



Interoperability of Data Parallel Runtime Libraries *

Guy Edjlali, Alan Sussman and Joel Saltz
Department of Computer Science
University of Maryland
College Park, MD 20742
{edjlali, als, saltz}@cs.umd.edu

Abstract

This paper describes a framework for providing the ability to use multiple specialized data parallel libraries and/or languages within a single application. The ability to use multiple libraries is required in many application areas, such as multidisciplinary complex physical simulations and remote sensing image database applications. An application can consist of one program or multiple programs that use different libraries to parallelize operations on distributed data structures. The framework is embodied in a runtime library called Meta-Chaos that has been used to exchange data between data parallel programs written using High Performance Fortran, the Chaos and Multiblock Parti libraries developed at Maryland for handling various types of unstructured problems, and the runtime library for pC++, a data parallel version of C++ from Indiana University. Experimental results show that Meta-Chaos is able to move data between libraries efficiently, and that Meta-Chaos provides effective support for complex applications.

1. Introduction

Distributed parallel programs are used to speed up the time to complete an application. To achieve this goal, such programs rely on partitioning data and computation among the available processors. They are considered difficult to write, to maintain and to modify. Many parallel programming paradigms have been developed, with the most important one for large scale scientific applications being data parallelism. Data parallel applications can currently be written using a high level language, for example High Performance Fortran (HPF) [12] or pC++ [3]. Data parallel programs can also be written using a sequential programming language and runtime libraries for performing communication. These libraries can be low level communication libraries such as MPI [16] or PVM [7], or application specific runtime libraries that encapsulate communication into higher level functions, such as Chaos [11] or LPARX [13]. However, inter-application communication to allow multiple data parallel programs to cooperate to solve a single problem is rare, because such programs are difficult to write and there are few tools

available to develop them.

To motivate our work, we present the following simple scenario. A client program, running sequentially or in parallel, requires the services of a parallel server, on the same or another (parallel) machine. The server could provide functionality that is not available in the client, or provide additional computational power to make the client run significantly faster. Let us be more concrete with our example: let the client be a sequential C program and the server be an HPF parallel program, and the two programs exchange one parameter: a matrix of size $M \times N$. The matrix is stored as a two-dimensional array in both the client and the server. In addition, suppose the array is distributed on the server in a block-cyclic fashion, to optimize the server computation. Because of the block-cyclic distribution of data on the server, the client process must communicate with every server process. This operation requires a *collective communication operation* [16]. The client must determine which parts of the matrix are going to be sent to each of the processes in the server. Similarly, each server process must determine which part of the array it will receive from the client, and the order in which the array elements will arrive. To determine this information, the client process needs to know the *data distribution* on the server. However, only knowing the data distribution is insufficient to perform the communication for this example. The matrix stored in the memory of the client is laid out in row major order (C style), while the matrix stored on (each processor of) the server is stored in column major order (Fortran style). Therefore the *mapping* between the data elements on the client side and on the server side has to be specified. This example illustrates the three points we focus on in this paper: collective communication, data distribution and data mapping.

In this paper, we present a *meta-library* approach that achieves direct application to application data transfer. By a meta-library, we mean a runtime library-based system that interacts with the data parallel libraries and languages used to implement the separate applications. The meta-library can handle any data distribution, and can be used for any data parallel library or language construct that distributes data for the sequential or parallel applications. The meta-library can be used to allow the exchange of data between separate (sequential or parallel) programs, and can also be used to allow data transfers between data managed by different data parallel libraries in the same application.

An example that illustrates the utility of the meta-library approach comes from a computational aerodynamics problem. Com-

* This research was supported by NASA under grant #NAG-1-1485 (ARPA Project Number 8874) and by ARPA under grant #F19628-94-C-0057. The Maryland IBM SP2 and Digital AlphaServer used for the experiments were provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and grants from IBM and Digital Equipment Corporation.

putational fluid dynamics (CFD) flow solvers often use different types of meshes to represent different physical structures. For example, the space around an airplane body may be modeled with a structured mesh, while the nose, wing and tail may be modeled with an unstructured mesh. The data parallel numerical solution techniques used for the flow fields employ algorithms specifically designed for each type of mesh and often use runtime library support optimized for a particular solution technique. To allow interactions between the different meshes at their shared boundaries, it is necessary for the different parallel libraries that distribute the meshes to exchange data. However, such functionality is not easily achieved for arbitrary libraries. The meta-library framework we describe in this paper provides the mechanisms needed to perform the communication.

In sequential programs, intra-application and inter-application communication is commonly used in a networked environment. Such programs often use low level communication calls (e.g., sockets) to move data between separate address spaces. Building manageable distributed applications with these low level communication calls can be difficult. Therefore many distributed applications instead use the Remote Procedure Call (RPC) [2] paradigm to hide many communication details from the application programmer. RPC extends the notion of a procedure call in a sequential program by allowing the transfer of both data (parameters and return values) and control over a network from one process to another. Two processes are involved in the RPC call: the client and the server. In addition, the CORBA object model [15] provides RPC-like capability for a distributed object model. RPC provides a simple, efficient programming model for heterogeneous sequential applications. Our meta-library framework can be seen as a step towards allowing RPC-like functionality for data parallel programs, by providing the infrastructure needed to pass data between parallel or sequential clients and servers.

Much effort has recently gone into languages and runtime libraries for combining task and data parallelism. For example, the data parallel language Fx [17] extended HPF to support task parallelism. However, none of that work addressed the problem of allowing programs written using different data parallel libraries or languages to inter-operate. All the systems that we will discuss in Section 6 are designed to work only within a single (extended) language or with one data parallel library. In addition, none of those designs allow multiple application-specific data parallel libraries to exchange data within a single program. Our meta-library based system works both for separate data parallel programs written using any data parallel library or language (that exports the required set of inquiry functions) and for a single data parallel program that uses multiple data parallel libraries to optimize performance.

We have developed a prototype implementation based on the meta-library approach. We call our runtime library Meta-Chaos. An implementation currently runs on the IBM SP2 multicomputer, an eight-node Digital Alpha cluster of SMPs and a cluster of workstations. The libraries and languages currently supported by Meta-Chaos include Fortran, C, HPF [12], pC++ [3], Multiblock Parti [1] and Chaos [11]. Our results indicate that this approach is feasible and that the overhead of our general approach is acceptable. Our results also show that the data parallel library extensions required by the meta-library are not difficult to implement, even by someone other than the implementor of Meta-Chaos.

The rest of the paper is organized as follows. Section 2 presents

a high level overview of interoperability between two data parallel libraries, while Section 3 discusses the system design of the meta-library. Section 4 describes an implementation of the meta-library, called Meta-Chaos, and Section 5 presents several experiments designed both to quantify the overhead costs encountered when an application uses Meta-Chaos and to show the benefits that can be obtained from using Meta-Chaos to structure an application. Section 6 compares the meta-library approach to previous work, and we conclude in Section 7.

2. Basic Concepts

Figures 1 and 2 provide a high-level view of interoperability between two data parallel libraries, for two different scenarios. Suppose we have programs written using two different data parallel libraries named **libX** and **libY**, and that data structure A is distributed by **libX** and data structure B is distributed by **libY**. Then the scenario presented in Figure 1 consists of copying multiple elements of A into the same number of elements of B, with both A and B belonging to the same data parallel program. On the other hand, the scenario presented in Figure 2 copies elements of A into elements of B, but A and B belong to different programs. Our system design provides a standard set of techniques for performing the copy operation.

The two examples show the main steps needed to copy data distributed using one library to data distributed using another library. More concretely, these steps are (1) specify the elements to be copied (sent) from the first data structure, distributed by **libX**, (2) specify the elements to be copied (received) into the second data structure, distributed by **libY**, (3) specify the correspondence (mapping) between the elements to be sent and the elements to be received, (4) build a communication schedule, by computing the locations (processors and local addresses) of the elements in the two distributed data structures, and (5) perform the communication using the schedule produced in step 4.

These five steps could be performed directly by the application programmer. However this supposes that the application programmer (a) knows the locations of the source data A1 on the processors where A is distributed by **libX** (perhaps requiring the programmer to look at internal data structures of **libX**), (b) knows the location of the destination data B1 on the processors where B is distributed by **libY** (perhaps requiring the programmer to look at internal data structures of **libY**), (c) can easily determine the mapping between the elements to be sent from A and the elements to be received into B (note that A and B may be in different programs), (d) can compute a communication schedule, and (e) can perform the communication.

The application programmer could perform these operations, but that requires the programmer to understand the management of both A and B, even if the management is encapsulated in a data parallel runtime library. The resulting code could be difficult to optimize across multiple parallel platforms, and any changes to the management of the data structures would have to be reflected in the implementation of the algorithm, so code maintenance would be difficult. Also, determining the mapping between elements in A1 and elements in B1 could be complex.

However, the data parallel library writer knows the internal data structures and behavior of the library and can optimize the performance of operations on data distributed by the library. This observation leads us to require the library builder to export some

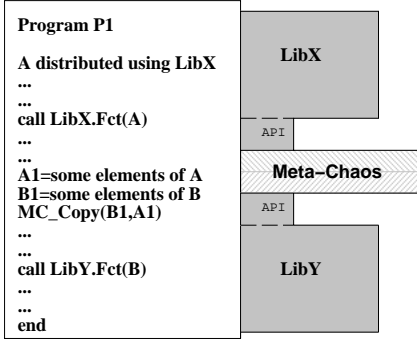


Figure 1. Communicating between two libraries within the same program

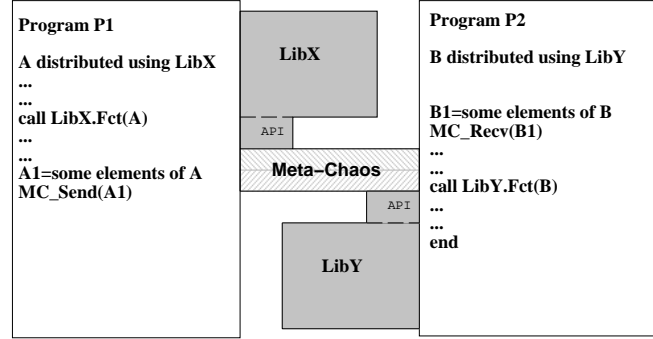


Figure 2. Communicating between libraries in two different programs

additional services, to support interoperability with other data parallel libraries. The application programmer specifies the data to be moved for both the source and destination. On the other hand, the library writer provides a way to determine the location of the data (processor and local address). The mapping between the source data and the destination data must also be specified. This can be done via a process we call *linearization*. Providing a linearization is the responsibility of the library writer, and will be described in Section 3.3.

A new piece of software, which we call a *meta-library*, must be written to compute the communication schedule at runtime. The meta-library uses the specification of the data to be sent and received by the application programmer(s) and the services exported by the data parallel libraries to build the schedule. The meta-library can also provide data transport routines that are appropriate for the execution environment (e.g., message passing). Using the meta-library, the application programmer only defines the source and destination of the data to be transferred, and the meta-library does the rest of the work.

3. System design

3.1. Options

There are at least three potential solutions to provide a mechanism for allowing data parallel libraries to interoperate. The first approach is to identify the unique features provided by all existing data parallel libraries and implement those features in a single integrated runtime support library. The major problem with such an approach is extensibility.

A second approach is to use a custom interface between each pair of data parallel libraries that must communicate. However, if there are a large number of libraries that must interoperate, say n , this method requires someone to write n^2 communication functions. So this approach also has the disadvantage of being difficult to extend.

The third approach is to define a set of interface functions that every data parallel library must export, and build a meta-library that uses those functions to allow all the libraries to interoperate. This approach is often called a framework-based solution, and is the one we have chosen for our design. This approach gives the task of providing the required interface functions to the data par-

allel library developer (or a third party that wants to be able to exchange data with the library). The interface functions provide information that allows the meta-library to inquire about the location of data distributed by a given data parallel library. Providing such functions does not prevent a data parallel library developer from optimizing the library for domain-specific needs.

3.2. Data specification

The data to be transferred are specified by the application programmer. The data are distributed and otherwise managed by one or more data parallel libraries. Most such libraries provide a compact way to describe groups of elements in a distributed data structure (e.g., a range of subscripts for a distributed array). We will call this description of a set of elements a *Region* type. Hence the library builder must specify the *Region* type for a given library, so that the meta-library will be able to map elements in the source data parallel library to elements in the destination library. For example, High Performance Fortran (HPF) [12] and Multiblock Parti [1] utilize arrays as their main distributed data structure; therefore the *Region* type for them is a regularly distributed array section. Chaos [11] employs irregularly accessed arrays as its main distributed data structure, either through irregular data distributions or accesses through indirection arrays. For Chaos the *Region* type would be a set of global array indices.

A *Region* type is dependent on the requirements of the data parallel library. The library builder must provide a *Region* constructor for each *Region* type to create regions and a destructor to destroy the *Regions* specified for that library.

A single *Region* is not always sufficient to characterize the data to be moved. Thus multiple *Regions* to be moved can be specified. *Regions* are gathered into an ordered group we call a *SetOfRegions*. A mapping between source and destination data structures therefore specifies a *SetOfRegions* for both the source and the destination.

3.3. Linearization

Linearization is the method by which the meta-library can define an implicit mapping between the source of a data transfer distributed by one data parallel library and the destination of the transfer distributed by another library. The source and destination data elements are each described by a *SetOfRegions*.

One view of the linearization is as an abstract data structure that provides a total ordering for the data elements in a `SetOfRegions`. The linearization for a `Region` is provided by the library writer.

We represent the operation of translating from the `SetOfRegions` S_A of A , distributed by `libX`, to its linearization, L_{S_A} , by ℓ_{libX} , and the inverse operation of translating from the linearization to the `SetOfRegions` as ℓ_{libX}^{-1} :

$$L_{S_A} = \ell_{libX}(S_A)$$

$$S_A = \ell_{libX}^{-1}(L_{S_A})$$

Moving data from the `SetOfRegions` S_A of A distributed by `libX` to the `SetOfRegions` S_B of B distributed by `libY` can be viewed as a three-phase operation:

1. $L_{S_A} = \ell_{libX}(S_A)$
2. $L_{S_B} = L_{S_A}$
3. $S_B = \ell_{libY}^{-1}(L_{S_B})$

The only constraint on this three-phase operation is to have the same number of elements in S_A as in S_B , in order to be able to define the mapping from the source to the destination linearization (the second operation).

The concept of linearization has several important properties. First, it is independent of the structure of the data, and thus very flexible. Any data structure can be transferred to any other data structure, so long as a mapping can be specified. For example, several elements of a distributed tree data structure can be transferred to several elements of a distributed array data structure as long as (a) a linearization is provided by the library used to create the distributed tree data structure, (b) a linearization is provided by the library used to create the distributed array structure, and (c) the number of elements specified to be transferred are the same in the tree and in the array. Second, linearization does not require the explicit specification of the mapping between the source data and destination data. The mapping is implicit in the separate linearizations of the source and destination data structures. Third, a linearization is only an abstract, not a physical object. No space must be allocated for the linearization of a `SetOfRegions` in the memory of either the source or the destination program. The meta-library can transfer data directly from the source `SetOfRegions` to the destination `SetOfRegions`, never building a data structure for the linearization. Fourth, a parallel can be drawn between a linearization and the marshal/unmarshal operations for the parameters of a remote procedure call. Linearization can be seen as an extension of the marshal/unmarshal operations to distributed data structures. Last, for optimization purposes, multiple linearizations can be provided by a library writer for the same `Region` type in a single data parallel library.

3.4. Example

Figure 3 shows a data copy from distributed array A into distributed array B , with the `SetOfRegions` defined as shown. For this example, a `Region` for the array is a regular section, and the order within a section is row major. The `SetOfRegions` for A and B define the 1-1 mapping between elements for the copy.

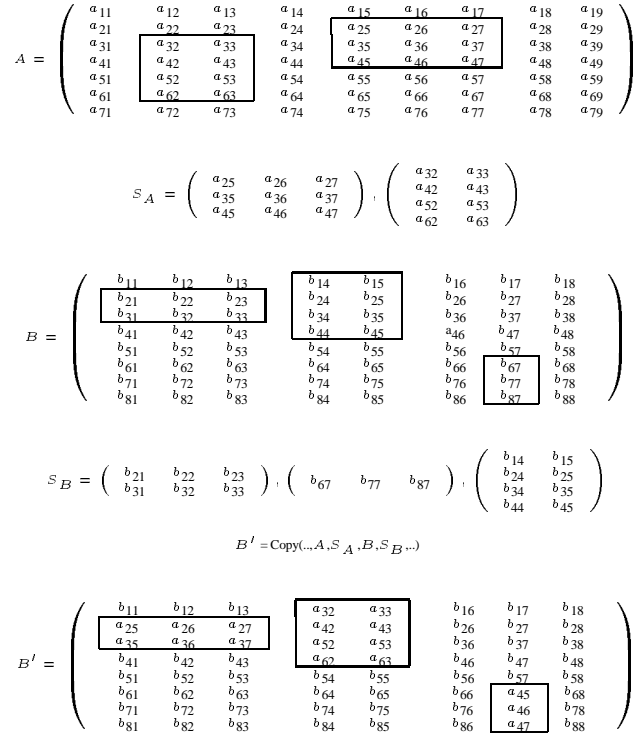


Figure 3. Before and after a data copy from distributed array A to distributed array B

3.5. Data parallel library extensions

A communication schedule describes the data motion to be performed for the specified transfer. From the `SetOfRegions` specified by the application programmer the meta-library can determine the elements to be moved, and where to move them. The meta-library applies the (data parallel library-specific) linearization mechanism to the source `SetOfRegions` and to the destination `SetOfRegions`. The linearization mechanism generates a one-to-one mapping between elements of the source `SetOfRegions` and the destination `SetOfRegions`.

Implementation of the schedule computation algorithm requires that a set of procedures be provided by both the source and destination data parallel libraries. These procedures are essentially a standard set of inquiry functions that allow the meta-library to perform operations such as (1) dereferencing an object to determine the owning processor and local address, and a position in the linearization, (2) manipulating the `Regions` defined by the library to build a linearization, and (3) packing the objects of a source `Region` into a communication buffer, and unpacking objects from a communication buffer into a destination `Region`.

4. Implementation - Meta-Chaos

We have implemented our system design for the meta-library on a network of four-processor SMP Digital Alpha Server 4/2100 workstations, on an IBM SP2, and on a network of Sun workstations.

We call the system Meta-Chaos, because it borrows several of its implementation techniques from Chaos.

4.1. Interface to the Library Builder

A major concern in designing the system was to require that relatively few procedures be provided by the data parallel library implementor, to ease the burden of integrating a new library into the Meta-Chaos framework. So far, implementations for several data parallel libraries have been completed, including the High Performance Fortran runtime library, the Maryland Chaos and Multiblock Parti libraries for various types of irregular computations, and the pC++ [3] runtime library, Tulip, from Indiana University. The pC++ implementation of the required functions was performed by the pC++ group at Indiana in a few days, using MPI as the underlying message passing layer, which shows that providing the required interface is not too onerous.

4.2. Communication cost

Once a communication schedule has been computed, Meta-Chaos uses the information in the schedule to copy data into contiguous communication buffers in each processor managed by the source data parallel library. Similarly, Meta-Chaos uses the information in the schedule to extract data from communication buffers into the memory of each processor managed by the destination data parallel library. The communication buffers are transferred between the source and destination processors using either the native message passing mechanism of the parallel machine (e.g., MPL or MPI on the IBM SP2), or using a standard message passing library on a network of workstations (e.g., PVM or MPI). Messages are aggregated, so that at most one message is sent between each source and each destination processor.

A set of messages crafted by hand to move data between the source and the destination data parallel libraries would require exactly the same number of messages as the set created by Meta-Chaos. Moreover, the sizes of the messages generated by Meta-Chaos are also the same as for the hand-optimized code. The only difference between the two set of messages would be in the ordering of the individual objects in the buffers. This ordering depends on the order of the bijection between the source objects and the destination objects used by Meta-Chaos (which is based on the linearizations provided by the source and destination data parallel libraries), and the order chosen by the hand-crafted procedure.

The overhead introduced by using Meta-Chaos instead of generating the message passing by hand is therefore only the computation of the communication schedule. Since the schedule can often be computed once and reused for multiple data transfers (e.g., for an iterative computation), the cost of creating the schedule can be amortized.

4.3. Non-array aggregate data structures

Meta-Chaos requires only linearizations (i.e. a one to one implicit mapping from the source to the destination `SetOfRegions`) to be able to exchange data between different data parallel libraries. Therefore Meta-Chaos is not constrained to work only for distributed arrays, but can also be used for other distributed aggregates, including pointer-based structures such as trees and graphs. The only constraint is that each library that supports non-array distributed data structures provide a method for linearizing the data structures it supports. For example, Meta-Chaos could be used to allow two C++ programs parallelized using pC++ constructs [3],

each containing compatible pointer-based data structures, to exchange parts of the data structures. More complex scenarios, with programs parallelized using different libraries and exchanging data between non-array data structures, can also be supported.

5. Experimental Results

We present two classes of experiments to evaluate the feasibility of using Meta-Chaos for efficient interaction between multiple data parallel libraries. The first class of experiments, in Sections 5.1 and 5.2, presents a set of application scenarios that quantify the overheads associated with using Meta-Chaos. The second class of experiments, in Section 5.3, is designed to show the benefits that Meta-Chaos can provide by allowing a sequential or parallel client program to exploit the services of a parallel server program implemented in a data parallel language (HPF).

Meta-Chaos is designed to operate efficiently for at least two significantly different patterns of communication. The first pattern, often called irregular communication, requires specifying each element separately (e.g., through an indirection array), while the second pattern is more regular, and usually specifies entire groups of elements in a compact way (e.g., with a regular section). To evaluate the overheads incurred in using Meta-Chaos, we present experiments that generate communication schedules and copy data for both types of communication patterns. These two patterns provide upper and lower bounds on the communication performance of Meta-Chaos, because they represent the least and most compact representations of the data to be moved. We also compare communication cost using Meta-Chaos to the communication cost of highly optimized and specialized data parallel libraries, which in these experiments are the Chaos and Multiblock Parti libraries.

5.1. Interaction between a structured and an unstructured mesh in one program

One scenario where communication between two different libraries can occur in the same program is when the program performs a sweep through a regular mesh followed by a sweep through an irregular mesh. Both meshes are defined in the same program, but the regular mesh is distributed regularly (using the Multiblock Parti library) while the irregular mesh is distributed irregularly (using the Chaos library).

Meta-Chaos is used to perform a copy operation between the regular and the irregular mesh. The schedule generated by Meta-Chaos is used multiple times, twice per time-step, to perform the data copies. All that must be done is to select the proper source and destination for each data copy, so that Meta-Chaos can generate message sends from the source mesh and receives into the destination mesh.

It is also possible to compute the communication schedule by treating the regular mesh generated by Multiblock Parti as an irregular mesh. To do that, a Chaos-style translation table has to be created to describe the pointwise data distribution. The translation table can be utilized by Chaos to directly compute a communication schedule for moving data between the regular and irregular meshes. However, the correspondence between the points in the regular mesh and the Chaos representation of the mesh must be stored explicitly.

For this experiment, the parallel programming environment is a 16 processor IBM SP2. The data is a two-dimensional array of double precision floating point numbers of size 256x256, regularly distributed by blocks in both dimensions onto the processors, us-

		Number of processors			
		2	4	8	16
Chaos	schedule	1099	830	437	215
	copy	64	52	38	33
Meta-Chaos with cooperation	schedule	1509	832	436	215
	copy	71	50	32	21
Meta-Chaos with duplication	schedule	2768	1645	1025	745
	copy	70	50	33	21

Table 1. Schedule build time (total) and data copy time (per iteration) for regular and irregular meshes in one program on SP2, in msec

ing the Multiblock Parti data distribution routines. The irregular mesh contains 65536 points, stored into a Chaos array irregularly distributed among the processors. The two data parallel libraries are both called from the same program.

There are two different ways to compute a schedule using Meta-Chaos; in Table 1 they are called *cooperation* and *duplication*. The terms refer to the way Meta-Chaos computes schedules. For *cooperation*, Meta-Chaos computes ownership of the source objects (processor, local address, etc.) in the processors running the source program using the functions provided by the source data parallel library. That information is sent to the processors running the destination program, which also compute the corresponding information for the destination objects using the destination parallel library functions and then compute the complete schedule for both the source and destination processors. The computed schedule is then sent to the source processors. On the other hand, when Meta-Chaos computes schedules with *duplication*, the source and destination processors first exchange data descriptors for both source and destination distributed data structures. This method assumes that, for two separate programs using Meta-Chaos to exchange data, both the source and destination processors know how to interpret the data descriptors (i.e. both sides have the code for both data parallel libraries).

The cost of the schedule computation for Chaos is dominated by the calls to the Chaos `dereference` function, which performs the translation from a global (sequential) array index into a processor number and local address. The Meta-Chaos implementation with *cooperation* also uses the same Chaos `dereference` function, which is why the schedule computation costs for the two methods are very similar. On the other hand, the Meta-Chaos implementation with *duplication* must call the Chaos `dereference` function twice for each array element to be copied, which explains why the cost of building the schedule with that method costs about twice as much as for the other two implementations.

The major difference in the cost of the data copy using Chaos

		Number of processors			
		2	4	8	16
Block Parti	schedule	19	11	10	9
	copy	467	195	101	53
Meta-Chaos with cooperation	schedule	29	29	20	25
	copy	396	198	102	52
Meta-Chaos with duplication	schedule	24	20	14	13
	copy	396	198	102	52

Table 2. Schedule build time (total) and data copy time (per iteration) for two structured meshes in one program on SP2, in msec

and using Meta-Chaos is that the Chaos implementation internally requires an extra copy of the data and also an extra level of indirect data access. These extra operations are necessary to implement the correspondence between the regular mesh representation of each array element for Multiblock Parti and the pointwise representation of the same element for Chaos. These factors cause the Chaos data copy to usually cost somewhat more than the Meta-Chaos version. However, the actual communication of the data, in terms of the messages generated by all three methods, is essentially identical: all the methods use the same total number of messages and the messages are the same size.

From this experiment, we see several advantages of Meta-Chaos over trying to use a single data parallel library in a manner for which it was not designed: (1) smaller memory requirements (Meta-Chaos does not have to explicitly maintain the mapping between the regular mesh representation of an array element and the pointwise representation), (2) ease of use (no extra memory allocation, no explicit mapping between objects in the two data parallel libraries), and (3) the data copy performs better (no extra internal copy, no extra indirect access).

5.2. Interaction between two structured meshes in the same program

The first set of experiments showed that Meta-Chaos is able to copy objects distributed by different data parallel libraries however they are distributed.

In this experiment, there is one program with two regular mesh data structures distributed by Multiblock Parti, and the program copies a section of one mesh to a section of the second mesh once per time-step. This scenario would occur, for example, in a multiblock computational fluid dynamics code, where inter-block boundaries must be updated at every time-step [1]. The copy operation can be completely expressed using Multiblock Parti functions, for both building the communication schedule and moving the data. This allows us to compare both the cost of computing a schedule and moving the data with Meta-Chaos to the cost of computing the same schedule and moving the data using only one data parallel

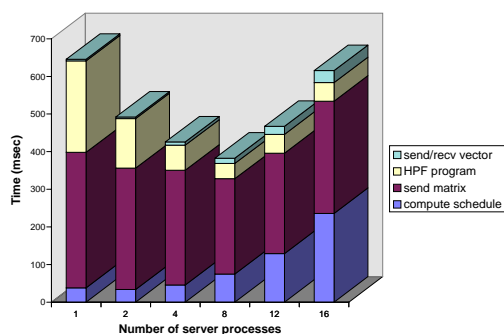


Figure 4. Total time for a sequential client. The server runs on four nodes, with up to four processes per node (at most one per processor).

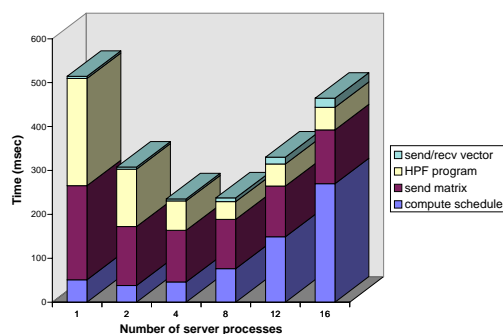


Figure 5. Total time for a four-process client running on four separate nodes. The server runs on four nodes.

library.

Table 2 shows the times to compute a schedule using Multiblock Parti, and using Meta-Chaos with both the *cooperation* and *duplication* implementations. The table also shows the time required to perform the data copy operation for all three methods. The programs was run on up to 16 processors on an IBM SP2. The two dimensional arrays of double precision floating point numbers for the meshes are each 1000x1000, and each array is distributed by blocks in each dimension across all the processors. Half the data in each array was involved in the data copy.

As was explained for the previous experiment, the time to compute the schedule with Meta-Chaos using the *cooperation* method is around twice the time required when using Multiblock Parti. Neither Multiblock Parti nor Meta-Chaos using the *duplication* method require any communication to build a communication schedule for this experiment. The overhead for building the schedule using Meta-Chaos is a little higher than for Multiblock Parti, which is not surprising since Multiblock Parti is optimized to build schedules for moving regular sections. On the other hand, the Meta-Chaos *cooperation* implementation requires some communication, since parts of the schedule are not computed on the processors that use it, so those parts must be sent to the right processors. Even though the cost of the communication for this method is not large, it still causes the schedule build to cost more than for the other two methods.

Since the data copy operations for Multiblock Parti and for both Meta-Chaos implementations are exactly the same (they all effectively generate the same schedule), the times for the data copy are essentially the same for all three methods. The only difference is that Meta-Chaos handles data copies within a processor (when parts of the source and destination mesh are on the same processor) more efficiently than does Multiblock Parti. Meta-Chaos performs a direct copy between the storage for the source and destination, while Multiblock Parti requires an intermediate buffer. This is only an issue for the two-processor case, because a large percentage of the data is copied locally, requiring no communication.

These results are encouraging because they show that the more

general Meta-Chaos library is able to generate a communication schedule with very little extra overhead compared to generating the same schedule using a special-purpose data parallel library that has been optimized to generate such schedules.

5.3. Client/server program interaction

This experiment presents the results of client/server-style program interaction. The structure of the data on the client and the server is completely managed by Meta-Chaos, meaning that neither needs to know anything about the structure of the data (e.g. whether or how it is distributed across multiple processors) in the other program. From this point of view, Meta-Chaos provides an analogue of a Unix pipe for the programmer to transfer data between the client and server programs.

To illustrate client/server interaction we have chosen a scenario in which the client uses the server as a high performance computation engine for performing a matrix vector multiply operation. Meta-Chaos is used to perform the copy operations for the matrix and the vectors, after computing the required communication schedules. These schedules are computed once and stored for reuse as needed.

We have implemented this scenario on an eight-node Digital Alpha cluster of four-processor SMPs, connected via OC-3 links to a Digital ATM Gigaswitch. The client program is a sequential Fortran program, or a parallel Fortran program parallelized using Multiblock Parti. The client program builds the matrix and multiple vectors, then sends data to the server program and receives the result vector. The server program is an HPF matrix-vector multiply program that distributes the matrix and vector across the processors, gets a matrix from the client, then repeatedly gets a vector from the client, computes the result vector and returns it to the client. The client and server are run on disjoint sets of nodes on the Alpha cluster, with each program allocated its own set of up to four nodes (16 processors). Meta-Chaos access the ATM switch via PVM, while HPF uses a high performance Digital implementation of the UDP protocol. The experiment has been performed using a 512x512 matrix of double precision floating point numbers.

Figures 4 and 5 show the times to (1) compute the schedules to

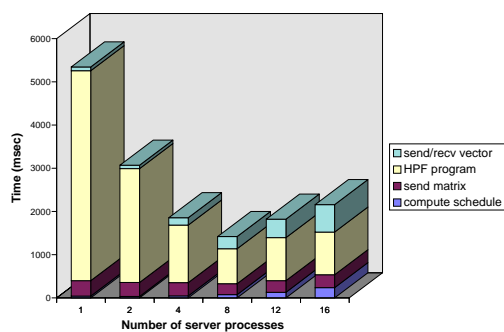


Figure 6. Total time for twenty vectors for a one-process client. The server runs on four nodes.

copy the matrix and the vectors between the client and the server, measured on the client, (2) send the matrix from the client to the server, measured on the client, (3) perform the matrix-vector multiply on the server, measured on the server, and (4) copy both the operand and the result vectors between the client and the server, computed by measuring in the client the total time to send the operand vector, compute in the server, and receive the result vector and then subtracting the time spent in the server (from measurement 3).

The figures show results for both one and four client processes (one per node). In all these experiments, the server is running on four nodes, with up to four processes per node (one per processor).

As is shown in the figures, the best performance is obtained from a server running with eight processes. This is because that configuration achieves the best balance between communication and computation. The time to compute the communication schedules decreases with increasing numbers of server processes up to four server processes, and increases thereafter because of contention for the ATM network among multiple server processes on the same node. In addition, building the schedules requires an all-to-all communication between the client and server processes, and a relatively small amount of data is sent, so adding more server processes increases the total number of messages required. The same all-to-all communication is required for copying the matrix and the vectors between the client and server. All these factors lead to the performance behavior shown, namely that, beyond eight server processes, the speedup from running the matrix-vector multiply on more server processors is offset by increased communication overhead. In addition, the HPF server program does not speed up beyond eight processors, because of increased internal communication costs in performing the matrix-vector multiply.

A more realistic determination of the benefits that can be achieved from Meta-Chaos requires performing more computation in the server, to amortize the cost of exchanging data between the client and server. Figure 6 shows the results from performing many matrix-vector multiplies using the same matrix. In that case, the communication schedules must only be computed once and the

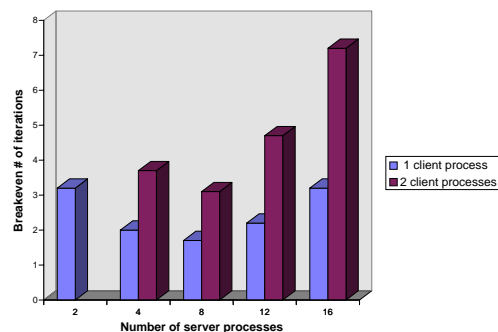


Figure 7. Break-even number of exchanged vectors, for a sequential and a two-process client, with one client process per node. The server runs on four nodes, with up to four processes per node (one per processor)

matrix is only sent once from the client to the server. The figure shows the time to compute the schedules to copy the matrix and vector between the client and the server, to send the matrix to the server, to perform the matrix-vector multiply operation and to copy both the operand and the result vectors between the client and the server for varying numbers of vectors and server processes. From the results shown in Figure 6, we can compute that a speedup of 4.5 is achieved when the server is an eight-process program, relative to performing the same computation in the client.

Figure 7 shows the number of vectors that must be multiplied by the same matrix to amortize the overhead of using a separate server program rather than computing the matrix-vector multiply within the client processes (e.g., with a library routine). From the results shown in Figure 7, we see that a sequential program could benefit greatly from using a parallel server to perform an expensive computation, using Meta-Chaos to do the communication between the programs. In this experiment, the server is not executing a particularly expensive computation, but performance gains are still possible after only a small number of matrix-vector multiply computations are done to amortize the cost of sending the matrix.

6. Related work

The software tool that provides the closest functionality to that of Meta-Chaos is HPF/MPI [6]. The HPF/MPI library extends the MPI point-to-point message passing functions to allow multiple HPF programs to communicate using MPI calls. Communication between two HPF programs using either Meta-Chaos or HPF/MPI would be written the same way, except for some syntactic differences. However, there are three main differences between Meta-Chaos and HPF/MPI. The first one is that Meta-Chaos can be easily extended to additional data parallel libraries and languages (so long as the libraries/languages provide the required interface functions), while HPF/MPI is restricted to only HPF programs. The second difference is related to the first one, in that HPF/MPI computes a communication schedule using a mechanism similar to

the *duplication* technique used in Meta-Chaos. HPF/MPI does not provide any mechanism equivalent to the Meta-Chaos *cooperation* technique for computing a schedule. For example, computing a communication schedule to transfer array elements irregularly distributed using the Chaos library using the *duplication* method would be very expensive. The third major difference in functionality between Meta-Chaos and HPF/MPI is that Meta-Chaos allows multiple data parallel libraries to communicate with one another in the *same* program. Overall, Meta-Chaos provide a superset of the functionality of HPF/MPI.

As HPF/MPI and other previous work has shown, integrating task and data parallelism can present significant advantages in both performance and ease of programming. Several research efforts have been working on the enhancement of data parallel languages to integrate data parallelism and task parallelism. Fx [17] adds compiler directives to HPF to specify task parallelism. Opus [10] is a set of HPF extensions that provides a mechanism for communication and synchronization through a shared data abstraction (SDA). Fortran M [4] extends Fortran77 for task parallel computations, and also introduces several data distribution statements. Braid [5] introduces data parallel extensions to the Mentat distributed object programming language. Integrating task and data parallelism within one language is an active area of research, but does not address the problem that there are many existing parallel codes that have been written using different data parallel libraries. The existence of many such data parallel libraries [8, 9, 11, 13, 14], and their related applications, provides strong support for the claim that no single library or language is sufficient for all potential applications. In addition, asking application programmers to write new codes, or rewrite existing codes, using only one data parallel language or library just so they can inter-operate is unlikely to succeed. Therefore a tool like Meta-Chaos is needed to allow existing parallel applications to inter-operate and also to allow new applications, written using whatever data parallel library or language the application programmer feels is appropriate, to inter-operate.

7. Conclusions and Future Plans

In this paper we have addressed the problem of interoperability between different data parallel libraries. With the mechanisms we have described, multiple libraries can exchange data in the same data parallel program or between separate data-parallel programs.

We have implemented the interoperability mechanism in a library called Meta-Chaos and have explored the behavior of the approach for several programs on two different parallel architectures. Our experimental results show that our framework-based approach can be implemented efficiently, with Meta-Chaos exhibiting low overheads, even compared to the communication mechanisms used in two specialized and optimized data parallel libraries. In addition, we exhibited the flexibility of the approach for applications using a client/server execution model.

We plan to make Meta-Chaos publicly available and encourage developers of data parallel libraries to provide the interface functions needed for Meta-Chaos to access data distributed using those libraries. We plan to apply the framework-based approach to new application areas, and are currently studying ways to incorporate distributed data parallel objects into the CORBA [15] object model, so that data parallel programs could interoperate with distributed object systems. Meta-Chaos could be used as the underlying mechanism for such an extension.

References

- [1] G. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):747–754, July 1995.
- [2] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.
- [3] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.
- [4] K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *Journal of Supercomputing Applications*, 8(2), 1994. Also available as CRPC Technical Report CRPC-TR93430.
- [5] E. West and A. Grishaw. Braid: Integrating Task and Data Parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 211–219. IEEE Computer Society Press, Feb. 1995.
- [6] I. Foster, D. Kohr, R. Krishnaiyer, and A. Choudary. Double standards: Bringing task parallelism to HPF via the message passing interface. In *Proceedings Supercomputing '96*. IEEE Computer Society Press, Nov. 1996.
- [7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manjekar, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.
- [8] U. Geuder, M. Hardtner, B. Worner, and R. Zin. Scalable execution control of grid-based scientific applications on parallel systems. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*. IEEE Computer Society Press, 1994.
- [9] W. Gropp and B. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 60–67. IEEE Computer Society Press, 1994.
- [10] M. Haines, B. Hess, P. Mehrotra, J. V. Rosendale, and H. Zima. Runtime Support for Data Parallel Tasks. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 432–439. IEEE Computer Society Press, Feb. 1995.
- [11] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Runtime and language support for compiling adaptive irregular programs. *Software-Practice and Experience*, 25(6):597–621, June 1995.
- [12] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.
- [13] S. Kohn and S. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 509–517. IEEE Computer Society Press, May 1994.
- [14] A. Lain and P. Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 820–826. IEEE Computer Society Press, Apr. 1995.
- [15] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1995.
- [16] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, 1996.
- [17] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 13–22, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.