Data Redistribution and Remote Method Invocation in Parallel Component Architectures

Felipe Bertrand
Randall Bramley
Indiana University
{febertra,bramley}@indiana.edu

Kostadin B. Damevski
Scientific Computing and Imaging Institute
University of Utah
damevski@cs.utah.edu

James A. Kohl
David E. Bernholdt
Computer Science and Mathematics Division
Oak Ridge National Laboratory
{kohlja,bernholdtde}@ornl.gov

Jay W. Larson
Mathematics and Computer Science Division
Argonne National Laboratory
larson@mcs.anl.gov

Alan Sussman
UMIACS and Department of Computer Science
University of Maryland
als@cs.umd.edu

***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.**

I. 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. Also, there are major application modeling and applied mathematics problems involved in combining multiple models.

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" processors and the other executes on a potentially distinct set of "N" processors, so for a data object to be shared between the two simulations a mapping between corresponding data elements must be made, and software infrastructure

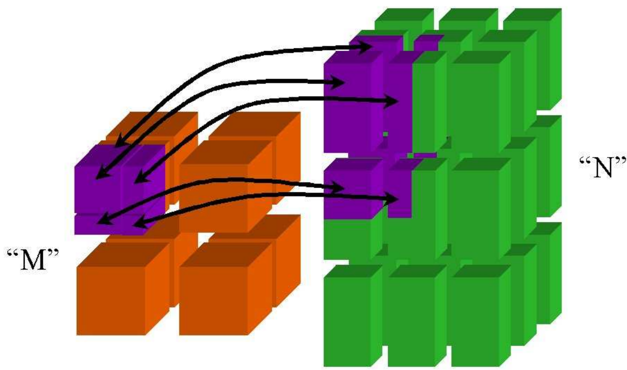

Fig. 1. The "M×N" Problem

must provide for the scheduling, synchronization and transfer of data elements defined by the mapping. This arrangement is illustrated in Figure 1, where M=8 and N=27, and multiple processors on the N side must export data values for each single processor 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.

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 processors. 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 processors is assigned to a new set of processors. This reassignment may try to achieve load balance on the receiving set of processors, but the essential problem is how to specify and define which receiving processor(s) gets data from which sending processor(s).

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) [1]. This methodology has been successful in the business software domain (e.g. CORBA [2], DCOM [3] and Enterprise JavaBeans [4]), and is migrating into the scientific domain in projects such as the Common Component Architecture (CCA) [5]. 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 [6] and the Center for Component Technology for Terascale Simulation Software [7] (part of the Scientific Discovery through Advanced Computing (SciDAC) program [8]), 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 [9] 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 V describes related work in M×N /coupling research. Section II 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 IV presents a survey of component-based M×N solutions that have been generated in conjunction with the CCA effort. Finally, Section VI concludes and looks toward future work in this area.

II. Overview

A. 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.

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, so 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 so a 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 [10].

In most scientific computing, the decomposition of the data in a problem to be solved using parallel computing is based on *domain* decomposition, while the decomposition provided by software component methodology is typically a *functional* decomposition. 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.

In parallel computing a *communication schedule* is the 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×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. In this context, 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 Figure 2). A set of identical component instances across a given direct-connected framework is called a *cohort* and constitutes a *parallel component*. In Figure 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

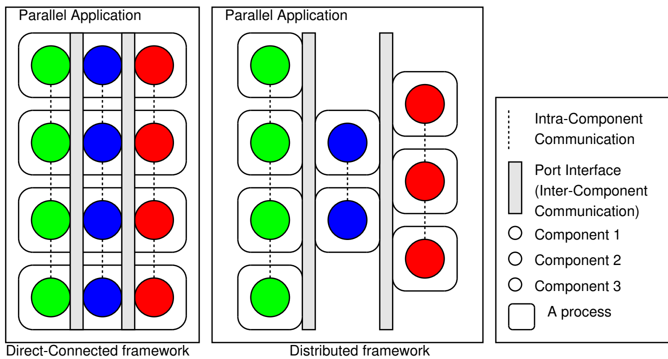

Fig. 2. Direct-connected and Distributed Frameworks

Remote Method Invocation (RMI), using a network communication library or other form of interprocess communication such as shared memory. The right side of Figure 2 shows how 3 components are interconnected in a distributed framework. Ideally, a framework would provide both direct and distributed connection mechanisms.

In the direct-connection case, M×N communication can happen between parallel programs running in separate framework instances. In this case the standard approach is to let programs communicate through intermediate *M×N components* that are instantiated co-located on both sides of a connection. The M×N components provide a basic API for parallel data transfer and redistribution between two parallel components (or for *self* connections, such as for transpose operations). The pair of M×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 figure 3. One such implementation of an M×N component is described in Section IV-A.

In the distributed case, an M×N component cannot in general be colocated to mediate the communication because components are distributed and the two communicating parallel components are on different machines. Here the M×N communication must occur inside 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 PRMI 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 IV-B and IV-C.

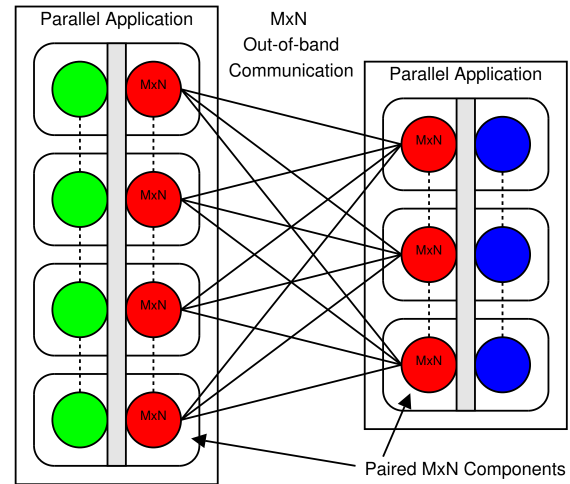

Fig. 3. M×N Component

B. Parallel Data Representation

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.

1) Linearization: Meta-Chaos [11], [12], 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 the 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 [13]. In this 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 overhead, no communication schedule is required in this system.

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 in order 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) Distributed Array Descriptor: The dominance of arrays in scientific computing calls, in general, 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 distributed array (DA) representations/packages can interoperate (even without data redistribution issues).

A top-down solution might be to standardize on a single DA package and force all components to either use it directly, or develop adapters that can convert between the *standard* and *native* DA representations. This solution would also require significant modifications to existing code, and the use of adapters might have serious consequences for performance and memory requirements (where in-place conversion between DA representations is not possible). Therefore, as a simpler starting point, the CCA has chosen to take a bottom-up approach, developing a *distributed array descriptor* (DAD) that provides global data distribution information and provides access to the local storage of each process's patch(es) of the distributed array.

Such a descriptor can be used to facilitate the conversion between DA representations, allowing the use of $2N$ distinct converters to/from the DAD's intermediate representation rather than N^2 converters directly coupling individual DA representations or packages. However, many components do *not* need to interact with a completely functional DA package, but rather they just need to be able to access the memory locations constituting the DA to acquire a specification of the global layout of the data. An example of this might include data-parallel components, which perform operations on their local portion of a distributed array. For parallel M×N data redistribution, direct access to the DA's local memory is often sufficient (on the receiving side, the DA may first have to be allocated using the DA package). Though there may be philosophical objections to short-circuiting the DA package's interface in this way, this is a practical approach to facilitating interoperability of components using distributed arrays in the near to medium term. The approach is highly pragmatic, and will work for most DA packages and representations we have investigated.

The DAD interface supports the description of dense multidimensional arrays. Work on other data structures, such as sparse matrices and particle fields is planned.

The general model of distributed arrays, as well as much of the specific terminology used in the CCA's DAD interface (version 1) [14] is largely patterned after the High Performance Fortran (HPF) [15], [16] distributed array 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 processors. 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 templates is also extremely flexible in the HPF model, allowing the expression of complex relationships in the distributions of multiple actual arrays.

With one exception, the types of distributions supported by the CCA distributed array descriptor are per-axis, so that the complete data distribution requires the specification of a distribution on each axis of the array. The supported distributions include:

- Collapsed: all elements of the axis belong to a single process.
- Block-Cyclic: 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.
- Generalized Block: a variant of the block distribution introduced by the Global Array [17] package that allows one block per process, but the blocks can be of different sizes.
- Implicit: a 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.

There is one additional distribution type supported in the CCA DAD which is global to the entire array rather than axis-specific:

- Explicit: allows completely arbitrary distributions to be specified as a collection of (multidimensional) rectangular patches, each assigned to a particular process. The patches must not overlap and must completely cover the template.

The generalized block, HPF-style implicit, and explicit data distributions allow description of more irregular data distributions than the more common block-cyclic distribution[1]. This facilitates support for applications with irregularly distributed arrays, but may pose a barrier for interoperability in some cases, because many DA packages do not support these more general distributions.

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.

C. Data Redistribution

Often, two components in a coupled simulation must both access a distributed array 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 tranfered once.

The API of the CCA M×N component (Section IV-A) implements both one-time and persistent communication primitives. The PRMI model (section II-D), however, only supports one-time communication because of the limitations of the RMI paradigm.

A *communication schedule* for distributed arrays specifies the destination process of each of the data elements in the source array and their locations in the destination processes. This schedule is computed prior to the transfer operation, and can be reused in consecutive transfers, and even for different arrays as long as they conform to the same distribution template.

Communication schedules can be expensive to calculate, especially if the varieties of source and destination templates are numerous. One way to simplify this operation is to have an *intermediate representation* in the mapping process from the source layout to the target layout. Rather than matching source and target templates directly, the system refers the layouts to this intermediate representation. It is important to notice that it is not necessary for the system to arrange the actual data according to this intermediate representation; it can exist only in an abstract form, as a theoretical reference for the computation of the communication schedule.

As mentioned above, environments such as Meta-Chaos and the Indiana MPI-IO-based M×N device use linearization to represent data distributions. The application has some control over the mapping to and from this linear representation. In the Indiana M×N implementation, each process on the receiver side broadcasts to the senders which chunks of data it requires, referencing them to the linearization. Linearization is a useful tool that can be used by a system to implement communication schedules. It does not imply serialization - the linearization is a logical process, but actual transfers can be carried out fully in parallel. Linearization simplifies the task of matching a variety of data structures, from multidimensional arrays to trees or graphs. However, as already noted, the application must often know about how the sender linearized the data

[1] Implicit and explicit distributions allow the expression of completely arbitrary data distributions, while generalized block is more limited, but still supports a broad range of irregular distributions.

to interpret the delinearized data at the destination.

Another approach to the problem of calculating communication schedules is to defer the task to the application. This is the strategy used in the Distributed CCA Architecture (DCA), described briefly in Section IV-C. In DCA the application must specify the destination of each chunk of data. It does so through special arrays that mimic MPI's all-to-all communication primitives.

D. 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 various 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. The semantics of an M×N remote method invocation must be defined. 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 components 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 parallel remote method invocation 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 issues of invocation order guarantees (see [18]).

In the CCA model, parallel port methods can define two kind of arguments: *simple* and *parallel*. The current CCA convention is that a *simple* argument must have the same actual value in all the processes originating a given method invocation. However, some frameworks may not actively enforce this policy because checking that the actual values match might incur in a performance penalty. Regardless, this policy ensures that a provider component can always assume that a *simple* argument has the same actual value in all the processes.

A *parallel* argument represents a data array or structure that is 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×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. Several strategies are being explored to deal with this problem. For simple layout patterns like homogeneous block distributions, however, there is no need for the application to specify any layout parameters.

Supporting more complex data distribution layouts requires a way for the *provides* component to tell the framework how the data has to be delivered *prior* to the data transfer. Two solutions are currently being explored in the CCA Forum. One is to allow 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 possibility is to pass to the provides side a *reference* to the data object on the uses side, and to delay the actual transfer of data until the provides side has specified its layout.

Another difference between regular, directly-connected, ports and distributed remote ports is that of *process participation*. Process participation is the problem of defining which processes in both the calling and callee components will participate in any given port method invocation. The problem of process participation is closely related to that of defining the scope of collective calls.

In a direct-connected framework 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 need not be aware of these details because it does not need to know which processes 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.

However, in distributed frameworks the framework must know which processes in the calling component are participating in the collective call. The framework must mediate some communication between those processes in order to redistribute the parallel data and deliver the arguments. 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 approach to solve this problem is to require that all processes in both components (caller and callee) participate in the remote method invocation. This is the strategy followed in the PRMI model by Damevski [19]. In this case, process participation is implicit and static (it does not change from invocation to invocation). Indeed, Damevski [19] provides two kinds of remote invocations: the all-to-all collective invocation just described and a one-to-one non-collective invocation in which one process of the calling component calls one process of the callee component. Although this strategy is less flexible than the direct framework case, it provides simplicity and maximum transparency to the application programmer, who need not be aware of the details of the M$\times$N communication.

Another approach is presented in the Distributed CCA Architecture (DCA) [20]. 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 in all port method invocations, while on the callee side all processes must participate. This solution is more flexible in defining process participation, but is tied to using an MPI-based framework. However, 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 analagous to the

DAD.

In current direct-connect frameworks there is generally 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. In order to overcome this difficulty, CCA has introduced the notion of *one-way* methods (adopted from CORBA [2]). 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 (that includes arguments with the *out* attribute).

III. 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 genericity 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 distributed array descriptor serves this purpose.
- Distributed components running at different locations. This is the distinction between a direct-connected or distributed framework.
- Asynchronous, nonblocking transfers of data between components to allow overlapping of computation and communications.
- Well-defined semantics for a broad range of possible parallel remote method invocations 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 intercomponent parallel communications is more desirable than requiring learning a completely new API.

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

IV. 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 [22], [23], [24], [25], PAWS [18], [26]. and Meta-Chaos [12], [11]. These systems all provide ways to describe the subsets of data that are to be collected and moved to a particular destination

Project	Parallel Data	Language	PRMI	Prod. Level
Dist. CCA Arch. (DCA)	MPI-based arrays	C	Yes	No
InterComm	Dense arrays	C/Fortran	No	Yes
Model Coupling Toolkit (MCT)	Dense/sparse arrays, grids	Fortran	No	Yes
MxN Component	SIDL	Babel	No	Yes
SciRun	SIDL	C	Yes	Yes

Fig. 4. M×N projects and features. Some CCA frameworks use Babel [21] for language interoperability, which provides SIDL bindings for C, C++ and FORTRAN.

process. This is often done by distilling a given data decomposition on a per dimension basis into subregions or subsampled 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.

A. 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 Distributed Array Descriptor (DAD) object (see Section II-B). 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 II-B), 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.

B. 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 [27], and has been leveraged before for similar problems [28]. Both PRMI and parallel data redistribution primitives are defined in an extension to the SIDL language [10], 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.

In order 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 subsetting mechanism is engaged to allow greater flexibility.

To enable parallel data redistribution, a distributed array type was added to the SIDL language. Instances of the distributed array 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 [26]. For more information on this approach see [19].

C. DCA: A Distributed CCA Framework based on MPI

The Distributed CCA Architecture (DCA) is a prototype of a parallel and distributed CCA-compliant framework, based on MPI. The DCA uses MPI constructions such as communicator groups and all-to-all MPI communication primitives to solve the challenges of a distributed framework, concentrating on the M×N problems of data redistribution, process participation, and PRMI semantics that are more complex than the ones supported by SCIRun2.

DCA uses MPI communicator groups to determine process participation. The stub generator that parses the SIDL source files automatically adds an extra argument to all port methods, of type MPI_Comm, that is used to communicate to the framework which processes participate in the parallel remote method invocation. Also, parallel arguments are identified in the SIDL file with the special keyword "parallel".

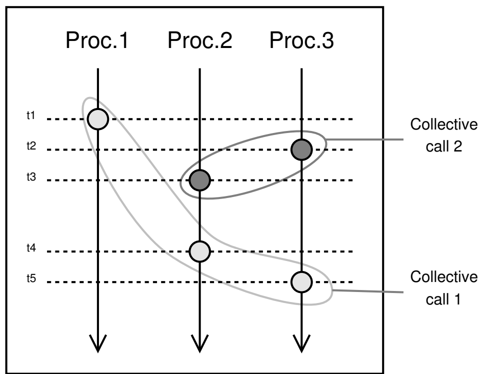

Fig. 5. The synchronization problem: if the PRMI call is delivered as soon as one process reaches the calling point, the remote component will block at t_1 waiting for data from processes 2 and 3, and will not accept the second collective call at t_2 and t_3. The remote component will be blocked indefinitely 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.

This communicator group defines the scope of parallel arguments (i.e. on which processes a parallel argument is deployed). Also, it is used to perform a *barrier synchronization*, required to ensure that the order of invocation is preserved when different but intersecting sets of processes make consecutive port calls. Figure 5 shows how a deadlock can occur unless barrier synchronization is performed. 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 Figure 5 disappears if process 1 participates in the second call).

DCA also employs the MPI all-to-all communication model to implement parallel data redistribution. This works by having the user define the data distribution layout using MPI data types, displacement and count arrays (see [29]). This information is passed to the framework at in the remote call using extra arguments automatically generated by the SIDL parser. This strategy of dealing with parallel data has the advantage of being familiar to MPI users and of being flexible by giving users the tools to describe their own data redistribution layout. This flexibility also has its disadvantages, because it places more responsibility on the user for defining the data distribution layouts. Dealing with MPI data types and displacement and count arrays is in general more complex than dealing with higher level abstractions like the CCA Distributed Array Descriptor (DAD). Also, although it would be beyond the current scope of the DCA, it would be possible to support the DAD as a layer on top of the DCA abstractions.

Component concurrency is achieved in two ways. On one hand, all CCA *Go* ports [2] are called at startup time, so all components that provide a Go port will be started concurrently. On the other hand, components can execute concurrently by using *one-way* methods. The DCA is more fully described in [20].

D. InterComm

InterComm [30] 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 [12], [11], but adds significant new functionality and provides much better performance. InterComm uses its own distributed array descriptor (DAD), and the CCA Data Group is currently integrating the models to have a single DAD by the end of 2004. 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

[2] *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.

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. 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, making 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 potentially hide the cost of data transfers behind other program activities.

E. The Model Coupling Toolkit

The Model Coupling Toolkit (MCT) [31] 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 [32], and to implement the coupling API for the Weather Research and Forecasting Model (WRF; http://www.wrf-model.org). 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 essentially implemented at the mesh level, 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 the following objects and services needed to construct distributed-memory parallel coupled models:

- A lightweight model registry that defines the MPI processes on which a module resides, and a process ID look-up table that obviates the need for intercommunicators between concurrently executing modules;
- A multifield data storage object that is the common currency modules use in data exchange;
- Domain decomposition descriptors, communications schedulers for intermodule parallel data transfer and intramodule parallel data redistribution, and the facilities to implement intermodule handshaking;
- A class encapsulating distributed sparse matrix elements and communication schedulers used in performing interpolation as parallel sparse matrix-vector multiplication in a multi-field, cache-friendly fashion;
- A 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);
- Spatial integral and averaging facilities that include paired integrals and averages for use in conservation of global flux integrals in intergrid interpolation;
- Registers for time averaging and accumulation of field data for use in coupling concurrently executing components that do not share a common timestep, or are coupled at a frequency of multiple timesteps;

- A facility for merging of state and flux data from multiple sources for use by a particular model (e.g., blending of land, ocean, and sea ice data for use by an atmosphere 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 [33]. Work is in progress to employ the Babel language interoperability tool to create MCT bindings for other programmming languages.

V. Related Work

The Data Reorganization Interface Standard (DRI-1.0) [34] 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 langage 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 Distributed Array Descriptor and M×N component.

VI. 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×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 intercomponent 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×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×N technology is only now starting to emerge as an important tool for composing parallel com-

ponents and even complete applications into larger cross-disciplinary simulations. M×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.

ACKNOWLEDGMENT

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] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. New York: ACM Press, 1999.

[2] Object Management Group, "CORBA component model," http://www.omg.org/technology/documents/formal/components.htm, 2002.

[3] Microsoft Corporation, "Distributed Component Object Model," http://www.microsoft.com/com/tech/dcom.asp, 2004.

[4] Sun Microsystems, "Enterprise JavaBeans downloads and specifications," http://java.sun.com/products/ejb/docs.html, 2004.

[5] 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 The Eighth IEEE International Symposium on High Performance Distributed Computing*, 1998.

[6] CCA Forum, "CCA Forum homepage," http://www.cca-forum.org/, 2004.

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

[8] U. S. Dept. of Energy, "SciDAC Initiative homepage," http://www.osti.gov/scidac/, 2003.

[9] J. Glimm, D. Brown, and L. Freitag, "Terascale Simulation Tools and Technologies (TSTT) Center," http://www.tstt-scidac.org/, 2001.

[10] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens, "Divorcing language dependencies from a scientific software library," in *Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Mar. 2001.

[11] M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman, and J. Saltz, "Runtime coupling of data-parallel programs," in *Proceedings of the 1996 International Conference on Supercomputing*, Philadelphia, PA, May 1996.

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

[13] F. Bertrand, Y. Yuan, K. Chiu, and R. Bramley, "An approach to parallel MxN communication," in *Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium*, Santa Fe, NM, October 2003.

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

[15] High Performance Fortran Forum, "High Performance Fortran language specification," *Scientific Programming*, vol. 2, no. 1–2, pp. 1–170, 1993.

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

[17] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A non-uniform-memory-access programming model for high-performance computers," *J. Supercomputing*, vol. 10, no. 2, p. 169, 1996.

[18] K. Keahey, P. Fasel, and S. Mniszewski, "PAWS: Collective interactions and data transfers," in *Proceedings of the High Performance Distributed Computing Conference*, San Francisco, CA, August 2001.

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

[20] F. Bertrand and R. Bramley, "DCA: A distributed CCA framework based on MPI," in *Proceedings of HIPS 2004, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*. Santa Fe, NM: IEEE Press, April 2004.

[21] Lawrence Livermore National Laboratory, "Babel homepage," http://www.llnl.gov/CASC/components/babel.html, 2004.

[22] J. A. Kohl and G. A. Geist, "Monitoring and steering of large-scale distributed simulations," in *IASTED International Conference on Applied Modeling and Simulation*, Cairns, Queensland, Australia, September 1999.

[23] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, "CUMULVS: Providing fault tolerance, visualization and steering of parallel applications," *The International Journal of High Performance Computing Applications*, vol. 11, no. 3, pp. 224–236, 1997.

[24] J. A. Kohl, "High performance computers: Innovative assistants to science," *ORNL Review, Special Issue on Advanced Computing*, vol. 30, no. 3/4, pp. 224–236, 1997.

[25] J. A. Kohl and P. M. Papadopoulos, "A library for visualization and steering of distributed simulations using PVM and AVS," in *High Performance Computing Symposium*, Montreal, CA, July 1995.

[26] 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 Computation, July 1998.

[27] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. Parker, "SCIRun2: A CCA framework for high performance computing," in *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*. Santa Fe, NM: IEEE Press, April 2004.

[28] K. Keahey and D. Gannon, "PARDIS: A parallel approach to CORBA," in *Proceedings of the High Performance Distributed Computing Conference*, 1997, pp. 31–39.

[29] M. P. I. Forum, "MPI: a message-passing interface standard," *International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, no. 3/4, pp. 159–416, Fall-Winter 1994.

[30] J. Lee and A. Sussman, "Efficient communication between parallel programs with InterComm," University of Maryland, Department of Computer Science and UMIACS, Tech. Rep. CS-TR-4557 and UMIACS-TR-2004-04, January 2004.

[31] J. W. Larson, R. L. Jacob, I. T. Foster, and J. Guo, "The Model Coupling Toolkit," in *Proceedings of the International Conference on Computational Science (ICCS) 2001*, ser. Lecture Notes in Computer Science, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, Eds., vol. 2073. Berlin: Springer-Verlag, 2001, pp. 185–194.

[32] L. Harper and B. Kauffman, "Community Climate System Model," http://www.ccsm.ucar.edu/, 2004.

[33] E. Ong, J. Larson, and R. Jacob, "A real application of the Model Coupling Toolkit," in *Proceedings of the 2002 International Conference on Computational Science*, ser. Lecture Notes in Computer Science, C. J. K. Tan, J. J. Dongarra, A. G. Hoekstra, and P. M. A. Sloot, Eds., vol. 2330. Berlin: Springer-Verlag, 2002, pp. 748–757.

[34] D. R. D. Forum, "Document for the data reorganization interface (dri-1.0) standard," http://www.data-re.org/, September 25 2002.