Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
J. Parallel Distrib. Comput. 66 (2006) 931–946
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Data redistribution and remote method invocation for coupled components

Felipe Bertrand[a,*], Randall Bramley[a], David E. Bernholdt[b], James A. Kohl[b], Alan Sussman[c], Jay W. Larson[d], Kostadin B. Damevski[e]

[a]Computer Science Department, Indiana University, USA
[b]Computer Science and Mathematics Division, Oak Ridge National Laboratory, TN, USA
[c]UMIACS and Department of Computer Science, University of Maryland, MI, USA
[d]Mathematics and Computer Science Division, Argonne National Laboratory, IL, USA
[e]Scientific Computing and Imaging Institute, University of Utah, UT, USA

Received 23 May 2005; received in revised form 30 September 2005; accepted 1 December 2005
Available online 3 May 2006

Abstract

With the increasing availability of high-performance massively parallel computer systems, the prevalence of sophisticated scientific simulation has grown rapidly. The complexity of the scientific models being simulated has also evolved, leading to a variety of coupled multi-physics simulation codes. Such cooperating parallel programs require fundamentally new interaction capabilities, to efficiently exchange parallel data structures and collectively invoke methods across programs. So-called "M × N" research, as part of the common component architecture (CCA) effort, addresses these special and challenging needs, to provide generalized interfaces and tools that support flexible parallel data redistribution and parallel remote method invocation. Using this technology, distinct simulation codes with disparate distributed data decompositions can work together to achieve greater scientific discoveries.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Common component architecture; Model coupling; Parallel data redistribution; Parallel remote method invocation

1. Introduction

Scientific computing is adopting more sophisticated scientific models that combine multiple physical models into a single advanced simulation experiment. For example, by applying several live simulation programs as dynamic boundary conditions, in place of the more traditional static boundary approaches, new results of a higher fidelity are possible. Projects now using this approach include biological cell modeling, climate modeling, space weather, fluid-structure coupling, and fusion energy simulation. Yet such *coupled* simulation models introduce a whole suite of complications, especially when each individual model can have a different temporal scale, spatial mesh organization, and/or distributed data decomposition. Individual simulation models are also often developed independently by different research teams, leading to challenging software integration obstacles. Combining multiple models also leads to major problems in application modeling and applied mathematics.

High-performance computing introduces an additional complication when the individual models are parallel programs: the "M × N" problem (pronounced "M by N"). In the M × N problem, two *parallel* simulation programs must cooperate and exchange data. However, one simulation executes on a set of "M" processes and the other executes on a potentially distinct set of "N" processes, so for a data object to be shared between the two simulations a mapping between corresponding data elements must be made, and software infrastructure must provide for the scheduling, synchronization and transfer of data elements defined by the mapping. This arrangement is illustrated in Fig. 1, where $M = 8$ and $N = 27$, and multiple processes on the N side must export data values for each single process on the M side. The set of operations required for such data manipulation is referred to as "parallel data redistribution" because the data is effectively translated from one distributed data decomposition into another.

* Corresponding author.
E-mail addresses: febertra@indiana.edu (F. Bertrand), bramley@indiana.edu (R. Bramley), bernholdtde@ornl.gov (D.E. Bernholdt), kohlja@ornl.gov (J.A. Kohl), als@cs.umd.edu (A. Sussman), larson@mcs.anl.gov (J.W. Larson), damevski@cs.utah.edu (K.B. Damevski).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.12.009

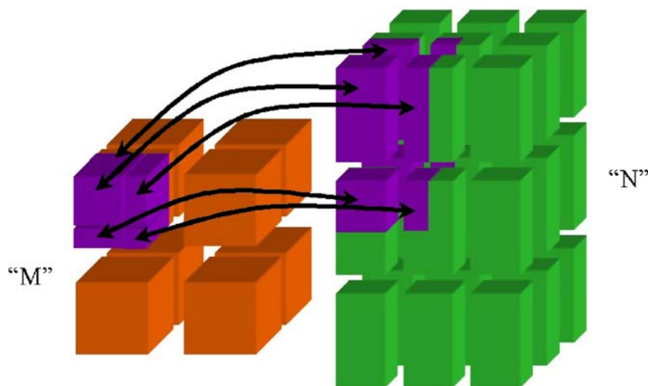


Fig. 1. The "M × N" problem.

Most of the literature in parallel computing about data redistribution deals with balancing work loads when shifting from one computational phase to another on the same set of processes. Examples include reassigning regions in an adaptive mesh refinement algorithm or a fluids-structures interaction simulation. In distributed computing, "data redistribution" refers to how a distributed data object from one set of processes is assigned to a new set of processes. This reassignment may try to achieve load balance on the receiving set of processes, but the essential problem is how to specify and define which receiving process(es) gets data from which sending process(es).

Such complications have made scientific simulation software increasingly unmanageable, prompting a variety of software development techniques to handle the complexity of integrating software modules, tools and libraries. One solution for managing this software complexity is an evolution of the object-oriented programming concept, known as *component-based software engineering* (CBSE) [46]. This methodology has been successful in the business software domain (e.g. CORBA [41], DCOM [39] and Enterprise JavaBeans [45]), and is migrating into the scientific domain in projects such as the Common Component Architecture (CCA) [2]. The CCA extends the concepts of *components*, *ports* and *frameworks* to high-performance scientific computing in parallel and distributed environments.

As part of the open CCA Forum [10] and the Center for Component Technology for Terascale Simulation Software [11] (part of the Scientific Discovery through Advanced Computing (SciDAC) program [48]), the M × N problem has been explored as a key enabling technology for component-based scientific simulation software. An M × N Working Group, in cooperation with a Scientific Data Components Working Group and the Terascale Simulation Tools and Technology [21] SciDAC Center, have been developing interfaces and technology that alleviate the burden on the scientific applications programmer in trying to assemble large coupled simulation applications. The work has emphasized the fundamental infrastructure required for two basic sets of capabilities, namely parallel data redistribution and collective method execution.

Using a generic description of each component's parallel data, a variety of data exchange operations can be applied to transparently couple parallel data objects configured at run time. Beyond parallel data exchange or redistribution capabilities, there is also the need for concatenating component "filters," e.g. for spatial and temporal interpolation or unit conversions. Such capabilities form the basis for a general, extensible M × N toolkit to encompass the full range of generalized model coupling technology. Generalizing the existing set of numerical interpolation and filtering schemes is a major undertaking and apart from some preliminary experiments is beyond the scope of our current work, which concentrates on parallel component interactions that can be solved by computer science middleware.

A related problem is when parallel components invoke methods on each other, referred to as *parallel remote method invocation* (PRMI). Although the term RMI originated in the Java community, here it refers to the general problem of interacting parallel object-oriented components. No well-defined widely accepted semantics exist yet for the possible wide range of types of parallel invocations. Methods could be invoked between serial or parallel callers and callees, and used to perform either coordinated parallel operations or to independently update local state in parallel. Such invocations could require data arguments or return results in either serial or parallel (decomposed or replicated) data arrangements.

Prototype M × N component and framework solutions have been developed to explore the desired capabilities. The remainder of this paper describes these solutions in more detail. Section 2 provides basic background on CBSE and component concepts, especially in the context of parallel and distributed environments. This overview includes a generalized view of parallel data structures and their underlying distributed data decompositions, as well as a review of the semantics and issues relating to PRMI. Section 3 lists some desired features of M × N systems. Then, Section 4 presents a survey of component-based M × N solutions that have been generated in conjunction with the CCA effort. Section 5 describes related work in M × N/coupling research. Finally, Sections 6 looks toward future work in this area and Section 7 concludes.

2. Overview

2.1. Definitions and overview

Within the CCA Forum, a *component* is a software unit which may be instantiated as part of a running process, or on a set of multiple processes, e.g., as an MPI job. It is therefore possible to have one component running as multiple processes, as well as have multiple components all running within one process. Partitioning a job into parallel processes typically implements *domain* or data decomposition, while partitioning a task into components implements a *functional* or computational phase decomposition. In any case, the decomposition into components is independent of any decomposition into parallel processes, and within the CCA a *component* can span multiple MPI-like parallel processes. The problems introduced by parallel components are particularly difficult when the problem has other forms of functional decomposition, sometimes based on accessing distributed or specialized data resources.

Communication between CCA components is through *ports* which employ a uses/provides design pattern. A *provides* port is a public interface that a component implements, that can be referenced and used by other components. A *uses* port is a connection end point that can be attached to a provides port of the same type. Once connected, the uses port becomes a reference to the provides port and the component can make method invocations on it. Interfaces in the CCA are specified with the Scientific Interface Definition Language (SIDL). SIDL is object oriented, designed with an emphasis on scientific computing [32].

In parallel computing a *communication schedule* is a sequence of message passing required to correctly move data among a set of cooperating processes. A communication schedule is typically the most difficult design issue in parallel programming and requires complex bookkeeping about data ownership by processes and the correct ordering of sends and receives to keep local copies up to date, and to avoid deadlock. Some parallel programming environments provide sophisticated aid in relieving users of this burdensome problem, and in some sense the $M \times N$ problem is the component version of the communication schedule problem.

Framework is the term used in the CCA to describe the execution environment of a component-based application. It is useful to distinguish *direct-connected frameworks* and *distributed frameworks*, although to an application user there is no difference in the interfaces. In direct-connected frameworks, all components in one process live in the same address space and a port invocation then looks like a refined form of library call (see left side of Fig. 2). A set of identical component instances across a given direct-connected framework is called a *cohort* and constitutes a *parallel component*. In Fig. 2 a cohort would be the circles in the same column. All external interactions between this parallel component and the rest of the components occur through port connections, whereas all internal interactions among the cohort occur out-of-band from the CCA framework (e.g. using MPI).

In contrast, components in a distributed framework each run in different sets of processes which may be distributed across multiple machines. In this case, port invocations become a refined form of remote method invocation (RMI), using a network communication library or other form of inter-process communication such as shared memory. The right side of Fig. 2 shows how 3 components are interconnected in a distributed framework. Ideally, a framework would provide both direct and distributed connection mechanisms.

All inter-component communication in distributed frameworks is $M \times N$. On the other hand, in the direct-connection case there is no $M \times N$ because the components are co-located in the same processes. However, $M \times N$ communication can still happen between parallel programs running in separate direct-connected framework instances. In this case the standard approach is to let programs communicate through intermediate *$M \times N$ components* that are instantiated co-located on both sides of a connection. The $M \times N$ components provide a basic API for parallel data transfer and redistribution between two parallel components (or for *self* connections, such as for

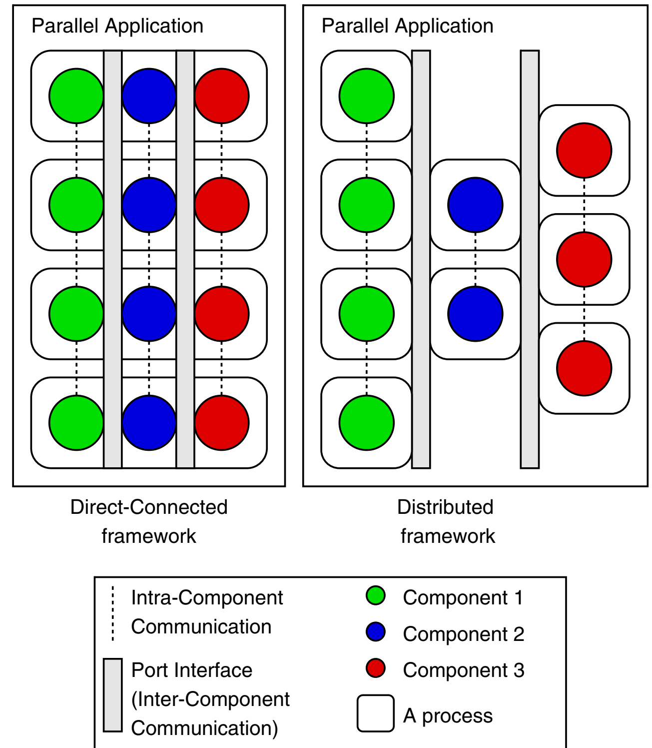


Fig. 2. Direct-connected and distributed frameworks.

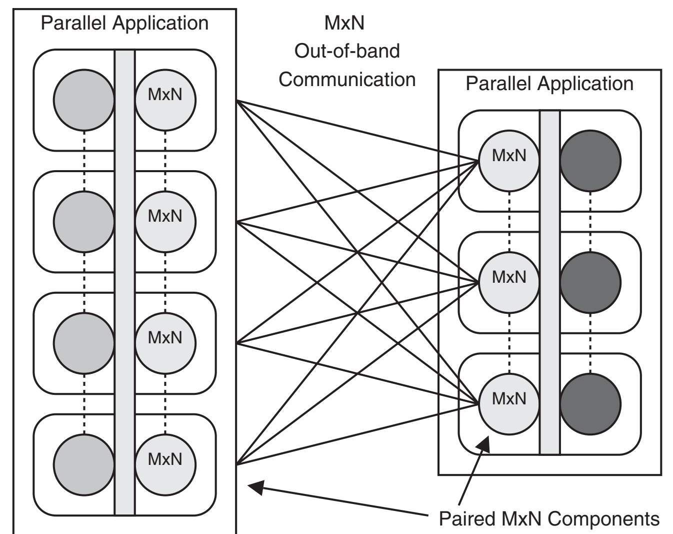


Fig. 3. $M \times N$ component.

transpose operations). The pair of $M \times N$ component instances for a given connection must communicate with each other using an internal mechanism that is out-of-band as far as the CCA specification is concerned. This scenario is shown in Fig. 3. One such implementation of an $M \times N$ component is described in Section 4.1.

In a purely distributed case, this $M \times N$ component cannot be co-located with both sides of the connection. Therefore, it is not possible to use an $M \times N$ component to mediate the communication like it is done in the inter-framework case

described above. Now the M × N communication must be handled by the framework as part of the port abstraction. Ports in distributed frameworks are based on the RMI paradigm, so that idea must be expanded to handle calls between pairs of parallel components. The framework must define all the semantics of this parallel RMI interaction, including various synchronization issues and the transfer of method arguments and the resulting return value(s). Two implementations of distributed frameworks that support parallel components and some basic PRMI capabilities are presented in Sections 4.2 and 4.3.

2.2. Parallel data representation and redistribution

When a data structure is distributed across the processes in a parallel environment, many different layouts are possible for the data. The data transfer and redistribution associated with an M × N or PRMI interaction requires representation of the data layout on both sides of the transaction in a uniform way that is understood by each component and the entity (usually another component or framework) that makes the transfer. The tools described in this paper take two primary approaches to representing the parallel decomposition of arrays.

2.2.1. Linearization

Meta-Chaos [43,17], a coupling library developed at the University of Maryland, introduced the concept of *linearization*. In this method, the elements of the source array are mapped to a linear, one-dimensional arrangement, which constitutes an abstract intermediate representation. The mapping between the source and target templates is therefore implicit and indirect. The application has complete control over the mapping to and from this linear representation.

Linearization is also used in the MPI-IO M × N device developed at Indiana University [6] and in the M × N facility in Argonne National Laboratory's model coupling toolkit (MCT) [34,24]. In the IU M × N system, each process on the receiver side broadcasts to the senders which chunks of data it requires, referencing them to the linearization. At the expense of this small communication overhead, no communication schedule is required. In the MCT M × N device, the root processes in each M × N cohort exchange their respective domain decomposition descriptors, broadcasting to their respective cohorts the remote descriptor. Each processor then computes its communication schedule for the parallel data transfer. These schedules are exposed to the user, and can be kept for future transfers, thus allowing amortization of this overhead.

Linearization simplifies the task of matching a variety of data structures, from multidimensional arrays to trees or graphs. However, the application must often know about how the sender linearized the data to make sense of the de-linearized data at the receiver's end. This typically involves implicit knowledge of the data structure on the sender's side, or the explicit transfer of information about the sender's linearization scheme to the receiver.

2.2.2. Distributed array descriptor

The dominance of arrays in scientific computing calls for a special level of support for distributed array (DA) data structures. In component-based applications, one issue that must be addressed is how components using different DA representations/packages can inter-operate (even without data redistribution issues). In this context, there are questions of both functionality and efficiency or performance. Presently available DA packages offer wide variations in the level of functionality they support. Some provide full-fledged DA objects with a rich set of operations, while others offer little more than a uniform way of organizing the distributed data and leave most or all of the operations to the user, or to separate libraries. An interoperable DA model with significant functionality would require significant additional code for some DA packages to conform, while an easier-to-support low functionality model might not provide users with enough utility to be worth adopting. From the efficiency standpoint, the principle question is what level of interoperability can be achieved without having to perform expensive copies of data between different DA packages?

The "best" answers to these questions depends significantly on the types of codes wanting to exchange DAs. If codes are being newly written, the simplest approach might be to design a full-functionality DA interface based on an existing package and use it in all codes wishing to interoperate. However if the target codes are already using disparate DA packages, it is generally not feasible to rewrite them to a new DA package. On the assumption that this latter situation will dominate the CCA user base initially, we have chosen to begin with a simple, bottom-up approach to DA interoperability. We have developed a distributed array *descriptor* (DAD) [4] that provides global data distribution information and provides access to the local storage of each process's part of the DA; the DAD provides no array operations.

A DAD could be created to describe an existing DA in one package, and then passed to a component using a different DA package. On the receiving side, the DAD provides enough info that the array could be "adopted" by the receiving DA package (i.e. creating a DA in the receiver's representation that references the local memory locations specified in the DAD) if possible, or the data in the array could be accessed directly and operated on. Though there are obvious limitations and concerns associated with this approach, it is a pragmatic way to facilitate the exchange of DA data in the near- to medium-term which will work with most DA packages and representations. The DAD could also be used as the basis for a more sophisticated DA infrastructure, including a higher functionality common DA interface and interconversion adapters. A higher-level approach will become more desirable and more practical for users as component software becomes more widespread.

The general model and much of the specific terminology used in the CCA's DAD interface is largely patterned after the high performance fortran (HPF) [23,28] DA model. Both DAD and HPF distinguish between array *templates* and the actual arrays that hold the data. Templates can be thought of as virtual arrays that specify the logical distribution of the array across the processes. Any number of actual arrays can be *aligned*, or mapped, to a given template, simplifying computation and reuse of communication schedules and other forms of pre-planning for data movement operations. The mapping of actual arrays onto

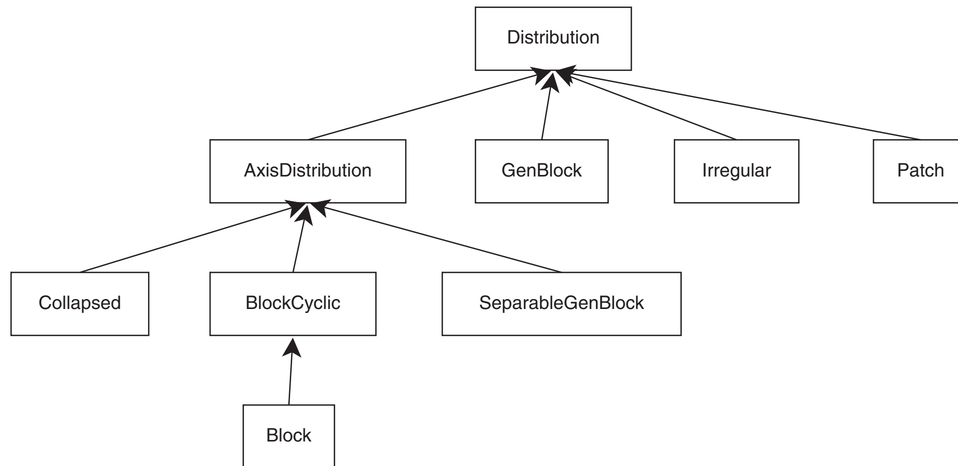


Fig. 4. The distributed array descriptor `Distribution` class hierarchy.

templates is flexible in the HPF model, allowing the expression of complex relationships in the distributions of multiple actual arrays. Thus far in the DAD, the primary focus has been on representing the various distributions before addressing alignment.

The DAD takes an object-oriented approach to the specification of the data distribution of a DA. Fundamentally, DAD distributions are of two kinds: those which can be decomposed into individual distributions for each axis of the multidimensional array, and those which cannot. The `Distribution` class hierarchy is depicted in Fig. 4 and explained below.

The supported distributions include:

- `Collapsed`: All elements of the axis belong to a single process.
- `BlockCyclic`: The elements are divided into regular blocks and distributed cyclically across all processes the axis. If blocks are sized so that each process receives exactly one block, this is often referred to simply as a *block* distribution, and the other extreme of one element per block is commonly known as a *cyclic* distribution. Intermediate sized blocks result in more than one block assigned per process.
- `Block`: A special case of `BlockCyclic` to simplify the common case of simple block distributions.
- `SeparableGenBlock`: A variant of the block distribution introduced by the Global Array [40] package that allows one block per process, but the blocks can be of different sizes. This class is capable of describing many irregular distributions, but since the descriptions are per-axis, it cannot describe completely arbitrary distributions.
- `GenBlock`: The tree-based generalized block distribution provided by InterComm [36,37], in which axes are recursively subdivided. This representation is more general than the `SeparableGenBlock` because it is not just a Cartesian product of irregularly-sized blocks on the individual axes. However it does not offer enough flexibility to represent completely arbitrary distributions.
- `Irregular`: Essentially the `Implicit` distribution type used in HPF that provides complete flexibility in how the data is distributed at the cost of one index element per data element, and potentially expensive queries into the descriptor. The DAD currently supports only one-dimensional distributions of this type, primarily because we have not yet identified applications which need higher-dimensionality representations of this type.
- `Patch`: Decomposes the array into multidimensional rectangular patches of arbitrary size. As with `Irregular`, this representation provides complete flexibility at a cost (in storage and query performance) which is intermediate between `Irregular` and the more specialized `AxisDistribution` classes.

The flexibility of the DAD allows for compact descriptions of many types of distributions. Using the most compact descriptor appropriate for a given distribution usually allows a DA package to provide better performance than is possible for a completely general, structureless linearization, such as the DAD's *implicit* distribution type. However there may be interoperability issues where certain DA packages cannot support all representations (particularly the most flexible ones). If it is not possible to change to a different DA package, the only possibility in such situations would involve copying the DA, redistributing the data into a supported distribution.

2.3. Data redistribution persistence

Often, two components in a coupled simulation must both access a DA over a long period of time using different distributions. The contents of the array can change, so a persistent mechanism for keeping the two local copies identical is required. In this context, we say that the two objects are *coupled*. This coupling can be asymmetrical, for example when one copy (the receiver's or *synchronized* copy), is actually a sampling of the larger, remote, copy. On the other hand, sometimes the data only need be transferred once.

The API of the CCA M × N component (Section 4.1) implements both one-time and persistent communication primitives. The PRMI model (Section 2.4), however, only supports

one-time communication because of the limitations of the RMI paradigm.

2.4. Parallel remote method invocation semantics in CCA

Supporting PRMI is a problem unique to the CCA. Commercial component systems support only serial RMI, having no need for the added complications of massive parallelism and the SPMD model for a component. The CCA programming model requires new semantics, policies, and conventions for invoking parallel methods and appropriately communicating function arguments and results. Synchronization is also a fundamental concern with PRMI, to ensure consistent invocation ordering and the coordination of parallel data arguments, and to avoid deadlocks and other failure modes.

Parallel remote ports are the CCA communication mechanism for distributed parallel components. Parallel remote ports differ from regular CCA ports in that they connect parallel components that are deployed in a distributed fashion over the network. Challenges in defining PRMI semantics include:

- Delivery of arguments. How are the method arguments from the M processes on the calling (client) side delivered to the N processes on the providing (server) side? If M is not equal to N, then *which* of the N providing processes services the invocation for a given set of the M invokers?
- Process participation. The framework must define a policy and implement a mechanism that allows components to make *collective* method calls. Seemingly independent method calls that each of a set of parallel processes make at some point in the execution of a component, must be grouped together and presented as a single effective invocation to the component(s) providing the port implementation. This grouping is primarily a logical one, and does not imply serialization of the invocation. For such a collective invocation to happen there must be a way for the calling component to tell the framework which processes are participating in a given port invocation.
- Concurrency issues. In a distributed framework the components run independently and for efficiency this concurrency should not be unnecessarily inhibited by CCA requirements. For that reason, the calling component cannot arbitrarily block until the providing component returns with the result of the call. This portion of the standard RMI model must be revised.
- Parallel consistency. Several other, low-level details of PRMI must be addressed by the framework. These details relate to the potential need for enforcing synchronization between the processes that participate in a collective call, and dealing with invocation order guarantees [26].

2.4.1. Argument passing in PRMI

In the CCA model, parallel port methods can define two kind of arguments: *simple* and *parallel*.

Simple arguments are the remote version of the regular arguments of a local library call; they are arguments local to each process. Even though the *formal* arguments are the same for all the processes participating in a call, the *actual* values of those arguments may be allowed to vary from process to process depending on the specific semantics of the PRMI model.

Parallel arguments represent data arrays or structures that are decomposed among a set of parallel component processes. Such parallel argument values must be gathered and transferred, and possibly redistributed according to the corresponding $M \times N$ layout (given some policy for handling cases where a one-to-one mapping between processes does not exist). The specific actions are strongly tied to the number of processes in the source and destination components, as well as some semantic knowledge of the given method being invoked, namely the expected form, if any, of the decomposed input data.

In the process of transferring a parallel argument, the framework must know the layout of the data at both the calling and the callee sides. The application must convey this information to the framework prior to the actual transfer of data. This is not a problem on the calling side, because the application specifies this information inside the parallel data arrays object, or maybe in a separate *layout* object such as a DAD that is passed as an extra argument to the port method.

On the callee side, however, the application does not have the opportunity to set the layout prior to the call. Prior to a call, the component is blocked waiting for remote port invocations.

Supporting complex data distribution layouts requires a way for the *provides* component to tell the framework how the data has to be delivered before the transfer of data. Two solutions are being explored in the CCA Forum. One allows the component to specify the layout using a special framework service before the call is received (for example from inside another method, or during initialization routines). The second does not transfer the parallel data in advance. Instead, a reference to the parallel argument is passed to the application, which can later set the layout and trigger the reception.

2.4.2. Concurrency in PRMI

Current direct-connect frameworks generally have a single thread of execution, even though that thread may take the form of an SPMD parallel program. In other words, only one component is active at any logical point in the program. In distributed frameworks simultaneous execution of components is possible and is generally desirable to increase concurrency. The problem is that the port system in distributed frameworks is RMI-based and the calling component will block until the called component has finished servicing the call. To overcome this difficulty, CCA has introduced the notion of *one-way* methods (adopted from CORBA [41]). In one-way methods the calling component continues execution immediately, without waiting for the remote invocation to complete. One-way methods must not have any return value, including arguments with the *out* attribute.

Concurrency involving more than two components can create race conditions, deadlocks, and other problems if the proper measures are not taken.

One scenario in which race conditions can develop is when two components are trying to make an invocation on a third component concurrently. If the calls are served by the processes as soon as the requests arrive, the order in which the calls are served may vary from process to process. This may

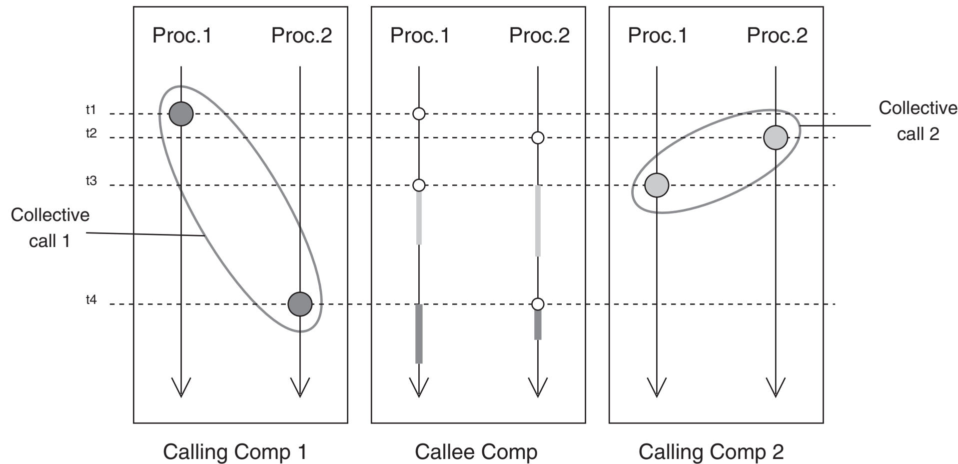


Fig. 5. Avoiding race conditions in PRMI. Components 1 and 2 are making concurrent PRMI calls to the central component. Process 1 of component 1 reaches the invocation point first, but the call is not initiated in the target component until process 2 reaches the invocation point too. Therefore the call arriving from component 2 is served first, and the order of invocation in both processes of the callee component is the same. To simplify this figure, the transmission delays have been removed.

cause incorrect messages and even deadlock if there is internal communication between the processes serving the call. Therefore the framework must ensure that the order of invocation at the destination component is the same in all processes, even when the requests from the calling components arrive intermingled. One way to guarantee this (see Fig. 5) is for the framework to hold the delivery of the call until all the requests from one component arrive. Unfortunately this imposes a synchronization point in the sender.

2.4.3. *PRMI as an extension of RMI*

We have identified two distinct approaches to PRMI. These approaches derive the semantics of PRMI from two other well known paradigms. The first one, $\mathrm{PRMI}_{\mathrm{RMI}}$, derives the semantics of PRMI from the RPC/RMI serial paradigm. The second one, $\mathrm{PRMI}_{\mathrm{SPMD}}$, derives the semantics from the SPMD, non-distributed, model of collective calls. Whereas $\mathrm{PRMI}_{\mathrm{RMI}}$ adds parallelism to the distributed model of RPC/RMI, the $\mathrm{PRMI}_{\mathrm{SPMD}}$ approach adds distribution to the already-parallel model of SPMD.

$\mathrm{PRMI}_{\mathrm{RMI}}$ preserves the character and properties of the serial RPC/RMI on which it is based. In particular, in $\mathrm{PRMI}_{\mathrm{RMI}}$ there is a single conceptual call delivered to the callee component. Both the calling and the callee components participating in a port call do so as whole, indivisible, entities. $\mathrm{PRMI}_{\mathrm{RMI}}$ provides opaqueness (or rather, encapsulation) regarding the remote component: details such as the number of processes that one component has remain hidden to the other side. In this model, a call generated by a single-process component is indistinguishable from a call generated by a ten-process component.

In $\mathrm{PRMI}_{\mathrm{RMI}}$, each simple formal argument is required to have the same actual value across the cohort of processes. Likewise, all the processes of the provides component must return the same actual value. This policy settles the $\mathrm{M} \times \mathrm{N}$ problem for simple arguments because regardless of the number of processes, the value assigned to any given argument or return value is the same.

In this model, the problem of process participation is solved by requiring that all the processes in a component participate in every call. This is a rigid approach, but it is simple and avoids certain deadlock situations that can happen with other models.

2.4.4. *PRMI as an extension of SPMD*

$\mathrm{PRMI}_{\mathrm{SPMD}}$ is an attempt to translate the SPMD style of parallel programming to CCA parallel ports. Exact SPMD semantics cannot be exactly replicated in part because of the issues of the handling of parallel arguments, the required synchronization for deadlock avoidance and the $\mathrm{M} \times \mathrm{N}$ cardinality mismatch. In addition to these, other constraints are inherent to the remote nature of PRMI, such as the inability to pass arguments by reference, and the possibility of network failures.

$\mathrm{PRMI}_{\mathrm{SPMD}}$ is based on *process pairing*. This means that each calling process is paired with a remote, callee process and communicates directly with it. For example, simple arguments are passed directly between paired processes, so they are not required to have the same actual value across the cohort.

Note that process pairing maintains SPMD semantics for cases in which $M \leqslant N$. When $N > M$, however, it won't be possible to pair some of the calling processes. The framework defines the default behavior for those situation.

In contrast with the previous model, in $\mathrm{PRMI}_{\mathrm{SPMD}}$ the calling component decides which processes participate in a collective PRMI call. This flexibility introduces a complication because the framework must provide a way for the user to define the group of processes that are involved in a call.

In a direct-connected framework (or in a simple SPMD program) the application developer can establish process participation through many means. For example, the caller component can communicate the set of participating instances to the callee component by passing an MPI communicator group or some other parallel runtime system object. This information could easily be sent as one of the functional arguments to the method invocation. The framework itself need not be aware of these details because it does not need to know which processes

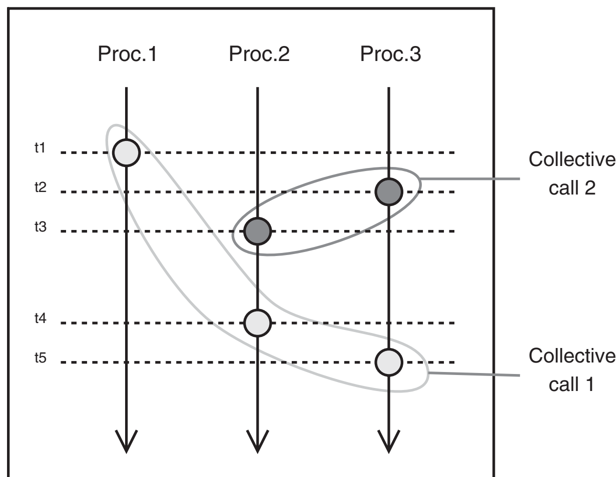


Fig. 6. The synchronization problem: If the PRMI call is delivered as soon as one process reaches the calling point, the remote component will block at t_1 waiting for data from processes 2 and 3, and will not accept the second collective call at t_2 and t_3. The remote component will be blocked indefinitely because processes 2 and 3 will never reach t_4 and t_5 to complete the first call. The solution is to delay PRMI delivery until all processes are ready.

participate in any call. It is the application's task to use the communicator group (or equivalent object) to link together the seemingly independent method calls and interpret them as a single collective invocation. This strategy is the standard procedure for collective calls in SPMD programming.

In distributed frameworks the framework must know which processes in the calling component are participating in the collective call because it must mediate some communication between those processes to redistribute the parallel data and deliver the arguments. Also, process participation on the caller and callee sides must be dealt with distinctly because the processes involved are different, and each side is unaware of the configuration on the other side.

One possible approach to communicate to the framework, the participation group is presented in the distributed CCA architecture (DCA) [5]. In DCA the application programmer can decide process participation on the calling side via an MPI communicator group that is passed as the last argument of the method (this argument is mandatory in all port methods). This solution is more flexible in defining process participation, but is tied to using an MPI-based framework. Regardless, any parallel remote invocation must somehow include sufficient information to identify the participating tasks at the caller and callee sides. For transparency and interoperability, it would be beneficial to include this information generically in each port's method interface specification in some way, for example through the use of descriptors analogous to the DAD.

A problem with allowing participation groups in $\mathrm{PRMI}_{\mathrm{SPMD}}$ is that a *barrier synchronization* is required to ensure that the order of invocation is preserved when different but intersecting sets of processes make consecutive port calls. Fig. 6 shows how a deadlock can occur unless barrier synchronization is performed. This situation is very similar to the race condition previously described, but it involves only two components. Again, the solution to this problem is to delay the delivery of remote invocations until all participating processes have reached the calling point by inserting a barrier before the delivery. In other invocation schemes where *all* processes must participate, the barrier is not required because all calls are delivered in order to the remote component (note that the problem shown in Fig. 6 disappears if process 1 participates in the second call).

In line with SPMD semantics, unpaired processes in the callee side are idle for the duration of the call. Sometimes this is undesired, because it may leave some computational nodes underused. Therefore, the $\mathrm{PRMI}_{\mathrm{SPMD}}$ framework must provide a service to incorporate these processes into the computation. One scenario in which this requirement is evident is one in which a small one-process "driver" component makes an invocation to a large, parallel component. Unpaired process that are used in this way are part of the computation but not part of the PRMI call, except possibly for the reception and sending of parallel arguments.

3. Characteristics of M × N systems

Before describing particular M × N implementations in detail, a list of characteristics will help in understanding relative strengths and weaknesses. Features which have been valued in at least one M × N context or another include

- Language interoperability between components. The languages C, C++, and Fortran are particularly needed in HPC scientific computing, and scripting interfaces like Python, Perl, and Matlab are also valued.
- Support of component concurrency. This means the implementation does not assume or rely upon models like alternating execution of components to assure correctness or integrity of the data.
- Complete generality in the values of M and N. So it should not be required that M and N are commensurate, or that $M < N$, etc.
- Scalability for large values of M and N. This implies that communications between the components is not serialized through a single data management process, and that the creation of communication schedules is not serialized.
- Generic mechanisms for description of data distributions. For dense arrays of values a DAD serves this purpose.
- Distributed components running at different locations. This is the distinction between a direct-connected or distributed framework.
- Asynchronous, non-blocking transfers of data between components to allow overlapping of computation and communications.
- Well-defined semantics for a broad range of possible PRMIs between components.
- No direct dependency on using a particular parallel run-time system like MPI or PVM.

The last item refers to the underlying run-time system and not necessarily the API presented to an end user. Some M × N systems have taken the view that presenting a user with a familiar interface like MPI for carrying out inter-component

Project	Parallel Data	Language	PRMI	Prod. Level
DCA	MPI-based arrays	C	Yes	No
InterComm	Dense arrays	C/Fortran	No	Yes
MCT	Dense/sparse arrays, grids	Fortran	No	Yes
MxN Comp.	SIDL	Babel	No	Yes
SciRun	SIDL	C	Yes	Yes

Fig. 7. M × N projects and features. Some CCA frameworks use Babel [35] for language interoperability, which provides SIDL bindings for C, C++ and FORTRAN.

parallel communications is more desirable than requiring learning a completely new API (Fig. 7).

Another potential feature is direct support for numerical conversions needed for model coupling, such as interpolation. However, this paper deals only with the computer science systems issues of the M × N problem because the numerical ones are often application dependent.

4. Implementations

M × N research within the CCA has ranged from generalized specifications of semantics to implementations of practical component frameworks. Specifications and implementations are likely to evolve as more applications use M × N technologies. Much of this research started from existing partial solutions, and currently no single framework supports the full desired range of M × N capabilities. Rather than trying to create a single framework with all of the capabilities, current work looks to bridge frameworks so that users can access more specialized features as needed.

Given the diversity of the parallel data layouts at the source and destination sides of an M × N transfer, generating an efficient communication schedule to move the various data elements to their correct destinations is difficult. Several tools have developed technology to address this issue, including CUMULVS [30,20,29,31], PAWS [26,3], Meta-Chaos [17,43], and MCT [34,24]. These systems all provide ways to describe the subsets of data that are to be collected and moved to a particular destination process. This is often done by distilling a given data decomposition on a per dimension basis into subregions or sub-sampled patches. While this approach is typically both practical and efficient for most common cases, it can require some complex data transformations in the worst case. This paper deals with only the most fundamental of data exchange and redistribution operations and does not address the numerical capabilities of true model coupling, including spatial and temporal interpolation, energy or flux conservation, data reductions, mesh mediation, and units conversion. These arduous, and typically application-specific tasks are beyond the scope of this initial M × N work.

4.1. M × N *parallel data redistribution components*

A preliminary M × N CCA component specification for direct-connect frameworks was developed using two distinct existing software tools to define and generalize parallel data redistribution—CUMULVS and PAWS. These tools have complementary models of parallel data sharing and coupling. PAWS is built on a "point-to-point" model of parallel data coupling, with matching "send" and "receive" methods on corresponding sides of a data connection. CUMULVS is designed for interactive visualization and computational steering, so provides protocols for persistent parallel data channels with periodic transfers, using a variety of synchronization options. A generalized M × N specification has been developed within CCA that covers both of these connection models with a single unified interface.

The CCA M × N interface has methods that define key operations in parallel data exchange. Parallel components can *register* their parallel data fields by providing a handle to a DAD object (see Section 2.2). The DAD interface provides run-time access to information regarding the layout, allocation and data decomposition of a given distributed data field. The M × N registration process allows a component to express the required DAD information for any dense rectangular array decomposition, and also indicates which access modes for M × N transfers with that data field are allowed (read, write or read/write). Parallel communication schedules can then be defined and applied to define M × N connections using a variety of synchronization options. M × N connections can provide either one-shot transfers or persistent periodic transfers that recur automatically, as defined when the connection is created.

For a given M × N transfer operation, each independent pairwise communication for the overall transfer is initiated when a single instance of the parallel source cohort (1 of *M*) invokes the `dataReady()` method, indicating that the state of its local portion of the data is consistent and "ready" for the transfer. A matching `dataReady()` call at the corresponding destination cohort process (1 of *N*) completes the given pairwise communication. When all such messages have been exchanged, according to the associated communication schedule, then the transfer is considered complete. By breaking down the overall M × N transfer into these independent asynchronous point-to-point transfers, no additional synchronization barriers are required on either side of the transfer. This feature allows efficient implementations for a variety of situations.

M × N connections can be initiated by either the source or destination components, or by a third party controller. Therefore, neither side of an M × N connection need be fully aware, if at all, of the nature of any such connections. This situation expedites the incorporation of existing parallel legacy codes into the component environment. Decisions about the connectivity of parallel data objects can be made dynamically at run-time, as no fundamental changes to the source or destination component codes are strictly necessary.

Several challenges remain to achieve a reliable and efficient M × N implementation. A variety of parallel data layouts must be recognized to decode the location of specific data elements in both the source and destination processes. Initial prototypes have focused on the dense data decompositions supported by the DAD interface (see Section 2.2), including "explicit array patch" decompositions for more arbitrary or optimized meshing schemes. To support more complex data structure

decompositions, a "particle-based" container solution is also under development.

4.2. SciRun2

One way of creating a distributed framework that supports parallel components is by utilizing the code generation process of the interface definition language (IDL) compiler. The IDL compiler can be used to perform the necessary data manipulations and provide consistent behavior for parallel component method invocations. This is the method that is used in SCIRun2 [52], and has been leveraged before for similar problems [27]. Both PRMI and parallel data redistribution primitives are defined in an extension to the SIDL language [32], the Scientific IDL extension developed as part of the CCA project.

In the SCIRun2 SIDL extension, the methods of a parallel component can be specified to be *independent* (one-to-one) or *collective* (all-to-all) with respect to RMI. *Collective* calls are used in cases where the parallel component's processes interoperate to solve a problem collaboratively. Collective calls are capable of supporting differing numbers of processes on the uses and provides side of the call by creating ghost invocations and/or return values. The user of a collective method must guarantee that all participating caller processes make the invocation. The system guarantees that all callee processes receive the call, and that all callers will receive a return value.

To accomplish this functionality, argument and return value data is assumed to be the same across the processes of a component (the component developer must ensure this). The constraint can be relaxed by using a parallel data redistribution mechanism, as described below. *Independent* invocations are provided for normal serial function call semantics. For each of these invocation types, the SIDL compiler generates the glue code that provides the appropriate behavior. This mechanism works regardless of the different numbers of processes with which each component may be instantiated. If the needs of a component change at run-time and the choice of processes participating in a call needs to be modified, then a sub-setting mechanism is engaged to allow greater flexibility.

To enable parallel data redistribution, a DA type was added to the SIDL language. Instances of the DA type can be defined as parameters of a method. At run-time, the instances are set by participating processes to the desired/available part of the global array. The data redistribution is automatically performed when the method invocation is made. The data redistribution mechanism described here is very similar to the one provided by PAWS [3]. For more information on this approach see [14].

Newer version of SCIRun2 seamlessly propagate the $M \times N$ infrastructure through the framework to the user in order to enable greater usability of the benefits added by parallel components. The user can enable parallel "slave" frameworks that represent resources (e.g. clusters, supercomputers) at the disposal of a "master" SCIRun2 framework. Through the SCIRun2 GUI, a user can instantiate, connect, and execute parallel components on a "slave" framework. The $M \times N$ infrastructure described is used to enable the communication of these components as needed by the application.

4.3. DCA: A distributed CCA framework

The DCA is a prototype of a parallel and distributed CCA-compliant framework, based on MPI. The DCA uses MPI [38] constructions such as communicator groups and data types to solve the challenges of a distributed framework, concentrating on the $M \times N$ problems of data redistribution, process participation, and PRMI.

PRMI in the DCA is a natural extension of the semantics of collective calls of the SPMD model of parallel programming. Under these semantics, processes in the calling component communicate one-to-one with processes on the callee component. Simple arguments are transferred independently between the pairs of coupled processes. In line with collective semantics, the framework does not require that simple arguments have the same actual value in all the participating processes. The application may choose to enforce this policy at a higher level in a case-by-case basis.

If, due to an $M \times N$ situation, a process in the calling component cannot be paired with a remote peer, the process' participation is restricted to the sending and receiving of *parallel* arguments, which are subject to data redistribution and are not transferred one-to-one. On the other hand, if a process in the callee component is not paired, it can still be activated (through a special framework service) and used for computational work, and can too send and receive parallel arguments. Unpaired processes on either side do not send or receive simple arguments. In summary, when all the calling processes can be paired (for example, when $M<N$), a PRMI call in the DCA is similar to a local, collective call in a SPMD environment.

DCA supports distribution of one-dimensional parallel arguments, according to layouts similar to those of HPF. By allowing the use of MPI data-types, data can be arranged in non-contiguous buffers, which is useful in situations such as sampling. Parallel arguments are identified in the SIDL file with the special keyword `parallel`.

In the DCA, parallel arguments are not transferred automatically during the method invocation. Instead, a reference to the parallel object is passed to the recipient. Using this reference, the receiving component is able to specify the desired parallel layout before the transfer is started. In addition, the component can then start the reception in a non-blocking manner, to overlap communications and computation. The same principle applies for both `in` and `out` parallel arguments.

Component concurrency is achieved in two ways. First, all CCA *Go* ports[1] are called at startup time, so all components that provide a `Go` port will be started concurrently. Second, components can execute concurrently by using `one-way` methods. Because `out` parallel arguments are handled with references and are not required to be immediately available, `one-way` methods in the DCA can have `out` parallel arguments. An early version of the DCA is more fully described in [5].

[1] `Go` ports are special CCA ports which are recognized by frameworks as a way to start a CCA application running. They are the component equivalent of the `main()` function in a C program.

4.4. InterComm

InterComm [36,37] is a framework for coupling distributed memory parallel programs, which correspond to CCA components, and is mainly targeted at coupled physical simulations. Such programs include those that directly use a low-level message-passing library, such as MPI. To date, InterComm has focused primarily on providing *efficient* communication in the presence of complex data distributions for multidimensional array data structures. InterComm is a descendant of Meta-Chaos [17,43], but adds significant new functionality and provides much better performance. InterComm uses its own DAD, and the CCA Data Group is in the process of completely defining the capabilities and interfaces of its DAD, as described in Section 2.2.2. In InterComm array distributions are classified into two types: those in which entire blocks of an array are assigned to processes, *block distributions*, and those in which individual elements are assigned independently to a particular process, *irregular* or *explicit* distributions. For block distributions, the data structure required to describe the distribution is relatively small, so can be replicated on each of the processes participating in the inter-program communication. For explicit distributions, there is a one-to-one correspondence between the elements of the array and the number of entries in the data descriptor, therefore, the descriptor itself is rather large and must be partitioned across the participating processes. InterComm provides primitives for specifying these types of distributions and has optimized the creation of reusable communication schedules for moving regions of both types from one array to another using point-to-point communication calls. A *linearization* is the method by which InterComm defines an implicit mapping between the source and destination of the transfer distributed by another library or over an unequal number of processes. This *linearization* is a one dimensional intermediate representation, the order of which is dependent on the order of the regions specified for the transfer. InterComm currently supports components written in multiple languages, including C, C++, Fortran77 and Fortran90.

In addition to providing runtime support for determining *what* data is to be moved between simulations, InterComm also provides support for decisions that must be made on *when* data is to be transferred [51]. Instead of requiring each program to contain logic to determine when a data transfer should occur using the communication schedules described above, programs only express potential data transfers with *import* and *export* calls, thereby freeing each program (component) developer from having to know in advance the communication patterns of its potential partners. The actual data transfers take place based on *coordination* rules determined by a third party responsible for orchestrating the entire coupled simulation, consisting of two or more components. The key idea for the coordination specification is the use of *timestamps* to determine when a data transfer will occur, via various types of matching criteria. In addition to the flexibility enabled by a separate coordination specification that makes it relatively easy to add new components and replace components with others having similar functionality, separation of control issues from data transfers enables InterComm to often hide the cost of data transfers behind other program activities.

4.5. The model coupling toolkit

Model coupling [8,16,7,49,53,47,9,1] frequently also requires $M \times N$ coupling because of the wide variance in workload by the individual models. Message-passing parallel models in mutual interaction impose a larger challenge known as the *parallel coupling problem* [33]. The parallel coupling problem comprises both architectural challenges (e.g., sequential vs. concurrent coupling), and parallel data processing challenges in the form of distributed mesh and field data description, parallel data transfer (the $M \times N$ problem), and data transformation (e.g., intermesh interpolation). As an example, the MCT [34,33,24] is a software package that extends MPI to ease implementation of parallel coupling between MPI-based parallel applications. Currently MCT is being employed to couple the atmosphere, ocean, sea ice, and land modules in the community climate system model [13,22], and to implement the coupling API for the Weather Research and Forecasting Model [44,50]. Because the form of model coupling used in climate and weather modeling is well-advanced, MCT internally implements $M \times N$ capabilities at a higher level than the other CCA projects. For example, distributed array descriptors are implemented for both physical mesh and field data, and MCT automatically provides the array data transfers as well as numerical interpolation and communication scheduling with a simpler and higher-level interface in Fortran90.

MCT provides objects and services needed to construct application-specific solutions to the parallel coupling problem. These facilities fall into three broad categories—data description, data transfer, and data transformation.

MCT's description of distributed data exchanged between models includes a physical mesh descriptor, a field data storage class, and a domain decomposition descriptor. Each of these classes works off of MCT's linearization approach to describing both field and mesh data. The data object for describing physical grids capable of supporting grids of arbitrary dimension and unstructured grids, and is capable of supporting masking of grid elements (e.g., land/ocean mask). The field data object is a standard intermediate representation into which model data is copied for transfer and transformation, and is similar to a Trilinos multivector. The decomposition descriptor employs a combination of the linearization and explicit approaches described in Sections 2.2.1 and 2.2.2.

MCT's $M \times N$ system comprises a singleton lightweight registry class and two types of communication schedule objects. The model registry defines the MPI processes on which each model's cohort resides, and a process ID look-up table that obviates the need for inter-communicators between distinct processor pools. Communications schedule objects exist to coordinate one-way $M \times N$ transfers and two-way $M \times N$ exchanges. These schedules are computed by each model by calling and MCT routine and providing an integer ID for the remote model, and its local domain decomposition descriptor. Blocking and non-blocking library routines are available to accomplish the $M \times N$ transfer operations.

Data transformation operations supported by MCT include intermesh interpolation, time averaging and accumulation, spatial integration and averaging to diagnose and enforce conservation of intermodel fluxes, and merging of data from multiple source models. MCT provides three classes and a variety of library routines to support these operations.

Intermesh interpolation in MCT is supported as a sparse matrix-vector multiply, which is an immediate corollary to MCT's linearization approach to model data. Two classes are provided to support this type of linear transformation. One class stores in COO format the non-zero matrix elements for the transformation and the other encapsulates all the computation and communication information needed to perform the simultaneous parallel linear transformation of multiple fields in a cache-friendly fashion. Conservation of fluxes is of importance in many coupled models and MCT provides spatial integral and averaging facilities that include paired integrals and averages for use in conservation of global flux integrals in intermesh interpolation.

Often, models that interact will have differing timesteps, or may be coupled at a frequency of multiple timesteps, creating a need for time integration of data. Averaging of state data and accumulation of fluxes form a popular approach to implementing data coupling in climate models, due to the separation of timescales between the ocean and other components of the climate system. MCT provides a class for storing running averages and integrals over an accumulation period, and a library routine that performs the accumulation.

In some coupled models, one subsystem may have and interface with the rest of the system that will require data from multiple models at some of the locations on its interface. For example, the atmosphere in a coupled climate model's interface with the Earth's surface will in places require as input a blend of land, ocean, and sea ice data. MCT provides a variety of library routines for merging data from multiple source models for use by a particular model.

MCT supports both sequential and concurrent couplings (and combinations thereof), and can support coupling of components running as multiple executable images if the implementation of MPI used supports this feature. MCT is implemented in Fortran90. The MCT programming model is scientific-programmer-friendly, consisting of F90 module use to declare MCT-type variables and invocation of MCT routines to create couplings [42,33,13]. Work is in progress to employ the Babel language interoperability tool to create MCT bindings for other programming languages.

5. Related work

The Data Reorganization Interface Standard (DRI-1.0) [15] is the result of a DARPA-sponsored effort targeted at the military signal and image processing community. DRI *datasets* are arrays of up to three dimensions (support for higher dimensions is optional). Block and block-cyclic *partitions* are supported, and local memory *layouts* are distinguished from the data distribution. The data types specified in the DRI standard include float, double, complex, double complex, integer, short, unsigned short, long, unsigned long, char, unsigned char, and byte. Reorganization operations in DRI are collective, and are handled at a low level. The user provides send and receive buffers and repeatedly calling DRI get/put operations until the operation is complete. The specification is language independent, but a C binding is included. Relative to the work discussed in this paper, the DRI can be thought of as a specialized and low-level DAD and M × N component.

$XChange_{m\times n}$ [1] is a middleware infrastructure for coupling components in distributed applications. $XChange_{m\times n}$ uses the publish/subscribe paradigm to link interacting components, and deal specifically with dynamic behaviors, such as dynamic arrivals and departures of components and the transformation of data "in-flight" to match end point requirements.

Another tool for model coupling is the distributed data broker (DDB) [16], which is a general purpose tool from UC Berkeley for coupling multiple parallel models that exchange large volumes of data. The DDB provides a mechanism for coupling codes with different grid resolutions and data representations.

Roccom [25] is an object-oriented software framework for high performance parallel rocket simulation. Roccom enables coupling of multiple physics modules, each of which models 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 Roccom so that other modules can share them with the permission of the owner module.

One commercial effort is MpCCI [19]. MpCCI provides support for running coupled simulation components within a single MPI environment, and also provides methods for mapping between grids in different components. However, the system requires substantial changes to individual model codes, although direct support for a limited set of commercial multiphysics codes is provided. Earlier versions of MpCCI required running in a single MPI environment, but more recent versions relax the single MPI environment constraint, employing a coupling server similar to the MCT flux coupler component.

6. Future research plans

As the technology for M × N and PRMI evolves beyond its fundamental data exchange capabilities and execution semantics, a complete spectrum of additional parallel model coupling functions is required to address the parallel coupling problem identified in Section 4.5. Such features include proper handling or hooks for: application discovery and matchmaking, enforcing parallel data dependencies, orchestration of coupling communications, interpolation in time and space, data translation and units conversion, and parallel process model optimization. Without integration of these crucial functions into the user's scientific simulation environment, it will be impractical to fully realize the potential of multiphysics coupled models for production-scale scientific discovery. This integrated collection of capabilities, for assembling and executing coupled scientific codes, is referred to as "Parallel Coupling Infrastructure," or PCI.

Development of PCI technology will require the incorporation of a diverse set of new and existing software tools that must work together seamlessly toward constructing and coordinating coupled simulation models. For example, several global meta-data registration tools exist for discovery in certain execution environments, e.g. the MDS repository for Globus [12]. These services can be applied for PCI application discovery, yet the specific meta-data associated with proper rendezvous and matchmaking must be carefully designed and populated. In addition, some mechanism must be designed for specifying the high-level data dependencies within a coupled model composition. An overall coupling service must coordinate the handshaking and synchronization of various model codes, to instigate connections and then orchestrate their cooperative symbiotic execution. This service might maintain efficient execution by dynamically adjusting communication schedules and possibly performing load balancing adjustments.

With respect to interpolation, which is applied to match up disparate data meshes and reconcile wide ranges of simulated time scales, there are a variety of domain-specific algorithms and techniques. Many schemes are available for spatial interpolation, to migrate data values from one spatial grid to another while conserving energy and fluxes. Similarly, efficient approaches to temporal interpolation attempt to optimize model algorithms to help match simulated execution times across time scales [18]. A number of curve-matching techniques can also be applied to extrapolate intermediate values within a time frame gap. Exacerbating the high level of complexity associated with these various spatial and temporal interpolation schemes is the fact that scientists are often *religious* about the choice of scheme for their particular domain. As such, the prudent approach for PCI is not to reinvent any interpolation schemes but rather provide generalized "hooks" for wrapping up the available algorithms.

Related to interpolation, though somewhat simpler, is the important task of units conversion, wherein specific data values are translated into the proper units to verify meaningful numerical results when incorporated into another model. The implementation for these translations are often simple linear functions that can be applied on a per element basis.

A subtle yet critical aspect of PCI is the optimization of the overall parallel process model. Many simulation codes can be organized as distinct collections of SPMD ("single program multiple data") process groups, however more complex coupled simulations might often be assembled in more flexible MPMD ("multiple program multiple data") arrangements. Such MPMD process organizations allow a single set of computing resources to be shared among different sub-models. This can occur over time, as different phases of execution trade back and forth, keeping the data to be coupled readily available in the memory of the shared processors. Or, a single larger processor allocation can be split into several subgroups, each one executing a distinct physics model. Determining the proper arrangement, or combinations thereof, requires a global understanding of the relative iteration timings and the changing data dependencies among the set of coupled models. If new coupled performance methodologies are designed, then either composition time or run time optimizations can be made to streamline the overall execution.

An evolutionary design path must be constructed to allow new and existing codes to incorporate progressively sophisticated mechanisms for model coupling. In its simplest form, different models can perform "I/O Driven" coupling by the mere exchange of data files, allowing unmodified execution flow within a given model. More elaborate schemes might allow a coordinated swapping of key memory regions, such as with time-shared MPMD codes. The most efficient and flexible form of coupling involves live parallel communication, where explicit messages are scheduled to transfer specific data elements from source to destination processor. By encapsulating these variations within a single PCI interface specification, it will be possible to incrementally develop the coupling capabilities for a given multiphysics simulation.

While this technology relates closely to recent studies and advances in "Scientific Workflow" and software experiment management, the needs and challenges of PCI surpass those for basic workflow. In a PCI environment, there are many *dynamic* relationships that exist or are created amongst the individual parallel codes at run time. A generalized PCI solution will avoid static hard-coding of the interdependencies between coupled models, allowing more flexibility in choosing precisely when and how parallel simulations will attach and share their data. This late binding of models preserves the "anonymity" of each individual model, enabling coupling with a variety of known *and* unforeseen physics models that might share the same internal data semantics. Then, rather than blindly relying on simple metrics for the scheduling of data coupling communications, PCI must perform cross-model data dependency analysis to ensure that complicated or cascading interdependencies are correctly maintained among all executing models. The missing piece in traditional workflow systems is a lack of accounting for this entanglement of decomposition information with data in the parallel data models. By building the new PCI capabilities on top of $M \times N$ and PRMI technology, these issues can be properly encapsulated and automated in the resulting development/execution environment.

7. Summary

Exchanging elements among disparate parallel or distributed data structures is merely the beginning of true technology for parallel model coupling and transparent data sharing. Depending on the nature of the actual data structures involved, significant data translations could be needed beyond the simple $M \times N$ mapping of data elements. If the source and destination data use different meshes or spatial coordinate representations, or are computed in different units or at different time frames, then several additional data translation and conversion components will be required to fully transform and share semantically comparable parallel data.

A wealth of interpolation and sampling schemes are available for translating data among desired spatial or temporal formats. Historically, such schemes carry with them an almost religious stigma, and there is much debate among scientists

on the merits of one scheme over another. We hope to extend our collection of interface specifications to include appropriate hooks for supporting various generic data transformations and conversions. Given sufficient flexibility in the arguments for these interfaces, a wide range of implementations can be built to cover common interpolation or conversion algorithms. Because of the wide variety of space and time discretizations used in scientific computing, there will always be a need to allow user created inter-component data modifications.

To utilize the resulting sequence of data transformations and data redistributions, a pipeline of components can be assembled. An important pragmatic issue that arises with such pipelining is how efficiently redistribution functions compose with one another. Techniques must be explored to operate on data *in place* and avoid unnecessary data copies. *Super-component* solutions could also be explored for some common cases by combining several successive redistribution and translation components into a single optimized component. This will require a uniform way of describing data distributions, such as the DAD for arrays, and with more difficulty, a uniform way of describing transformations.

In the near term, the primary research goal of this effort will be to develop higher-level operations on top of these fundamental $M \times N$ data transfer functions. The complexity of the current port interfaces alludes to the low-level "assembly-language" nature of our current understanding of this technology. More user-friendly simplifications will be developed for the most common operations, to make this technology more readily available and practical for everyday usage.

$M \times N$ technology is only now starting to emerge as an important tool for composing parallel components and even complete applications into larger cross-disciplinary simulations. $M \times N$ connections are needed for more than just computations: dynamically inserting data from large sensor arrays into a running computation (such as weather modeling) or accessing data in parallel from distributed scientific databases will mean connecting non-computational components with computational ones. The basic issues of the meaning of PRMI, efficient redistribution of data, and shielding users from the complexities of parallel codes interacting at run-time are the same.

Acknowledgments

This work is supported by National Science Foundation Grants CDA-0116050 and EIA-0202048, and by the U.S. Department of Energy's Scientific Discovery through the Advanced Computing (SciDAC) initiative, through the Center for Component Technology for Terascale Simulation Software, of which Argonne, Lawrence Livermore, Los Alamos, Oak Ridge, Pacific Northwest, and Sandia National Laboratories, Indiana University, and the University of Utah are members.

Research at Oak Ridge National Laboratory is supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

[1] H. Abbasi, M. Wolf, K. Schwan, G. Eisenhauer, A. Hilton, XChange: coupling parallel applications in a dynamic environment, in: IEEE International Conference on Cluster Computing, 2004.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, B. Smolinski, Toward a common component architecture for high-performance scientific computing, in: Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing, 1998.

[3] P. Beckman, P. Fasel, W. Humphrey, S. Mniszewski, Efficient coupling of parallel applications using PAWS, in: Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computation, 1998.

[4] D.E. Bernholdt, CCA distributed array descriptor (DAD), ⟨http://www.cca-forum.org/~data-wg/dist-array/⟩.

[5] F. Bertrand, R. Bramley, DCA: a distributed CCA framework based on MPI, in: Proceedings of HIPS 2004, Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, IEEE Press, Santa Fe, NM, 2004.

[6] F. Bertrand, Y. Yuan, K. Chiu, R. Bramley, An approach to parallel $M \times N$ communication, in: Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium, Santa Fe, NM, 2003.

[7] T. Bettge, A. Craig, R. James, V. Wayland, G. Strand, The DOE parallel climate model PCM: the computational highway and backroads, in: V.N. Alexandrov, J.J. Dongarra, C.J.K. Tan (Eds.), Proceedings of the International Conference on Computational Science (ICCS) 2001, Lecture Notes in Computer Science, vol. 2073, Springer, Berlin, 2001, pp. 148–156.

[8] F.O. Bryan, B.G. Kauffman, W.G. Large, P.R. Gent, The ncar csm flux coupler, NCAR Technical Note 424, NCAR, Boulder, CO, 1996.

[9] California Institute of Technology, Center for simulation of dynamic response of materials homepage, ⟨http://www.cacr.caltech.edu/ASAP/⟩, 2003.

[10] CCA Forum, CCA Forum homepage, ⟨http://www.cca-forum.org/⟩, 2004.

[11] Center for Component Technology for Terascale Simulation Software (CCTTSS), CCTTSS SciDAC Center web page, ⟨http://www.cca-forum.org/ccttss/⟩, 2004.

[12] B. Clifford, Globus monitoring and discovery, in: Proceedings of GlobusWORLD 2005, Boston, MA, 2005.

[13] A.P. Craig, B. Kaufmann, R. Jacob, T. Bettge, J. Larson, E. Ong, C. Ding, H. He, cpl6: the new extensible high-performance parallel coupler for the community climate system model, Internat. J. High Perf. Comput. Appl. 19 (2005) 309–328.

[14] K. Damevski, Parallel RMI and M-by-N data redistribution using an IDL compiler, Master's Thesis, The University of Utah, May 2003.

[15] Data Reorganization (DRI) Forum, Document for the data reorganization interface (dri-1.0) standard, ⟨http://www.data-re.org/⟩, September 25, 2002.

[16] L.A. Drummond, J. Demmel, C.R. Mechoso, H. Robinson, K. Sklower, J.A. Spahr, A data broker for distributed computing environments, in: Proceedings of the International Conference on Computational Science, 2001, pp. 31–40.

[17] G. Edjlali, A. Sussman, J. Saltz, Interoperability of data-parallel runtime libraries, in: International Parallel Processing Symposium, IEEE Computer Society Press, Geneva, Switzerland, 1997.

[18] W.R. Elwasif, D.B. Batchelor, D.E. Bernholdt, L.A. Berry, E.F. D'Azevedo, W.A. Houlberg, E.F. Jaeger, J.A. Kohl, S. Li, Coupled fusion simulation using the common component architecture, in: Computational Science—ICCS 2005 Fifth International Conference, Proceedings, Part I, Lecture Notes in Computer Science, vol. 3514, Springer, Atlanta, Georgia, USA, 2005, pp. 372–379.

[19] Fraunhofer Institute for Algorithms and Scientific Computing (SCAI), MpCCI: multidisciplinary simulations through code coupling, ⟨http://www.scai.fraunhofer.de/mpcci.html⟩, 2005.

[20] G.A. Geist, J.A. Kohl, P.M. Papadopoulos, CUMULVS: providing fault tolerance, visualization and steering of parallel applications, Internat. J. High Perf. Comput. Appl. 11 (3) (1997) 224–236.

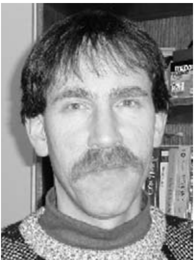
[21] J. Glimm, D. Brown, L. Freitag, Terascale Simulation Tools and Technologies (TSTT) Center, ⟨http://www.tstt-scidac.org/⟩, 2001.
[22] L. Harper, B. Kauffman, Community climate system model, ⟨http://www.ccsm.ucar.edu/⟩, 2004.
[23] High Performance Fortran Forum, High Performance Fortran language specification, Sci. Programming 2 (1–2) (1993) 1–170.
[24] R. Jacob, J. Larson, E. Ong, M × n communication and parallel interpolation in ccsm3 using the model coupling tookit, Internat. J. High Perf. Comput. Appl. 19 (2005) 293–308.
[25] X. Jiao, M. Campbell, M. Heath, Roccom: an object-oriented, data-centric software integration framework for multiphysics simulations, in: Proceedings of the 2003 International Conference on Supercomputing, ACM Press, New York, 2003, pp. 358–368.
[26] K. Keahey, P. Fasel, S. Mniszewski, PAWS: collective interactions and data transfers, in: Proceedings of the High Performance Distributed Computing Conference, San Francisco, CA, 2001.
[27] K. Keahey, D. Gannon, PARDIS: a parallel approach to CORBA, in: Proceedings of the High Performance Distributed Computing Conference, 1997, pp. 31–39.
[28] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., M. Zosel, The High Performance Fortran Handbook, MIT Press, Cambridge, MA, 1994.
[29] J.A. Kohl, High performance computers: innovative assistants to science, ORNL Review, Special Issue on Adv. Comput. 30 (3/4) (1997) 224–236.
[30] J.A. Kohl, G.A. Geist, Monitoring and steering of large-scale distributed simulations, in: IASTED International Conference on Applied Modeling and Simulation, Cairns, Queensland, Australia, 1999.
[31] J.A. Kohl, P.M. Papadopoulos, A library for visualization and steering of distributed simulations using PVM and AVS, in: High Performance Computing Symposium, Montreal, CA, 1995.
[32] S. Kohn, G. Kumfert, J. Painter, C. Ribbens, Divorcing language dependencies from a scientific software library, in: Proceedings of the 11th SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, 2001.
[33] J. Larson, R. Jacob, E. Ong, The model coupling toolkit: a new fortran90 toolkit for building multi-physics parallel coupled models, Internat. J. High Perf. Comput. Appl. 19 (2005) 277–292.
[34] J.W. Larson, R.L. Jacob, I.T. Foster, J. Guo, The model coupling toolkit, in: V.N. Alexandrov, J.J. Dongarra, B.A. Juliano, R.S. Renner, C.J.K. Tan (Eds.), Proceedings of the International Conference on Computational Science (ICCS) 2001, Lecture Notes in Computer Science, vol. 2073, Springer, Berlin, 2001, pp. 185–194.
[35] Lawrence Livermore National Laboratory, Babel homepage, ⟨http://www.llnl.gov/CASC/components/babel.html⟩, 2004.
[36] J. Lee, A. Sussman, Efficient communication between parallel programs with InterComm, Technical Report CS-TR-4557 and UMIACS-TR-2004-04, University of Maryland, Department of Computer Science and UMIACS, January 2004.
[37] J.-Y. Lee, A. Sussman, High performance communication between parallel programs, in: Proceedings of 2005 Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models (HIPS-HPGC 2005), IEEE Computer Society Press, Silver Spring, MD, 2005appears with the Proceedings of IPDPS 2005.
[38] Message Passing Interface Forum, MPI: a message-passing interface standard, Internat. J. Supercomputer Appl. High Perf. Comput. 8 (3/4) (1994) 159–416.
[39] Microsoft Corporation, Distributed component object model, ⟨http://www.microsoft.com/com/tech/dcom.asp⟩, 2004.
[40] J. Nieplocha, R.J. Harrison, R.J. Littlefield, Global arrays: a non-uniform-memory-access programming model for high-performance computers, J. Supercomputing 10 (2) (1996) 169.
[41] Object Management Group, CORBA component model, ⟨http://www.omg.org/technology/documents/formal/components.htm⟩, 2002.
[42] E. Ong, J. Larson, R. Jacob, A real application of the model coupling toolkit, in: C.J.K. Tan, J.J. Dongarra, A.G. Hoekstra, P.M.A. Sloot (Eds.), Proceedings of the 2002 International Conference on Computational Science, Lecture Notes in Computer Science, vol. 2330, Springer, Berlin, 2002, pp. 748–757.
[43] M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman, J. Saltz, Runtime coupling of data-parallel programs, in: Proceedings of the 1996 International Conference on Supercomputing, Philadelphia, PA, 1996.
[44] D. Schaffer, Coupling implementation of the wrf i/o api, ⟨http://www-ad.fsl.noaa.gov/ac/schaffer/mct_wrf_io_api.html⟩, 2004.
[45] Sun Microsystems, Enterprise JavaBeans downloads and specifications, ⟨http://java.sun.com/products/ejb/docs.html⟩, 2004.
[46] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press, New York, 1999.
[47] G. Toth, I.V. Sokolov, K.J. Kane, T.I. Gombosi, D.L. de Zeeuw, A.J. Ridley, O. Volberg, K.C. Hansen, W.B. Manchester, I.I. Roussev, Q.F. Stout, K.G. Powell, Space weather modeling framework: modeling the Sun-Earth system faster than real time, AGU Fall Meeting Abstracts, 2004, B325+.
[48] U.S. Dept. of Energy, SciDAC Initiative homepage, ⟨http://www.osti.gov/scidac/⟩, 2003.
[49] S. Valcke, A. Caubel, R. Vogelsang, D. Declat, Oasis3 ocean atmosphere sea ice soil user's guide, Technical Report TR/CMGC/04/68, CERFACS, Toulouse, France, 2004.
[50] WRF Oversight Board, Weather Research and Forecasting Model, ⟨http://www.wrf-model.org/⟩, 2000.
[51] J. Wu, A. Sussman, Flexible control of data transfers between parallel programs, in: Proceedings of the Fifth International Workshop on Grid Computing—GRID 2004, IEEE Computer Society Press, Silver Spring, MD, 2004.
[52] K. Zhang, K. Damevski, V. Venkatachalapathy, S. Parker, SCIRun2: a CCA framework for high performance computing, in: Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), IEEE Press, Santa Fe, NM, 2004.
[53] S. Zhou, Coupling earth system models: an ESMF-CCA prototype, ⟨http://webserv.gsfc.nasa.gov/ESS/esmf_tasc/⟩, 2003.

Felipe Bertrand is a Software Engineer at Delmos Space, a private Spanish company operating in aerospace activities. His interests are scientific computing and high-performance component systems. He received his Ph.D. from Indiana University and his master's degree in electrical engineering from the Universidad Politecnica de Madrid.

Randall Bramley completed his B.S. degree in mathematics and computer science in 1981, a M.S. in computational mathematics in 1985, and a Ph.D. in computer science from the University of Illinois in 1989. He was a senior research scientist at the Center for Supercomputing Research and Development at the University of Illinois until 1992, at which time he joined the IU Computer Science Department. He is currently director of the Scientific Computing Program at IU.

David E. Bernholdt is a Senior R&D Staff Member in the Computer Science and Mathematics Division of Oak Ridge National Laboratory. Prior to joining ORNL, he held positions at the Northeast Parallel Architectures Center at Syracuse University and a postdoctoral fellowship at the Pacific Northwest National Laboratory. David received his Ph.D. in 1993 from the University of Florida. His research focuses on software environments for high-performance scientific computing.

James Kohl is a Research Scientist at Oak Ridge National Laboratory in the Computer Science and Mathematics Division. He received his Ph.D. in Electrical/Computer Engineering from University of Iowa (1994), an M.S. in Electrical Engineering (1989) and a B.S. in Computer/Electrical Engineering (1988) from Purdue University. Kohl has been involved in a number of parallel computing and visualization projects for scientific simulation, at ORNL, the IBM T. J. Watson Research Center (1992), and Argonne National Laboratory (1983-1990). Notable projects include the Common Component Architecture (CCA) and "M $\times$ N" Parallel Data Redistribution, the CUMULVS system for interactive visualization, coupling, computational steering and fault tolerance, PVM, MatView and XPVM. Kohl is a member of the Order of the Engineer, Eta Kappa Nu, Tau Beta Pi and Mensa International.



Alan Sussman is an Assistant Professor in the Computer Science Department at the University of Maryland, College Park. His research interests include compilers and runtime systems for parallel computers, high performance database and I/O systems, coupled multiphysics simulations, and medical informatics. He received his Ph.D. in computer science from Carnegie Mellon University in 1991 and his B.S.E. in Electrical Engineering and Computer Science from Princeton University in 1982.

Jay Larson is a software engineer in the Mathematics and Computer Science division at Argonne National Laboratory and is a senior fellow in the Computation Institute at the University of Chicago. He has published in the fields of nonlinear dynamics, plasma physics, climate, weather forecasting, and high-performance computing. In recent years his work has focused primarily on the development of high-performance software infrastructure for the earth system modeling community, most notably as co-lead developer of the Model Coupling Toolkit. He holds a B.A. in mathematics and physics from Drake University and a Ph.D. in plasma physics from the College of William and Mary in Virginia.

Kostadin Damevski is a Research Assistant at the Scientific Computing Institute at the University of Utah. He is working towards his Ph.D. in Computer Science from the School of Computing at the University of Utah. His research interests include development of software and tools for high-performance scientific computing.