Interoperability of Data Parallel Runtime Libraries with Meta-Chaos *

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

It is notoriously difficult to achieve good performance from complex data parallel programs running on distributed memory parallel machines. Therefore a number of runtime libraries targeted at particular application domains have been developed over the last several years. These libraries often provide capabilities that do not exist in widely available data parallel languages, such as High Performance Fortran (HPF) [14]. Examples of such data parallel runtime libraries include AMR++ and P++ [17, 20] for adaptive grid applications, Grids [9] for structured and unstructured grid problems, Multiblock Parti [1] and GMD [11] for multigrid and multiblock codes, LPARX [15] for codes using adaptive finite difference methods, Chaos [7] and PILAR [16] for unstructured grid problems, GA [18] for computational chemistry and Aztec [12], PETSc [10], ScaLAPACK [5, 6] and Syisda [3] for linear algebra operations.

Although this list is not exhaustive, it shows the amount of effort that has gone into developing data parallel runtime libraries optimized for various types of applications. However, a major

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

weakness of all these libraries is the inability to communicate with any other of the libraries in an efficient and user-friendly manner. Interoperability between such libraries is highly desirable for at least two reasons. First, interoperability allows a single application to use multiple libraries for different data structures within the same program. An example of this scenario would be a computational fluid dynamics (CFD) code that requires both a structured and unstructured grid to model the space around a surface with complex geometry. Different libraries could be used to efficiently parallelize the computations on the two grids, but some mechanism is required to transfer data between the grids at their boundaries. We present such a mechanism in this paper. Second, interoperability would allow separate programs, parallelized using different libraries, to communicate using either a client-server or peer-to-peer model. One example of a peer-to-peer model is a complex physical simulation, such as shipboard fire modeling. Such an application would require communication between the different libraries that were used to parallelize the structural mechanics code used to model the ship walls, the CFD code used to model airflow through the room with the fire, and the flame code used to provide a detailed simulation of the fire. Again, the framework described in this paper provides the mechanisms needed to perform the communication.

Many scenarios can be described for a client, running sequentially or in parallel, that could use the services of a parallel server, on the same or another (parallel) machine. Such servers could provide functionality that is not available in the client (e.g a database, or computational routines), or provide additional computational power to make the client run significantly faster. An example of this scenario is an image processing client that wants to access data from one or more satellite image database parallel servers. The servers could return all the data for a query to the client, with the client computing an output image, or the servers might also be used as computational engines to produce a partial output image, with the combination of partial output images from the various servers occurring in the client. The framework described in this paper also provides the mechanisms required for direct communication in client-server applications. We will return to a version of the client-server scenario in the experiments describe in Section 5.4.

This paper addresses the problem of allowing interoperability between various data parallel runtime libraries, and introduces a runtime library called Meta-Chaos that can be used to solve the problem. The role of Meta-Chaos is to allow efficient and transparent interoperability between data structures distributed by a user (or compiler) using multiple data parallel libraries and/or data parallel languages. The multiple data parallel libraries can be used within a single program, or can be used in multiple programs. In either case, Meta-Chaos is able to transfer data between the various data parallel libraries directly and efficiently, as will be shown from the experimental results in Section 5.

Meta-Chaos hides the distribution of data across the processors by one data parallel library from other such libraries. The key concept in Meta-Chaos for allowing the data distribution to be hidden is *virtual linearization*. Linearization is the method by which Meta-Chaos organizes data structures into a canonical form, so that any data parallel library that provides functions to order the individual elements of a data structure (e.g. an array) can communicate with any other data parallel library that also provides the functions required by Meta-Chaos. The underlying communication layer required by Meta-Chaos is a point-to-point message passing library, such as MPI [24] or PVM [8], or a vendor-specific library such as MPL for the IBM SP2.

The rest of the paper is organized as follows. Section 2 presents an example application that requires communication between two data parallel libraries in a single program, while Section 3 discusses several potential solutions to the problem of interoperating data parallel libraries. Section 4 discusses the concepts used both to implement the Meta-Chaos runtime library and to employ Meta-Chaos in an application. Section 5 presents several experiments both to quantify the overheads encountered in using Meta-Chaos and to describe the behavior of programs using Meta-Chaos to structure their computations. We conclude in Section 6.

2 A motivating example - interaction between a structured and an unstructured mesh

```
! Loop 0: timestep loop
       do time=time_start,time_stop
! Loop 1: sweep over a structured mesh
          forall ( i = 2:n1-1:1 , j = 2:n2-1:1 )
             a(i,j) = a(i,j-1)+a(i-1,j)+a(i+1,j)+a(i,j+1)
          end forall
! Loop 2: exchange boundary information
!         between the structured and unstructured mesh
          forall ( i = 1:Reg2IrregDim:1 )
             x(Reg2Irreg_Irreg(i)) = a(Reg2Irreg_Reg1(i),Reg2Irreg_Reg2(i))
          end forall
! Loop 3: sweep over an unstructured mesh
          forall ( i = 1:Nedges:1 )
             y(ia(i)) = y(ia(i)) + (x(ia(i))+x(ib(i)))/4
             y(ib(i)) = y(ib(i)) + (x(ia(i))+x(ib(i)))/4
          end forall
! Loop 4: exchange boundary information
!         between the unstructured and structured mesh
          forall ( i = 1:Reg2IrregDim:1 )
             a(Reg2Irreg_Reg1(i),Reg2Irreg_Reg2(i)) = x(Reg2Irreg_Irreg(i))
          end forall
!
       end do
```

Figure 1: A program that uses both regularly and irregularly distributed data

In computational fluid dynamics (CFD) flow solvers, different types of meshes may be used 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 HPF-like pseudo-code in Figure 1 represents a simplified version of a code that uses two interacting meshes. Loops 1 and 3 in the code show sweeps through a structured and an unstructured mesh, respectively. These loops are similar in form to loops found in structured and unstructured CFD codes. Loops 2 and 4 in the figure show the copies of data between the two meshes.

The structured mesh is represented by array a, and is regularly distributed (e.g. by blocks in each dimension) across the processors of the parallel machine. The node data in the unstructured mesh is represented by arrays x and y, which are irregularly distributed across the processors, both with the same distribution. x and y are accessed through the indirection arrays ia and ib. ia and ib are regularly distributed across the processors, and represent the edges connecting the nodes in the unstructured mesh.

The interface between the two meshes is defined by a mapping between elements of a and elements of x. The mapping is represented in Figure 1 by the arrays Reg2Irreg_Irreg, Reg2Irreg_Reg1 and Reg2Irreg_Reg2. For index i of the mapping, Reg2Irreg_Irreg(i) provides the index of x that corresponds to indexing a with Reg2Irreg_Reg1(i) in the first dimension and Reg2Irreg_Reg2(i) in the second dimension.

The techniques presented in this paper can be used to describe the mapping defined by the various Reg2Irreg arrays in the example, and also to copy the data between the a and x arrays.

This can be implemented as one data parallel program or can be implemented as two data-parallel programs interacting with one another. This last interaction can utilize a peer to peer interaction model or a client-server model.

3 Mechanisms to support interoperability

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. Such an approach requires extensive redesign and implementation effort, but should allow for a clean and efficient integrated system. However, existing runtime libraries cover only a subset of potential application domains, and it would be difficult to reach a consensus on an exhaustive set of features to provide in an all-inclusive library. Another 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. This approach would allow a data copy between two libraries to be expressed by a call to a specific function. 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 so-called 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 to implement in the Meta-Chaos runtime library. This approach gives the task of providing the required interface functions to the data parallel 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.

To summarize, we have adopted the framework-based approach for several reasons:

- It is very difficult to define an all-encompassing interface that supports all application domains efficiently.

- We expect that interactions between libraries will be relatively infrequent and restricted to simple coarse-grained operations, such as copying a large section of an array distributed by one library to a section of an array distributed by another library. This encourages the use of specialized and optimized libraries in the computation portions of an application, to provide the best possible performance.

- This approach does not preclude the use of the custom interface method. Whenever a custom interface function is not available for two libraries, the framework-based approach can be used to perform the communication.

- Adding a new data parallel library to interoperate with all existing libraries is relatively simple, since the code for no other library must be modified. All that is required is to provide the interface functions for the new library. Therefore this approach is easily extensible.

4 Meta-Chaos

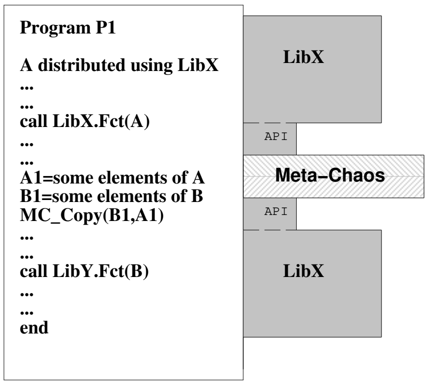


Figure 2: Meta-Chaos for communicating between two libraries within the same program

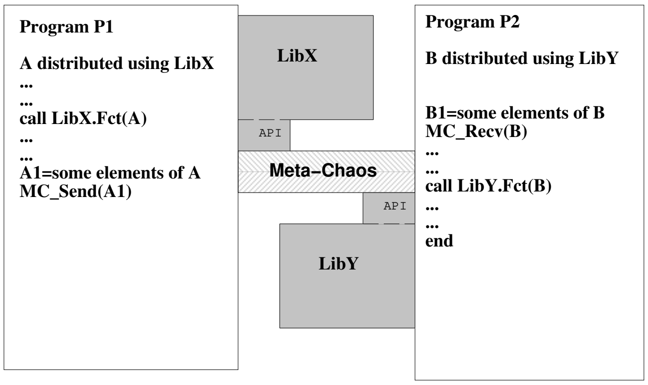


Figure 3: Meta-Chaos for communicating between libraries in two different programs

This section is divided into three parts. We first describe in detail how Meta-Chaos works, concentrating on the concept of a *virtual linearization* that completely describes the mapping between the source and destination data structures. We then discuss the Meta-Chaos library interface from the viewpoint of a user who wants to use multiple data parallel libraries through either a compiler or in a hand parallelized application, and finally provide a short example of using Meta-Chaos to copy data between two data parallel programs written in HPF.

4.1 Meta Chaos Mechanism Overview

Figures 2 and 3 provide a high-level view of interoperability between two data parallel libraries using Meta-Chaos, 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 2 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 3 copies elements of A into elements of B, but A and B belong to different programs. In either scenario, Meta-Chaos is the *glue* that binds the two libraries, and performs the copy.

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 which will 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.
5. Perform the communication using the schedule produced in step 4.

The goal of Meta-Chaos is to allow easy data parallel library interoperability. Meta-Chaos provides functions that support each of the 5 steps just described. In the following sections, we describe the mechanisms used by Meta-Chaos to specify the data elements involved in the communication (steps 1 and 2), the *virtual linearization* (step 3), and the schedule computation (step 4). Step 5 uses the schedule computed by step 4 to perform data copy, and uses system-specific transport routines (e.g. *send* and *receive* on a distributed memory parallel machine).

4.1.1 Data specification

We define a Region as a compact way to describe a group of elements in global terms for a given library. A Region is an instantiation of a Region type, which must be defined by each data parallel library.

For example, High Performance Fortran (HPF) [14] and Multiblock Parti [1, 25] utilize arrays as their main distributed data structure, therefore the Region type for them is a regularly distributed array section. Chaos [7, 13, 23] employs irregularly accessed arrays as its main distributed data structure, either through irregular data distributions or accesses through indirection arrays. For Chaos a 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 to create regions and a destructor to destroy the Regions specified for that library. The library builder also implicitly defines a *linearization* of a Region, as we will discuss further in Section 4.1.2. Depending on the needs of the data parallel library, Regions are allowed to consist of collections of arbitrarily complex objects. However, throughout this paper, we will concentrate on Regions consisting of arrays of objects of basic, language-defined types (e.g. integer, real, etc.).

Regions are gathered into an ordered group called a SetOfRegions. A mapping between source and destination data structures therefore specifies a SetOfRegions for both the source and the destination.

Figure 4 shows an array `A`. For this example, a Region for the array is a regular section. Two Regions, r_A^1 and r_A^2 are illustrated. Together they define a SetOfRegions S_A.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} & a_{19} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} & a_{29} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} & a_{39} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} & a_{49} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} & a_{59} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} & a_{69} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} & a_{79} \end{pmatrix}$$

$$r_A^1 = \begin{pmatrix} a_{25} & a_{26} & a_{27} \\ a_{35} & a_{36} & a_{37} \\ a_{45} & a_{46} & a_{47} \end{pmatrix} \qquad r_A^2 = \begin{pmatrix} a_{32} & a_{33} \\ a_{42} & a_{43} \\ a_{52} & a_{53} \\ a_{62} & a_{63} \end{pmatrix}$$

$$S_A = r_A^1\ ,\ r_A^2 = \begin{pmatrix} a_{25} & a_{26} & a_{27} \\ a_{35} & a_{36} & a_{37} \\ a_{45} & a_{46} & a_{47} \end{pmatrix}, \begin{pmatrix} a_{32} & a_{33} \\ a_{42} & a_{43} \\ a_{52} & a_{53} \\ a_{62} & a_{63} \end{pmatrix}$$

Figure 4: Regions and SetOfRegions for a distributed array A

4.1.2 Linearization

Linearization is the method by which Meta-Chaos defines the 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. For example, the linearization of a Region that is owned by a single processor would be defined by the memory order of the region. More specifically, if the Region is an array section, and the array is laid out in row major order (as for C-style arrays), then the linearization of the section is the row major ordering of the elements of the regular section. This can be extended to a data structure distributed across multiple processors by viewing the data structure as existing in the memory of a single abstract processor, and producing the linearization from the viewpoint of that memory.

In the example in Figure 4, an array A and two Regions, r_A^1 and r_A^2 were shown. The linearizations for r_A^1 and r_A^2 are illustrated in Figure 5 by $L_{r_A^1}$ and $L_{r_A^2}$, respectively.

To extend the concept of a *linearization* to a SetOfRegions, we define the linearization of a SetOfRegions as the linearization of the first Region in the set followed by the linearization of the remaining Regions.

In Figure 4, we introduced a SetOfRegions S_A that is the aggregation of the two Regions r_A^1 and r_A^2. In Figure 5, we represent the linearization of the SetOfRegions S_A by L_{S_A} .

We represent the operation of translating from the SetOfRegions S_A of A, to its linearization, L_{S_A}, by ℓ, and the inverse operation of translating from the linearization to the SetOfRegions

$$L_{r_A^1} = \boxed{a_{25} \quad a_{26} \quad a_{27} \quad a_{35} \quad a_{36} \quad a_{37} \quad a_{45} \quad a_{46} \quad a_{47}}$$

$$L_{r_A^2} = \boxed{a_{32} \quad a_{32} \quad a_{42} \quad a_{42} \quad a_{52} \quad a_{52} \quad a_{62} \quad a_{62}}$$

$$L_{S_A} = a_{25} \quad a_{26} \quad a_{27} \quad a_{35} \quad a_{36} \quad a_{37} \quad a_{45} \quad a_{46} \quad a_{47} \mid a_{32} \quad a_{33} \quad a_{42} \quad a_{43} \quad \cdots$$
$$\cdots \quad a_{52} \quad a_{53} \quad a_{62} \quad a_{63}$$

Figure 5: Linearization for Regions and SetOfRegions

as ℓ^{-1}:

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

Then moving data from the SetOfRegions S_A to the SetOfRegions S_B can be viewed as a three-phase operation:

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

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).

We show in Figure 6 a 2-dimensional array B, with 3 Regions. The three Regions are gathered into a SetOfRegions S_B. L_{S_B} is the linearization of S_B. Figure 7 shows the result of a data copy operation applied to source S_A and destination S_B.

The concept of the linearization has several important properties:

- 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.
- It 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.
- It is only an abstract, not a physical object. No space is allocated for the linearization of a SetOfRegions in the memory of either the source or destination program. Meta-Chaos transfers data directly from the source SetOfRegions to the destination SetOfRegions, never building a data structure for the linearization.

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} & b_{17} & b_{18} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} & b_{27} & b_{28} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} & b_{36} & b_{37} & b_{38} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} & \mathrm{a}_{46} & b_{47} & b_{48} \\ b_{51} & b_{52} & b_{53} & b_{54} & b_{55} & b_{56} & b_{57} & b_{58} \\ b_{61} & b_{62} & b_{63} & b_{64} & b_{65} & b_{66} & b_{67} & b_{68} \\ b_{71} & b_{72} & b_{73} & b_{74} & b_{75} & b_{76} & b_{77} & b_{78} \\ b_{81} & b_{82} & b_{83} & b_{84} & b_{85} & b_{86} & b_{87} & b_{88} \end{pmatrix}$$

$$r_B^1 = \begin{pmatrix} b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}, \; r_B^2 = \begin{pmatrix} b_{67} & b_{77} & b_{87} \end{pmatrix}, \; r_B^3 = \begin{pmatrix} b_{14} & b_{15} \\ b_{24} & b_{25} \\ b_{34} & b_{35} \\ b_{44} & b_{45} \end{pmatrix}$$

$$S_B = r_B^1, r_B^2, r_B^3$$

$$L_{S_B} = \begin{array}{|cccccc|ccc|ccccc} \hline b_{21} & b_{22} & b_{23} & b_{31} & b_{32} & b_{33} & b_{67} & b_{77} & b_{87} & b_{14} & b_{15} & b_{24} & b_{25} & \cdots \\ \hline \cdots & b_{34} & b_{35} & b_{44} & b_{45} \\ \hline \end{array}$$

Figure 6: Array B, regions r_B^1, r_B^2 and r_B^3 , setOfRegion S_B, and the linearization of S_B, L_{S_B}

4.1.3 Communication schedule computation

The communication schedule describes the data motion to be performed. Meta-Chaos uses the SetOfRegions specified by the user to determine the elements to be moved, and where to move them. Meta-Chaos 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 each element of the source SetOfRegions and the destination SetOfRegions. The high-level algorithm for computing the communication schedule is shown in Figure 8.

The 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 Meta-Chaos to perform operations such as:

- dereferencing an object in a SetOfRegions to determine the owning processor and local address, and a position in the linearization,
- manipulating the Regions defined by the library to build a linearization, and
- packing the objects of a source Region into a communication buffer, and unpacking objects from a communication buffer into a destination Region.

A major concern in designing Meta-Chaos was to require that relatively few procedures be provided by the data parallel library implementor, to ease the burden of integrating a new

$$B\text{=MC\_Copy}(..,A,S_A,B,S_B,..)$$

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & a_{32} & a_{33} & b_{16} & b_{17} & b_{18} \\ a_{25} & a_{26} & a_{27} & a_{42} & a_{43} & b_{26} & b_{27} & b_{28} \\ a_{35} & a_{36} & a_{37} & a_{52} & a_{53} & b_{36} & b_{37} & b_{38} \\ b_{41} & b_{42} & b_{43} & a_{62} & a_{63} & b_{46} & b_{47} & b_{48} \\ b_{51} & b_{52} & b_{53} & b_{54} & b_{55} & b_{56} & b_{57} & b_{58} \\ b_{61} & b_{62} & b_{63} & b_{64} & b_{65} & b_{66} & a_{45} & b_{68} \\ b_{71} & b_{72} & b_{73} & b_{74} & b_{75} & b_{76} & a_{46} & b_{78} \\ b_{81} & b_{82} & b_{83} & b_{84} & b_{85} & b_{86} & a_{47} & b_{88} \end{pmatrix}$$

Figure 7: Result of moving data from S_A to S_B

For each object i in the source SetOfRegions:

1. Determine the corresponding object belonging to the destination SetOfRegions using the source and destination linearizations.
   Call this object j.

2. Determine the owner (processor and local address) of object i, P_i, using the inquiry functions provided by the source data parallel library.

3. Determine the owner of j, P_j, using the inquiry functions provided by the destination data parallel library.

4. In the schedule for P_i, insert a send of object i to P_j

5. In the schedule for P_j, insert a receive into object j from P_i

Figure 8: Meta-Chaos schedule computation algorithm

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++ [4] 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.

4.1.4 Moving data using a communication schedule

Meta-Chaos uses the information in the communication schedule in each processor of the source data parallel library to move data into contiguous communication buffers. Similarly, Meta-Chaos uses the information in the schedule to extract data from communication buffers into the memory of each processor of 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 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 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 (the linearization), 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.

4.2 Meta-Chaos applications programmer interface (API)

An applications programmer can use Meta-Chaos to copy objects from a source distributed data structure managed by one data parallel library to a destination distributed data structure managed by another data parallel library. The distributions of the two data structures across the processors of the parallel machine or network of workstations are maintained by the two data parallel libraries.

There are four steps that an applications programmer must perform to completely specify a data transfer using Meta-Chaos:

1. specify the objects to copy from the source distributed data structure,
2. specify the objects in the destination distributed data structure that will receive the objects sent from the source,
3. compute the communication schedule to move data from the source to the destination distributed data structure, and
4. use the communication schedule to move data from the source to the destination distributed data structure.

The first two steps require the user to define the objects to be sent from the source distributed data structure and the objects to be received into at the destination. This is done using Regions, as was described in Section 4.1.1. A Meta-Chaos routine is then used to gather multiple Regions into a SetOfRegions. The applications programmer must create two SetOfRegions, one for the source and one for the destination distributed data structure.

The third step is to compute the communication schedule, both to send data from the source data structure and receive data into the destination. Meta-Chaos provides the routine to compute the schedule for the user, given the source and destination SetOfRegions. The sender SetOfRegions is mapped to the receiver SetOfRegions using the linearization (as described in Section 4.1.2).

The final step is to use the communication schedule to perform the data copy operation. Meta-Chaos provides functions for efficiently moving data using the schedule.

4.3 Example

Figure 9 illustrates the sequence of calls required to allow two HPF programs to exchange data using Meta-Chaos.

```
program source

integer , dimension(200,100) :: B
!hpf$ distribute B (block,block)
integer, dimension(2)::Rleft,Rright

! define the source array section
Rleft(1)  =50    Rleft(2)  =50
Rright(1) =100   Rright(2) =100
regionId=CreateRegion_HPF(2,
        Rleft(1),Rright(1))
src_setOfRegionId=MC_NewSetOfRegion()
MC_AddRegion2Set(RegionId,
        src_setOfRegionId)

! compute the schedule
schedId=MC_ComputeSched(HPF,
        B, src_setOfRegionId)
! move data from the source using
! the schedule schedId
call MC_DataMoveSend(schedId,B)
```

```
program destination

integer , dimension(50,60) :: A
!hpf$ distribute A (block,block)
integer, dimension(2)::Rleft,Rright

! define the destination array section
Rleft(1)  =1     Rleft(2)  =10
Rright(1) =50    Rright(2) =60
regionId=CreateRegion_HPF(2,
        Rleft(1),Rright(1))
dest_setOfRegionId=MC_NewSetOfRegion()
MC_AddRegion2Set(RegionId,
        dest_setOfRegionId)

! compute the schedule
schedId=MC_ComputeSched(HPF,
        A,dest_setOfRegionId)
! move data into the destination using
! the schedule schedId
call MC_DataMoveRecv(schedId,A)
```

Figure 9: Example of HPF inter-program communication using Meta-Chaos

Two programs are shown in Figure 9, a source program and a destination program. Each owns one HPF distributed array, and the programs use Meta-Chaos to copy an array subsection from the source to the destination. The program performs the following operation, in Fortran90 array syntax:

$$A[1:50:1, 10:60:1] = B[50:100:1, 50:100:1]$$

In the HPF programs, the user defines the source and destination array sections with the `CreateRegion_HPF` interface function provided by the implementor of HPF runtime library interface functions. The Meta-Chaos functions `MC_NewSetOfRegion` and `MC_AddRegion2Set` are used to create the SetOfRegions for the source and destination. Building the communication schedule is then performed by the call to the Meta-Chaos `MC_ComputeSched` function, which is a collective operation across the processors of both the source and destination program. The Meta-Chaos `MC_DataMove` calls then perform the communication between the source and destination, using the schedule. This is also a collective operation across both the source and destination, with the source program using a call for sending data and the destination program using a call for receiving data.

The communication schedule can be computed once and then reused, for example in an iterative computation. The communication schedule is also symmetric, meaning that it can be used to copy data either from the source program to the destination program, or to copy data from the destination to the source. The only change required would be to switch the calls to `MC_DataMoveSend` and `MC_DataMoveRecv` between the programs.

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, 5.2 and 5.3, provide a set of application scenarios that quantify the overheads associated with using Meta-Chaos. We have designed these experiments to allow a comparison with the performance of highly optimized and specialized data parallel libraries, in this case the Maryland Chaos and Multiblock Parti libraries. The second class of experiments, in Section 5.4, 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).

The first set of experiments, in Sections 5.1 and 5.2, shows two data parallel libraries exchanging data between a structured mesh (distributed by Multiblock Parti [1]) and an unstructured mesh (distributed by Chaos [7]), both within a single program and between two separate programs. These experiment are designed to quantify the overhead of using Meta-Chaos, first by comparing the cost of exchanging the data using Meta-Chaos vs. using a data parallel library (Chaos) to copy data within a single program (although Chaos is not optimized for moving regular meshes) and second by allowing the program to be split into two separate programs that exchange data using Meta-Chaos. The next experiment, in Section 5.3, describes a single program that uses two regular meshes as its main data structures. The two meshes are distributed by the same data parallel library (Multiblock Parti), so that copying data between the meshes can also be performed using only Multiblock Parti. This experiment quantifies the overhead of using Meta-Chaos compared to using the specialized and optimized Multiblock Parti library (in exactly the way the data parallel library was designed to be used). The last set of experiments, in Section 5.4, explores the interaction between two programs, one written in HPF and the other using Multiblock Parti. The two programs interact using a client/server programming model, with the HPF program acting as a computational server for the Multiblock Parti program. These experiments show the feasibility of using Meta-Chaos, on a network or in a parallel machine, for direct data transfers between a (sequential or parallel) client and data parallel programs running as data or computational servers.

5.1 Interaction between a structured and an unstructured mesh in the same program

The program for this experiment iterates over 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 by the Multiblock Parti library while the irregular mesh is distributed by the Chaos library. The algorithm is the one shown in the example from Figure 1. The algorithm has four phases: a sweep through a regular mesh, a remapping from the regular to the irregular mesh, a sweep through the irregular mesh and a remapping back from the irregular to the regular mesh. The four phases are placed inside a time-step loop, as they would be to use an iterative method for solving a partial differential equation.

For each parallel loop in the algorithm, neither the loop bounds nor the communication pattern change between iterations of the time-step loop. Therefore communication schedules can be generated before the first iteration of the time-step loop and re-used for all time-steps. To effectively parallelize the algorithm, each parallel loop is transformed into two loops, an inspector loop and an executor loop [22]. The inspector loop computes a communication schedule and is executed only once. The executor loop performs both the computation from the original loop and the communication required by the schedule, for every time-step. All three libraries (Meta-Chaos, Multiblock Parti and Chaos) separate the function of building a communication schedule from using one to copy data. Multiblock Parti and Chaos schedules are used to communicate within a distributed mesh and Meta-Chaos schedules are used to copy data between the meshes.

The data distribution, partitioning of the computation and the communication generation related to computation within a mesh are performed by calls to Multiblock Parti and Chaos functions. Copying data between the regularly distributed and the irregularly distributed meshes requires a specification of the points to be redistributed in each mesh, and also a specification of the mapping between the points in the meshes. Meta-Chaos is used to perform the copy, by specifying the Multiblock Parti SetOfRegions from which the data will be sent and the Chaos SetOfRegions that will receive the data. The mapping between the two SetOfRegions is provided by the linearization of the Regions. Meta-Chaos generates a communication schedule by inquiring about the distribution of data distributed by each of the two data parallel libraries. The schedule 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 of each the 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.

To demonstrate the efficiency of using Meta-Chaos, we consider the following factors:

- the time to compute the schedule to remap the data using Meta-Chaos compared to the time required using Chaos functions, and
- the time to remap data using the schedule generated by Meta-Chaos compared to the time required for the schedule generated by Chaos.

These comparisons allow us to show the efficiency of Meta-Chaos as compared to that of the Chaos library. Since Chaos has been highly optimized for various types of irregular communication, it should provide a good standard against which to compare the efficiency of the Meta-Chaos library.

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, using 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.

The time for the Multiblock Parti and Chaos inspector and executor loops, for the sweeps through the regular and irregular meshes, have been totaled and are presented in Table 1. These times are provided to show the cost of performing the mesh sweeps, including intra-mesh communication, for comparison to the cost of the inter-mesh communication. Table 2 shows the time required to build the communication schedule for copying data between the regular

	Number of processors			
	2	4	8	16
inspector	1533	1340	667	684
executor	91	66	65	53

Table 1: Inspector time (total) and executor time (per iteration) for regular and irregular meshes in one program on IBM SP2, in msec

and irregular meshes, using either Meta-Chaos or Chaos functions. The copy time is the time required to copy data using the schedule from the regular mesh to the irregular mesh and back, for one complete time-step.

There are two different ways to compute a schedule using Meta-Chaos; in Table 2 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). With both sets of data descriptors available, the communication schedule can be computed separately in the processors running the source and destination programs. The *duplication* method is not practical for some cases, such as when Meta-Chaos is used to communication between two separate programs and at least one of the programs does not have a compact data descriptor (e.g. a Chaos translation table, which is the same size as the data array).

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, 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 data copy using Chaos 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

		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 2: Schedule build time (total) and data copy time (per iteration) for regular and irregular meshes in one program on IBM SP2, in msec

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 3 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:

- 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 - that is done implicitly in the linearization),
- ease of use (no extra memory allocation, no explicit mapping between objects in the two data parallel libraries), and
- the data copy performs better (no extra internal copy, no extra indirect access).

5.2 Interaction between a structured and an unstructured mesh in two separate programs

The applicability of Meta-Chaos is not limited to communicating between two data parallel libraries in the same program. Meta-Chaos can also be used to communicate between data parallel libraries in two different programs. Implicitly controlling the coupling between two data parallel programs, using Meta-Chaos to perform the communication, has been explored in another recent paper [21]. To show the performance of this feature, the same algorithm that was described in the previous section has been implemented in two separate programs. The first

		Number of processors for P_{irreg}		
		2	4	8
Number of processors for P_reg	2	1350	726	396
	4	1377	738	403
	8	1381	718	398

Table 3: Time for Meta-Chaos schedule computation for 2 separate programs on IBM SP2, in msec

		Number of processors for P_{irreg}		
		2	4	8
Number of processors for P_{reg}	2	63	61	66
	4	55	33	36
	8	61	32	21

Table 4: Time for the Meta-Chaos data copy for 2 separate programs on IBM SP2, in msec per iteration

program, called P_{reg}, performs the regular mesh computation using Multiblock Parti, while the second program, called P_{irreg}, performs the irregular mesh computation. A data copy between the regular and irregular meshes must be divided into one operation for each program : a send operation from the source program and a receive operation into the destination program. In one time-step, each program acts as the source for one data copy and the destination for another copy.

The meshes are the same as in the previous example. The two programs were run on non-overlapping nodes of the 16 node IBM SP2, for up to 8 nodes for each program. The timings for building the communication schedule and copying the data for each time-step are presented in Tables 3 and 4, respectively. The Meta-Chaos *cooperation* implementation for building the schedule is used for the timings, because the *duplication* method would require transferring a Chaos translation table between the programs, which is very expensive.

Since the Meta-Chaos schedule computation is done with the *cooperation* method, most of the work is performed in P_{irreg}. Therefore increasing the number of processors for P_{reg} does not decrease the time for computing the schedules. However the time does decrease when using more processors for P_{irreg}, almost linearly for these experiments.

First, the time for the data copy is symmetric from the viewpoint of either P_{reg} or P_{irreg}. That is because in one iteration both programs are both the source and destination for the data copy. Second, the performance of the data copy operation is limited by whichever program runs on fewer processors, say P_{irreg}. For a fixed number of processors for P_{irreg}, increasing the number of processors for P_{reg} (for cases in which P_{reg} runs on more processors than P_{irreg}) does not decrease the data copy time. One processor for P_{irreg} will send and receive the same total amount of data, no matter how many processors are used by P_{reg}. However, the number of messages sent and received will increase, because the communication is all-to-all. But adding more processors to P_{reg} increases the total bandwidth available for communication on the SP2. For this experiment, these two factors effectively canceled each other out, keeping the total time constant for increasing numbers of processors for P_{reg}.

5.3 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. However, the high cost of the 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

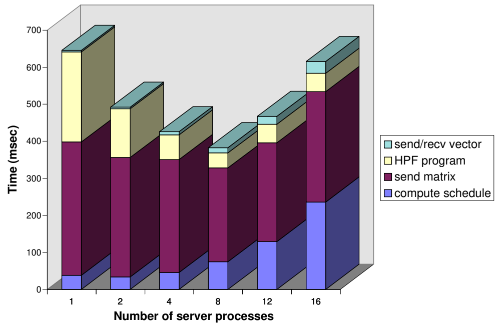
Table 5: Schedule build time (total) and data copy time (per iteration) for two structured meshes in one program on IBM SP2, in msec

`dereference` function prevented us from getting a true estimate of the overhead generated when using Meta-Chaos. We will now try to evaluate the overhead of using Meta-Chaos with another experiment.

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 [2]. 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 library.

Table 5 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 of 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



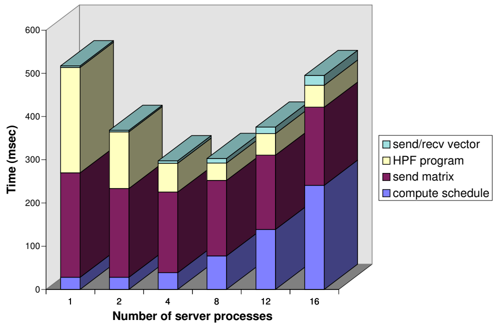


Figure 10: Total time for a sequential client. The server runs on four nodes, with up to four processes per node (at most one per processor).

Figure 11: Total time for a two-process client running on two separate nodes. The server runs on four nodes.

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 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.

5.4 Client/server program interaction

This experiment presents the results of client/server-style program interaction. The interesting part of such an experiment for Meta-Chaos is that each of the client and server programs can be run as either a sequential or a parallel program. Such a scenario is important because it shows that Meta-Chaos can be utilized to perform direct communication between a client and a server program. 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

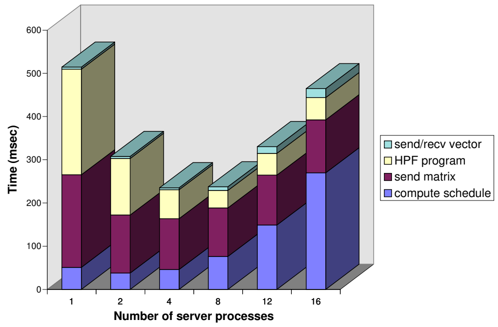


Figure 12: Total time for a four-process client running on four separate nodes. The server runs on four nodes.

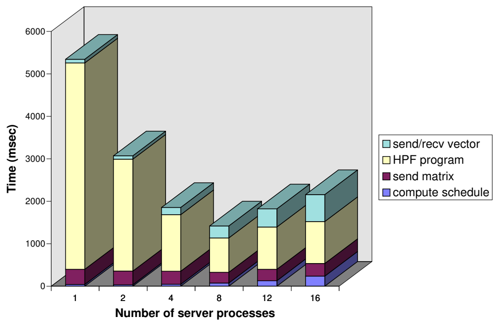


Figure 13: Total time for twenty vectors for a one-process client. The server runs on four nodes.

server as a high performance computation engine for performing a matrix operation. Specifically, the client may be either a sequential or a parallel program, and the server program performs a matrix-vector multiply, with the client sending one matrix to the server, then sending multiple vectors to the server (one at a time) and in return receiving multiple result vectors (also one at a time).

Meta-Chaos is used to perform the copy operations for the matrix and the vectors, after computing the required communication schedules. There are three schedules to compute: to copy the matrix from the client to the server, to copy the operand vector from the client to the server, and to copy the result vector from the server to the client. These schedules are computed once and stored for reuse as needed.

We have implemented this scenario on an eight-node Digital Alpha farm of four-processor SMPs, connected via OC-3 links to a Digital ATM Gigaswitch. The client program is a Fortran program and, if it is parallel, uses Multiblock Parti to distribute the matrix and vectors across the processors. The client program builds the matrix and multiple vectors, then sends data to the server program and receives results. It does none of its own computation. 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 farm, with each program allocated its own set of up to four nodes (16 processors). Meta-Chaos and Multiblock Parti access the ATM switch via PVM, while HPF uses a high performance Digital implementation of the UDP protocol.

To simplify analysis of the results, the experiment has been performed using a 512x512 matrix of double precision floating point numbers. Since the matrix is square and Meta-Chaos schedules are symmetric (i.e. they can be used to copy data either from the source program to the destination program, or to copy data from the destination to the source), only two schedules must be computed: one schedule for copying the matrix from the client to the server, and one schedule for copying a vector either way between the client and the server.

Figures 10, 11 and 12 show the times to:

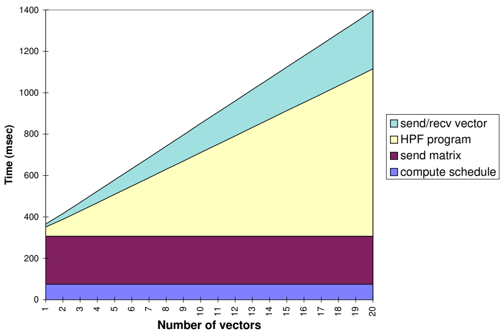


Figure 14: Total time, broken down by various functions, for varying numbers of vectors exchanged between the client and server. The client runs sequentially and the server is an eight-process program running on four nodes (using two processors per node).

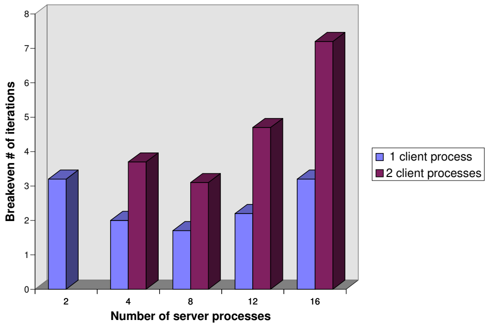


Figure 15: Break-even number of exchanged vectors, for a sequential and a two-process client, with one client process per node. The server runs on four nodes, with up to four processes per node (one per processor)

- compute the schedules to copy the matrix and the vectors between the client and the server, measured on the client,
- the time to send the matrix from the client to the server, measured on the client,
- the time to perform the matrix-vector multiply on the server, measured on the server, and
- the time to 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 subtracting the time spent in the server (from the previous measurement).

To smooth out variations in the timings, the programs were run 100 times, and the average measured values were used. The three figures are for varying numbers of client processes, up to four (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. Figures 13 and 14 show the results of performing many matrix-vector multiplies using the same matrix. In that case, the communication schedules must only be computed once and the matrix is only sent once from the client to the server. The figures show 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. Figure 13 shows these times for twenty matrix-vector multiplies, when the client is a sequential program and the server runs on four nodes (up to 16 processors). Figure 14 shows total time as a function of the number of matrix-vector multiplies when the client is a sequential program and the server is an eight-process program running on four nodes (the best case for the HPF server). From the results shown in Figure 13, 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 14 emphasizes the point that, in many cases, much of the overhead from using a computation server (the time to compute schedules and to send the matrix) can be amortized over multiple computations.

Figure 15 provides another view of the results from Figures 10 and 11. The figure 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). For example, when the client is running on only one processor, and when the server is running on 4 processes, the cost of computing the schedule and sending the matrix and vectors is amortized after about two matrix-vector multiply operations. This means that, for this configuration of client and server processes, if more than two vectors will be multiplied by the same matrix, it is faster to use Meta-Chaos to copy data between the client and server and perform the computation in the HPF server program, rather than do the computation in the client. No result is shown for the two-process client, two-process server, because in that case the communication overhead would never be amortized. The best break-even points shown are obtained for an eight-process server, which is not surprising since that configuration provides the best overall server performance.

From the results shown in Figure 15, we see that a sequential (or parallel) 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 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 introduced the concept of a *virtual linearization* that defines a canonical form for distributed data structures. Using this canonical form, we can build software to copy data structures between data parallel libraries.

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, and good performance, 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 [19] 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] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. In *Proceedings Supercomputing '93*, pages 578–587. IEEE Computer Society Press, November 1993.

[3] C. Bischof, S. Huss-Lederman, X. Sun, A. Tsao, and T. Turnbull. Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach. In *Proceedings of Scalable High Performance Computing Conference*, 1994.

[4] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[5] J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. The design and implementation of the ScaLAPACK, LU, QR, and Cholesky factorization routines. *Scientific Programming*, 1995.

[6] Jaeyoung Choi, David W. Walker, and Jack J. Dongarra. The design of scalable software libraries for distributed memory concurrent computers. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 792–799. IEEE Computer Society Press, April 1994.

[7] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[8] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[9] 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.

[10] 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.

[11] Rolf Hempel and Hubert Ritzdorf. The GMD communications library for grid-oriented problems. Technical Report 589, GMD, November 1991.

[12] S. Hutchinson, J. Shadid, and R. Tuminaro. Aztec user's guide, version 1.0. Technical Report SAND95-1559, Sandia National Laboratories, Oct 1995.

[13] Yuan-Shin Hwang, Bongki Moon, Shamik D. Sharma, Ravi Ponnusamy, Raja Das, and Joel H. Saltz. Runtime and language support for compiling adaptive irregular programs. *Software–Practice and Experience*, 25(6):597–621, June 1995.

[14] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[15] S.R. Kohn and S.B. 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.

[16] Antonio Lain and Prithviraj Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 820–826. IEEE Computer Society Press, April 1995.

[17] M. Lemke and D. Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independant development of structured grid applications. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing (CONPAR'92 - VAPP V)*, 1992.

[18] J. Nieplocha, R. Harrison, and R. Littlefield. Global arrays: a portable shared-memory programming model for distributed memory computers. In *Proceedings Supercomputing '94*, 1994.

[19] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1995.

[20] R. Parsons. A++/P++ array classes for architecture independant finite difference computations. In *Proceedings of OONSKI'94, The Object-Oriented Numerics Conference*, April 1994.

[21] M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman, and J. Saltz. Runtime coupling of data-parallel programs. In *Proceedings of the 1996 International Conference on Supercomputing*. ACM Press, May 1996. To appear.

[22] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[23] Joel Saltz, Ravi Ponnusamy, Shamik D. Sharma, Bongki Moon, Yuan-Shin Hwang, Mustafa Uysal, and Raja Das. A manual for the CHAOS runtime library. Technical Report CS-TR-3437 and UMIACS-TR-95-34, University of Maryland, Department of Computer Science and UMIACS, March 1995.

[24] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, 1996.

[25] Alan Sussman, Gagan Agrawal, and Joel Saltz. A manual for the Multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1 and UMIACS-TR-93-36.1, University of Maryland, Department of Computer Science and UMIACS, December 1993.