PARDIS: A Parallel Approach to CORBA [*]

Katarzyna Keahey and Dennis Gannon
Department of Computer Science
Indiana University
215 Lindley Hall
Bloomington, IN 47405
{*kksiazek, gannon* }@*cs.indiana.edu*

Abstract

This paper describes PARDIS, a system containing explicit support for interoperability of PARallel DIStributed applications. PARDIS is based on the Common Object Request Broker Architecture (CORBA) [15]. Like CORBA, it provides interoperability between heterogeneous components by specifying their interfaces in a meta-language, the CORBA IDL, which can be translated into the language of interacting components. However, PARDIS extends the CORBA object model by introducing SPMD objects representing data-parallel computations.

SPMD objects allow the request broker to interact directly with the distributed resources of a parallel application. This capability ensures request delivery to all the computing threads of a parallel application and allows the request broker to transfer distributed arguments directly between the computing threads of the client and the server. To support this kind of argument transfer, PARDIS defines a distributed argument type — distributed sequence *— a generalization of CORBA sequence representing distributed data structures of parallel applications.*

In this paper we will give a brief description of basic component interaction in PARDIS and give an account of the rationale and support for SPMD objects and distributed sequences. We will then describe two ways of implementing argument transfer in invocations on SPMD objects and evaluate and compare their performance.

1. Introduction

Advances in research on network protocols and bandwidths, and innovations in supercomputer design have made practical the development of high-performance applications whose processing is distributed over several supercomputers. These applications make use of the combined computational power of several resources to increase their performance, and exploit the heterogeneity of diverse architectures and software systems by assigning selected tasks to platforms which can best support them. Experiences of the I-WAY [5] networking experiment demonstrated that this way of approaching high-performance computing has enormous potential for solving important scientific problems [**?**].

At the same time another development in distributed object-oriented technology, the Common Object Request Broker Architecture (CORBA) [15] has made it possible to seamlessly integrate heterogeneous distributed objects within one system. CORBA provides interoperability between different components by specifying their interfaces in a meta-language, the CORBA Interface Definition Language (IDL), which is translated into the language of interacting components by a compiler. Code generated in this way may contain calls to a part of the framework called the Object Request Broker (ORB), which allows the interacting objects to locate each other, and contains network communication libraries providing network transport in a distributed domain.

High performance applications composed of many distributed, heterogeneous components have previously been developed in an ad hoc fashion trying to explicitly combine different communication libraries and languages and developing special-case tools. Systems constructed in this way usually require extensive modifications to the original application code and result in software which is complex, and difficult to debug and maintain. Implementing these systems requires substantial effort on the part of the pro-

[*] Copyright 1997 IEEE. Published in the Proceedings of IEEE 6th International Symposium on High Performance Distributed Computing, August 5-8, 1997, Portland, OR. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. This work was supported by DARPA through Army contract DABT63-94-C-0029 ARPA

grammer and makes tuning and optimizing the code difficult and time-consuming. Our research is based on the stipulation that applying the CORBA approach to distributed parallel computations will enable the programmer to develop high-performance heterogeneous scenarios quickly and efficiently.

In this paper, we describe our initial experiments with PARDIS, a distributed system which employs the key idea of CORBA — interoperability through meta-language interfaces, to implement interaction of distributed parallel applications. PARDIS extends the CORBA object model by the notion of an *SPMD object*. SPMD objects allow the request broker to interact directly with the distributed resources of a parallel application, taking advantage of locality and multiple processing resources whenever possible. To support distributed argument transfer, PARDIS introduces the notion of a *distributed sequence* — a generalization of a CORBA sequence representing distributed data structures of interacting parallel applications. We will describe two methods of argument transfer used in invocations on SPMD objects, and show how the application-level knowledge of data distribution can be employed to increase the performance of operation invocation on SPMD objects.

In brief, this paper makes the following contributions :

- describes the basic concepts underlying our vision of a parallel approach to CORBA and their interaction
- presents two methods of argument transfer in invocations made on SPMD objects and their performance analysis
- demonstrates that taking advantage of knowledge about local data distribution can bring performance improvement even in the presence of only one physical network link to support communication between the distributed locations of interacting objects.

PARDIS is an on-going project. In its final shape it is meant to be fully interoperable with vendor-supplied implementations of CORBA.

2. SPMD Objects and Distributed Sequences

CORBA defines a framework based on the concept of a *request broker*, which delivers requests from *clients* to *objects*, defined as an encapsulated entities capable of performing specific services. CORBA does not specify how an object may satisfy a request. In particular, if an object uses more than one computing resource (henceforth called a *computing thread*) in processing a request, this fact is invisible to the client and the request broker, which regard the object as a single, encapsulated entity.

There is a class of services which can be efficiently implemented by a Single Program Multiple Data (SPMD) computation — a collaboration of computing threads, each of which is working on a similar task. Those computations are very often associated with a distributed memory model, and support distributed data structures. It may be useful for an object providing such services to make the existence of the multiple computing resources visible to the request broker, since the distributed resources can make it necessary to deliver argument values (or their parts) for one request to different destinations, and interact with multiple resources in delivering the request.

PARDIS supports this notion by introducing *SPMD objects*, which can be defined as objects associated with a set of one or more computing threads visible to the request broker, and are capable of satisfying services if and only if a request for them is delivered to all the computing threads.

2.1. Programming with SPMD Objects — An Example

From the point of view of a system designer, programming with SPMD objects is not crucially different from programming with CORBA objects. Consider a simple example, in which a programmer wants to build a distributed scenario composed of two components: a parallel application A, computing simple diffusion simulation on an array distributed over the nodes A is executing on, and a parallel application B, which wants A to compute diffusion on data provided by B and to use the result of this computation. We will show how to use PARDIS in order to implement this system by making application A an SPMD object, and application B its client.

As in CORBA, the first step consists of specifying an interface to the object. In our example, application A will perform the "diffusion" service, which takes as an input argument the number of diffusion timesteps, and a diffusion array which it later returns. An IDL interface to this object would look like this:

```
interface diff_object {
   void diffusion(in long timestep,
                  inout diff_array darray);
};
```

In this specification `diff_array` is a *distributed argument* type; when delivered to the server, this argument will be distributed over the address spaces of its computing threads according to a previously specified template. We will discuss distributed sequences in detail in the next section.

Based on this specification, the IDL compiler will generate stubs translating the definitions above into the language of package (for example HPC++ [3]), in which the client and server are implemented. The stub code contains calls to communication libraries provided by PARDIS. Linked to the object's implementation, it allows the request broker to invoke methods on the object; the client can use it to invoke methods on remote objects.

For example, the C++ stub class generated for `diff_object` will offer the following functionality on the client's side:

```
class diff_object: public PARDIS::Object{
  static diff_object*
    _bind(char* obj_name, char* host_name);
  static diff_object*
    _spmd_bind(char* obj_name,
               char* host_name);
  void diffusion(int, diff_array&);
  void diffusion(int, diff_array_nd&);
  void diffusion_nb(int,
                    future<diff_array>&);
  void diffusion_nb(int,
                    future<diff_array_nd>&);
};
```

Note that the diffusion operation is represented by four methods: a method operating on distributed arguments (`diff_array`), a method operating on their non-distributed versions (`diff_array_nd`) and their non-blocking counterparts. The choice of method depends on the kind of binding established to the object implementation as explained below.

Through this proxy, the client can make calls on possibly remote objects, implemented using systems different from the client's, as if they were implemented in terms of the client's package and without the need to explicitly handle their remoteness. All client B needs to do in order to request the diffusion service is establish a binding between an object proxy and a concrete implementation, and invoke the `diffusion` method:

```
diff_object* diff =
   diff_object::_spmd_bind("example",
                           HOST1);
diff->diffusion(64, my_diff_array);
```

PARDIS provides a naming domain for objects. At the time of binding the client has to identify which particular object of a given type it wants to work with; specifying a host is optional. There are two operations which a parallel client can use to establish a binding between the client's stub representing an object and the object's implementation:

- `_spmd_bind` is a collective form of bind; it has to be called by all the computing threads of a client and should be used by clients wishing to act as one entity in interactions with objects. After `_spmd_bind`, every invocation to the object must be called by all the threads that participated in the bind call, and will result making one request on the object. If a request operates on distributed arguments, a proxy method using distributed mapping should be used. It is assumed that all threads will invoke the request with identical values of non-distributed arguments (such as `timesteps` in this example).
- `_bind` is non-collective and always establishes one binding per thread, so invoking it from all threads of a parallel program would establish multiple bindings either to the same object, or to different objects of the same type depending on arguments to `_bind`. After this form of bind, proxy methods using non-distributed mapping of distributed arguments should be used; the invocations are non-collective. This kind of interaction can be useful to parallel clients which want to interact in parallel with multiple distributed objects.

On the server's side, PARDIS uses the CORBA C++ mapping through inheritance [15] to invoke operations on the object. All the programmer of the server needs to do, is provide the implementation of an object computing diffusion simulation, and instantiate that object. In the case of both the client and the server the generated stub code contains all the code necessary to perform argument marshaling.

As the example of the client's stub shows, PARDIS supports non-blocking invocations returning *futures* (similar to ABC++ futures [14]) as its "out" arguments. This allows the client to use remote resources concurrently with its own, and provides the programmer with an elegant way of representing results which are not yet available. PARDIS also allows the server to interrupt its computation in order to process outstanding requests; full discussion of these capabilities is beyond the scope of this paper, for details refer to [13].

Principles applied in this simple scenario can be used to construct more complex interactions composed of multiple parallel applications, as well as units visualizing or otherwise monitoring their progress (see [13] for an example). Interoperability with CORBA will eventually enable PARDIS to integrate many existing systems based on this technology.

2.2 Distributed Sequences

In order to make full use of interaction with SPMD objects in a distributed environment, the programmer needs to be able to define and manipulate argument data structures distributed over the address spaces of the computing threads of an SPMD object. At this time, PARDIS provides one such structure, a generalization of the CORBA sequence, called a distributed sequence.

The distributed sequence from the example in the preceeding section can be defined in IDL as

```
typedef dsequence<double,1024,(BLOCK,BLOCK)>
   diff_array;
```

This definition represents a bounded distributed sequence of 1024 elements of type `double`, uniformly blockwise distributed on the client's as well as the server's side. In this definition, `double` can be replaced by any non-distributed type defined in IDL, ranging from basic types to complex user-defined types such as arrays, structures or interfaces. Both the length and distribution are optional in the definition of the sequence. Leaving the distribution unspecified allows interacting objects to trade sequences of different distributions at client and server, and providing run-time length specifications allows the objects to grow and shrink sequences between interactions.

For parallel C++ programs built directly on top of run-time system libraries (rather than built in terms of a parallel C++ package), a sequence is mapped to a class which behaves like a distributed one-dimensional array with additional length and distribution parameters, in a style similar to CORBA sequence mapping. The code fragment below shows an example functionality of code generated for sequence of doubles with no fixed length or distribution:

```
//IDL
typedef dsequence<double> ds_double;

//C++
class ds_double{
public:
   ds_double();
   ds_double(unsigned int length,
             DistTempl* dist = default);
//*** conversion constructor:
   ds_double(unsigned int local_length,
             double* data,
             Boolean release=FALSE);
   ds_double(const ds_double& s);
   ~ds_double();
   ds_double& operator=(const ds_double& s);

   unsigned int length() const;
   void length(unsigned int len);
   double_proxy operator[] (int index);
   void redistribute(DistTempl* dist);
   double* local_data();
   unsigned int local_length();
};
```

`operator[]` provides access to the elements of the sequence with location transparency. It is currently an error to access element beyond the value of the length of a sequence. The length of an unbounded sequence can be changed at run-time using the `length` method; if a sequence is shrunk, the data above the length value will be discarded, if a sequence is lengthened, new elements will be added to the ownership of the computing thread which owned the last elements of the old sequence. The programmer can use the `redistribute` method to redistribute elements of a sequence whose distribution is not preset.

At present, it is assumed that most invocations of the methods on the sequence will be SPMD-style, that is they will be called collectively by all the computing threads. This assumption was made in order to provide interoperability with packages based on run-time systems which do not include support for global pointers and cannot handle asynchronous access to an arbitrary context. In later versions, PARDIS will support two run-time system interfaces capturing the functionality of message passing and one-sided run-time systems which will allow us to take advantage of these two styles in our mapping.

Although the distributed sequence offers limited support for remote data access, its main purpose is to be used as a *container* for data, not provide its management. The conversion constructor (as specified in the mapping) allows the programmer to create a sequence based on his or her memory management scheme, with no data ownership. Similarly, the local access operations can be used to convert a sequence to the programmers memory management scheme.

An "in" argument on the client's side must set the length and distribution of a distributed sequence before it can be used. An "out" argument (represented as a managed type [15]) should be initialized by a distribution template before calling the operation which returns it; otherwise a uniform blockwise distribution will be assumed. The distribution of return values is always assumed to be blockwise. The server can set the distribution of a distributed sequence which is an "in" parameter to any of its operations before registering; otherwise, the distribution for that sequence will default to uniform blockwise.

An alternative to the default distribution is provided by the `PARDIS::Proportions` object, which can be constructed by proportion array or numbers (up to a point). For example, if the distribution of `diffusion_array` in operation `diffusion` of the `diffusion_object` from previous example is not predefined, the server can specify it by performing the following assignment prior to object registration:

```
_diff_object_sk::diffusion_myarray =
   new DistTempl(Proportions(2,4,2,4));
```

This will cause the elements of argument `myarray` to be distributed over the address spaces of threads 0, 1, 2 and 3 in proportions 2:4:2:4 when the diffusion operation is invoked.

The experimental mapping described here, although it provides easy integration with PARDIS to any data-parallel application implemented in C++, is not yet a fully satisfactory solution. For a truly seamless integration, the sequence will map directly to constructs present in the programmer's package (such as for example distributed vector in HPC++ PSTL [3]). We are currently working on formulating direct mappings for the HPC++ PSTL and POOMA [1] libraries.

2.3 General Design Components of PARDIS

PARDIS is a distributed software system consisting of an IDL compiler, communication libraries, object repository databases and facilities responsible for locating and activating objects. The relationship between these components is depicted in figure 1. As in other CORBA implementations, the IDL compiler translates the specifications of objects into "stub" code containing calls to communication libraries and generating requests to locating and activating agents.

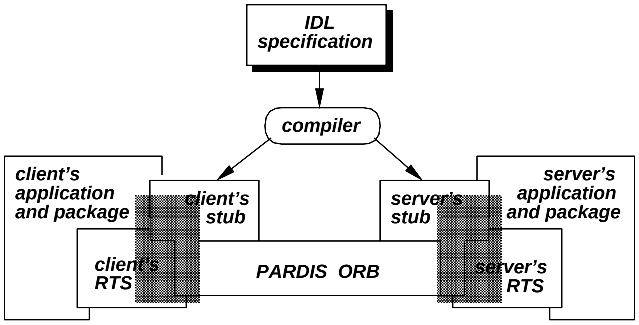

Figure 1. Interaction of main components of PARDIS: the shaded areas in the picture denote the PARDIS run-time system interface.

In order to provide support for interaction with SPMD objects and distributed sequences, PARDIS may need to issue calls to the run-time system underlying a parallel application. A generic run-time system interface has therefore been built into PARDIS libraries and may also be used by the compiler-generated stubs. To date only one run-time system interface has been specified; it encompasses the functionality of message-passing libraries and has been tested using applications based on MPI [7] and the Tulip [2] run-time system. In the future PARDIS will provide an alternative run-time system interface capturing the functionality of the more flexible one-sided run-time systems.

3. Two Methods of Distributed Argument Transfer — Experimental Performance

We have investigated two methods of implementing transfer of distributed arguments in invocations made on SPMD object. This section describes our experiments and their results.

3.1. Hardware and Description of Experiment

In the experiments described below we measure the time of invocation made by a client executing on a a 4-node SGI Onyx R4400 on an SPMD object executing on a 10-node SGI PC R8000. The network transfer is conducted over a 155 MB/s ATM link using the LAN Emulation protocol. During the experiments, the machines as well as the link were dedicated.

Both the client and the server were relying on the MPICH [12] (v 1.0.12, compiled to use shared memory) implementation of MPI [7] for their internal communication. Although the hardware we used supports shared memory, our experiments were based on a distributed memory model. The current version of PARDIS uses NexusLite to provide network transport; since we do not use the asynchronous features of Nexus, no threads additional to the implementation of client and server are spawned, and the sends and receives for large data sizes are in practice synchronous operations. Refer to [10] for details on Nexus implementation.

In order to bring out the asymmetry of interaction (different number of interacting processes at client and server, and different hardware) in our invocations we were including one "in" argument sent only from the client to the server. Both client and server assume uniform blockwise distribution of the sequence. The performance analysis was based on averages obtained over 1000 blocking invocations on the server. We would like to stress that the results given in this section are only intended to show *relative* performance of the two methods. PARDIS is still under development and no optimizations have yet been applied.

3.2. Centralized Argument Transfer

In this method of argument transfer, the SPMD object makes available only one network connection to clients. This connection is waited on by one of the SPMD threads which we will subsequently call a *communicating thread*. All other computing threads are communicating with this thread through the PARDIS interface to the run-time system underlying the object implementation. Similarly, a parallel client also designates a communicating thread which handles requests and their arguments.

On invocation, the computing threads of the client first synchronize, marshal arguments and then the request is sent to the server as one message. The communicating thread of server receives the request, unmarshals arguments and performs the request; after the invocation the server's computing threads synchronize and the communicating thread informs the client of the completion status of the request. The distributed arguments are gathered and scattered by the communicating threads of the client and server as part of the marshaling or unmarshaling process (see figure 2). This process is performed by PARDIS using the interface to the run-time system and is invisible to the programmer.

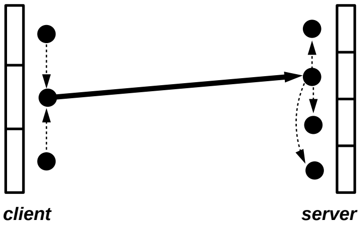

Figure 2. Centralized Argument Transfer: the dotted lines show run-time system communication taking place during argument marshaling and unmarshaling, the thick black line shows network transfer.

The main advantage of centralized argument transfer is its simplicity and for this reason it is often used in hand-coded solutions to inter-MPP communication. In our experience, it is also the most practical method of communication with parallel applications executing on front-end based architectures such as T3D.

Let t_c denote time of invocation and argument transfer in the centralized method. It can then be described as:

$$t_c = t_{gather}(n) + t_p + t_T + t_u + t_{scatter}(m)$$

where t_p is the time of packing the data, t_u is the time of receiving and unpacking the data, t_T denotes the time from the beginning of the send operation to the end of the receive operation, and n and m are the numbers of computing threads of the client and server respectively. We will investigate how these times influence the total invocation time in different configurations of client and server. In our measurements we also included the time it took to complete the process of sending the sequence as it proved to influence the results ($t_{p\&s}$ denotes time of packing and sending, time of packing is constant). Since it is likely that invocations on SPMD objects will most often involve transferring large arguments, we will concentrate on evaluating the efficiency of this method for a relatively large sequence composed of 2^{17} of elements of type double. Table 1 summarizes the results.

The results show that the increase in the time of invocation is accounted for by two main factors: the cost of gather and scatter (t_{gather} and $t_{scatter}$ in the table above) and by the increase in time of send and receive ($t_{p\&s}$ and t_u) as the number of computing threads on either side goes up. Since exactly the same operations are involved in packing, sending and receiving the message each time, we hypothesize that the latter effect is due to scheduler interference. It appears that the computing threads are descheduled on issuing system calls and that increasing the number of computing threads decreases the probability that a particular thread will be scheduled at any time. Communication always takes place between a particular pair of threads and is synchronous for large data sizes, so this behavior will cause the time of send to increase, leading also to the increase of total invocation time. However, even assuming that this effect could be eliminated, from the times of gather and scatter we can see that the time of argument transfer would still grow with the number of client's and server's resources.

3.3. Multi-Port Argument Transfer

In order to enable multi-port argument transfer, each computing thread of the SPMD object opens a network connection on a separate port. These connections become a part of object reference for this particular object and are accessible to clients wanting to connect. Similarly, parallel clients also open multiple connections, one per thread, so that each computing thread of the client can communicate directly with each thread of the server.

In the centralized method, argument transfer is a part of the invocation message; all information associated with a request is sent in one message. However, sending the invocation to every computing thread instead of having only one thread broadcast it to others could lead to contention between different invoking clients (as within the same object some threads might accept invocation from one client while others accept invocation from another resulting in calling different methods). In this method, we will therefore separate the invocation from the argument transfer. The invocation header will be delivered using the centralized method as above, and upon its receipt the computing threads will await argument transfer on network ports. As in the case of the centralized method, the client's threads synchronize on making the invocation and the server's threads synchronize after the invocation is completed.

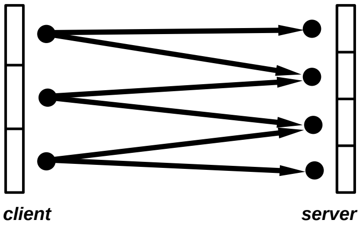

Figure 3. Multi-Port Argument Transfer

Transferring the arguments from each thread may involve sending them to more then one destination (see figure 3). Based on information provided by the ORB, the client's

	$n = 1$ $t_{gather} = 0.74$				$n = 2$ $t_{gather} = 33.6$				$n = 4$ $t_{gather} = 43.2$			
m	t_c	$t_{p\&s}$	t_u	$t_{scatter}$	t_c	$t_{p\&s}$	t_u	$t_{scatter}$	t_c	$t_{p\&s}$	t_u	$t_{scatter}$
1	417	380	16.7	0.2	497	421	17.1	0.2	571	486	15.9	0.2
2	442	382	20.5	21.3	529	430	20.3	20.2	634	528	20	18.9
4	451	385	21.1	25	538	433	21.2	24.6	685	571	21.1	25.5
8	461	394	21.8	25.8	552	446	21.7	26.2	697	577	21.6	26.7

Table 1. Time of invocation using the centralized method of argument transfer: m is the number of server's processes, n is the number of client's processes, the time is given in milliseconds.

threads first calculate to which of the server's threads they should send data. Each thread then marshals the part of data it owns, and sends it. The server's threads receive all the data transfers associated with a given request and unmarshal them according to information contained in the transfer header. Timing invocation and argument transfer of this method on a sequence of 2^{17} of elements of type double is summarized in table 2. Here, the times of send, unpacking and receive, and packing (marshaling) (t_{send}, t_u and t_p respectively) represent the maximum over all threads involved; time of post-invocation synchronization ($t_{exit\_barrier}$) comes from the communicating thread.

$n = 1$					
m	t_{mp}	t_p	t_{send}	t_u	$t_{exit\_barrier}$
1	420	37.2	338	23.5	0.03
2	417	38.4	348	18.3	165
4	408	35.1	347	8.1	256
8	412	30.9	356	3.5	307

$n = 2$					
m	t_{mp}	t_p	t_{send}	t_u	$t_{exit\_barrier}$
1	431	15.9	361	23.6	0.03
2	425	16.4	358	12.6	3.9
4	412	17	352	7.5	169
8	393	16.4	336	3.5	240

$n = 4$					
m	t_{mp}	t_p	t_{send}	t_u	$t_{exit\_barrier}$
1	367	13.1	285	25.8	0.03
2	376	13.8	298	13.5	3.9
4	368	13.4	296	6.4	8.3
8	336	13.1	261	3.4	129

Table 2. Time of invocation using the multiport method of argument transfer: m is the number of server's processes, n is the number of client's processes, the time is given in milliseconds.

These results indicate that two send operations initiated by separate computing threads of the client are completed at roughly the same time on the server's side. For example, the time spent in exit barrier for $n = 1, m = 2$ corresponds to roughly half of the send operation, which means that the sends were sequentialized. When $n = 2, m = 2$ however, the threads are nearly synchronized on the post-invocation barrier which means that they completed receive operations at roughly the same time. We verified this behavior for different cases by comparing the amount of time different processes of the server spent in the barrier and conclude that data transfer from two separate computing threads of the client did not happen sequentially, but was interleaved.

This fact helps explain the decrease in the time of send as we increase the number of threads at client and server beyond a certain threshold. We assume that it is more probable that *any* of a number of threads will be scheduled to receive or send than that *a particular* thread will be scheduled (this behavior holds till m or n exceed the capacity of the machines). If this assumption is correct and more than one send operation is in progress at the same time, then the higher probability of the sending or receiving process being scheduled will result in quicker response to the send operation and decrease overall send time. Note that the time of send initially increases as we begin to increase the number of client's and server's threads which we attribute to the scheduler interference, which caused similar behavior in the centralized method.

Finally, the results show that the time of packing and unpacking for any given thread decreases as n and m increase, since each thread becomes responsible for smaller chunks of data. Further, these operations are performed in parallel to at least some degree (for packing: compare cases when $n = 1, m = 1$ and $n = 2, m = 1$; for unpacking: note that when $n = 2, m = 2$ the time spent in barrier is 3.9 ms while the time of unpacking is 12.6 ms) and thus contribute to the overall decrease in invocation time. We expect that this effect will be amplified in cases which require data translation (not present in our experiments) or more sophisticated marshaling. Note that this method allows us to use the full processing capability of client and server for argument transfer.

Overall, we can describe the time of invocation in the multi-port method as

$$t_{mp} = t_p(n) + t_T + t_u(m)$$

Since t_p and t_u decrease as n and m increase it is reasonable to assume that in the absence of other factors t_{mp} will decrease with the increase of resources on client's and server's side.

In all the cases shown above the sequence can always be divided very efficiently (only the minimum number of sends in each case), and all the threads of the sender (the client) are sending the same amount of data, so that none are faster than others. Experiments show that cases when the sequence is split unevenly are of comparable efficiency (for example for $n = 3$ and $m = 5$ in the same experiment the timing of the invocation was 370 milliseconds).

3.4. Comparison

So far we considered the behavior of the two methods in the context of fixed argument size and changing client and server configurations. The graph below compares the effective bandwidth of an "in" argument transfer, including all the invocation overhead, for different data sizes in the most powerful client-server configuration considered ($n = 4$ and $m = 8$).

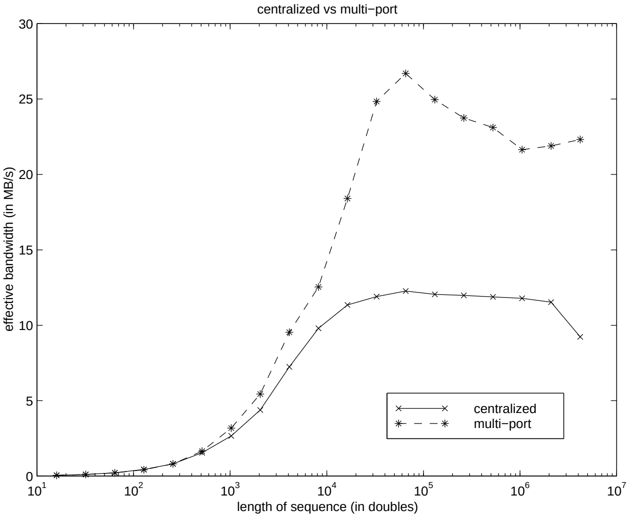

Figure 4. Performance of centralized versus multi-port method of argument transfer configured at $n = 4$, $m = 8$

The multi-port method peaks at 26.7 MB/s for sequence of length 2^{17} doubles. The highest bandwidth for the centralized method is 12.27 MB/s for sequence of length 2^{16} doubles. The data indicates that for small data sizes the performance of both methods is nearly the same, and that for large data sizes the multi-port method significantly outperforms the centralized method. This relationship is similar for other configurations; although invocation times of the multi-port method fluctuate, we have not found a case in which it would underperform the centralized method.

From the experimental results it is evident that these methods behave differently with the increase of resources on either client's or server's side. In the case of the centralized method, the time of argument transfer grows with the increase of computational resources at client and server, as the time of both gather and scatter grows. In the case of multi-port transfer however, the time decreases because we take advantage both of data locality (communication is direct, no need for gather and scatter) and employ the full computational power of client and server in parallel for data marshaling and other communication processing. Further, in the particular hardware configuration used in this experiment, the multi-port method allowed us to better utilize the network link by reducing the scheduler interference.

4. Related Work

Many researchers have investigated the design and efficiency of tools and environments allowing the programmer to build distributed high-performance systems. This research primarily centers on two areas: multimethod runtime systems and metacomputing environments.

Multimethod run-time systems, such as Nexus [8] and Horus [17], integrate diverse transport mechanisms and protocols under one interface. This allows the programmer to treat a collection of supercomputers connected by a network as one virtual metacomputer, knowing that the most optimal communication method will be applied to communication between any two nodes of this virtual machine. This low-level approach, although very effective for many applications, still requires the programmer to write his or her code in terms of the interface to a given run-time system. In contrast, our approach does not interfere with the run-time system or package used by a given application which allows the programmer to choose run-time system interface best suited to his or her needs. As a consequence, the programmer does not need to rewrite the application code and can reuse already existing components in building meta-applications.

Large-scale metacomputing environments such as Legion [11], Globus [9] or WWVM [6] focus on providing interoperability of many diverse components. They address problems of scheduling, I/O systems, component compilation and resource management. NetSolve [4] provides interfaces to standard scientific tools such as Matlab and allows client-server interaction between computing units. It also attempts to load-balance its applications. Our focus is different; we are trying to provide explicit abstractions geared specifically towards interoperability of parallel objects rather then develop an environment integrating diverse components. Far from attempting to incorporate all of their features, PARDIS could exist as one of the communication subsystems in the environments mentioned above.

Active research is also being done on optimizing the performance of CORBA for high-speed networks. The TAO project [16] focuses on developing a high-performance, real-time ORB providing quality of service guarantees, optimizing the performance of network interfacing and ORB components. This research is concerned mainly with increasing performance by optimizing the architectural components of CORBA, not by introducing new concepts on the level of object model.

5. Conclusions and Future Work

In this paper we have introduced the concept of SPMD objects and simple distributed argument structures, distributed sequences, which support it. SPMD objects and distributed sequences are designed to provide the programmer of parallel objects with an easy and efficient way of integrating his or her applications into a heterogeneous, distributed environment. These concepts were implemented in PARDIS, a system based on CORBA design principles which in its final version will support interoperability with CORBA.

We have also presented two different methods of implementing argument transfer in method invocations on SPMD objects containing distributed arguments. Results obtained using the multi-port method show that by exploiting data locality in different computing threads of client and server, and employing all the available computing power for argument transfer processing, it is possible to reduce the time of operation invocation on SPMD objects even in the presence of only one network connection between client and server. Further, a very desirable characteristic of the multi-port method is that the time of argument transfer decreases with the increase of computational resources of client and server. This shows that SPMD objects are not only useful from the point of view of programmer convenience, but also provide an efficient solution for communication between distributed parallel objects.

Our most immediate plans focus on investigating different strategies of distributed argument transfer in different hardware configurations and under different assumptions about argument distribution. Once these issues are resolved, we plan to continue our work on direct mapping strategies for concrete packages such as HPC++ and POOMA. This will enable us to test the capabilities of PARDIS on real world applications and provide insight into the design of other distributed argument structures.

References

[1] S. Atlas, S. Banerjee, J. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A High Performance Distributed Simulation Environment for Scientific Applications. In *Supercomputing '95 Proceedings*, December 1995.

[2] P. Beckman and D. Gannon. Tulip: A Portable Run-Time System for Object-Parallel Systems. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[3] P. Beckman, D. Gannon, and E. Johnson. Portable Parallel Programming in HPC++. In *International Conference on Parallel Processing*, 1996.

[4] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. In *Supercomputing '96 Proceedings*, November 1996.

[5] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-Way: Wide-Area Visual Supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2):123–131, 1997.

[6] K. Dincer and G. C. Fox. Building a World-Wide Virtual Machine Based on Web and HPCC Technologies. In *Supercomputing '96 Proceedings*, November 1996.

[7] M. P. I. Forum. *MPI:A Message-Passing Interface Standard*. June 1995.

[8] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Multimethod Communication for High-Performance Metacomputing Applications. In *Supercomputing '96 Proceedings*, November 1996.

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *to appear in The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

[10] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. *Technical Memorandum ANL/MCS-TM-189*, May 1994.

[11] A. S. Grimshaw and W. A. Wulf. Legion — A View From 50,000 Feet. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computaiton*, August 1996.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1996.

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

[14] M. L. Norman, P. Beckman, G. L. Bryan, J. Dubinski, D. Gannon, L. Hernquist, K. Keahey, J. P. Ostriker, J. Shalf, J. Welling, and S. Yang. Galaxies Collide on the I-WAY: An Example of Heterogenous Wide-Area Collaborative Supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2):132–144, 1997.

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

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

[17] D. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A High-performance Endsystem Architecture for Real-time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[18] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A Framework for Protocol Composition in Horus. In *Proceedings of Principles of Distributed Computing*, 1995.