

Efficient Communication Between Parallel Programs with InterComm*

Jae-Yong Lee and Alan Sussman
UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742
{jylee, als}@cs.umd.edu

Abstract

We present the design and implementation of *InterComm*, a framework to couple parallel components that enables efficient communication in the presence of complex data distributions within a coupled application. Multiple parallel libraries and languages may be used in different modules of a single application. The ability to couple such modules is required in many emerging application areas, such as complex simulations that model physical phenomena at multiple scales and resolutions, and remote sensing image data analysis applications.

The complexity of the communication algorithms is highly dependent on the distribution of data across the processes in a distributed memory parallel program. We classify the distributions into two types - one that represents a data distribution in a compact way so that the distribution information can be replicated, and one that explicitly describes the location of each data element, so can be very large, requiring that the distribution information be distributed across processes as is the data.

InterComm builds on our previous work on the *Meta-Chaos* program coupling framework. In that work, we showed that the steps required to perform the data exchanges include locating the data to be transferred within the local memories of each program, generating communication schedules (the patterns of

interprocessor communication) for all processes, and transferring the data using the schedules. In this paper we describe the new algorithms we have developed, and show how those algorithms greatly improve on the *Meta-Chaos* algorithms by reducing the high cost of building the communication schedules. We present experimental results showing the performance of various algorithmic tradeoffs, and also compare against the original *Meta-Chaos* implementation.

1 Introduction

Distributed memory parallel programs take advantage of multiple processors to speed up execution of an application. Since data is distributed across the multiple processes running the application, the program must perform interprocessor communication to share data elements among the processes. Communications between processes can be performed using low level communication libraries such as MPI [18] or PVM [9], or by higher level data parallel libraries or languages that encapsulate communication operations into higher level abstractions, such as the *Chaos* [17] library for applications with irregular distributions of data across processes or *Global Arrays* [16] or *KeLP* [8] for applications with regular block distributions. However, in many modern, complex applications, multiple parallel and/or sequential programs may be required to cooperate to solve a complex problem, such as a multi-scale, multi-resolution physical simulation [14]. To date, few communication mechanisms have been developed to support fast, easy-to-use, direct communication between parallel programs. Such communication is not

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #ACI-9619020 (UC Subcontract #10152408), and #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B517095, and NASA under Grants #NAG5-11994 and #NAG5-12652.

straightforward because each program (a *component*) may use a different programming language or data parallel runtime library.

Suppose that we have two parallel programs that cooperate to solve a complex problem, and they have distributed data structures *ds1* and *ds2*, respectively. To transfer a set of data elements in *ds1* to a set of elements in *ds2*, all processes of the sender program and of the receiver program may need to be involved in the data transfer, because data elements in *ds1* and *ds2* may be distributed across the processes. Thus, each sender process must determine which data elements of *ds1* it should send to the various receiver processes. Similarly, each receiver process must determine which data elements should be written for each message received from a sender process. This implies that each process in the sender or the receiver program must acquire information about the data distribution across processes of both *ds1* and *ds2*.

InterComm is a runtime library that achieves direct data transfers between data structures managed by multiple data parallel languages and libraries in different programs. Such programs include those that directly use a low-level message-passing library, such as MPI. Each program does not need to know in advance (i.e. before a data transfer is desired) any information about the program on the other side of the data transfer. All required information for the transfer is computed by InterComm at runtime. As was already described, such a data transfer requires that all processes of the sender and receiver programs locate data elements involved in the data transfer and that a mapping be specified between the data elements in the two data structures. Using the data distribution and mapping information already described, InterComm generates all the information required to execute direct data transfers between the processes in the sender program and the receiver program (a customized all-to-all communication pattern [18]), and stores the information in a *communication schedule* [6].

In this paper, we describe in Section 2 the algorithms we have developed to efficiently build communication schedules for data transfers on different classes of distributed data structures. We describe various choices that can be made in the algorithm designs, and provide experimental results that indicate which variations work best under different scenarios in Sec-

tion 3. We summarize our results and describe future directions for this work in Section 4.

1.1 Prior related work

There have been several efforts to model and manage parallel data structures and provide support for coupling of parallel applications that use those data structures. While some of them provide similar methods to distribute parallel data structures and represent the data distribution, they employ various strategies to transfer such distributed data between application components.

Parallel Application Work Space (PAWS) [3, 12] provides the ability to share data structures between parallel applications. PAWS supports scalar values and parallel multi-dimensional array data structures. An application defines a *global domain* to provide each process with a *global* view of a data structure across all processes. The global domain is divided into *subdomains*, with each subdomain assigned to one process, representing a *local* view of a part of the data structure. The layout (*representation* in PAWS) of a data structure consists of global and sub-domains. Since the shape of a PAWS sub-domain can be arbitrarily defined by an application, multi-dimensional arrays in PAWS can be partitioned and distributed in a completely general way. A PAWS controller is a process that links applications and parallel data structures in the applications. A PAWS application registers itself as an active application with the PAWS controller when it starts execution. The application also registers the data structures that it will share with other applications. To transfer data elements between data structures of two applications, the PAWS controller establishes a connection between those data structures using information in its registry, and uses the parallel layout of both data structures to compute communication schedules for the data transfer. On the other hand, InterComm communication schedules are generated directly in the processes of the parallel applications without a separate controller process, allowing schedules to be computed in parallel.

Collaborative User Migration, User Library for Visualization and Steering (CUMULVS) [10] is a middleware library that facilitates the remote visualization and steering of parallel applications, and sup-

ports sharing parallel data structures between programs. Although it supports multi-dimensional arrays like PAWS, arrays cannot be distributed in a fully general way. A multi-dimensional array is partitioned into chunks in each dimension (as in High Performance Fortran (HPF) [13]), or each data element is explicitly distributed (assigned to a process) by the application. In addition, the application programmer must export a topology for processes that represent the ownership of each data chunk. A receiver program (a *visualizer*) in CUMULVS is not a parallel program. It specifies the data it requires in a request that is sent to the parallel sender program. After receiving the request, the sender program generates a sequence of connection calls to transfer the data.

Rocom [11] is an object-oriented software framework for high performance parallel rocket simulation. Multiple physics modules have been developed to model various parts of the overall problem to build a comprehensive simulation system. A physics module builds distributed objects (data and functions) called *windows* and registers them in Rocom so that other modules can share them with the permission of the owner module. A window may be partitioned into multiple *panes* for parallelism, and each process in a module may have multiple panes. For example, if a window has a multi-dimensional array as its data attribute, the array can be partitioned into subarrays, each of which can be a pane.

The Common Component Architecture (CCA) Forum [2, 4] has been developing a set of common interfaces to provide interoperability among high performance application components. The CCA MxN working group [5] has designed interfaces to transfer data elements between parallel components running with different numbers of processes in each parallel component (hence MxN). InterComm can be used as the runtime support for a general implementation of the CCA MxN services.

Model Coupling Toolkit (MCT) [15] is a system that has been developed for the Earth Systems Modeling Framework (ESMF) [7]. ESMF has developed various earth systems simulation components and a *flux coupler* component. The flux coupler serves to transfer data between the physics simulations component using the MCT functionality. In MCT, a *globalSegmentMap* is defined to describe the distribution

of a data structure across processes. The *globalSegmentMap* describes each continuous chunk of memory for the data structure in each process. Using *globalSegmentMaps*, MCT can generate a *router* – a communication scheduler that tells processes how to transfer data elements between a simulation component and the flux coupler. Therefore, all data transfers between two physics components are executed through the flux coupler.

Meta-Chaos [6] is a meta-library that interacts with data parallel libraries and languages to achieve direct data transfer between different parallel data structures. To move data between data structures in different applications, Meta-Chaos locates data elements to be transferred in both the sender and receiver processes and then generates communication schedules. Since data is distributed across process, Meta-Chaos requires communication among processes to determine which process owns data elements and where the data elements are located in the processes. In other words, each process needs to communicate with all other processes to obtain data distribution information. *Dereference* functions are used to request such information, and must be provided for each data parallel library or language. Meta-Chaos spends most its time in building communication schedules in the dereference function. In this paper, we present efficient algorithms that generate communication schedules by directly using data distribution information supplied by each application component via an InterComm API, without using dereference functions. Therefore, InterComm can be used with any parallel program that can describe its data distribution, including explicit message passing programs using, for example, MPI. We will also show, in Section 3, that the InterComm algorithms perform much better in absolute terms than those in Meta-Chaos, and also scale better.

2 Scheduling Communication between Parallel Programs

In this section we describe algorithms for generating communication schedules that completely describe the pattern of communication between the processes in a source program and the processes in a destination program. More specifically, the schedule for each process in the source program specifies the data elements

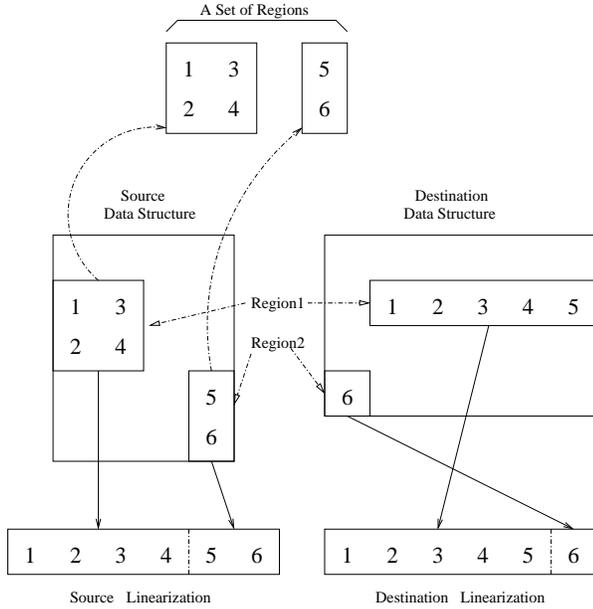


Figure 1. Sets of regions and linearization

to send from that process and which processes in the receiver to send them to. Similarly, the schedule for each process in the receiver program specifies which data elements will be received into, and which sender processes will send them.

2.1 Background

Application programmers must be able to specify the data elements that will participate in a data transfer. Many common data structures allow specifying the set of data elements to be transferred in a compact way (e.g., for an array, a sub-array requires specifying two corners of the sub-array). However, in the worst case the set can be enumerated. We call this set a *region* [6]. InterComm provides several methods to describe a region, so that application programs can easily specify the data elements that will participate in a data transfer. In this paper, we concentrate on multi-dimensional array data structures, since those are the only ones currently supported by InterComm. Since a single region is not always adequate to describe the data to be moved, multiple regions can be gathered into a set of regions. Figure 1 shows an example of a set of regions. As shown, the regions do not have to be the same in the source and destination programs.

Linearization [6] provides an implicit mapping between data elements in the source set of regions and

the destination set of regions, so that the mapping does not have to be explicitly specified for each data transfer. Data elements in a set of regions are *flattened* into an abstract linearized space. By linearizing both the source and destination sets of regions, InterComm can compute a total ordering for the data elements in each of the set of regions, providing an implicit one-to-one mapping between the source and destination linearizations. Figure 1 shows an example of how linearization works for two-dimensional arrays laid out in memory in column major order (i.e. Fortran-style).

Data parallel programs employ *data descriptors* that describe the global distribution of data structures across the processes executing the program. We say that a process in a program *owns* a data element if the data descriptor says that the local memory for that element is located in the process. InterComm supports data descriptors that fall into two classes, both representing partitionings of multi-dimensional arrays across the processes in a parallel program. One class of descriptors supports a *block* array data distribution, which can be represented in a compact, essentially implicit way. We will take advantage of this descriptor being compact, so that it can be replicated easily and cheaply over multiple processes. The other kind of descriptor supports an irregular, explicit distribution of the elements in an array, and can also be easily extended to non-array data structures. Such a descriptor cannot be described in a compact way because the descriptor must enumerate all the data elements, and therefore its size is proportional to the size of the data structure. Such a *non-compact* descriptor explicitly represents the distribution of the data elements, describing the process that owns each data element and the memory location of the element in that process. Replicating a non-compact descriptor across processes requires a large amount of memory for a large array, therefore such a descriptor must be partitioned over the processes in a parallel program [17].

In the communication schedule building algorithms, three naming schemes, or *views*, of a data element are used. A *global view* describes the location of a data element in the entire data structure. In this *global reference space*, for a distributed array data structure a data element is named with a global index in each dimension of the array. The second naming scheme is a *local view*, in which a given process ad-

dresses data elements within its own address space. In this *local reference space*, for a distributed array data elements are also named with indexes in each array dimension, but only for the part of the array owned by a given process. The last naming scheme is a *linearization view*. Since linearization determines a total ordering of the data elements specified by a set of regions, data elements to be transferred may be named by their indices in a *linearization space*.

Data descriptors provide the information necessary to translate data elements named in a global reference space into names in a local reference space (and vice versa). In InterComm, regions are specified in the global reference space. Data elements in a set of regions can be mapped into the linearization space using the region specifications and the data descriptor. Therefore we can also translate local names into the linearization space (and vice versa). The InterComm algorithms translate names across all three views to produce communication schedules from data descriptors and region specifications in both a source and destination program.

2.2 A general schedule building algorithm

We address two types of environments for executing parallel programs. One is a local cluster or distributed memory parallel machine environment and the other is a heterogeneous wide-area Grid environment. Message passing is usually much faster within a cluster than across a wide area network (WAN). The algorithms we describe to generate communication schedules provide several options that can be used depending on the computation environments of the communicating programs. For example, the algorithm can attempt to reduce the number of expensive messages across the WAN in a Grid environment, perhaps at the expense of additional message passing within a cluster.

Algorithm 1 shows the five high-level steps needed to transfer data between two parallel programs. These steps will be executed in each of the processes of either or both of the source and destination programs (i.e. communication schedule building is a collective operation [18] across the processes in both the source and destination programs). The first four steps in the algorithm compute communication schedules needed to perform the data transfer in the last step. For steps

Algorithm 1

General steps to transfer data between data structures in two parallel programs

- 1: Retrieve distribution information about the source data structure
 - 2: Retrieve distribution information about the destination data structure
 - 3: Compute partial communication schedules
 - 4: Transfer the schedules to the processes that need them
 - 5: Transfer data elements using the schedules
-

1 and 2, each process needs to collect distribution information for all the data elements for which it will be responsible for computing schedules. In step 3, each process generates partial communication schedules using information about data distributions and data specifications (a set of regions) for two data structures (one local and the other remote). In step 4 each process transfers the computed schedules to the processes that will use them for the data transfer. Finally, the two programs can transfer data using the communication schedules. Some of the steps do not have to be performed in some of the algorithms, depending on how the workload for building the schedules is assigned to the processes. For example, processes do not need to send computed schedules to other processes if they compute only schedules for data elements that they own. We describe three specific algorithms for computing communication schedules, based on the types of the source and destination data descriptors involved in the data transfer, each of which is designed to optimize performance for that case:

- two compact data descriptors,
- one compact and one non-compact data descriptor, and
- two non-compact data descriptors.

The following sections describe the algorithms for each case in more detail. For each case, we discuss not only the algorithms, but also workload balancing problems and issues related to building schedules in a wide-area environment.

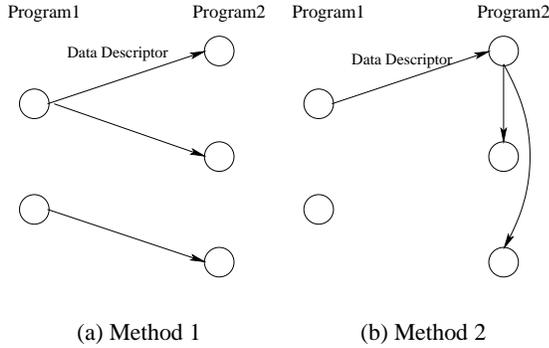


Figure 2. Two methods for exchanging information

2.3 Two compact data descriptors

A compact descriptor for a data structure can be cheaply replicated across all the processes that participate in building the communication schedule because of its small size. With a compact descriptor, all processes in a program have access to complete data distribution information for any such data structure. The only additional information a process needs to compute a schedule (or a part of a schedule – a *partial schedule*) for a given data transfer are the data specifications for the source and destination (the sets of regions), and the distribution information for the data structure managed by the other program (the compact data descriptor of the other side). The following sections describe both how processes obtain that information and how the schedule building algorithms use that information.

2.3.1 Exchanging compact data descriptors and data specifications

Some processes will be assigned the responsibility for computing parts of the communication schedules for a data transfer, as described in Section 2.3.2. Because both the compact data descriptors and the data specifications for the source (sending program) and destination (receiving program) for the transfer can be replicated, since the replicated information is small, the processes in the two programs can exchange the information in an inexpensive way via point-to-point message passing and/or broadcast operations. Since all the processes sending information can send the replicated

information, and all the processes receiving information require the same information, we have (at least) two options for how to send the information. Figure 2 shows two methods. In Figure 2(a), each sending process sends to the information a disjoint set of receiving processes. Suppose we have m sending processes and n receiving processes. Each sending process sends the information to a disjoint set of $\lfloor \frac{n}{m} \rfloor$ or $\lfloor \frac{n}{m} \rfloor + 1$ receiving processes. With the number of processes in each program and a process ID available at runtime, each process can determine which processes in the other program it has to send information to and receive information from. Although this method requires n messages to be transmitted between the two programs, it is efficient because all the sending processes can send messages at the same time, and also order the messages to minimize network contention at the receiving processes. The second method, shown in Figure 2(b), has a *representative* sender process send the information to a representative receiving process. The representative receiver process broadcasts the information to all other receiving processes. Although this approach takes two steps, this method can reduce network traffic, and could perform better than the first method when two parallel programs run on different clusters with a wide-area network connection between them. In that environment, the second method requires only one expensive message across the WAN and one broadcast within a cluster, while the first method requires n expensive messages between clusters.

2.3.2 Responsibility for computing schedules

Which processes generate communication schedules? Since any process can generate schedules from information about the source and destination data structures, there are (at least) two options for specifying which program will generate the schedules. The first option is to have both sets of source and destination processes compute schedules. In this case, all processes of both programs need to acquire information about the data descriptors and data specifications from processes of the other program. In the second option, either the source (or destination) set of processes computes communication schedules for the processes in both programs, and sends the computed schedules to the processes in the destination (source) program.

The processes in the destination (source) program only send the data descriptor and data specification information to the processes in the program computing the schedule. *Responsible* processes compute communication schedules from information about both source and destination data structures while *non-responsible* processes send the required information to responsible processes and in return receive communication schedules that have been computed in the responsible processes.

What part of the schedules does each process compute? Since any process can obtain complete information for computing schedules after receiving a data descriptor and data specification from the other program, any process can compute any part of the schedule. For load balancing in computing the schedules, we must look at the linearizations for both the source and destination data structures, which implicitly provides the mapping between elements in the two data structures. We can partition the linearization space into as many sets as there are processes that will compute the schedule (the processes in one or both of the programs, as described above), and have each process compute the schedule for one set. This approach will balance the workload evenly. However, this method requires a collective communication across all processes in both programs, to send the computed schedules to all other processes in the program (a process may compute a partial schedule for every other process, depending on the source and destination data descriptors and data specifications). However, a process does not need to send computed schedules to other processes in the same program, if it only computes a schedule for data elements that it owns. However, we cannot then determine how well the workload will be balanced, because each process may own different numbers of data elements from the data specification. Although this second approach may not achieve perfect workload balance, the collective communication for sending schedules to other processes in the first approach can be expensive. In our current implementation, each process that participates in the schedule computation computes schedules for the data that it owns, so it does not need to send partial schedules to other processes in the same program. However, each such process must still send partial schedules to the processes in the other program, if only one of the two

programs performs the entire schedule computation.

2.3.3 Schedule Generation Algorithm

In this section, we describe how to compute schedules from two compact data descriptors and two data specifications for the source and destination data structures. The algorithm runs in each process of both the source and destination program, but parts of the algorithm only execute in the program that does the schedule computations (if only one program is computing the schedules).

Algorithm 2 Computing schedules with two compact data descriptors and two sets of regions

- 1: Send a data descriptor and a set of regions for the data structure in this program (the *local* data structure) to one or more processes in the other program, if necessary
 - 2: Receive a data descriptor and a set of regions for the data structure in the other program (the *remote* data structure), from a process in the other program
 - 3: Compute the intersection I between locally owned data elements and the regions specified for the local data structure
 - 4: Map the data elements in I into the linearization for the local data structure
 - 5: Compute the intersections J_i between the data owned by each remote process i and the set of regions for the remote data structure
 - 6: For each J_i , map it into the linearization for the remote data structure
 - 7: For each remote process i , compute the intersection of the linearizations of I and J_i
 - 8: For each pair (I, J_i) , translate each data element name in the intersection from the global reference space into the local reference space
 - 9: For each pair (I, J_i) , generate entries into the communication schedules for each contiguous piece in the intersections of their linearizations (one entry for the local elements, and one entry for the corresponding remote elements)
 - 10: Send the communication schedules to processes in the other program, if necessary
-

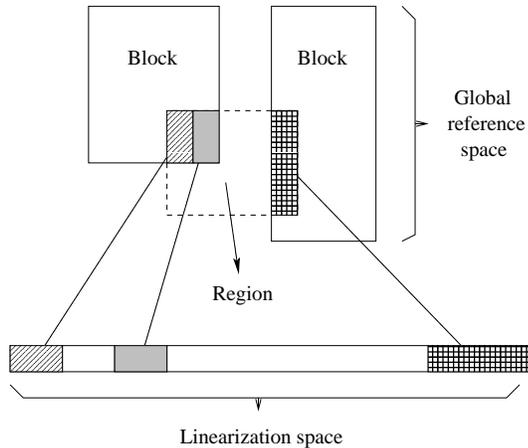


Figure 3. Mapping intersections of sets of data elements and regions into the linearization space

Algorithm 2 shows the steps needed to generate communication schedules with two compact data descriptors. The steps in this algorithm correspond to the first four steps in Algorithm 1. Each process that is responsible for building parts of the schedules receives the data distribution and data specification information for the remote data structure, as was described in Section 2.3.1 (lines 1 and 2 in Algorithm 2). In line 3, each responsible process computes the intersections of the regions specified for the data structure in its program with the data elements that it owns in the local data structure, because each process computes communication schedules only for the data elements that it owns. For the remote data structures, however, the process does not know which remote processes own data elements that correspond to the locally owned elements. Therefore each process computes the intersections of the regions for the remote data structure with the data elements owned by each process in the remote data structure, in line 5. As was discussed in Section 2.1, the linearization provides a total ordering of the data elements to be transferred. By mapping the intersections of the regions and data elements owned by the local and remote processes into the linearization space, the algorithm can match data elements in each responsible process with the corresponding data elements in the remote processes (lines 4 and 6). Figure 3 shows how the intersections of the regions and the data elements owned by a process can be mapped

into the linearization space. In this example, it is assumed that data elements in an array are stored in column major order. The algorithm then computes the intersections of the linearizations for the local and remote data structures for each remote process, in line 7.

Although the algorithm determines how data elements in the local and remote data structures match through the linearizations, the data elements must be specified in the communication schedules in the local reference space (a process ID, local address pair). The algorithm must therefore translate global references into local references using the information in the compact data descriptor (line 8). Note that a set of contiguous data elements in the intersection of the local and remote linearizations is contiguous in the memory layouts of both the local and the remote data structures. Finally, the algorithm generates the communication schedule entries for each set of contiguous data elements in the intersection of the local and remote linearizations (line 9). If processes in both programs compute the schedules, then an communication entry is only for the local references, specifying which local data elements will be transferred, and to/from which remote processes. If only processes in one program compute the schedules, the processes in that program must send schedules back to the processes in the other program (line 10), as described in Section 2.3.2. In this case, schedule entries must be made for both the local process and the corresponding remote process for a matched pair of data elements, using the local views for those processes.

2.4 One compact and one non-compact data descriptor

For this case, InterComm could exchange data descriptors as in the compact-compact case described in Section 2.3.1. However, it may not be efficient to send a complete non-compact data descriptor because the descriptor can be very large. We describe two algorithms, differing based on where schedules are computed. One algorithm computes schedules using the processes in the program that employs a non-compact data descriptor for its data structure. The other algorithm uses processes in both programs. The following sections describe the details of the algorithms and dis-

cuss their performance with respect to the workload balancing and the communication cost in Grid environment.

2.4.1 Responsibility for computing schedules

Which processes generate communication schedules? There are two algorithms, depending on which processes are responsible for computing communication schedules. The first algorithm has the program with a non-compact data descriptor compute schedules. One program obtains both data descriptors, computes communication schedules for both programs and sends schedules back to the other program. Since only one of the data descriptors must be sent with that method, an algorithm using that method does not have to send the non-compact data descriptor. The algorithm sends the compact data descriptor to all processes in the program that has the non-compact data descriptor, and those processes compute the schedules for processes in both programs. The schedules computed in the non-compact descriptor processes must be sent back to the compact descriptor processes. With this algorithm, InterComm can avoid transferring a non-compact data descriptor between programs. However, the workload for building the schedule will be imbalanced since the program with a compact data descriptor is idle while the program with a non-compact data descriptor computes schedules.

The second algorithm has both programs compute schedules. For the program with a compact descriptor to compute schedules, it requires the information from the non-compact data descriptor in the other program. Therefore, processes in the program with the non-compact data descriptor must transfer the information to processes in the program with the compact data descriptor. To minimize information to be transferred between programs, InterComm extracts distribution information for the data elements involved in the data transfer in the program with the non-compact descriptor, and sends only that information to the proper processes in the program with the compact descriptor. Therefore, InterComm requires a proportional amount of data distribution information about the non-compact data descriptor to be transferred between processes. This procedure may require that all processes with a non-compact data descriptor send information to all

processes with a compact data descriptor. Although the algorithm requires more communication between two programs than the first algorithm, it employs the processes in both programs to compute schedules, to balance the workload.

As will be seen from the experimental result in Section 3, the second algorithm performs better when both programs are run on the same cluster or on machines with a local area network connection, because both programs participate in computing schedules and the cost to transfer part of a non-compact data descriptor is not very expensive. However, the second algorithm is worse than the first if the two programs are executed on machines connected across a wide-area network (e.g., each program runs on a different cluster, with the clusters connected via a WAN) because of the high cost to transfer part of the non-compact data descriptor between the two programs.

With either algorithm, the size of the schedules to be sent between programs can be much larger than for the two compact data descriptor case in Section 2.3, because the algorithms must generate a separate schedule entry for each individual data element to be transferred, because of the explicit nature of the non-compact data descriptor. Schedule entries for the compact-compact case can be aggregated to compactly describe sets of data elements, making those schedules smaller. More specifically, the total size of all the schedules for the compact/non-compact case is proportional to the size of the data to be transferred. To transfer the compact data descriptor and the corresponding set of regions, the algorithm can use either of the two options shown in Figure 2.

What part of the schedules does each process compute? We first discuss the algorithm that has only the program with a non-compact data descriptor compute schedules. To compute schedules, a process must obtain information about the data distributions for corresponding parts of the the data structures in both programs. For the two compact descriptor algorithm described in Section 2.3.2, each process computes schedules for the data that it owns. That can be done because each process obtains complete data distribution information for the data structures in both programs. However, a non-compact data descriptor is not replicated across processes, instead it is partitioned across the processes running its program. So each process

in the non-compact descriptor program contains partial distribution information about its data structure. One option for assigning the workload is for a process in the non-compact descriptor program to compute schedules for the data elements for which it has distribution information. This approach may cause poor load balance if the distribution information about the data to be transferred is not uniformly partitioned across the processes. Moreover, this method requires a collective communication across the non-compact descriptor processes, to send the schedules to the processes that own the data, as well as a collective communication between the non-compact and compact descriptor programs to send the computed schedules to the compact descriptor program. An alternative approach is for a process in the non-compact descriptor program to compute schedules for the data that process owns. This approach eliminates the collective communication across the non-compact descriptor processes to send schedules, but requires a collective communication to collect the distribution information about the data elements a process owns onto that process. This second approach may also cause poor load balance, because each non-compact descriptor process may own different numbers of data elements that are involved in the data transfer. We have therefore chosen a third alternative to balance the workload for computing schedules across processes. In this option, all non-compact descriptor processes compute schedules for the same number of data elements, by assigning a contiguous range in the linearization space to each process. The algorithm in which the processes in both programs compute schedules is similar to this option, with the only difference being that the linearization space is partitioned across the processes in both programs. We explain the details of the algorithm in the next section.

2.4.2 Schedule Generation Algorithm

In this section, we describe how to compute schedules from one compact and one non-compact data descriptor, and two data specifications for the source and destination data structures. The algorithm runs in each process of both the source and destination program, but parts of the algorithm only execute in the program that does the schedule computations (if only one pro-

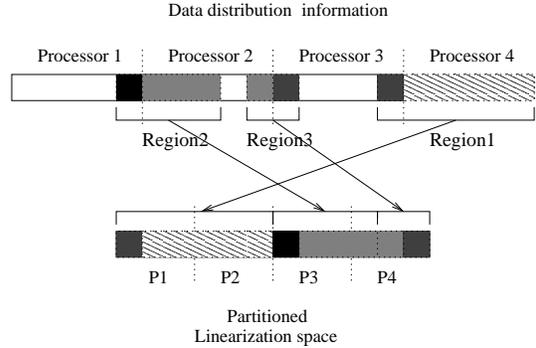


Figure 4. An example of the required distribution information

gram is computing the schedules).

As we said in the previous section, workload imbalance is a major problem for the first two methods described in Section 2.4.1. The workload imbalance comes from having each process compute schedules for different numbers of data elements in the linearization. To make the workload balanced, the linearization space can be evenly partitioned across all responsible processes, and each responsible process computes schedules for its part of the partitioned linearization space. Algorithm 3 shows the steps to compute schedules with one compact and one non-compact data descriptor.

Since processes in the program with a non-compact data descriptor must obtain the compact data descriptor and corresponding data specification to compute schedules, they acquire that information using one of the two methods described in Section 2.3.1 and Figure 2 (line 1 in Algorithm 3). Although a non-compact data descriptor may be very large, each responsible process needs distribution information about the data elements to be transferred in the part of the linearization space it is responsible for. Figure 4 shows how the data distribution information is partitioned across processes, and what information each responsible process requires to compute schedules. In this example there are four processes with a partial non-compact data descriptor and the linearization is partitioned into four parts. As seen in Figure 4, the data distribution information that each process needs may be owned by any other non-compact descriptor process. In other words, each such process must send and receive data distribution information to/from all

Algorithm 3 Computing schedules with one compact and one non-compact data descriptor and two sets of regions

- 1: The compact descriptor program sends its descriptor and corresponding data specification to the program with the non-compact data descriptor; the non-compact data descriptor program receives the information from the compact descriptor program
 - 2: The non-compact descriptor program sends data distribution information from the non-compact descriptor corresponding to its data specification to the other processes in the non-compact descriptor program (or in both programs, if both compute schedules), based on the partitioned linearization space;
 - 3: The non-compact data descriptor program (or both programs, if both compute schedules) receives data distribution information for the part of the linearization that each process is responsible for
 - 4: Responsible processes (either non-compact descriptor processes or all processes) compute the intersections J_i between the data owned by each compact data descriptor process i and the set of regions for the remote data structure
 - 5: For each J_i , the responsible processes map it into the linearization for the compact data structure
 - 6: For only the part of the linearization space each process is responsible for, compute pairs of matched data elements from the compact and the non-compact data descriptors in the linearization space
 - 7: For each pair of matched data elements, responsible processes translate from the global reference space to the local reference space
 - 8: Responsible processes generate partial communication schedule entries for each pair of matched data elements from the compact and the non-compact data structures
 - 9: Responsible processes send partial schedules back to all processes
 - 10: Receive partial schedules from the responsible processes
-

other responsible processes (lines 2 and 3). The linearization space is partitioned into as many parts as there are processes in both programs, if both programs are used to compute schedules. After obtaining information about the data distributions and data specifications for both the local and remote data structures, each responsible process can compute the intersections between the data blocks from the compact data descriptor and the data specification for the compact data structure, and map the intersections into the linearization space, as previously described in Section 2.3.3 (lines 4 and 5). Since each responsible process already has data distribution information about the irregularly distributed data elements in the part of the linearization space it is responsible for, it can find pairs of matched data elements from the non-compact and the compact data structures (line 6). As in Section 2.3.3, the algorithm must translate locations of data elements in the linearization space into local references (process IDs and local offsets). This part of the algorithm is exactly the same as for the remote block distributed data elements described in Section 2.3.3. The non-compact data descriptor must explicitly describe the global address, process ID, and local offset for each irregularly distributed data element. Therefore, each responsible process can translate positions in the global reference space into local references (line 7). Then schedules can be generated for both the compact and the non-compact descriptor processes (line 8). Since the linearization space is evenly partitioned across all responsible processes, each such process may compute schedules for data elements that it does not own. Therefore, each responsible process must send schedules to all other processes (line 9). Non-compact descriptor processes must send schedules back to all other non-compact data descriptor processes, as well as to all compact descriptor processes. Finally, all processes receive the communication schedules (line 10). Partial schedules sent between processes are aggregated to reduce the number of messages. The overall procedure to send partial schedules to the processes that require them employs a collective communication across all processes. However, the total number of messages sent depends on whether the processes in both programs or in only the non-compact descriptor program compute schedules. Figure 5 shows how many messages are required to send and receive sched-

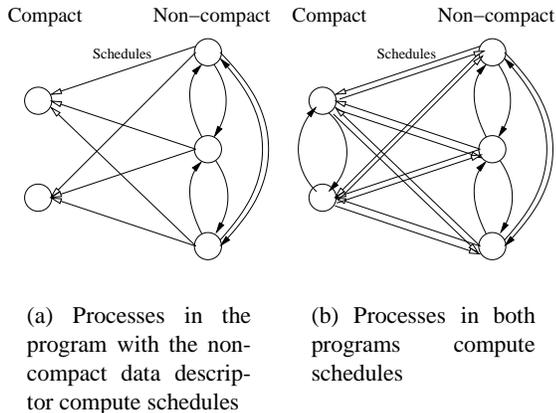


Figure 5. Messages to send and receive schedules depend on which processes are responsibility for computing schedules

ules in both cases. When only processes with the non-compact descriptor compute schedules, each non-compact descriptor process sends a message with the schedules to all compact descriptor processes, as well as to all other non-compact descriptor processes as seen in Figure 5(a). The total number of messages to send schedules to right processes is $m \times n$ (inter-program) + $(m - 1)^2$ (within program), where m and n are the number of non-compact and compact descriptor processes, respectively. However, all compact and non-compact processes must send/receive schedules to/from all other processes when both programs compute schedules, as seen in Figure 5(b). This method requires $(m + n - 1)^2$ messages, with $2 \times m \times n$ of them are inter-program messages. As was noted in Section 2.4.1, the algorithm requires more inter-program messages to send back schedules, if the processes in both programs are used to compute schedules. So if the two programs are run on different clusters connected via a WAN, it may not provide the best performance to employ processes in both programs to compute schedules, because of the relatively high message passing costs across the WAN. Figure 6(b) shows an alternative method for sending schedules from the responsible processes, when only the processes in the program with the non-compact descriptor compute schedules. In this method, a non-compact descriptor process sends the partial schedules needed by *all* the compact descriptor processes to the com-

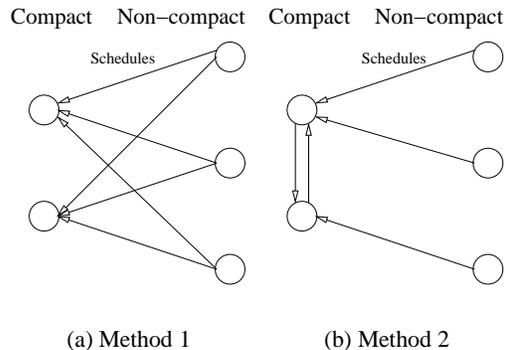


Figure 6. Two methods to send schedules back

compact descriptor process that sent it the compact data descriptor. A compact descriptor process then distributes the received schedules to all other compact descriptor processes. The schedule distribution work within the compact descriptor processes is very similar to that within the non-compact descriptor processes. Moreover, the compact descriptor processes do the distribution work at the same time the non-compact descriptor processes do theirs. This alternative requires messages to distribute schedules within processes with a compact data descriptor. So the total number of messages to send back schedules is n (inter-program) + $(m - 1)^2$ (within program) + $(n - 1)^2$ (within program) while the original algorithm sends $m \times n$ (inter-program) + $(m - 1)^2$ (within program) messages. In a Grid environment, using the alternative may be more efficient because it reduces the number of messages passed across the WAN to n , instead of $m \times n$.

2.5 Two non-compact data descriptors

In this case, the processes in both programs only have partial data distribution information. While necessary, it is very expensive for the two programs to exchange the non-compact data descriptors, since the size of a non-compact data descriptor is proportional to the size of the entire data structure.

2.5.1 Responsibility for Computing Schedules

In Section 2.4.2, we described an algorithm that has each responsible process extract the data distribution information necessary for other processes to compute

the schedules. The size of the data distribution information to be transferred among processes is proportional to the size of the actual data to be transferred. As for the other algorithms, we could assign only one program to compute schedules, as in Section 2.4.1. For the processes to compute schedules, they must receive data distribution information for the data structures on both programs. Therefore, the size of the data distribution information to be transferred is fixed, and is not dependent on which program computes the schedules. We therefore assign the workload of building the schedules to the processes in both programs.

2.5.2 Schedule Generation Algorithm

Algorithm 4 Computing schedules with two non-compact data descriptors

- 1: Send data distribution information about the data elements to be transferred in the local data structure to all local and remote processes, based on the partitioned linearization
 - 2: Receive data distribution information about both the local and the remote data structures for the part of the linearization that this process is responsible for
 - 3: Generate partial communication schedule entries for each pair of matched local and remote data elements
 - 4: Send partial schedules back to all processes of both programs
-

Algorithm 4 shows the steps to generate schedules with two non-compact data descriptors. As we said in Section 2.4.2, the linearization space is evenly partitioned across all the processes that compute the schedules. In this case, that means it is partitioned across all processes in both programs. Since all processes compute schedules for the same number of matching data elements, the workload is almost perfectly balanced across all the available processes.

To collect data distribution information for the part of the linearization it is responsible for, each process performs the same actions as described in Section 2.4.2 (lines 1 and 2 of Algorithm 4). The only difference is that each process collects distribution information for both the local and the remote data struc-

tures. This collection procedure requires a collective communication across all processes of both programs. After collecting the necessary data distribution information for both data structures, each process can generate schedule entries for each pair of matching data elements (line 3), because a non-compact data descriptor explicitly maps a global address in either the local or remote data structure to the process that owns (ownership) the data element (and its local address in that process). After generating the schedules, each process must send the schedules to the processes that need them (line 4). This activity requires another collective communication, with the total size of the schedules transferred proportional to the number of data elements that are involved in the data transfer. In Section 2.4.2, we discussed an alternative algorithm to reduce the number of messages between programs, when one program has a compact descriptor and one has a non-compact data descriptor. The original algorithm requires a collective communication to send schedules to processes. In the alternative algorithm, each process in one program sends all schedules for processes in the other program to exactly one process in the other program. Next the the processes redistribute the received schedules to the processes that need them, with that redistribution occurring only within the processes of one program (so that no communication between programs is needed, potentially across a WAN). This alternative algorithm can also be used for sending schedules between programs that each have non-compact data descriptors. Since the alternative algorithm reduces the number of messages transmitted between the two programs, it may have better performance than the original algorithm. However, we have not yet implemented the alternative algorithm for the two non-compact data descriptor case.

3 Performance Evaluation

3.1 Experimental environment and notation

We have experimented with the different InterComm algorithms to compare the times to generate communication schedules, and also compared against the Meta-Chaos schedule building implementation. Note that all variants for the same input specification (data descriptors and data specification) must produce

Algorithm	Variant/Label	Description
Two compact data descriptors	Deref	Meta-Chaos, using dereference functions
	BothCom	Both programs compute schedules
	OneCom	One program computes schedules
One compact and one non-compact descriptor	Deref	Meta-Chaos
	BothCom	Both programs compute schedules
	OneCom/AllBack	One program computes schedules/Send schedules back to all processes
	OneCom/OneBack	One program computes schedules/Send schedules back to one process
Two non-compact data descriptors	Deref	Meta-Chaos
	InterComm	InterComm algorithm

Table 1. The methods to generate communication schedules, with labels used in the performance graphs

the same communication schedule. We experimented with two types of computation environments. The first is a local cluster environment where all programs run on a single cluster connected via a local area network. The second is a distributed Grid computation environment where programs run on two clusters connected by a wide-area network (in this case, Internet2). For the experiments in the local cluster environment, we have run two parallel programs on a 50-processor Linux cluster at the University of Maryland, building schedules to copy distributed data between the programs. Each processor is a 650MHz PentiumIII machine with 768MB of memory, and the processors in the cluster are connected via channel-bonded Fast Ethernet, providing a 200Mb/sec connection for each processor. For experiments in the Grid environment, we ran one program on the 50-processor cluster just described and the other program on a 22-processor Linux cluster at Ohio State University. Each processor in that cluster is a 933MHz PentiumIII machine with 512MB of memory, and the processors are connected via Fast Ethernet. In all the experiments, a single process of a parallel program was assigned to each processor.

We employed two types of distributed datasets. One dataset is a multi-dimensional array, block distributed across processes in all dimensions, of the type supported by High Performance Fortran [13] or the Multi-block Parti library [1]. That dataset can be described with a compact data descriptor. For all of the experiments, the data is a two-dimensional array of double precision floating point numbers of size 1024×1024 . The second dataset is an explicitly (irregularly) dis-

tributed array, of the type supported by the Chaos library [17], that must be described with a non-compact data descriptor. That data is an array of 2^{20} double precision floating point numbers, with array elements randomly assigned to processes so that each process owns approximately the same number of elements. Note that the base data type of the data (e.g., int, float, double) does not affect the time to generate communication schedules, since all operations for computing schedules employ offsets from the beginning of the (array) data structure, in terms of the number of elements, not the number of bytes. The operations for moving the data using the computed communication schedules take into account the size of the base data type, to perform the message sends and receives specified in the schedule. The experimental space we explore is effectively three dimensional. The first dimension is for different combinations of the types of data descriptors in the source and destination program (e.g., compact-compact). The second dimension is the number of processes in the source and destination parallel programs. Each program was run on up to 16 processes, so that we can measure times for 1×1 through 16×16 processes (# sender processes \times # receiver processes), to show the scalability of the algorithms. The last dimension is the number of data elements to transfer. To show scalability with respect to the amount of data to transfer, we computed schedules that transfer $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$ and *all* the data elements from the datasets, randomly selecting which data elements are transferred (for block distributed arrays, we randomly selected contiguous subarrays, not individual data el-

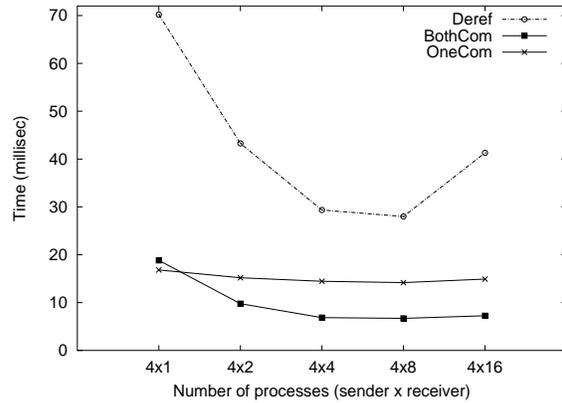
ements). We also measured how well the workload is balanced. For all the experiments, the execution times shown are computed from the times for five runs, showing the average value after removing the smallest and largest value. However, in general the execution times of those five runs did not vary much, because the experiments were run when the cluster(s) was (were) otherwise unloaded. As was previously discussed, InterComm generates essentially identical communication schedules as Meta-Chaos, so both systems have the same performance for transferring the data using the communication schedules.

To make the graphical presentation of the performance results easier to decode, we summarize the labels used for the various algorithm options in Table 1. For the InterComm algorithms with a compact data descriptor, there are two sets of options. One set is for how to transfer data descriptors between the programs and the other set is for where to compute schedules. When the programs are run on a single cluster, there is almost no performance difference for the two methods for transferring data descriptors, because they both require the same number of messages. We would expect that the method that has only representative processes exchange compact data descriptors would perform better in a Grid environment. However, the two clusters used for the experiments in the Grid environment are connected via *Internet2*, which provides very high bandwidth between the clusters. We therefore did not see much difference in the performance of the two methods, Thus we do not show experimental results for the representative method, to simplify the presentation. In the rest of this section, we present experimental results for building communication schedules. In Sections 3.2 and 3.3, we show experimental results on the local cluster environment. We compare experimental results in the Grid environment with those in the local cluster environment in Section 3.4.

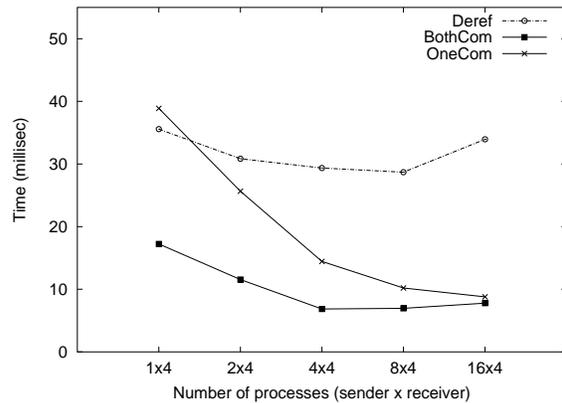
3.2 Scalability

3.2.1 Number of processes

Figures 7, 8 and 9 show the effects of scaling the number of processes in the sender and receiver programs. All processes were run on the 50-processor cluster described earlier, and half the data in the datasets was involved in the data transfer.



(a) Varying the number of receiver processes



(b) Varying the number of sender processes

Figure 7. Varying the number of processes, for two compact descriptors, transferring half of a 1024×1024 array between the sender and receiver programs. The line labeled Deref is for the Meta-Chaos library, while the others are labeled with the algorithm variations for InterComm, as described in Table 1.

Two compact data descriptors: Figure 7 shows the times to generate communication schedules when both the sender and receiver program have compact descriptors for their data structures involved in the transfer.

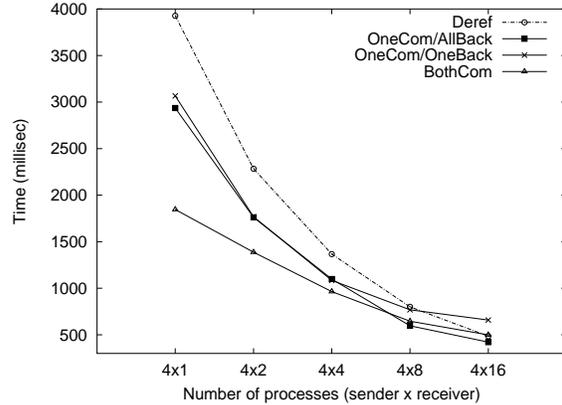
Figure 7(a) shows the effect of varying the number of receiver processes, fixing the number of sender

processes at four. Since Meta-Chaos computes schedules only in the receiver processes, its performance improves as more receiver processes are added. However, the Meta-Chaos performance decreases when there are too many receiver processes (the 4×16 point in the graph), because Meta-Chaos requires two communication operations within the receiver processes – one to *dereference* data elements and one to send computed partial schedules to other receiver processes.

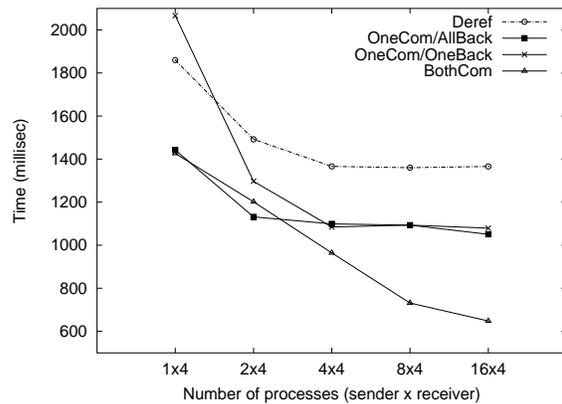
When both sender and receiver processes compute the schedules (the BothCom line in Figure 7(a)), performance is determined by the program that is run with fewer processes, because a process in that program performs more work on average than a process in the other program. We therefore see that performance improves with increasing numbers of receiver processes until the number of processes in both programs is the same (4×4). Beyond that point, performance is limited by the sender processes. When only sender processes compute the schedules (the OneCom line), performance is not affected by increasing the number of receiver processes. In comparing the InterComm algorithms to the Meta-Chaos algorithm, the best InterComm algorithm takes from 15% to 25% the time of the Meta-Chaos algorithm.

Figure 7(b) shows the effects of varying the number of sender processes, fixing the number of receiver processes at 4. Meta-Chaos performance is fairly constant because schedules are computed on the receiver processes. Meta-Chaos performance does improve slightly with more sender processes, because more sender processes are available to perform local dereferencing operations. However, Meta-Chaos performance degrades with too many sender processes (the 16×4 point), because the receiver processes must send partial schedules back to more sender processes. The InterComm algorithm that computes schedules in both programs (the BothCom line) improves in performance until the numbers of sender and receiver processes are the same, again because the overall performance is limited by the program running with fewer processes. Beyond that point (the 8×4 and 16×4 points), performance degrades slightly because each receiver process must send data descriptors to multiple (2 or 4) sender processes. The algorithm that computes the schedule in one sender process (OneCom) speeds up as the number of sender processes increases,

because it computes schedules only in the sender processes. For this scenario, the best InterComm algorithm takes from 25% to 50% the time of the Meta-Chaos algorithm.



(a) Varying the number of receiver processes



(b) Varying the number of sender processes

Figure 8. Varying the number of processes, for one compact and one non-compact descriptor with the sender program transferring half of a 1024×1024 array and the receiver program transferring into half of a 2^{20} element explicitly distributed array. The line labeled Deref is for the Meta-Chaos library, while the others are labeled with the algorithm variations for InterComm as described in Table 1.

One compact and one non-compact data descriptors: Figure 8 shows performance results when the sender program data structure is a two-dimensional, block distributed array of size 1024×1024 with a compact descriptor, and the receiver program has an explicitly distributed dataset of 2^{20} double precision floating point numbers with a non-compact descriptor. As we said before, we have two sets of options. One of them is how to send schedules back. We support only the send-back option that all processes send schedules back to all processes when both programs compute schedules. As we will see later, the performance is the worst when both programs compute schedules in Grid environment since it requires much expensive communication between two programs while it is the best in the local cluster environment. The send-back option affects the performance in Grid environment. Since the performance with both computing programs is not good and it should be used in the local cluster environment, we provide only the send-back method that all processes send schedules back to all processes when both programs compute schedules.

Figure 8(a) varies the number of receiver processes, fixing the number of sender processes at 4. When both the Meta-Chaos and InterComm (the OneCom/AllBack and OneCom/OneBack) algorithms compute schedules in the receiver processes (in the program with the non-compact data descriptor), they show similar performance characteristics with performance improving greatly as having more receiver processes. The InterComm algorithm that has both programs compute schedules (BothCom) shows good performance with small numbers of receiver processes, since the sender processes also compute schedules. Although sender processes compute schedules, performance degrades with more receiver processes (4×8 and 4×16) since the algorithm requires too much communication among all sender and receiver processes. Although the InterComm implementation that sends partial schedules back to the sender processes from the receiver processes has not yet been highly optimized, the best InterComm algorithm is always better than the Meta-Chaos algorithm.

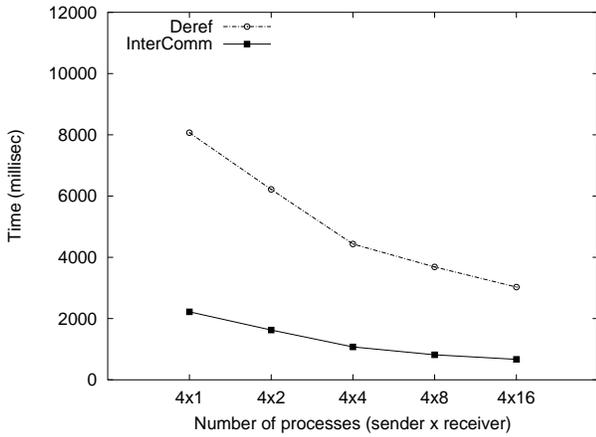
Figure 8(b) shows the effects of varying the number of sender processes on performance. The Meta-Chaos and InterComm algorithms (Deref and OneCom) that have only one program compute schedules improve

in performance until the number of sender processes is the same as the number of receiver processes. Although the InterComm algorithms compute schedules in the receiver processes, a small number of sender processes decreases performance because each sender process must send its compact data descriptor to more than one receiver process and also receives larger schedules from the receiver processes. For the InterComm algorithm that has both programs compute schedules (BothCom), performance improves linearly because more processes compute schedules. Although this method requires a large amount of communication among processes, the cost is not very high in the local cluster environment. However, in Section 3.4 we will see that this method does not perform well in a Grid environment because of high communication costs. In many cases in the local cluster environment, the InterComm algorithms take less than 60% of the time for the Meta-Chaos algorithm to generate the same schedules.

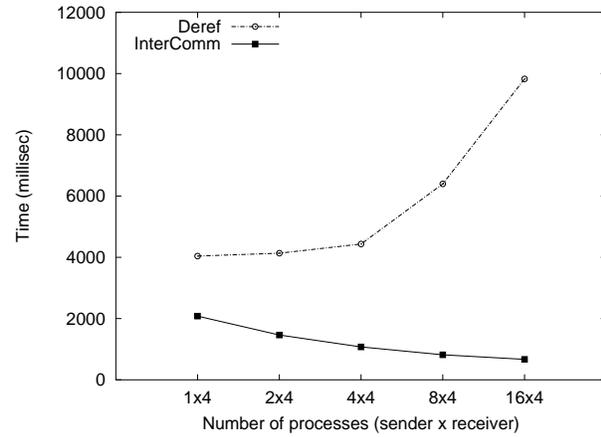
Two non-compact data descriptors: Figure 9 shows communication schedule building performance when both programs have an explicitly distributed 2^{20} element double precision floating point array. In this case, InterComm currently implements a single algorithm.

In Figure 9(a), both the Meta-Chaos and InterComm algorithm performance improves as the number of receiver processes increases. The Meta-Chaos algorithm performance improves because schedules are computed in the receiver processes. The InterComm algorithm takes less time because schedules are computed using all sender and receiver processes, which is why the InterComm algorithm performs better than the Meta-Chaos algorithm. In this experiment, the InterComm algorithm takes from 20% to 30% of the time for the Meta-Chaos algorithm.

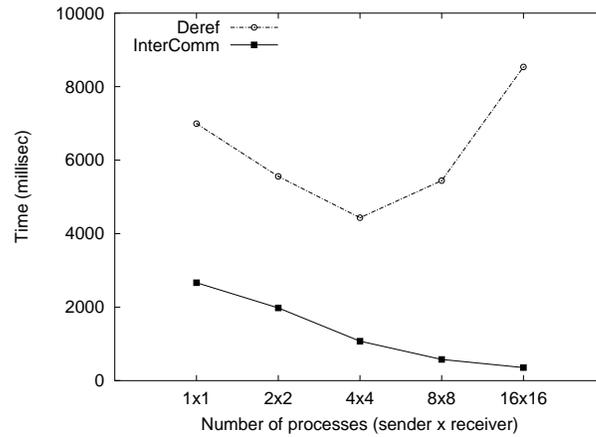
Figure 9(b) shows the effect of varying the number of sender processes. With more sender processes, InterComm performance improves, because performance depends on the total number of processes in both the sender and receiver programs. In fact, the performance is almost the same as in Figure 9(a). However, Meta-Chaos performance decreases as the number of sender processes increases. The Meta-Chaos algorithm computes schedules in the receiver processes, but the receiver processes must send schedules back to



(a) Varying the number of receiver processes



(b) Varying the number of sender processes



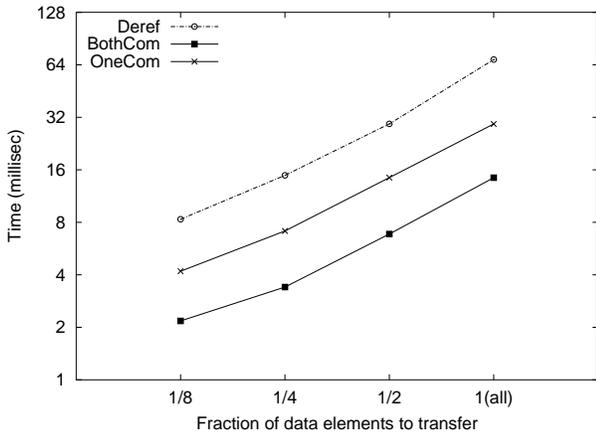
(c) Varying numbers of both sender and receiver processes

Figure 9. Varying the number of processes for two non-compact descriptors with both sender and receiver programs transferring half of a 2^{20} element explicitly distributed array. The line labeled Deref is for the Meta-Chaos library and the other is for the InterComm library.

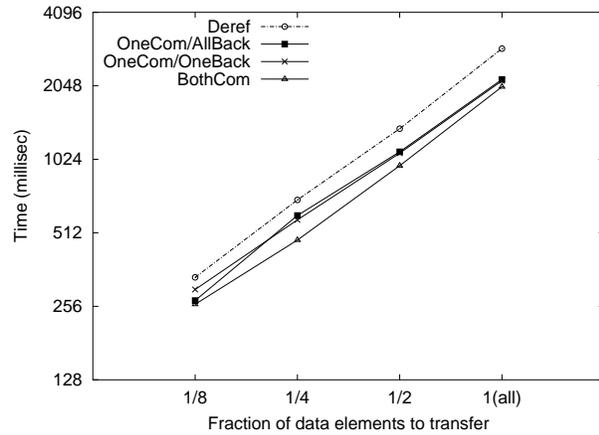
the sender processes. When there are more sender processes, each receiver process must send schedules to more sender processes. In this case, InterComm takes from 7% to 50% of the time for Meta-Chaos.

Figures 9(a) and 9(b) show that the Meta-Chaos algorithm speeds up with more receiver processes, but slows down with more sender processes, while InterComm speeds up with more processes in either the sender or receiver programs. Figure 9(c) shows another view of performance, when the number of pro-

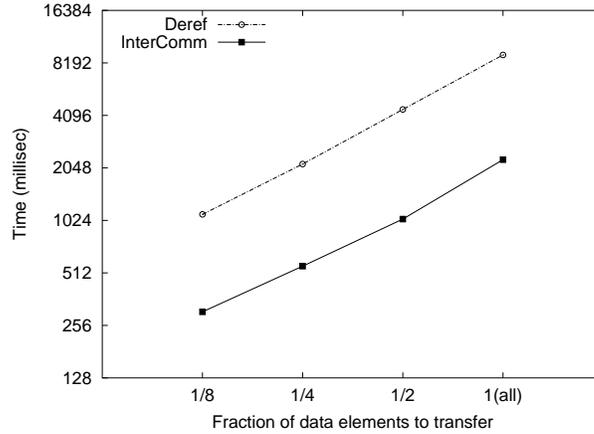
cesses in both the sender and receiver program increases. InterComm performance increases because it can make effective use of all available processes. However, Meta-Chaos performance only improves up to 4×4 processes, and then gets worse. This result implies that the performance gain from having more receiver processes is less than the performance loss from more sender processes when there are more than 4×4 processes. In this experiment, InterComm takes from 4% to 35% of the time for Meta-Chaos.



(a) Varying the number of transferred data elements with two compact data descriptors



(b) Varying the number of transferred data elements with one compact and one non-compact data descriptors



(c) Varying the number of transferred data elements with two non-compact data descriptors

Figure 10. Varying the number of transferred data elements for 4 sender and 4 receiver processes. The data structure with the compact data descriptor is a 1024×1024 block distributed array and the data structure with the non-compact descriptor is a 2^{20} explicitly distributed array. The graphs are labeled as described in Table 1.

3.2.2 Number of data elements to transfer

Figure 10 shows the effects of varying the amount of data to transfer, for the three combinations of data descriptors using 4 sender and 4 receiver processes. As the amount of data that must be transferred increases, the time required to compute the schedules increases. Note that the times increase approximately linearly

with the amount of data to transfer. In results not shown, we have seen similar results for other numbers of sender and receiver processes. Both the Meta-Chaos and InterComm algorithms show good scalability with respect to the number of data elements to transfer, but InterComm has better absolute performance.

3.3 Workload Balance

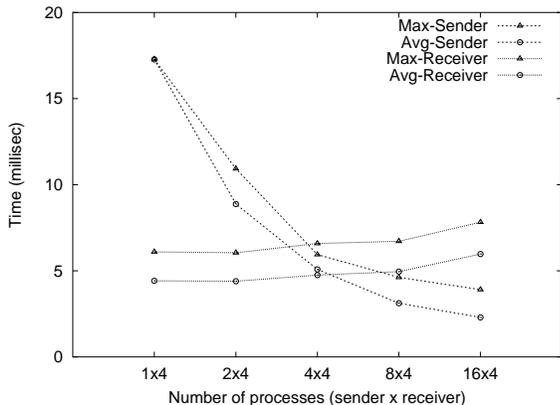


Figure 11. InterComm maximum and average times across sender processes (Max-Sender and Avg-Sender) and receiver processes (Max-Receiver and Avg-Receiver) to compute schedules. The sender and receiver programs transfer half of a 1024×1024 array, both with compact data descriptors. The algorithm is the All/Both one from Table 1.

In Meta-Chaos, the processes of the sender program are idle while the processes of the receiver program compute schedules. In other words, the workload is poorly balanced across the programs. However, the workload within the receiver processes is well-balanced because Meta-Chaos distributes the schedule building work evenly across those processes.

Figure 11 shows how well the workload is distributed for one of the InterComm algorithm variants for two compact data descriptors. The algorithm shown has all processes in the sender and receiver programs exchange their compact data descriptors and compute schedules completely locally, so that no schedules need to be sent between processes. In Figure 11, *Max-Sender* and *Avg-Sender* show the maximum time and the average times for the sender processes, while *Max-Receiver* and *Avg-Receiver* are the times for the receiver processes. Since the number of receiver processes is fixed at 4, the times for *max2* and *avg2* do not change much for increased numbers of sender processes. The times for *Max-Sender* and *Avg-Sender* decrease with more sender processes. We see

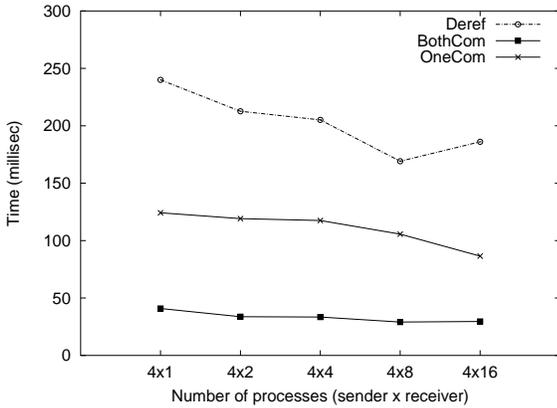
that the workload across the two programs depends on the number of processes in each of the programs. When we have m sender and n receiver processes, a sender process has approximately $\frac{n}{m}$ times as much work to perform as a receiver process. Within a program, the maximum time for a process to do its schedule computations is from 15% to 40% more than the average times across all processes in that program. Although this workload imbalance can be a problem, the InterComm algorithms for two compact data descriptors are still much faster in most cases than the Meta-Chaos ones, as shown in Figure 7.

The InterComm algorithms for one compact and one non-compact data descriptor compute schedules in either one program or both programs, while Meta-Chaos computes schedules on only one program. While we do not show detailed experimental results, the InterComm algorithm that computes schedules in both programs showed an almost perfect workload balance across all processes in both programs, since all sender and receiver processes are assigned the same number of data elements to compute schedules for. Although Meta-Chaos and the InterComm algorithm that computes schedules in only one program have the processes in one program idle during the schedule computation, our experimental results showed an almost perfect workload balance across the processes in the program computing the schedule (the one with the non-compact descriptor).

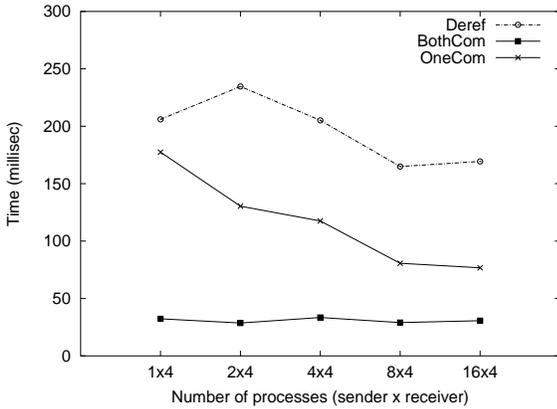
Finally, experiments for two programs both with non-compact data descriptors show that InterComm balances the workload almost perfectly, because all sender and receiver processes are assigned the same number of data elements to compute schedules for.

3.4 Running in a Grid Environment

Figures 12, 13 and 14 show the effects of scaling the number of processes in a Grid environment. Sender processes were run on the Maryland cluster and receiver processes on the Ohio State cluster. All experiments in this section are identical to the ones on scalability described in Section 3.2.1, except they were run in the Grid environment. We investigate scalability with respect to the number of processes in the Grid environment, and compare the results with those for the local cluster environment.



(a) Varying the number of receiver processes



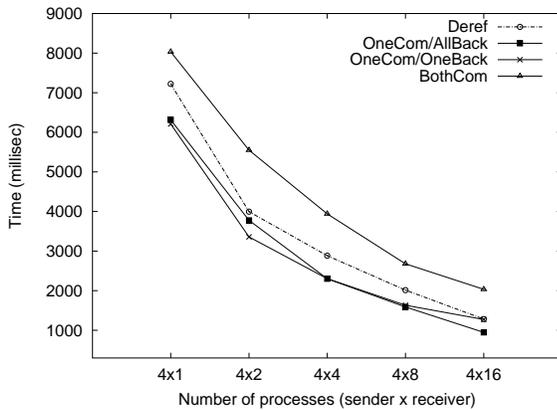
(b) Varying the number of sender processes

Figure 12. Varying the number of processes, in a Grid environment, for two compact descriptors, transferring half of a 1024×1024 array between the sender and receiver programs. The line labeled Deref is for the Meta-Chaos library, while the others are labeled with the algorithm variations for InterComm, as described in Table 1.

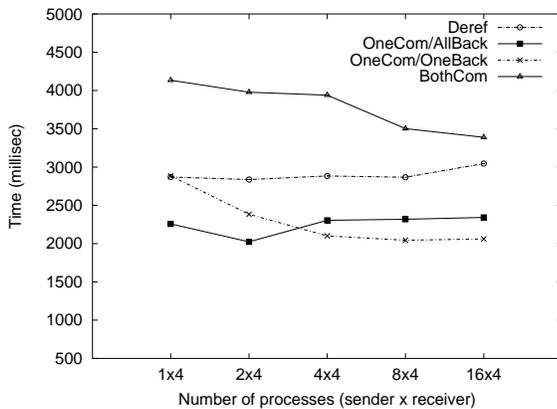
Two compact data descriptors: Figure 12 shows the times to generate communication schedules in the Grid environment when both the sender and receiver programs have compact data descriptors. Comparing with Figure 7 on a single cluster, Figure 12 shows similar relative performance, although the absolute perfor-

mance for both InterComm and Meta-Chaos are much better in the local cluster than in the Grid environment. We also see that the curves for Meta-Chaos in the Grid environment are flatter than the ones in the single cluster experiment, since communication costs are higher in the Grid environment and become a significant part of the time to compute schedules. The InterComm algorithms require that the two programs exchange messages for transferring the compact data descriptors. However, Meta-Chaos may require many large messages be transferred between the two programs to send back schedules. So Meta-Chaos may spend a large fraction of its time in communication. For the two different InterComm algorithms, the one that has only the processes in one program computes schedules (OneCom) takes relatively more time than the method that uses the processes in both programs (BothCom) in the Grid environment, compared to the local cluster environment seen in Figure 7. Since in that case the processes in the program computing the schedules must send them back to processes in the other program, and such messages are over the WAN, the performance with one program computing schedules is much worse than the method with both programs computing schedules. In this Grid experiment, the best InterComm algorithm (BothCom) takes only 12% to 18% of the time for the Meta-Chaos algorithm.

One compact and one non-compact data descriptor: For this case, we performed the same experiment as for the one compact and one non-compact descriptor case discussed in Section 3.2.1, but in the Grid environment. Figure 13 shows the results, varying separately the number of sender and receiver processes. The curves in Figure 13(a) have a similar shape to those in Figure 8(a) for the local cluster environment, but the absolute times are again much higher in the Grid environment. The InterComm algorithm that has one program compute schedules (OneCom) performs a little better than the Meta-Chaos algorithm, as seen in Figure 8(a). However, the InterComm algorithm that has both programs compute schedules (BothCom) is even worse than the Meta-Chaos algorithm in the Grid experiment, while that algorithm is the best one in the single cluster experiment. The poor performance is because the InterComm algorithm with both programs computing schedules requires a large amount of communication between processes in the two programs,



(a) Varying the number of receiver processes



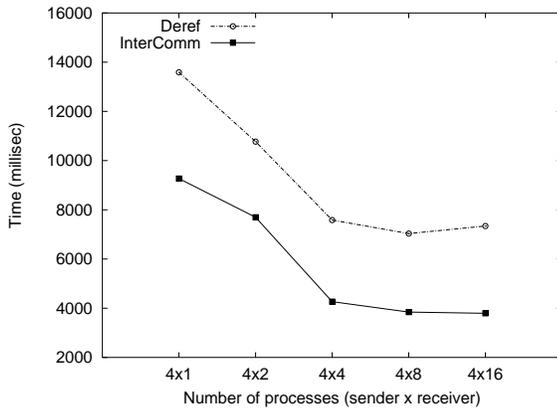
(b) Varying the number of sender processes

Figure 13. Varying the number of processes in a Grid environment for one compact and one non-compact descriptor, with the sender program transferring half of a 1024×1024 array and the receiver program transferring half of a 2^{20} element explicitly distributed array. The line labeled Deref is for the Meta-Chaos library, while the others are labeled with the algorithm variations for InterComm, as described in Table 1.

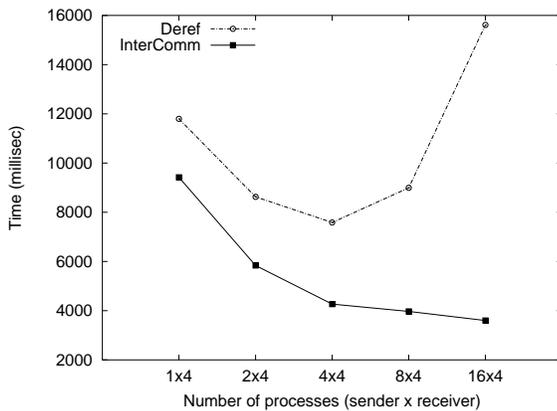
and communication is more expensive in the Grid environment than in the local cluster.

In Figure 13(b), the Intercomm BothCom algorithm also shows the worst performance, and very little performance improvement with more sender processes, while that algorithm showed a linear performance improvement in the local environment, as seen in Figure 8(b). In the Grid environment, the performance gain from using more processes to compute the schedules is cancelled out by the high communication cost between processes in the two programs. The other performance curves, for the InterComm OneCom algorithm and for Meta-Chaos also show little improvement with an increased number of sender processes, since the number of processes that compute schedules is fixed. In Figure 13(b), Meta-Chaos does not show a performance improvement with more sender processes because spreading the sender computation costs across additional sender processes (to locate data elements to be transferred in each sender process) is dominated by the more expensive communication cost in Grid environment. However, the InterComm algorithm that computes schedules in one program and sends schedules back to just one process (OneBack) performs better than the algorithm that sends schedules back to all processes (AllBack) when there are many sender processes, even though both algorithms have the same performance in the local cluster. This behavior is also an effect of the Grid environment. The AllBack algorithm requires many messages via the WAN, while the OneBack algorithm sends only one message per process via the WAN, and many messages on the LAN where each program is running.

Two non-compact data descriptors: We performed the same experiments as for the local cluster for two programs that both have non-compact data descriptors in the Grid environment. Figure 14 shows the results in the Grid environment, and looks similar to Figure 9. The InterComm algorithm performs much better than the Meta-Chaos algorithm in the local cluster environment and in the Grid environment. However, the absolute performance of the InterComm (and Meta-Chaos) algorithm in the Grid environment is not as good as in the local cluster environment, since the InterComm (and Meta-Chaos) algorithm spend much more time in communication between the two programs in the Grid environment.



(a) Varying the number of receiver processes



(b) Varying the number of sender processes

Figure 14. Varying the number of processes in a Grid environment for two non-compact descriptors, with both the sender and receiver programs transferring half of a 2^{20} element explicitly distributed array. The line labeled Deref is for the Meta-Chaos library, and the other is for the InterComm library.

4 Conclusions and Future Works

We have presented various algorithms implemented in InterComm to allow parallel applications to directly exchange data without intermediary processes. The algorithms are optimized to take advantage of the properties of different types of distributed data structures,

including how the data structures are distributed across the processes of the parallel application, and also are tailored to perform well on both single parallel machines or clusters, and in distributed Grid computation environments. We have shown that the algorithms for clusters and Grid environments are scalable and perform much better than the corresponding Meta-Chaos algorithms.

While our current InterComm implementation only supports data transfers between separate programs/components of an application, it is straightforward to extend the implementation to support data transfers between data structures with different data distributions within a single application component.

The InterComm algorithms fall into three classes, depending on the types of the data distributions in the two programs. We have described several options that can be selected from for each class of algorithms. One goal of some of the options is to make computation of communication schedules efficient in a heterogeneous Grid environment. We are working on methods to automatically select the best options for a given data transfer at runtime, based on the computation environments for the two programs (numbers of processes, speed of the network connections between the programs, etc.). We are also continuing to work on making the schedule building algorithms more efficient. Finally, we are working with application scientists in multiple areas, including earth and space science, to employ InterComm to support complex, multi-scale, multi-resolution physical simulations.

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] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1999.
- [3] P. Beckman, P. Fasel, W. Humphrey, and S. Mniszewski. Efficient coupling of parallel applications using PAWS. In *Proceedings of the 7th*

- IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 1998.
- [4] Common Component Architecture (CCA) Forum. <http://www.cca-forum.org/>.
- [5] Common Component Architecture (CCA) MxN working group. <http://www.csm.ornl.gov/cca/mxn/>.
- [6] G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries. In *Proceedings of the Eleventh International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1997.
- [7] Earth System Modeling Framework (ESMF). <http://www.esmf.ucar.edu/>.
- [8] S. Fink, S. Kohn, and S. Baden. Efficient runtime support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1):61–82, Apr. 1998.
- [9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation Series. MIT Press, 1994.
- [10] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. *International Journal of High Performance Computing Applications*, 11(3):224–236, Aug. 1997.
- [11] X. Jiao, M. T. Campbell, and M. T. Heath. Roccom: An object-oriented, data centric software integration framework for multiphysics simulations. In *Proceedings of 17th Annual ACM International Conference on Supercomputing*, June 2003.
- [12] K. Keahey, P. Fasel, and S. Mniszewski. PAWS: Collective Interactions and Data Transfers. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 2001.
- [13] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.
- [14] T. Kurc, U. Catalyurek, X. Zhang, J. Saltz, M. Peszynska, R. Martino, M. Wheeler, A. Sussman, C. Hansen, M. Sen, R. Seifoullaev, P. Stoffa, C. Torres-Verdin, and M. Parashar. A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies. *Submitted to Concurrency and Computation: Practice and Experience*, Apr. 2003.
- [15] J. W. Larson, R. Jacob, I. Foster, and J. Guo. The model coupling toolkit. In *Proceedings of International Conference on Computational Science*, 2001.
- [16] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10:197–220, 1996.
- [17] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, Aug. 1995.
- [18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference, Second Edition*. Scientific and Engineering Computation Series. MIT Press, 1998.