the diffusion object and the visualizer are still computing, it will block, otherwise it returns immediately.

In order to work with PARDIS, the POOMA programmer has to define a class inheriting from the skeleton class based on the interface definition and implementing the three declared methods as shown below (`modify_grid` and `grid_values` were implemented as straightforward calls to POOMA objects).

```
class diffusion_server: public diffusion_server_sk {
   public:
     ...
     double do_diffusion(int iterations) {
        ... /* POOMA initialization */
        for(int i=1; i<iterations; i++) {
           ... /* compute POOMA timestep */
           process_priority_requests();
           process_registered_requests(grid_values);
        }
```

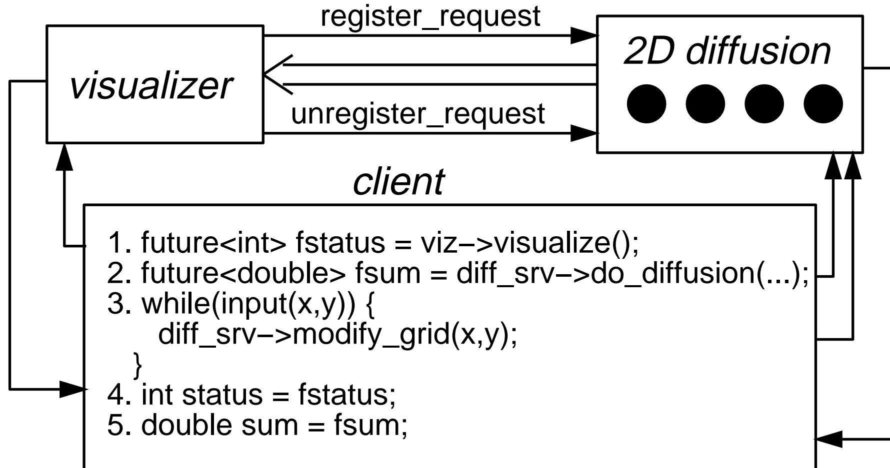

Figure 3: Distributed scenario in PARDIS

```
    }
    void modify_grid(int x, int y);
    sequence<double>* grid_values();
};
```

This example was implemented using PARDIS, and ran successfully with all three interacting entities placed on remote hosts: the diffusion object ran in parallel on an SGI PC; the visualizer and the client were implemented in C++ and ran on SGI and solaris platforms.

5 Summary and Future Work

This paper describes a model of interaction which enables objects to interact on a peer-to-peer basis while retaining the client-server relationship implied by the object model of CORBA and similar systems. Specifically, it shows how futures can be used to return multiple results from non-blocking invocations and introduces priority and registered requests which embody the demand-driven and data-driven modes of interaction in the object model. Introducing these constructs allows the programmer to exploit concurrency in distributed scenarios, overlap communication and computation between interacting objects and provides convenient abstractions for doing so.

The model of interaction has been prototyped in PARDIS, a system based on the CORBA object model, whose final goal is to implement a set of features allowing interoperability of parallel objects. So far, interaction between non-parallel servers and a limited level of interaction with parallel servers has been implemented. Preliminary experiences in building distributed scenarios using parallel components implemented within the POOMA framework indicate that the proposed model is capable of providing flexible and convenient interaction with parallel objects.

The most immediate work on PARDIS will focus on investigating different models of invocations on parallel objects in the context of the interaction model just described. Work on adapting the

OMG model to reflect the structure of data-parallel objects and integrating it with the model of interaction is the next goal. Eventually, PARDIS will incorporate much of the design features of CORBA, in particular its layered architecture and interoperability support.

Acknowledgments

The author is obliged to John Reynders and all members of the POOMA team for their help and comments in implementing this project, and to Dennis Gannon for the discussion of some of the concepts described here.

References

[BG96] P. Beckman and D. Gannon, *Tulip:a portable run-time system for object-parallel systems*, 10th International Parallel Processing Symposium, April 1996, pp. 532–536.

[CCK92] P. Carlin, M. Chandy, and C. Kesselman, *The Compositional C++ Language Definition*, Tech. Report CS-TR-92-02, CalTech, 1992.

[FKOT94] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke, *Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems*, Technical Memorandum ANL/MCS-TM-189 (1994).

[GBB+93] D. Gannon, F. Bodin, P. Beckman, S. Yang, and S. Narayana, *Distributed pC++: Basic ideas for an object parallel language*, Journal of Scientific Programming **2** (1993), 7–22.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel virtual machine. A users' guide and tutorial for networked parallel computing*, MIT Press, 1994.

[GK96] Dennis Gannon and Katarzyna Keahey, *Distributed parallel environment — a sketch*, POOMA '96 Abstracts, February 1996.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI. Portable parallel programming with the message-passing interface*, MIT Press, 1994.

[Gri93] Andrew S. Grimshaw, *The Mentat Computation Model Data-Driven Support for Object-Oriented Parallel Processing*, Tech. Report CS-93-30, University of Virginia, May 1993.

[Hal85] Robert H. Halstead, *Multilisp: A Language for Concurrent Symbolic Computation*, ACM Transactions on Programming Languages and Systems **7** (1985), no. 4, 501–538.

[JP94] C. R. Johnson and S. G. Parker, *A computational steering model applied to problems in medicine*, Supercomputing '94, 1994, pp. 540–549.

[NBB+96] Michal L. Norman, Peter Beckman, Greg L. Bryan, John Dubinski, Dennis Gannon, Lars Hernquist, Kate Keahey, Jeremiah P. Ostriker, John Shalf, Joel Welling, and Shelby Yang, *Galaxies Collide on the I-WAY: An Example of Heterogenous Wide-Area Collaborative Supercomputing*, accpted for publication by The International Journal of Supercomputer Applications (1996).

[OEPW96] W.G. O'Farrell, F. Ch. Eigler, S. D. Pullara, and G. V. Wilson, Parallel Programming Using C++, ch. ABC++, MIT Press, 1996.

[OMG95a] OMG, *CORBAservices: Common Object Services Specification.* , OMG Document, March 1995.

[OMG95b] OMG, *The Common Object Request Broker: Architecture and Specification. Revision 2.0*, OMG Document, June 1995.

[RHea96] J. V. W. Reynders, P. J. Hinker, and J. C. Cummings et al., Parallel Programming Using C++, MIT Press, 1996.

[vEGS92] T. von Eicken, D. Culler S. Goldstein, and K. Schauser, *Active messages: a mechanism for integrated communication and computation*, Tech. Report UCB/CSD 92/#675, University of California, Berkeley, March 1992.