Collective Objects: An Object-Oriented Tool for Collective Operations in Distributed Parallel Computation

Katarzyna Keahey
Dennis Gannon
{*kksiazek, gannon* }*@cs.indiana.edu*
Indiana University
215 Lindley Hall
Bloomington, IN 47401

May 6, 1996

Abstract:

This paper describes the collective object, a new abstraction providing support for collective operations common in parallel programming. The collective object is introduced in the context of research aiming to produce a design of a distributed object-oriented environment suitable for parallel computation, and will constitute a part of the object model of this environment. We give a formal definition of the collective object and supporting constructs and conclude with some examples and preliminary results concerning application of the collective object.

1 Introduction

Challenging object-oriented technology to create interoperability between distributed and heterogeneous modules has lead to the development of the Common Object Request Broker (CORBA) [OMG95] standard from The Object Management Group (OMG). The success of CORBA relies on the introduction of an object model which allows the designers of distributed, heterogeneous applications to express their programs completely in terms of object interactions. The idea of object encapsulation, enforced through interfaces, lets the programmer separate the services provided by an object from implementation details. Our research concentrates on formulating an environment which would use ideas developed by CORBA to bring interoperability of heterogeneous and distributed modules into the domain of parallel programming. We are therefore looking for a set of abstractions which would let the programmers of parallel applications formulate their ideas in its terms. Towards that end, this paper introduces the concept of a collective object, a new abstraction providing support for collective operations, such as barrier, gather, or reduce.

So far no object-oriented abstraction supporting collective operations has been defined. We feel that such abstraction is needed since it would allow programmers to formulate their own collective methods instead of relying on a limited set of system primitives. Conventional objects do not have the functionality needed to perform collective operations. In particular they lack the ability to accept and process an invocation coming from several sources as one service. It is therefore necessary to introduce an abstraction which will both draw on the functionality and elegance of object-oriented programming and provide this service in a way which is simple, intuitive, and makes efficient implementation possible.

This paper defines the concept of a *collective object*: a parallel abstraction designed to implement collective operations associated with a set of clients. Services of the collective object can include standard collective operations and other events which could be common to a set of clients, such as main thread events. A collective service is performed in answer to a *collective invocation*, that is a set of requests coming from all the clients registered with a given collective object. Each of the registered clients has to request the service exactly once; in response to this set of requests the collective method is provided only once.

The key difference between a collective invocation and an ordinary invocation is that an ordinary method will be invoked many times in response to many invocations. The state of the object that the method is invoked on, can change between invocations and the side-effects, if any, produced by this methods will be repeated. A collective method on the other hand, will be executed only once, and its execution will be completed only if all of the clients join in the invocation.

The notion of a collective invocation gives rise to questions about client synchronization. In particular we would want to know if all clients need to communicate their requests to the collective object before the collective method is executed, and whether it is safe to assume that they will all return from the call at the same time. Most collective operations do not require synchronizing with all clients, either on entry, or on exit. In the gather operation for example, each client can be "serviced" independently. A sum reduction can be executed on behalf of the clients which have already joined the invocation while waiting for other clients to contribute their input. Similarly, on exit from a collective method, only some operations (such as barrier) will require the clients to synchronize. The decision about whether to synchronize or not is therefore left to the programmer, and support is provided to monitor dependencies between individual clients, thus letting the programmer avoid synchronization whenever possible.

Dependencies between individual clients will arise mostly from argument instantiation and return in a non-synchronized invocation. In general, since each of the invoking clients can bring different argument values into the collective method, collective objects require techniques of argument handling which would associate the invoking clients with argument values instantiated by them. This function is fulfilled by the Collective Associative data structures (CAs) which maintain a mapping between the clients of a collective object, arguments instantiated by them and semaphores signaling argument instantiation. Associating argument values with semaphores prevents accessing uninstantiated arguments and lets the programmer write code without having to worry about the asynchronous nature of collective entry.

2 Motivation

This research grew out of our experiences in distributed parallel programming, in particular the work on the distributed version of the Self Consistent Field (SCF) program [NBB$^+$96] which was part of the I-WAY project presented at SuperComputing '95. SCF [HO92] simulates the interaction of galaxies by performing N-body computation on their gravitational fields. The computation is data-parallel in character, and in its distributed version it consists of several components distributed over different supercomputers.

The data exchange in the SCF application relies on a reduce-and-broadcast operation consisting of reducing the vector of coefficients describing the gravitational field of galaxies and then broadcasting it to all computational units. Our philosophy in implementing distributed SCF was to create interoperability between existing heterogeneous components, rather than redesign them to use a common communication base. Therefore, in addition to local reduce-and-broadcast operations, each component participated in a global reduce-and-broadcast, exchanging the vector of gravitational coefficients with other distributed components. The global reduce-and-broadcast operation was implemented by sclib, a library written specially to work with this project.

This experiment provided an insight into the general needs of a distributed object-oriented environment. In particular, it made clear the necessity to support a wide range of collective operations enabling collaboration between distributed parts of data-parallel computations. At this point we were faced with two choices: either to provide collective services as library calls, or to place entities capable of supporting collective services in the object model of our environment and thus give the programmer the opportunity to define his or her own collective operations. We chose the latter.

Our primary motivation was flexibility. Programming in terms of library calls is limiting; a programmer wanting to implement collective operations not supported by the library has to resort to programming in terms of low-level primitives, which is both arduous and error prone. Library designers have recognized this problem and efforts have been made to relax the rigidity of fixed library interfaces [For95], but still within the narrow limits of a library.

The collective object offers the programmer a flexible tool for developing his or her own collective operations. Through shifting the support for coordinating invocation from many processes onto the system and providing suitable argument support, the collective object lets the programmer concentrate on the semantics of the collective operation itself and disregard the mechanics of interaction of the participating processes.

Further, viewing collective operations as method invocations in the context of a parallel system, rather than as library primitives, leads to interesting ideas and generalizations. One of them is giving the notion of a collective invocation non-blocking semantics, and representing values returned from this invocation as futures. One of our experiments demonstrates how this idea can be used to overlap different levels of communication without loss of clarity in programming.

Finally, we found it very convenient to think in terms of the collective object abstraction. It deals with concepts frequently and widely used in parallel programming in a way that is very accessible to the programmer. We found that it gave us a fresh way of looking at parallel programs which lead to interesting ideas on how to improve them.

3 Defining the Abstraction

3.1 Extensions to the CORBA Object Model

In the CORBA model, *objects* (servers or service providers) are defined as encapsulated entities which can perform certain actions (services) upon request. *Clients* are entities capable of requesting services to be performed by specific objects. A *request* is an event that communicates to the service provider that a client wants a specific service performed; the request can carry with it some input data which can be used by the server in performing the service and it may demand some output which is a product of the service. A *service* is an action performed by the object to satisfy a request.

Let C be a set of clients $\{c_1, c_2, \ldots, c_n\}$. Let r_s be a request for a service s and let (c_i, r_s) denote a request for service s issued by $c_i \in C$. A *collective request* $R_{(C,s)}$ for service s from C is a set of requests $R_{(C,s)}$ such that $\forall c_i \in C,\ (c_i, r_s) \in R_{(C,s)}$ and $(c_i, r_s) \in R_{(C,s)} \wedge (c_j, r_s) \in R_{(C,s)} \Rightarrow i \neq j$. Informally, $R_{(C,s)}$ is a set of requests of the same kind such that every client in C makes the request exactly once. $R_{(C,s)}$ can be satisfied either by providing service s n times, if $n = \mid C \mid$, or providing s only once if s is a collective service.

A *collective service* s is a service performed in response to a collective request $R_{(C,s)}$ only once; input carried by any $(c_i, r_s) \in R_{(C,s)}$ can be used by the service provider in performing s, and any output produced by s can be returned to any $c_i \in C$. An entity capable of providing such services is called a collective object, that is, a *collective object* associated with C, $COLL_C$, is an object capable of providing collective services to C.

Like regular CORBA objects, the services provided by a collective object are defined by an interface. Rather than introducing a new keyword to CORBA IDL, we decided to make this distinction between a collective and ordinary object visible through requiring that the collective object inherit form a generic collective object defined by the Distributed Object-Oriented and Parallel (DOOP) environment. The generic collective object interface will additionally make available many common collective operations. An example IDL definition might look like this:

```
interface my_collective: DOOP::collective_object {
   void reduce_and_broadcast(inout vector coefficients);
   void one2all(inout long val);
   void exchange(in array x, out array y);
};
```

3.2 Semantics of Collective Invocation

We will call the set of operations executed by the collective object in order to perform a collective service a *collective method*, and the process of invoking a collective method, a *collective invocation*.

From the client's perspective invoking a collective method is not different from any other invocation; in particular the signature of the collective method that the client uses for invocation does not reflect the existence of other clients. We will relax the invocation semantics of the CORBA standard to support non-blocking method invocations; thus the client's invocation of a collective method could be *blocking* or *non-blocking*. In the first case, after issuing the request, the client blocks until the collective object signals that the request is satisfied and all the requested return values have been instantiated. In the second case, after issuing the request, the client proceeds with its computation until it needs the results of the request; it then waits for the results to be returned.

Let m be a collective method corresponding to the service requested by r. A request r_c from a client c *enters* m when the request has been communicated to the collective object, and the arguments (if any) carried by r_c have been instantiated. A *collective entry* to m is a set of request entries to m such that $r_c \in R_{(C,s)}$. After a request from the first client enters the collective method, m is ready to be activated and start executing towards satisfying that request. If we impose the additional condition that all clients have to enter before m can start executing, the collective entry is *synchronized*. Otherwise the entry is *non-synchronized*.

Most collective operations do not need to globally synchronize on entry to the collective object; dependencies between individual clients are handled on the level of argument instantiation by the Collective Associative data structures (see section 3.4). However, the clients may want to synchronize on entry to the collective method to ensure that some side-effects of their actions, such as I/O operations for example, complete before entering the collective method (as they may affect it).

Symmetric to the parallel entry is the notion of a *collective exit*. A client *exits* a collective method when the work that the collective object was doing on behalf of that client's request is completed, all the return values requested by the client are returned to the client process, and if the invocation was blocking, control is returned to the client's process. A *collective exit* is a set of client exits from m. If m has to finish executing on behalf of all the clients before any are allowed to exit, the collective exit is *synchronized*, otherwise it is *non-synchronized*.

The meaning of synchronization on exit is that a client can expect the results produced by the collective method to be available to other clients at the same time as they are made available to it. The character of the collective entry or exit can be enforced by the implementator of the method by calling a synchronizing function at entry or exit.

As is the case with ordinary methods, methods of the collective object may or may not affect its state. We will call the first kind *state-dependent* methods and the second kind *state-independent* methods. State-independent services of the collective object can be executed concurrently if the implementation allows it; every state-dependent method must await the completion of the previously invoked state-dependent method.

3.3 Activation and Interaction of Collective Objects

The activation of collective objects is performed by an entity external to both the clients and the object. The activating entity is responsible for delivering to the collective object the references to the clients.

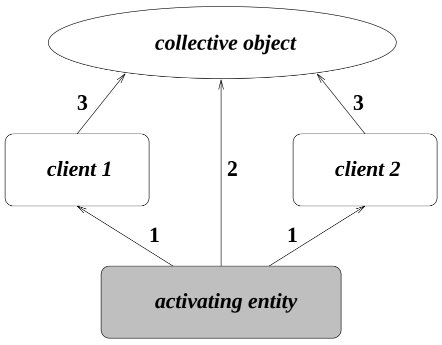

Figure 1: Binding to the collective object. The activating entity first obtains references to collective object clients (1), then delivers them to the collective object (2), finally the clients bind to the collective object (3)

From the client's perspective the process of binding to the collective object appears to be no different then any other service request. The request for binding will however be satisfied only if this client is registered with the collective object. Furthermore, subsequent invocations to collective methods will be performed only if all clients have established a binding.

A collective object can be a client of another collective object provided that the sets of clients registered with both objects are identical. It follows that the collective object can call its own methods internally.

3.4 Collective Associative Data Structures as Arguments

Support for the collective invocation calls for a mechanism of argument instantiation and return which would give the programmer convenient access to data associated with different clients. This mechanism needs to be general enough to handle situations when only some of the clients participating in the collective call bring or request meaningful argument values (as in one-to-many broadcast) and scenarios when all the clients bring in or return different values (as in exchange or scatter). Further, argument handling should be capable of dealing with the fact that in a non-synchronized invocation, values from different clients may be instantiated at different times. The task of dealing with those issues is fulfilled by the Collective Associative data structures.

A *Collective Associative data structure* (CAs) is a set of mappings, relating domains participating in collective argument transfer, such as argument values and client references. We will say that a CAs refers to type T if the argument values it relates are of type T. Every CAs of a collective object $COLL$ contains at least one mapping: a function from the set of client references registered with $COLL$ to instances of T.

In certain cases, only some of the participating clients will be sending meaningful input to the collective object or expecting meaningful return values. In order to avoid unnecessary value transfer in such cases, and yet inform the collective object that the argument values have been instantiated, the CAs use *empty values*. An empty value associated with type T is an entity, which has the typecode of T, but does not belong to a set of legal values of T. The empty value simply carries the information "this instance of T is not active".

It is useful to associate each client reference in a CA with a *semaphore* which indicates if a given client has already instantiated its argument value. The association of argument value and semaphore, induced in this way, corresponds to the concept of a future [Hal85] and allows the programmer to write non-synchronized collective operations without the need to explicitly check for argument instantiation; any attempt to access non-instantiated argument value will simply cause the CAs to block until the responsible client instantiates that value. Another useful mapping is an ordering on the set of client references which lets the programmer of the collective object view the argument structure as a vector.

A collective method has two signatures: the signature seen by the clients, the *internal signature*, and the signature as it appears to the collective method, the *external signature*. These signatures are bound by the following relationship: an argument or return type T in the external signature is represented as a CAs referring to T in the internal signature. The CORBA tie mechanism [OMG95] can be used to associate the client invocation with the collective method in the general case.

Following the CORBA model of argument passing, the collective object supports argument transfer in three modes: `in`,`out` and `inout`. The lifespan of the `in` and `inout` arguments lasts throughout the duration of the collective call independent of the time of exit of the client that instantiated them. The programmer can decide to cause the return of the `out` or `inout` arguments before the method completes its execution by invoking `CAreturn` on the corresponding argument structure. This mechanism is similar to `rtf` in Mentat [Gri93].

3.5 Examples

This section will present a few examples of programming with the CAs. The figure below illustrates of what a very simple interface to a CAs might look like in C++ syntax:

```
template<class T>
class CA {
public:
  T& operator()(client_ref);
  T& operator()(order_type);
  CA<T>& operator=(T&);
  order_type assoc(T&);
  client_ref assoc(T&)
  client_ref assoc(order_type);
  order_type assoc(client_ref);
  T& next_available();
  void CAreturn();
};
```

In this example, operator() method retrieve values associated with a certain client or its ordering, operator= assigns a value to all argument values managed by the CAs and the assoc methods retrieve information about the mappings maintained by the CAs. The next_available method returns the next available argument value (in the centralized implementation); the assoc methods can be used to determine which client supplied it. The functionality of CAs can be extended by the implementator, either by adding new mappings or more operations. A very useful extension for example would be to support generic programming algorithms such as applying a function to all arguments held by the CAs, or reducing across the CAs.

Example 1: Reduce-and-broadcast function (centralized implementation). In this example val is an inout argument, which both brings in and returns a value. Note that the efficiency of this implementation could be improved by using the next_available method to retrieve the next instantiated argument.

```
void reduce(CA<double>& val) {
   double total=0;
   for(int i=0; i<registered_clients(); i++)
      total += val(i);
   val = total;
   val.CAreturn();
}
```

Example 2: One-to-all broadcast. This implementation of broadcast relies on the fact that exactly one client placed the call with an instantiated value; if only empty values were passed next_available will never return, if more than one meaningful value is passed, the first instantiated value will be broadcast.

```
void one2all(CA<int>& val) {
   val = val.next_available();
   val.CAreturn();
}
```

Example 3: Exchange. The exchange operation permutes the elements of the in_array according to the key provided in exchange_index (both are in arguments). The results are returned in the out_array (an out argument). This operation can be used for data exchange, for example in the binary-exchange fast Fourier transform [KGGK94].

```
void exchange(CA<array> in_array, CA<array>& out_array, CA<int> exchange_index) {
   for(int i=0; i<number_of_clients; i++)
      out_array(i) = in_array(ex_index(i));
   out_array.CAreturn();
}
```

4 Implementation

We will consider two implementations of a collective object: a *centralized* implementation, where the collective object is associated with only one context, understood as a distinct address space, and a *disseminated* implementation, when the implementation of the collective object is distributed over the contexts of its clients, and its computations are effectively performed by the clients. In both cases we assume that each client of the collective object is associated with a different context.

4.1 Centralized Implementation

In a centralized implementation the collective object can either reside in the context of one of the servers or in a context of its own. The situation when the collective object is given resources independent of the clients' resources allows the clients to delegate the execution of a collective operation to a different resource and proceed with other computations until the results of the collective operation are needed. The idea of treating the collective operation as an invocation, which could return a future rather than block, lets the programmer implement this concept easily.

We will now present a few preliminary experiments demonstrating how this feature of the collective objects can be used. The first experiment tries to determine if improvements obtained in this way are worthwhile; the second series of experiments relates to the SCF project whose efficiency could be improved by using futures from collective invocation. The collective object implementation is based on the mpich implementation of MPI over UNIX sockets; the collective object and its clients are UNIX processes and no additional scheduling facilities are used. The experiments were run on an SGI Power Challenge with 10 processors.

Experiment 1

In this experiment five computational servers were using the services of a collective object in order to perform a reduce-and-broadcast operation, implemented as a method of collective object as shown in Example 1. The work of the clients consisted of reducing a vector of 600 elements across all clients, and a computation C, unrelated to the reduction. In the first version of the experiment, the collective invocation is blocking; in the second version it is non-blocking and returns futures which are consumed by the servers only after C completes. These experiments were contrasted with an implementation of the same computational servers based directly on MPI, that is the servers were using calls to `MPI_Reduce` and `MPI_Broadcast`, rather than collective methods, in order to perform the reduction, and then performing C. The table below summarizes the results, in seconds of wall clock time.

Experiment	time (in seconds)
MPI direct	0.0175
collective (blocking)	0.02165
collective (non-blocking)	0.0150

The performance of a blocking collective object reduction was worse than that of the direct MPI implementation which could have been due to two factors. First, the implementation of the collective object was written on top of MPI and involved additional data copies. Second, while MPI was likely performing a tree reduction, the collective object was performing a sequential reduction. Still, overlapping computation and reduction in the case of non-blocking collective invocation allowed the collective object to outperform the library (at the cost of an additional resource).

Experiment 2

In this experiment we used the collective object to experiment with the distributed SCF program, described in section 2. As mentioned before the local reductions (confined to a particular supercomputer) are performed separately from the global reductions (over all supercomputers involved). For the components of the SCF computation written in pC++ the following code was developed:

```
(1) pcxx_ReduceDoubleAdd((void*)sinsum,elem_count);
(2) pcxx_ReduceDoubleAdd((void*)cossum,elem_count);
(3) sclib_ReduceDoubleAdd((double*)sinsum,elem_count);
(4) sclib_ReduceDoubleAdd((double*)cossum,elem_count);
(5) pcxx_BroadcastBytes(flag, elem_count*sizeof(double), sinsum);
(6) pcxx_BroadcastBytes(flag, elem_count*sizeof(double), cossum);
```

where (1) and (2) are local reductions, (5) and (6) local broadcasts, and (3) and (4) are calls to a distributed library which implements both a global reduction and a broadcast. Let t_1 be the time of execution of (1) or (2), t_2 time of execution of (5) or (6), and $T + l$ time of execution of (3) or (4), where l is the time needed to initiate the remote reduction and T is the remaining time of the remote reduction. The time spent on executing the code fragment above is $2(t_1 + T + l + t_2)$. We have rewritten this code using the collective object in the following way:

```
pcxx_ReduceDoubleAdd((void*)sinsum,elem_count);
dfsin = sinsum;
coll->rda(dfsin,elem_count);
pcxx_ReduceDoubleAdd((void*)cossum,elem_count);
dfcos = cossum;
coll->rda(dfcos,elem_count);
pcxx_BroadcastBytes(flag, elem_count*sizeof(double), ((double*)dfsin));
pcxx_BroadcastBytes(flag, elem_count*sizeof(double), ((double*)dfcos));
```

Here, `dfsin` and `dfcos` are futures which are consumed by the broadcast operations. Rearranging instructions of the original code fragment and using collective invocation with futures lets us overlap local and remote collective operations. The fact that this optimization has not even been noticed until collective objects were invented testifies to the power of this abstraction.

Since `sclib_ReduceDoubleAdd` and a blocking call to the collective object are functionally identical, it can be calculated that using the collective objects in this way should save $t_1 + t_2 + l$ time. Further, since `rda` is a state-independent operation, a collective object implementation can service it concurrently with other invocations of the same method. In this case the savings go up to $T + t_2$ (the assumption being that $T \gg t_1, t_2, l$ and $t_1 + l > t_2$ which is true of this particular example).

We tested this theory by combining the collective object implementation from the previous experiment and the relevant fragments of the pC++ implementation of SCF and configuring the system to include two pC++ components, using 2 processors each, and a collective object implemented in a separate context. The average of obtained results are summarized below (in milliseconds of wall clock time):

Experiment	time (in milliseconds)
blocking rda	11.34927
non-blocking rda	9.36438
state-independent	7.76205
t_1	0.79551
t_2	0.28878
T	4.48039

The results of using non-blocking calls to the reduce operations involving only one collective object confirmed our predictions, In the case of state-independent implementation of the collective object, although definite improvement was noticed, it fell slightly under our expectations, which can probably be attributed to increasing computational load of the machine. Even so, it gave us savings of over 30% in communication time. It seems safe to assume that in the absence of competition for resources the savings would only go up with the increase of the number of processors used by the computational servers (as t_1 and t_2 will grow).

During our experiments with distributed SCF, in the perfectly balanced case, time spent on remote collective operation constituted 15% of total computation [NBB+96]. Additional resources for collective object computations were available, and in fact used. Thus, the optimization just described could have saved us some time and could prove even more profitable in computations which are more communication-intensive then SCF.

4.2 Disseminated Implementation

In many cases using the centralized implementation of the collective object is not practical. In the exchange algorithm from example 3 using a centralized implementation would mean that instead of being sent directly to its recipient, the data would be routed through the collective object, an additional and unnecessary link in the communication. Besides, additional resources for collective object computations may not always be available or practical to use.

Disseminated implementation of a collective object therefore presents an interesting alternative. Although the invocation interface that is used by the client of the collective object is no different than in the case of the centralized implementation, operationally the calls to the collective object are performed using resources assigned to the clients. This opens the possibility of using traditional collective operation algorithms such as tree reduction. As in the case of centralized implementation, the programmer can take advantage of the non-blocking invocation combined with futures to overlap collective operations with local computation; in this case however only the nodes which do not participate in the execution of a collective method will benefit by it, so the distribution of the time saved in this way can be very irregular.

The efficiency and practicality of this solution has not yet been determined; research aimed at answering these questions is currently underway.

5 Conclusions

In this paper we have introduced and described the collective object, a new parallel programming abstraction providing support for collective operations which can be flexibly defined by the programmer. We have discussed the collective invocation semantics and argument structures which support non-synchronized invocation coming from many clients. The collective object has been introduced as one of the abstractions underlying an object-oriented design of a distributed environment supporting interoperability of parallel modules. The ideas developed here are based on our experiences with distributing a real parallel application.

We have provided illustrations of how the collective object can be used to implement collective communication between its clients. Further, we have shown that the collective objects can be used to advantage in overlapping remote and local communication and demonstrated with a real life example that such situations actually arise.

The research described above is still in progress. In particular, the potential of different implementations of the collective object has not yet been fully evaluated. We are also investigating the practicability of relaxing the definition of a collective object to support group invocation and opportunities that arise out of incorporating it into the object model of a parallel programming system. In the final version of this paper more complete performance results will be provided for the disseminated implementation of the collective object.

References

[For95] Message Passing Interface Forum, *MPI:A Message-Passing Interface Standard*, June 1995.

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

[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.

[HO92] L. Hernquist and J.P. Ostriker, *A Self-Consistent Field Method for Galactic Dynamics*, The Astrophysical Journal **386** (1992), 375–397.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing*, The Benjamin/Cummings Publishing Company, Inc., 1994.

[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).

[OMG95] OMG, *The Common Object Request Broker: Architecture and Specification. Revision 2.0*, OMG Document, June 1995.

[WO95] Gregory Wilson and William O'Farrell, *An Introduction to ABC++*, 1995, draft.

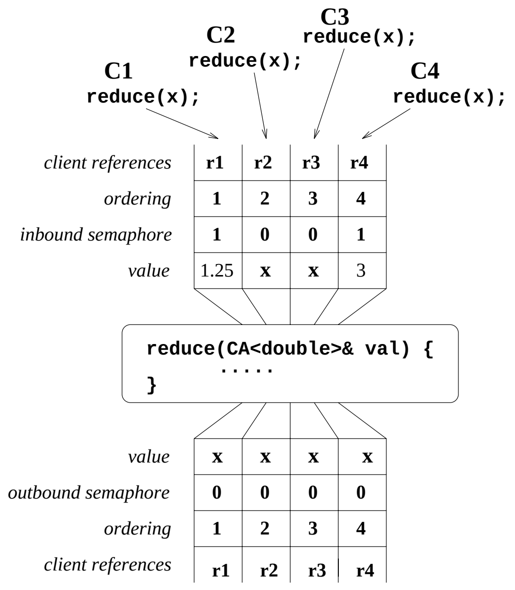

Figure 2: CAs in argument passing; note that the variable x in the client's invocations is of type double while to the collective method it appears as CAs referring to type double.