CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **26**:644–661
Published online 17 April 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3025

XpressSpace: a programming framework for coupling partitioned global address space simulation codes

Fan Zhang [1,*,†], Ciprian Docan [1], Hoang Bui [1], Manish Parashar [1] and Scott Klasky [2]

[1] *NSF Cloud and Autonomic Computing Center, Rutgers University, Piscataway NJ, USA*
[2] *Oak Ridge National Laboratory, Oak Ridge TN, USA*

SUMMARY

Complex coupled multiphysics simulations are playing increasingly important roles in scientific and engineering applications such as fusion, combustion, and climate modeling. At the same time, extreme scales, increased levels of concurrency, and the advent of multicores are making programming of high-end parallel computing systems on which these simulations run challenging. Although partitioned global address space (PGAS) languages attempt to address the problem by providing a shared memory abstraction for parallel processes within a single program, the PGAS model does not easily support data coupling across multiple heterogeneous programs, which is necessary for coupled multiphysics simulations. This paper explores how multiphysics-coupled simulations can be supported by the PGAS programming model. Specifically, in this paper, we present the design and implementation of the XpressSpace programming system, which extends existing PGAS data sharing and data access models with a semantically specialized shared data space abstraction to enable data coupling across multiple independent PGAS executables. XpressSpace supports a global-view style programming interface that is consistent with the PGAS memory model, and provides an efficient runtime system that can dynamically capture the data decomposition of global-view data-structures such as arrays, and enable fast exchange of these distributed data-structures between coupled applications. In this paper, we also evaluate the performance and scalability of a prototype implementation of XpressSpace by using different coupling patterns extracted from real world multiphysics simulation scenarios, on the Jaguar Cray XT5 system at Oak Ridge National Laboratory. Copyright © 2013 John Wiley & Sons, Ltd.

Received 20 October 2011; Revised 8 January 2013; Accepted 11 March 2013

KEY WORDS: coupled multiphysics simulation workflows; programming system; partitioned global address space

1. INTRODUCTION

In recent years, partitioned global address space (PGAS) has emerged as a promising programming model to improve the productivity of developing applications for emerging large scale parallel systems with increasing levels of concurrency. The PGAS model provides a distributed shared memory space abstraction and gives each process flexible remote access to it through language-level one-sided communication. PGAS languages, such as Unified Parallel C (UPC) [1], Co-Array Fortran [2], and Titanium [3], extend existing C, Fortran, and Java languages respectively, and support language-level constructs to explicitly express parallelism and data distributions. PGAS programming libraries, such as Global Arrays toolkit [4], provide a shared memory style programming environment in the context of distributed array data structures. As PGAS programming model is gaining attention in both academia and industry, newer members of PGAS language family, such as IBM X10 [5] and Cray Chapel [6], have been developed as part of Defense Advanced Research Projects Agency High Productivity Computing Systems initiative.

*Correspondence to: Fan Zhang, NSF Cloud and Autonomic Computing Center, Rutgers University, Piscataway NJ, USA.
†E-mail: zhangfan@cac.rutgers.edu

At the same time, emerging scientific and engineering simulations are increasingly focusing on end-to-end phenomena and are composed of multiple coupled parallel component applications that interact by sharing data at runtime. For example, in the Center for Plasma Edge Simulation Full-ELM coupling framework, the gyrokinetic edge component, XGC0 [7], and the MHD core component, M3D-OMP [8] run independently on different numbers of processors and exchange data as part of the coupled fusion simulation workflow. Similarly, in the Community Earth System Model (CESM) [9], separate simulation components are coupled as part of a multiphysics model to simulate the interaction of the earth, ocean, atmosphere, land surface, and sea ice. As a result, coupling and data sharing substrates that can facilitate such data exchange between independent parallel component applications in an efficient and scalable manner, are critical for these coupled applications.

However, building these coupled multiphysics simulation workflows from heterogeneous PGAS programs presents several challenges. First, although the PGAS model provides a shared memory abstraction for data exchange between parallel processes within single program, it does not easily support data coupling across heterogeneous programs. Multiple programs would have multiple PGAS shared memory spaces that are isolated from each other, and moving data across the different PGAS memory spaces is not supported by current PGAS runtimes. Furthermore, most existing coupling systems [10–12] are designed by using message-passing libraries such as MPI or PVM and are targeted at components applications based on the fragmented memory model, which is conceptually mismatched with the PGAS shared memory space model. As a result, the integration of these tools with PGAS-based applications leads to a mixing of PGAS and MPI/PVM runtimes and can result in degraded communication performance. Supporting coupled simulation workflows within the PGAS model thus requires specialized runtimes that support coupling and interactions across PGAS programs.

In this paper, we present the design, implementation, and experimental evaluation of XpressSpace, a programming system that extends the existing PGAS data access model by defining a semantically specialized shared data space abstraction that spans heterogenous PGAS programs and supports coupling and interactions between them. The contributions of our work are threefold. First, XpressSpace defines a global-view programming interface that conforms with the PGAS shared memory model. Coupled component applications share or exchange data through a shared data space abstraction by using one-sided data access operators. Second, XpressSpace enhances the basic data access interface with the ability to explicitly define coupling patterns between interacting applications that are part of the simulation workflow. Coupling patterns can be easily adapted based on user-defined specifications without requiring source code modification. Third, XpressSpace implements a lightweight runtime system for efficient and scalable memory-to-memory data redistribution between the coupled applications. Our current prototype implementation of XpressSpace and the discussion in this paper focus on UPC and Global Arrays; however, the ideas presented in this paper are applicable to most PGAS languages. In this paper, we also evaluate the performance and scalability of a prototype implementation of XpressSpace by using different coupling patterns extracted from real world multiphysics simulation scenarios, on the Jaguar Cray XT5 system at Oak Ridge National Laboratory.

The rest of the paper is structured as follows. Section 2 provides background material and outlines research challenges related to coupling heterogeneous PGAS programs. Section 3 presents an overview on the XpressSpace system architecture. Section 4 presents the high-level programming abstractions provided by XpressSpace and demonstrates the use of its programming interface by using a simple example. Section 5 describes the design and implementation of the XpressSpace runtime. Section 6 shows the experimental evaluation. Section 7 discusses related work, and Section 8 concludes the paper.

2. BACKGROUND AND CHALLENGES

This section describes performance issues related to multiphysics codes coupling and the challenges of coupling heterogeneous PGAS programs.

DOI: 10.1002/cpe

2.1. Multiphysics code coupling challenges

Application-to-application interactions in most coupled simulation workflows are based on exchanging data corresponding to coupled application variables. In parallel applications, data structures such as global multidimensional arrays are decomposed and distributed across the application processes, and exchanging data between different coupled simulation programs typically requires moving data from one application running on M processes to another application running on N processes, that is, the M×N problem. Efficiently redistributing the data from M to N application processes is a fundamental problem that has to be addressed by the supporting programming systems. Key associated technical challenges include:

Representation of global array data decomposition. Applications in coupled simulation workflow should have a unified way to specify the data decomposition of the coupled variables. First, it should clearly define the mapping from data elements of the distributed data structure (e.g., array) to application processes. Furthermore, it should be flexible and support different types of data distributions. Canonical distributions for array data structures supported by languages such as High Performance Fortran (HPF) [13], for example, Block, Cyclic, and Block-Cyclic, are a good example of such a representation and is used in the prototype described in this paper.

Computation of communication schedule. Communication schedule is the sequence of data transfers required to correctly move data between coupled applications. This information enables the senders in the M×N redistribution to determine the local data elements that needs to be transferred as well as the receivers for the data. Creation of the communication schedule requires matching the data decomposition of overlapped data domain for the coupled applications. The algorithm for computing communication schedule should be dynamic and scalable to handle the increasing complexity caused by the dynamic interactions between heterogeneous coupled applications, which run on thousands of processor cores.

Support for different coupling patterns. Tight coupling and loose coupling are two typical coupling patterns in coupled simulation workflows. In tight coupling, coupled applications progress at approximately the same speed and exchange data at each simulation time step. However, in loose coupling, coupled applications progress at different speeds and exchange data in an on-demand (possibly opportunistic) and asynchronous manner. In most cases, tightly coupled simulation workflow rely on direct transfers where data is moved directly from the memory of processes running one application to the memory of the processes running the other. Loose coupling often uses an intermediate data staging service, such as memory of a dedicated set of coupling servers, for asynchronous data sharing between the coupled applications.

Explicit definition of coupling patterns. In most current coupled simulation workflows, data coupling patterns are embedded into the source code of the coupled application programs by carefully matching data send and receive calls. This method presents the following drawbacks. First, when the number of coupled application changes, users have to reprogram the data coupling logic for all the application programs involved. Second, it is hard to reuse the implementation of the coupling patterns across application because the data coupling logic is implemented for one specific application context. Addressing these issues requires high-level programming interfaces that enable users to explicitly define the coupling patterns.

2.2. Challenges in coupling heterogeneous PGAS programs

The memory model presented by PGAS languages provides the abstraction of a global memory address space that is logically partitioned, and each process has a local section of this address space. One important advantage of PGAS over message passing models (such as MPI) is its global-view and data-centric programming model. Although message passing presents users with a more fragmented-view and control-centric programming interface, PGAS users can utilize high level language constructs to express data parallelism. For example, by using UPC, a user can define a global array by using the *shared* keyword in a single statement. Similarly, in the Global Arrays library, the collective operation *NGA_Create()* would create an *n*-dimensional shared array. Important tasks such as data decomposition and mapping array data elements to physical processors are handled by the language runtime. This is in contrast with message passing programming systems such as

DOI: 10.1002/cpe

MPI where applications have to explicitly manage the low level details of data decomposition and distribution, such as synchronization and process-to-process data communication.

Extending the PGAS model to enable the coupling heterogeneous PGAS programs requires developing a global memory address space abstraction that can be shared between multiple programs and a new programming interface for this abstraction that is consistent with the global-view and data-centric programming model. Users should be able to program coupled simulation workflow by expressing coordination, data movement, and coupling patterns at a higher level, and not be required to handle lower-level details such as computing communication schedule and programming matching data send and receive operations. Furthermore, coupling heterogeneous PGAS programs also implies that the new programming framework must support the sharing for data between applications developed using different PGAS languages or libraries. For example, it must support coupling a UPC-based land model and a Global Arrays-based sea model to simulate climate change.

Existing data coupling software tools are mostly designed for MPI-based applications and do not match well with PGAS shared-memory model. For example, by using the Model Coupling Toolkit (MCT) [10, 11] or InterComm [12] library, users have to specify the detailed data decomposition for each coupled variable to compute communication schedules. To do this by using the PGAS model, users need to specify the locality for each data element of the global data structures (e.g., array) by using language features such as *data affinity* in UPC, *data locale* in Chapel, or *data place* in X10. In addition, most existing coupled simulation workflows are developed as single MPI executables composed of multiple submodules, and data exchanges are hard-coded into the source code of each submodule. Using this approach, it is difficult to couple heterogeneous executables and to build workflows with dynamic data coupling patterns.

3. SYSTEM ARCHITECTURE OVERVIEW

The XpressSpace programming system extends the existing PGAS programming model to support coupling and interactions between heterogenous PGAS programs. It provides a simple programming interface that is designed to be consistent with the PGAS global-view programming model and extends them to enable users to compose PGAS applications into coupled application workflows. For a coupled application workflow, users must specify the coupled variables that are to be shared and define the coupling patterns (i.e., the structure of the workflow) in a separate XML file. Details of how data coupling is performed is abstracted from the users and is handled by the XpressSpace runtime.

The XpressSpace system architecture consists of two main components as illustrated in Figure 1, the XpressSpace bootstrap server and one or more independently running applications that are part of the coupled application workflow. The bootstrap server acts as a rendezvous point and is used to establish connections between independent PGAS (e.g., UPC [1] or Global Arrays [4]) executables that are part of the workflow and have no prior knowledge of each other. Both of these components are built on top of a layered architecture that is composed of the *DART Communication Layer*, the *XpressSpace Runtime Layer*, the *XpressSpace Core API*, and a *High-level User API*. In our current prototype, the high-level user API supports UPC and Global Arrays. These layers are described in the succeeding texts.

DART Communication Layer. This layer is based on our previous work on DART [14] and provides asynchronous messaging and data transport services to the other system components. The DART communication layer support different network architectures, including Cray XT Portals [15], Cray Gemini [16], IBM BlueGene/P Deep Computing Messaging Framework [17], InfiniBand, and TCP/IP, and implements remote direct memory access (RDMA) based data movement. DART provides an remote procedure call-like programming abstractions and hides the complexities of the underlying communication systems such as coordination and buffer management (e.g., RDMA buffers setup, registration, unregistration, and cleanup).

XpressSpace Runtime Layer. The runtime layer implements four major functional modules. The *bootstrap module* is used by the XpressSpace bootstrap server to parse the user-defined XML file, which is used to specify the workflow structure and to extract details about the coupling and data

DOI: 10.1002/cpe

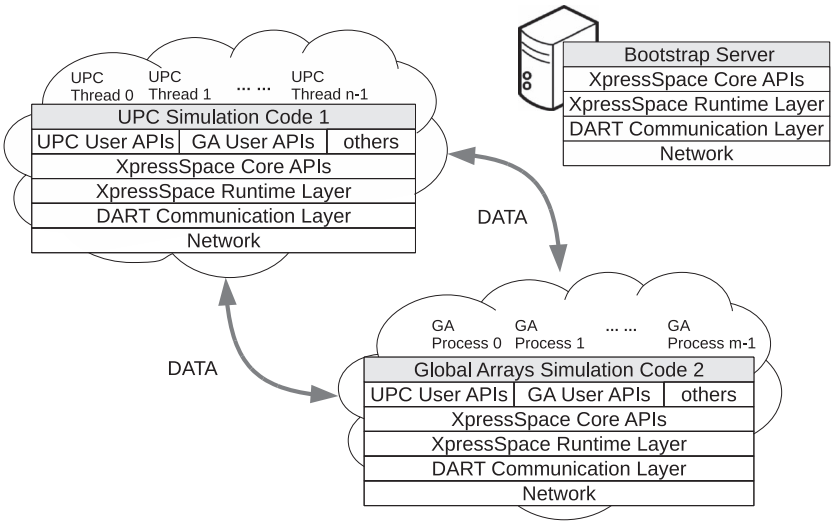

Figure 1. Schematic overview of the XpressSpace layered system architecture.

exchange patterns. The bootstrap server uses this information to generate data exchange schedules for the coupled variables and distributes these schedules to the corresponding application processes.

The XpressSpace runtime layer maintains Distributed Coupled Variables Directory (DCVD), which is a distributed hash table (DHT) that spans across the processes of the coupled applications and efficiently supports the computation of communication schedules. The DCVD maintains information about the coupled variables such as their distributions and data decompositions. In a coupled application workflow, an application can query the DCVD for the details of its coupled variables and can, for example, use it to compute necessary communication schedules. Each application in the coupled application workflow uses the *DCVD module* to build and maintain the DHT.

The index of the DHT is derived from the multidimensional data structures (e.g., arrays) used by the applications using the Hilbert space filling curve (SFC) [18], which maps the multidimensional index onto a one-dimensional address space. The key of each DHT entry is an interval (*start_index*, *end_index*) from this one-dimensional address space, which in turn is derived from the multidimensional data structure, for example, from the index corresponding to the region of interest of a multidimensional index. The data value of each DHT entry records the location of the data corresponding to that region.

The *data storage module* allocates and manages RDMA memory buffers to facilitate asynchronous data sharing. Data at the producer application processes is cached in local memory buffers from where it can be accessed on-demand by the consumer application processes. The *data transfer module* performs the RDMA-based parallel data transfers between the coupled workflow applications and implements both asynchronous pull and synchronous push transfer modes (details will be discussed in Section 5) to meet the requirements of different application coupling patterns.

XpressSpace Core API and **High-level User API**. XpressSpace provides a small set of core operators that exposes the functionalities from the runtime layer. The XpressSpace core API is independent of any high-level PGAS language constructs and provides binding for both C and Fortran. It is the building block that can be used to develop customized high-level user APIs for various PGAS languages. The high-level user API layer is a thin layer on top of the core API that support the global-view programming abstraction for different PGAS languages or libraries.

4. XPRESSSPACE PROGRAMMING INTERFACE

As noted earlier, XpressSpace provides a semantically specialized shared space abstraction [19] that extends across the coupled heterogeneous PGAS applications and supports data exchange between

DOI: 10.1002/cpe

Table I. XpressSpace User API.

[upc/ga]_xs_init()	Initialize the XpressSpace runtime.
[upc/ga]_xs_regsiter_var()	Register coupled variables with the XpresSpace shared data space.
[upc/ga]_xs_couple_var()	Perform parallel data transfer for the registered coupled variables.
[upc/ga]_xs_unregsiter_var()	Unregister coupled variables from the XpresSpace shared data space.
[upc/ga]_xs_finalize()	Release XpressSpace resources and terminate.

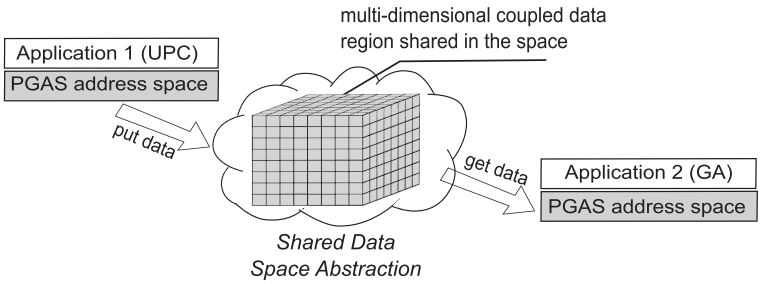

Figure 2. The XpressSpace shared space abstraction for coupling different PGAS programs.

these applications. The abstraction provides a tuple space-like model [20] and can be associatively accessed by the applications that are part of the workflow and enables decoupled interaction and coordination between these applications. XpressSpace also provides a high-level user API (Table I) that builds on top of the shared space abstraction. Although the current implementation focuses on cartesian grids and the PGAS distributed array defined over these grids, the proposed abstraction and approach can be applied to more general data domain and data structures.

This section introduces the XpressSpace shared space abstraction and describes the high-level user API it provides. This section also presents an example to illustrate how the XpressSpace API can be used to implement a coupled application workflow composed of both UPC and Global Arrays applications. As shown in Figure 2, even though coupled Application1 and Application2 have their own separate and isolated PGAS address spaces local to their processes, they can share and exchange data associated with the coupled variables by using the XpressSpace extended shared space abstraction. An XpressSpace-based coupled application implementation consists of two steps, inserting the XpressSpace global-view programming interface within the data coupling sections of the application source codes and explicitly defining the workflow structure and data coupling patterns within a coupling pattern specification (CPS) XML file. To use XpressSpace for coupling, the PGAS application source codes have to be modified as follows:

1. Define the data distribution templates for coupled variables. XpressSpace currently focuses on multidimensional array data structures and supports three common data distribution types, standard blocked, cyclic, and block-cyclic, as defined by most data parallel languages [13]. The *data_distribution* data structure is used to describe the data decomposition scheme. For example, for a general n-dimensional global array, it consists of three parameters: processors layout, data distribution type, and data block size. An n-tuple $(p_1, ..., p_n)$ specifies the number of processors in each dimension of the data domain. For the block-cyclic distribution, the block size also needs to be specified. An n-tuple $(b_1, ..., b_n)$ is used to specify the block size in each dimension of the data block.
2. Associate each coupled variable with a data distribution template by using the *xs_register_var()* call. In addition, users have to specify the region of the domain that need

DOI: 10.1002/cpe

to be coupled, for example, the relevant region of a global array. Note that only data associated with the coupled region is transferred at runtime. The size of the region is dependent on the application.

3. Perform coupling. Within the coupling section of the source codes, users have to insert *xs_couple_var()* initiate the data transfer. Note that this function can be invoked only on the coupled variables that are already registered with *xs_register_var()*.

This simple API provides users with the global-view operators on distributed global coupled data structures, for example, a global array, which is consistent with the PGAS memory model. However, the API does not provide users with direct control on coupling details such as which application to interact with or what direction the data is transferred. These coupling details have to be expressed in the separate CPS XML file, which contains the following key elements:

```
<coupling>
   <couple_var attribute = "Temperature">
      <schedule attribute="push">
          <from> 1 </from>
          <to> 2 </to>
          <start_ts> 1 </start_ts>
          <end_ts> 2000 </end_ts>
          <ts_interval> 1 </ts_interval>
      </schedule>
   </couple_var>
   <couple_var attribute = "Velocity">
      <schedule attribute="push">
          <from> 1 </from>
          <to> 3 </to>
          <start_ts> 1 </start_ts>
          <end_ts> 1000 </end_ts>
          <ts_interval> 4 </ts_interval>
      </schedule>
   </couple_var>
</coupling>
```

Listing 1. Sample CPS file for the coupled application workflow example.

- **coupling**: Serves as a container for other elements and must be used as the root element. The children elements are **couple_var** describing the coupled variables.
- **couple_var**: Represents a container for a coupled variable in the application. The expected children would be zero or more **schedule** elements.
- **schedule**: Records detailed information about coupling process between two applications. Programmers can use children elements **from** and **to** to specify the names of the sender and receiver applications for the coupled variables, use elements **start_ts** and **end_ts** to specify the time step scope for the coupling, and use **ts_interval** to specify the frequency of data coupling. Another configuration parameter that is part of **schedule** is the **attribute** field, which specifies the mode of data exchange. If it is specified as *push*, the data transfer mode will be Direct-Push, and it is specified as *pull*, the data transfer mode will be Cache-Pull (details of these modes are explained in Section 5). In both modes, the element **ts_interval** becomes optional because data transfer is triggered by the requests from the coupled applications.

The most important benefit provided by the XpressSpace programming interface is the ability to program with a global-view model without having to describe the mechanics of the coupling such as matching data receive and send functions. To illustrate this, we provide a simple example in Figure 3. The coupled simulation workflow in this example is composed of three iterative PGAS applications, of which App1 and App2 are implemented in UPC, and App3 is implemented by using

DOI: 10.1002/cpe

App1

```
/* Define two 2D global arrays */
shared [BLOCK_SIZE] double
      temperature[dimx][dimy],
      velocity[dimx][dimy];
int time_step = 1;
int app_id = 1; //unique application id
... ...

/* Register coupling variables, specify:
 (1) the unique variable name
 (2) assocated distribution template
 (3) pointer to the global array
 (4) coupled data region */
upc_xs_register_var("Temperature",
    &struct_template,
    &temperature[0][0],
    0, 0, 0, dimx-1, dimy-1, 0);

upc_xs_register_var("Velocity",
    &struct_template,
    &velocity[0][0],
    0, 0, 0, dimx, dimy/2-1, 0);

/* Code computation stage */
... ...

/* Code coupling stage */
upc_xs_couple_var("Temperature",
      &temperature[0][0], time_step);
upc_xs_couple_var("Velocity",
      &temperature[0][0], time_step);

time_step++;
```

App2

```
/* Define one 2D global array */
shared [BLOCK_SIZE] double
      temperature[dimx][dimy];
int time_step = 1;
int app_id = 2; //unique application id
... ...

/* Register coupling variables, specify:
 (1) the unique variable name
 (2) assocated distribution template
 (3) UPC pointer to the global array
 (4) coupled data region */
upc_xs_register_var("Temperature",
    &struct_template,
    &temperature[0][0],
    0, 0, 0, dimx-1, dimy-1, 0);

/* Code computation stage */
... ...

/* Code coupling stage */
upc_xs_couple_var("Temperature",
      &temperature[0][0], time_step);

time_step++;
```

App3

```
/* Define dimensions and blocks size
   for the 2D global array */
int dimensions[2] = {dimx, dimy};
int block_size[2] = {blk_x, blk_y};
int time_step = 1;
int app_id = 2; //unique application id
... ...

/* Create "Velocity" global array */
int handle_velocity = NGA_Create(
      C_DBL, 2, dimensions,
      "Velocity", block_size);

/* Register coupling variables, specify:
 (1) the unique variable name
 (2) assocated distribution template
 (3) GA handle to the global array
 (4) coupled data region */
ga_xs_register_var("Velocity",
   &struct_template,
   handle_velocity,
   0, 0, 0, dimx-1, dimy/2-1, 0);

/* Code computation stage */
... ...

/* Code coupling stage */
ga_xs_couple_var("Velocity",
      handle_velocity, time_step);

time_step++;
```

Figure 3. Programs for the coupled simulation workflow example.

the Global Arrays library. The three PGAS applications run concurrently on different sets of processors, and data is transferred from App1 to App2 and App3 during the coupling stage in each time step, and the applications runs for 2000 time steps, that is, $1 <= time\_step <= 2000$. There are two coupled variables; in this workflow are 2-dimensional global arrays *'Temperature'* and *'Velocity'*. During coupling in each time step, *Temperature* is redistributed from App1 to App2. Also, data region $< 0, 0, 0; dimx - 1, dimy/2 - 1, 0 >$ of App1 global array *Velocity* is redistributed to data region $< 0, 0, 0; dimx - 1, dimy/2 - 1, 0 >$ of App3 global array *Velocity* every four time steps when $1 <= time\_step <= 1000$. The CPS XML file shown in Listing 4 specifies the coupling details for each application.

5. IMPLEMENTATION OF THE XPRESSSPACE RUNTIME

5.1. Automatic detection of data locality

As discussed in the previous section, XpressSpace does not require users to specify the data decomposition details for each coupled variable. Instead, users only need to define data distribution types at a high level by using the XpressSpace global-view programming interface. While this hides low-level programming complexity from the uses, the XpressSpace runtime has to effectively determine locality for the data elements of the shared global array before performing actual parallel data transfers. At runtime, each process of the coupled applications constructs a list of descriptors to record what data is locally computed and stored. In case of arrays defined on a Cartesian grid, these are geometric descriptors. A geometric descriptor contains two cartesian coordinates to specify the bounding box of data region mapped to the local process as illustrated in Figure 4. Figure 4 shows two examples of how data decomposition information is internally represented within the XpressSpace runtime for UPC programs. The execution vehicle for a UPC program is called a UPC thread, which can be implemented either as full-fledged OS processes or as threads. In this paper, we will use the term UPC thread instead of process for description of the UPC-based application scenarios. Coupled variable *'Temperature'* is a 4×4 2D global array, which is decomposed across 4 UPC threads by using the block-block distribution by application Code1 and is decomposed across 2

DOI: 10.1002/cpe

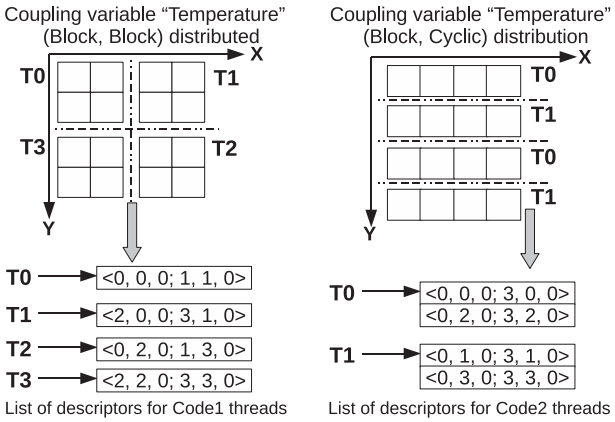

Figure 4. XpressSpace geometric descriptors for a Cartesian grid representing the data decomposition of the coupled variables.

UPC threads by using the block-cyclic distribution by application Code2. In this case, the geometric descriptor $< 0, 0, 0; 1, 1, 0 >$ for UPC thread T0 in Code1 indicates that data associated with the 2×2 subgrid bounded by coordinates $< 0, 0, 0 >$ and $< 1, 1, 0 >$ is currently stored at UPC thread T0. This geometric descriptor data structure forms the basis for the DCVD service.

The XpressSpace runtime uses two methods to inspect for data locality. First, for global arrays with standard data distributions such as blocked, cyclic, and block-cyclic, the XpressSpace runtime can easily determine what array data elements have local affinity because these distribution patterns are static. For example, the blocked distribution evenly decomposes an array among application processes, and each process owns one contiguous part of the array, while the cyclic distribution distributed an array elements across processes in a round-robin fashion. In the block-cyclic distribution, an array is divided into user-defined size blocks, and the blocks are cyclically distributed among processes. Second, for global array with irregular distributions, XpressSpace leverages the data locality detection features of PGAS languages or libraries to construct the geometric descriptors. For example, the *upc_threadof(shared void *ptr)* function can return the rank of UPC thread that has affinity to the shared array element pointed by *ptr*. Similarly, Global Arrays provides the library function *NGA_Distribution()*, which can find out the regions of the global array that a calling process owns.

5.2. *Dynamic computation of communication schedules*

Once data locality information is successfully determined, the DCVD service is built and uses a DHT to index the data decomposition for each coupled variable. At runtime, DCVD services as distributed directory and responds queries from other coupled applications. The process of generating the communication schedule consists of three steps as described in the succeeding texts. The description uses the example presented in Figure 4 and uses Figure 5 to illustrate how application Code1 and Code2 build their communication schedules for shared global array *'Temperature'* using the DCVD service.

First, DCVD uses the Hilbert space filling curve to map the *n*-dimensional domain to a one-dimensional index space. By using this mapping, we can uniquely identify a data point in the domain by using either *n*-dimensional coordinates or an one-dimensional index value. Similarly, a continuous data region can be represented by the geometric descriptor described earlier or a set of pairs of lower and upper indices in the one-dimensional index space. For example, as shown in Figure 5, the data region $< 0, 0, 0; 1, 1, 0 >$ stored in UPC thread T0 belonging to Code1 can be described by the interval $< 4 - 7 >$.

Second, a hash table is created by using the linearized one-dimensional index space as hash table keys and the data locality information as hash table values. The hash table records the mapping

DOI: 10.1002/cpe

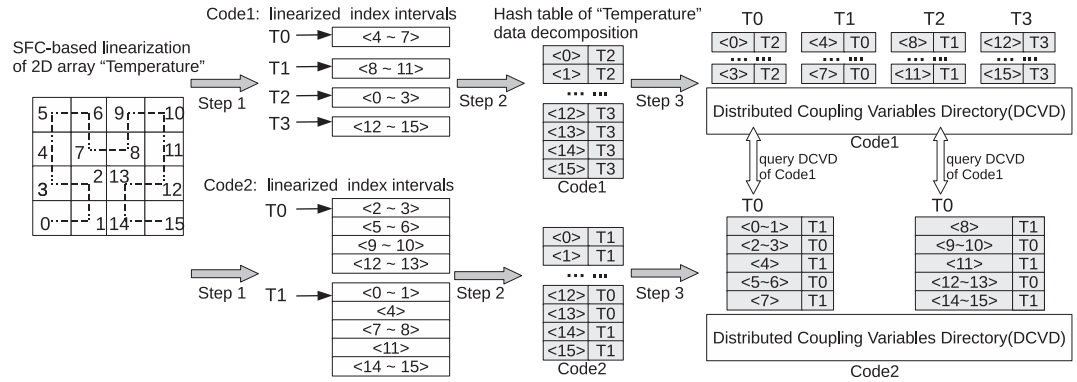

Figure 5. Building XpressSpace communication schedules.

between each data element and its storage location. For example, a key-value pair $(< 12-15 >, T3)$ for Code1 indicates that data elements in the region $< 12-15 >$ are associated with UPC thread T3. This association has two meanings in a coupled application workflow. When Code 1 is producing the coupled data, the key-value pair indicated where the corresponding data elements produced by T3 are stored. When Code1 is consuming the coupled data, the associated one-dimensional index intervals are used as the key to query the DCVD service of another application to determine the location of the data elements. The hash table created is distributed by dividing up the index space so as to load-balance the query load from consumer applications.

Third, consumer application processes concurrently query the DCVD DHT of the producer applications. Each querying process specifies data regions of interest within the global domain in its query request. The DCVD DHT returns the corresponding storage locations of the data elements of interest back to the querying process. To optimize performance, the XpressSpace runtime creates a in-memory local communication schedule cache to keep retrieved data location information. During each application time step, application processes or threads first look for communication schedules within this local cache. If the lookup succeeds, it is not necessary to query the DCVD of remote application. This optimization can significantly reduce the cost of querying the DCVD for applications where the coupling pattern is repeated, which is the case for many coupled scientific simulations.

5.3. Data transfer modes

XpressSpace supports two different data transfer modes, *Direct-Push* and *Cache-Pull*, to support data transfer requirements of the tight-coupling and loose-coupling patterns respectively. In a tightly coupled application workflow, the coupled applications usually progress at the same speed and the data transfer for the coupled variables is performed every time step in a synchronous manner. In this case, it is desirable to push coupled variables from the sender to the receiver application as soon as the data is produced. In a loosely coupled application workflow, applications may run at very different speeds, and asynchronous interactions are required.

Direct-Push employs a sender-driven push approach to support fast and synchronous data transfers appropriate for tight-coupling patterns. By using XpressSpace, we found that each process of the receiver application registers a memory buffer that can be accessed remotely by using the RDMA one-sided *put* operation. At every time step, processes of the sender application uses the communication scheduled to *put* the produced data of the coupled variable and region of interest directly into the remote receive buffer of the corresponding receiver process.

Cache-Pull employs a receiver-driven pull approach to support on-demand and asynchronous data transfer of the coupled variable and is appropriate for loose-coupling patterns. To implement this approach, each process of the sender application allocates a local memory buffer for data caching, and this buffer is accessible remotely by using the RDMA one-sided *get* operation. At every time

DOI: 10.1002/cpe

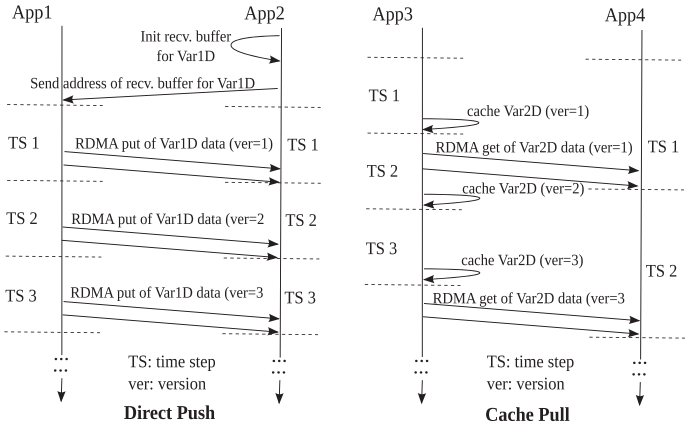

Figure 6. Sequence diagrams for the XpressSpace data transfer modes.

step, processes of the sender application save relevant-produced data into the local buffer and continue to next time step. When the receiver application required data for the coupled variables, it will *get* the latest data from remote cache buffer. XpressSpace tags each data object cached in the local memory buffer with a version attribute. Because of memory space constraint, only a limited number of data objects and versions can be cached. By default, older versions of the data objects are evicted out of local memory buffer following a FIFO schedule. However applications can enforce blocking semantics when the buffers are full, making the coupling pattern more synchronous.

Figure 6 uses sequence diagrams of two simple coupling examples to illustrate the *Direct-Push* and *Cache-Pull* data transfer modes. In the first case, workflow applications App1 and App2 are tightly coupled, and the coupled variable *'Var1D'* (a 1D global array) is redistributed from App1 to App2 at every time step. In the second case, workflow applications App3 and App4 are loosely coupled and running at different speeds with varied time-step lengths. Coupled variable *'Var2D'* (a 2D global array) is redistributed from App3 to App4, but the coupling frequency is not determined. Data is transferred only when receiver App4 demands it, and only the latest version of *'Var2D'* is fetched by App4 processes.

6. EXPERIMENTAL EVALUATION

The prototype implementation of XpressSpace was evaluated on the Cray XT5 Jaguar system at Oak Ridge National Laboratory. Jaguar XT5 has 18,688 compute nodes, and each compute node has a 2.6 GHz dual hex-core AMD Opteron processor, 16GB memory, and a SeaStar2+ router that interconnects the nodes via a fast network with a 3D torus topology.

The XpressSpace evaluation presented in this section consists of three distinct sets of experiments. The first set of experiments aims to evaluate the scalability of XpressSpace with increasing numbers of application processes and increasing coupled data sizes. The second set of experiments evaluates the performance of XpressSpace parallel data transfers for an M×N data redistribution by using a two-way tight coupling scenario in which two coupled application codes exchange data synchronously during the coupling stage of each time step. The third set of experiments are aimed at analyzing the overheads of computing the communication schedules by using DCVD. In addition, for the second and third sets of experiments, we compare the performance of XpressSpace with the MCT library, which is used to implement the coupler in the CESM 1.0 climate modeling system. The experiments used skeleton codes, which capture the interaction behaviors of real applications while removing the complexity of the computational aspects. We used the Berkeley UPC version 2.12 to develop our UPC skeleton codes, Global Arrays version 5.0 to develop the GA skeleton codes, and MCT 2.6.0 to develop the MPI skeleton codes.

DOI: 10.1002/cpe

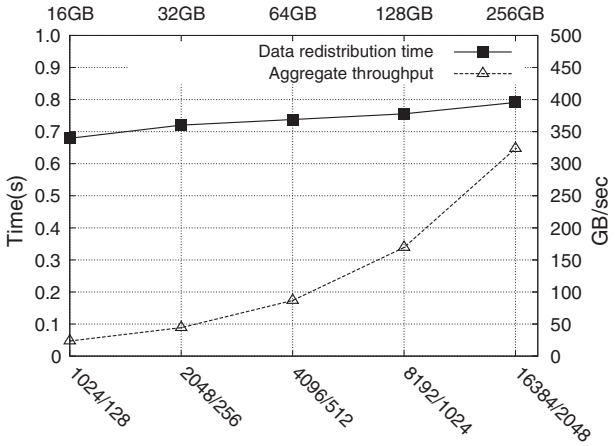

Figure 7. Transfer time and aggregate throughput for parallel data redistribution and data transfers by using XpressSpace. The bottom X axis represents the number of application processes that produce data / the number of application processes that consume data. The top X axis represents the size of the redistributed data.

6.1. Experiment 1: XpressSpace scalability

Coupled application workflows simulating complex phenomena such as climate modeling and plasma fusion science typically run on a large number of processor cores and significant amounts of data is transferred between the coupled simulation components. The scalability experiments presented in this section evaluates the ability of the XpressSpace framework to support parallel data redistribution for a different numbers of application processes and data sizes.

This experiment uses a one-way tightly coupled workflow composed of two heterogeneous PGAS applications, a UPC data producer application and a Global Arrays data consumer application, which interact at runtime by using XpressSpace. The two applications perform parallel computations over a common two-dimensional computational domain, and each application is assigned a distinct set of processor cores. The coupled variable was a two-dimensional global array that is decomposed by using a standard blocked distribution, and its data is redistributed from the producer to the consumer application during coupling.

This is a weak scaling experiment. Each process executing the producer application generate a fixed 16 MB of data per iteration. The number of data producing processes is varied from 1024 to 16,384 and the number of data consuming processes from 128 to 2048. Consequently, the size of data transferred is varied from 16 GB to 256 GB. In the experiment, each application ran for 50 iterations to simulate 50 parallel data redistributions and data transfers between the coupled applications.

Figure 7 presents the average transfer time and aggregate throughput required by XpressSpace to complete the parallel data redistributions and data transfers. The results show good overall scalability with increasing number of processes and data sizes. When that data size was increased from 16 GB to 256 GB (i.e., 16 fold), the transfer time increased less than 15%. The performance degradation is mainly due to contention at the shared network links, which is caused by the increasing number of concurrent data transfers at larger scales. However, this small increase in transfer time is acceptable when considering the scale of the application. Furthermore, the aggregate data transfer throughput maintained a near linear performance. When the number of data producing processes was increased from 1024 to 16,384, the throughput increased almost 13 times from 25 GB/s to 325 GB/s.

6.2. Experiment 2: Evaluation of parallel data transfer

This experiment uses a two-way tightly coupled workflow composed of two applications, App1 and App2, which exchange data at runtime. There are two coupled variables *'Var1'* and *'Var2'*, both of

DOI: 10.1002/cpe

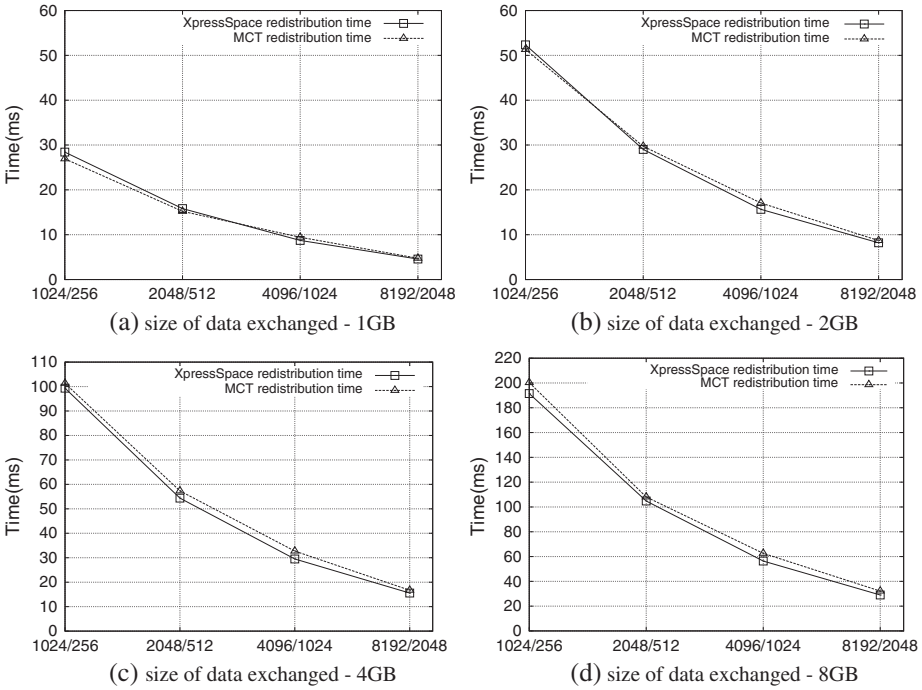

Figure 8. Data exchange time (redistribution and transfer) in millisecond for *Var1* and *Var2*. The bottom *X* axis is the number of processes running App1 / the number of processes running App2.

which are two-dimensional shared global arrays and are decomposed by using a standard blocked distribution. The two global arrays have the same size. *'Var1'* is redistributed from App1 to App2 and *'Var2'* is redistributed in the reverse direction from App2 to App1. In the XpressSpace version of the workflow, App1 and App2 are implemented as two independent UPC programs that run as separate executables, and the Direct-Push data transfer mode is used. In the MCT version of the workflow, App1 and App2 are implemented as two sub-modules of a single MPI program and ran as a single executable. We refer to the number of processes running App1 as num_proc_app1, the number of processes running App2 as num_proc_app2, and the total exchanged data size as $size\_data$. In our experiments, each process ran a single UPC thread or MPI process. The value of num_proc_app1 was varied from 1024 to 8192, the value of num_proc_app2 was varied from 256 to 2048, and $size\_data$ was increased from 1 GB to 8 GB.

The results of the experiment are plotted in Figure 8 and Figure 9. Figure 8 plots the average data exchange time required by XpressSpace and MCT. Figure 9 plots the average bidirectional bandwidth measured at each process for App1 and App2. As shown in Figure 8(a–d), the overall performance for the two frameworks are comparable. When redistributing a fixed size of data, the exchange time decreased with increasing numbers of processes. This is because the amount of data exchanged by each process decreased, requiring less time for data transfer. Furthermore, it can be seen from Figure 9(a–d) that XpressSpace has better performance than MCT when the size of the data exchanged is large. For example, the performance of XpressSpace and MCT is very close when that size of the data exchanged is 1 GB. However, the difference between that achieved bidirectional bandwidth per process for the two implementations becomes large as size of data exchanged increases beyond 1 GB (i.e., 2GB, 4GB, and 8GB). This performance gap is because MCT uses two-sided MPI message passing as the underlying data transport mechanism, which requires two memory copies at each end of the data transfer, and requires enforcing MPI's message matching semantics. In contrast, XpressSpace can directly push data into RDMA buffers at the receiving processes without requiring extra memory copies. The one-sided semantics can more effectively utilize the capabilities of the underlying RDMA hardware than MPI's two-sided message passing semantics.

DOI: 10.1002/cpe

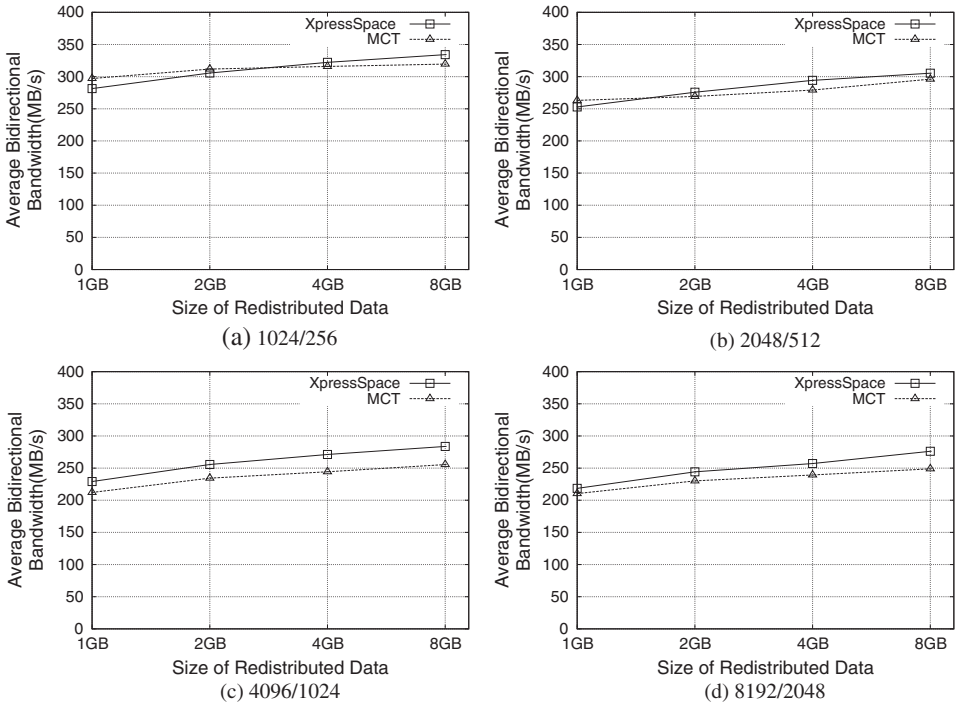

Figure 9. Average bidirectional bandwidth per process for redistribution of *Var1* and *Var2*.

Figure 9(a–d) highlight another interesting trend—the bidirectional bandwidth per process decreases with increasing numbers of processes for both XpressSpace and MCT implementations. With the size of the data exchanged, remaining constant larger numbers of processes implies that data redistribution will require more parallel data communications, which in turn will cause more contentions at the shared network links and the resulting impact on achievable throughput. This also explains why the rate of change of the data redistribution time (Figure 8(a–d)) decreases as the number of processes increases. As seen in these figures, the slopes of data redistribution time curves for both XpressSpace, and MCT decreases when the number of processes running App1 and App2 is increased from 1024/256 to 8192/2048.

6.3. Experiment 3: Evaluation of the overheads for computing communication schedules

This experiment evaluates the overheads of computing communication schedules in XpressSpace. As in Experiment 2, the workflow used in this set of experiments is composed of two coupled applications App1 and App2, and two coupled variables *'Var1'* and *'Var2'* that are exchanged during coupling.

Table II lists the average time or computing communication schedules for the M×N data redistribution required during coupling. As expected, the results in Table II show that in both cases, that is, using XpressSpace and MCT, increasing the size of the coupled data does not affect the time required to compute the communication schedules. However, increasing the scale of the application did cause the time required to generate the schedules to increase.

In the XpressSpace case, each process running App1 and App2 queries a number of DCVD processes to get information about the location of the desired data region of the coupled variable. The increasing number of queried DCVD processes results in increased communication costs and an overall increase in the times required to generate the communication schedules. Note that the number of queried DCVD processes is determined by the types of data distributions involved and how the data location information is distributed in each application's DCVD but not the data size of the coupled data. For example, when the number of processes running an application becomes larger, its DCVD distributed across a larger number of processes and the data location information is more fragmented within DCVD, which results in higher query costs.

DOI: 10.1002/cpe

Table II. Communication schedule computation time in seconds.

(a) XpressSpace

Array size	1 GB	2 GB	4 GB	8 GB
1024/256	0.0119	0.0194	0.0218	0.0211
2048/512	0.0226	0.0237	0.0268	0.0290
4096/1024	0.0276	0.0337	0.0385	0.0462
8192/2048	0.0365	0.0490	0.0509	0.0624

(b) MCT

Array size	1 GB	2 GB	4 GB	8 GB
1024/256	0.0130	0.0160	0.0131	0.0141
2048/512	0.0344	0.0334	0.0354	0.0387
4096/1024	0.05963	0.0682	0.1260	0.1371
8192/2048	0.0774	0.0922	0.1567	0.1797

The results in Table II also show that XpressSpace has lower overheads for computing communication schedules than MCT. As shown in the table, XpressSpace scales well as the number of data producing and consuming processes increase and is about two times faster than MCT when the number of processes is 8192/2048. The main advantages of XpressSpace are twofold. First, each application process can issue concurrent DCVD queries, and the communication schedule information is received asynchronously by using RDMA buffers that can be directly accessed by remote DCVD processes. Second, each application process only queries for relevant data locality information, and the size of this information is usually very small.

7. RELATED WORK

Programming abstractions and languages for multidisciplinary coordination: Several research projects have explored new programming abstractions and languages, or extensions to existing parallel languages to enable multidisciplinary, multiscale and multiphysics coordination. Opus [21] extends data parallel languages with the Shared Data Abstraction mechanism, which can be used as computation servers as well as shared data repositories to support the coupling of multiple programs. Orca [22] is a programming language for implementing parallel applications on loosely coupled distributed system. It provides a shared data abstraction that can be accessed by using user-defined high-level operations. This abstraction enables interaction and coordination between coupled application processes. Our work extends the PGAS parallel programming model to support both tightly and loosely coupled interactions between heterogeneous PGAS applications. Specifically, our XpressSpace programming system extends existing PGAS data sharing and data access models with a semantically specialized shared data space abstraction and operators to enable data coupling across multiple independent PGAS executables.

Data coupling software tools: Several research groups, including our group, have addressed coupling of parallel simulations such as the M×N coupling problem where a component running on M nodes is coupled with another component running on N nodes. To tackle this problem, the M×N working group of the Common Component Architecture (CCA) [23] forum developed a set of software tools for parallel data redistribution for coupled simulation including MCT [10, 11], InterComm [12], Parallel Application Workspace [24], and Meta-Chaos [25]. The CCA forum also defined a set of standard interfaces for coupled components to promote interoperability between components developed by different groups. Thus, InterComm programmers can use an XML job description file to describe the components in a coupled simulation. However, most CCA software tools build coupled simulation workflows as a single MPI program composed of different communicator groups, which limits their applicability. Furthermore, these tools were originally developed for message passing based parallel programs and do not consider more recent memory models and language features such as those introduced by PGAS languages.

DOI: 10.1002/cpe

A coupler based data sharing and exchange framework has been used in real world multiphysics simulations. In the CESM climate modeling system, different simulation models such as atmosphere, land, sea ice, and ocean are connected by a single coupler component. Synchronization and data flow is completely controlled by the coupler component. Data exchanges between different models go through a two-step transfer, first from the source model to the coupler, and then redistributed from the coupler to the destination model. The Mesh-based Parallel Code Coupling Interface [26] library also implements a coupling server and each parallel process communicates with this Mesh-based Parallel Code Coupling Interface coupling server engine. Performance, scalability, and resource limitations at a central coupler are some of the issues with the coupler-based coupled simulation system, especially when coupling is tight and involves large amounts of data.

Our work on XpressSpace explores an asynchronous approach to M×N coupling and is based on a common shared space data model that was used to couple heterogeneous applications. Our work also develops a distributed and scalable implementation that is based on memory-to-memory data transfers. These transfers can occur directly between the nodes running the coupled simulations, enabling the approach to support a wide range of different coupling workflow patterns.

Coupling through PGAS communication systems: Networking communication libraries such as Aggregate Remote Memory Copy Interface (ARMCI) [27] and GASNet [28] provides RDMA-based one-sided data access to data associated with distributed array data structures and are used for low level communications and data transfers in PGAS runtime systems, such as UPC, Global Arrays and Chapel. Although users can implement parallel M×N data redistribution by using PGAS-compatible ARMCI or GASNet APIs, there are two constraints. First, these libraries only support data communications between processes or threads within a single PGAS executable. In contrast, XpressSpace can be used to exchange data between heterogeneous PGAS executables. Second, ARMCI and GASNet define a very basic and general programming APIs using which users have to build the coupled application workflows. In contrast, XpressSpace provides a simple high level programming interface that hides implementation complexities from users and can be used to directly specify interaction and coordination patterns of coupled application workflows.

8. CONCLUSION

This paper presented the design and implementation of XpressSpace—a programming system that extends existing PGAS data sharing and data access models with a semantically specialized shared data space abstraction to enable data coupling across multiple independent PGAS executables. XpressSpace supports a global-view style programming interface that is consistent with the PGAS memory model and provides an efficient runtime system that can dynamically capture the data decomposition of global-view data structures such as arrays and enable fast exchange of these distributed data structures between coupled applications. XpressSpace also provides a simple programming interface that enables users to compose PGAS applications into coupled application workflows—users specify the coupled variables that are to be shared and define the coupling patterns (i.e., the structure of the workflow) in a separate XML file, and details of how data coupling is performed is abstracted from the users and is handled by the XpressSpace runtime, which implements efficient direct memory-to-memory data redistribution by using RDMA.

The paper also presented an experimental evaluation of XpressSpace, which demonstrated its performance and scalability. The evaluation compared the performance of XpressSpace with MCT. Our future work includes further optimizing parallel data transfers by using data locality aware mapping of application processes to physical processor cores. We will also explore support for using the directed acyclic graph to compose and represent more complex coupled simulation workflow scenarios.

ACKNOWLEDGEMENTS

The research presented in this work is supported in part by the National Science Foundation (NSF) via grant numbers DMS 1228203 and IIP 0758566, by the DoE ExaCT Combustion Co-Design Center via subcontract number 4000110839 from UT Battelle, by the DoE Scalable Data Management, Analysis, and Visualization

DOI: 10.1002/cpe

Institute via the grant numbers DE-SC0007455, by the NSF Center for Remote Data Analysis and Visualization via subcontract number A10-0064-S005, by the DoE Partnership for Edge Physics Simulations via grant numbers DE-SC0008455 and DE-FG02-06ER54857, and by an IBM Faculty Award. The research and was conducted as part of the NSF Cloud and Autonomic Computing Center at Rutgers University and the Rutgers Discovery Informatics Institute (RDI2).

REFERENCES

1. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K. Introduction to UPC and language specification. *Technical Report CCS-TR-99-157*, George Washington University, 1999.
2. Numrich RW, Reid J. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum* 1998; **17**:1–31.
3. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A. Titanium: a high-performance java dialect. *Concurrency and Computation: Practice & Experience* 1998; **10**:825–836.
4. Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H, Aprà E. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications* 2006; **20**:203–231.
5. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V. X10: an object-oriented approach to non-uniform cluster computing. *Proceedings of the 20th Annual ACM Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, San Diego, CA, USA, 2005; 519–538.
6. Chamberlain BL, Callahan D, Zima HP. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications (IJHPCA)* 2007; **21**:291–312.
7. Chang CS, Ku S, Weitzner H. Numerical study of neoclassical plasma pedestal in a tokamak geometry. *Physics of Plasmas* 2004; **11**:2649–2667.
8. Park W, Belova EV, Fu GY, Tang XZ, Strauss HR, Sugiyama LE. Plasma simulation studies using multilevel physics models. *Physics of Plasmas* 1999; **6**:1796–1803.
9. Collins WD, Bitz CM, Blackmon ML, Bonan GB, Bretherton CS, Carton JA, Chang P, Doney SC, Hack JJ, Henderson TB, Kiehl JT, Large WG, McKenna DS, Santer BD, Smith RD. The community climate system model version 3 (CCSM3). *Journal of Climate* 2006; **19**(11):2122–2143.
10. Larson J, Jacob R, Ong E. The model coupling toolkit: a new fortran90 toolkit for building multiphysics parallel coupled models. *International Journal of High Performance Computing Applications (IJHPCA)* 2005; **19**:277–292.
11. Jacob R, Larson J, Ong E. M×N communication and parallel interpolation in community climate system model version 3 using the model coupling toolkit. *International Journal of High Performance Computing Applications (IJHPCA)* 2005; **19**:293–307.
12. Lee JY, Sussman A. High performance communication between parallel programs. *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4 - Volume 05*, Denver, CO, USA, 2005; 177.2.
13. Koelbel CH, Loveman DB, Schreiber RS, Steele GL, Jr., Zosel ME. *The High Performance Fortran Handbook*. MIT Press: Cambridge, MA, USA, 1994.
14. Docan C, Parashar M, Klasky S. DART: a substrate for high speed asynchronous data IO. *Proceedings of 17th International Symposium on High Performance Distributed Computing (HPDC'08)*, Boston, MA, USA, 2008; 219–220.
15. Brightwell R, Hudson T, Pedretti K, Riesen R, Underwood K. Implementation and performance of portals 3.3 on the Cray XT3. *Proceedings of International Conference on Cluster Computing (CLUSTER'05)*, Boston, MA, USA, 2005; 1–10.
16. Alverson R, Roweth D, Kaplan L. The gemini system interconnect. *IEEE 18th Annual Symposium on High Performance Interconnects (HOTI'10)*, Mountain View, CA, USA, 2010; 83–87.
17. Kumar S, Dozsa G, Almasi G, Heidelberger P, Chen D, Giampapa ME, Blocksome M, Faraj A, Parker J, Ratterman J, Smith B, Archer CJ. The deep computing messaging framework: generalized scalable message passing on the blue gene/P supercomputer. *Proceedings of 22nd Annual International Conference on Supercomputing (ICS'08)*, Island of Kos, Greece, 2008; 94–103.
18. Sagan H. *Space-filling Curves*. Springer: New York, NY, USA, 1994.
19. Docan C, Parashar M, Klasky S. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Proceedings of International Symposium on High Performance and Distributed Computing (HPDC'10)*, Chicago, Illinois, 2010; 25–36.
20. Gelernter D. Generative communication in Linda. *ACM Transaction on Programming Language System* 1985; **7**(1):80–112.
21. Chapman B, Haines M, Mehrota P, Zima H, Van Rosendale J. Opus: a coordination language for multidisciplinary applications. *Scientific Programming* 1997; **6**(4):345–362.
22. Bal HE, Kaashoek MF, Tanenbaum AS. Orca: a language for parallel programming of distributed systems. *IEEE Transaction on Software Engineering* 1992; **18**(3):190–205.

DOI: 10.1002/cpe

23. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes L, Parker S, Smolinski B. Toward a common component architecture for high-performance scientific computing. *Proceedings of 8th International Symposium on High Performance Distributed Computing (HPDC'99)*, Redondo Beach, California, 1999; 13.
24. Fasel P, Mniszewski S. PAWS: collective interactions and data transfers. *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC'01)*, San Francisco, California, 2001; 47–54.
25. Edjlali G, Sussman A, Saltz JH. Interoperability of data parallel runtime libraries. *Proceedings of the 11th International Symposium on Parallel Processing (IPPS'97)*, Geneva, Switzerland, 1997; 451–459.
26. Joppich W, Kurschner M. MPCCI—a tool for the simulation of coupled applications. *Concurrency and Computation: Practice & Experience* 2006; **18**:183–192.
27. Nieplocha J, Carpenter B. ARMCI: a portable remote memory copy libray for ditributed array libraries and compiler run-time systems. *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, 1999; 533–546.
28. Nishtala R, Hargrove PH, Bonachea DO, Yelick KA. Scaling communication-intensive applications on blue gene/P using one-sided communication and overlap. *Proceedings of 23th IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Rome, Italy, 2009; 1–12.

DOI: 10.1002/cpe