DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows

Ciprian Docan & Manish Parashar
Center for Autonomic Computing
Rutgers University, Piscataway NJ, USA
{docan,parashar}@cac.rutgers.edu

Scott Klasky
Oak Ridge National Laboratory
P.O. Box 2008, Oak Ridge, TN, 37831, USA
klasky@ornl.gov

ABSTRACT

Emerging high-performance distributed computing environments are enabling new end-to-end formulations in science and engineering that involve multiple interacting processes and data-intensive application workflows. For example, current fusion simulation efforts are exploring coupled models and codes that simultaneously simulate separate application processes, such as the core and the edge turbulence, and run on different high performance computing resources. These components need to interact, at runtime, with each other and with services for data monitoring, data analysis and visualization, and data archiving. As a result, they require efficient support for dynamic and flexible couplings and interactions, which remains a challenge. This paper presents *DataSpaces*, a flexible interaction and coordination substrate that addresses this challenge. *DataSpaces* essentially implements a semantically specialized virtual shared space abstraction that can be associatively accessed by all components and services in the application workflow. It enables *live* data to be extracted from running simulation components, indexes this data online, and then allows it to be monitored, queried and accessed by other components and services via the space using semantically meaningful operators. The underlying data transport is asynchronous, low-overhead and largely memory-to-memory. The design, implementation, and experimental evaluation of *DataSpaces* using a coupled fusion simulation workflow is presented.

Categories and Subject Descriptors

C.5.1 [**COMPUTER SYSTEM IMPLEMENTATION**]: Large and Medium ("Mainframe") Computers

General Terms

Performance, Design, Management, Experimentation

Keywords

Code coupling, Workflows, Data redistribution, I/O, RDMA

1. INTRODUCTION

Emerging parallel/distributed scientific and engineering applications simulate large scale complex physical phenomena to get new insights, find more accurate solutions, and enable the development of realistic models for the phenomena being analyzed. These applications consist of multiple heterogeneous and coupled processes that need to dynamically interact and exchange data on-the-fly during execution. This includes interactions between parallel codes that simulate different components of the physical phenomena and the support processes required to ensure simulation progress as well as the validity and correctness of results, to visualize and analyze data, monitor execution and completion, etc. These processes can often run independently on distinct and distributed resources, and on-demand resulting in interaction, coordination and coupling patterns that are complex and highly dynamic.

For example, consider an application that monitors seismic activity using a wireless sensor network [26], and uses simulation models to predict the possibility of an earthquake. Such an application workflow is composed of concurrently running and interacting processes that monitor the environment, collect and store data, process and analyze this data, run the coupled simulations, and signal corrective actions. Similarly, in fusion science, multiple parallel codes that simulate the edge and core turbulence of a fusion reactor, run concurrently and are coupled through an integrated predictive plasma [8, 25, 20] simulation workflow. These coupled codes typically run at large scales, i.e., thousands or tens of thousands of compute nodes, and produce large amounts of data. These codes also interact dynamically at runtime, and their interactions patterns may depend on the computations and can change over the runtime of the applications.

Emerging end-to-end simulation workflows, such as the examples presented above, thus require support for data-intensive couplings between heterogeneous processes. These applications can run at different spatial and temporal scales, at different levels of parallelism, at different times, and possibly at different locations, and as a result, these couplings have to be dynamic, flexible and often asynchronous. Efficiently supporting such coupling behaviors and associated interaction and coordination patterns presents several challenges. For example, coupling data between applications or application components that run on distinct resources, e.g., clusters or parallel machines, and possibly on different number of processors, requires efficient data transport and data

redistribution. Furthermore, communication and coordination patterns often depend on the state of the computation and are only known at runtime. This requires that the communication schedules are efficiently determined at runtime without imposing synchronization or the collection of global information about the data distribution. Similarly, monitoring or analysis components (possibly instantiated dynamically) may require efficient access to data or data subsets of interest independent of the data distribution or local representation. Finally, data may often have to be transformed before it can be consumed and this transformation has to be performed in-transit.

Implementing these complex and varied interaction and coordination behaviors by using low level messaging or existing parallel programming frameworks such as MPI can quickly become a significant and non trivial effort. For example, in the case of MPI, matching sends and receives must be explicitly programmed for each interaction. As application workflows become more dynamic and complex, MPI-based implementation can quickly become infeasible. Meanwhile, while required asynchronous interactions can be implemented using files, the size of the data involved (typically tens of GB) and the associated latencies of the file systems make this an unattractive option for many applications, especially those involving strong physical coupling, such as the fusion example.

Clearly, we need a flexible interaction and coordination framework with simple high-level abstractions that can support the dynamic and asynchronous data-intensive coupling patterns required by the targeted application workflows. In this paper, we present a framework that addresses the requirements outlined above. Specifically, we present the design, implementation, and experimental evaluation of DataSpaces. DataSpaces provides the abstraction of a semantically specialized, distributed, virtual shared space that can be associatively accessed by all application components and services. It enables *live* data to be extracted from running simulation components, indexes this data online, and then allows it to be monitored, queried and accessed by other components and services via the space using semantically meaningful operators. The DataSpaces design enables it to be scalable and efficiently implemented on and across various high-performance systems and clusters, and can be used by application components to efficiently realize a variety of dynamic coupling behaviors. In fact, DataSpaces is currently being used to support the strong physical coupling requirements of coupled fusion simulation workflow described above.

The rest of this paper is organized as follows. Section 2 presents the DataSpaces conceptual model. Section 3 presents the architecture and design of DataSpaces, and Section 4 describes its implementation. Section 5 presents an experimental evaluation and discusses the results, Section 6 presents the related work, and Section 7 concludes the paper.

2. A CONCEPTUAL OVERVIEW OF DATASPACES

As mentioned in the introduction, DataSpaces provides an abstraction of a virtual shared space that can be associatively accessed by application components to realize various interaction and coordination behaviors. The DataSpaces virtual shared space abstraction is semantically specialized – it uses an addressing scheme that is specialized to the application formulation, for example, the geometric discretization and coordinate system that defines the application data domain.

The goal of the DataSpaces abstraction is to enable the *live* data of interest, which is extracted from a running application, to be efficiently indexed, and asynchronously accessed and processed by other components in the application workflow. The data of interest can be dynamically specified as tuples. Tuples are essentially key-value pairs, and the keys are defined using an application-specific address space. For example, the key field may be composed of (1) a set of coordinates or a similar geometric descriptor which is meaningful to the application to define a region of interest in the address space such as a grid block or mesh region, along with (2) an application property or attribute function such as temperature, velocity, potential. The value field may contain the data associated with the function and can be defined over the specified region.

Note that the DataSpaces abstraction is dynamic in that application components/services can attach to existing DataSpaces as long as they are aware of the addressing used by the space. Similarly, an application component/service can dynamically detach from DataSpaces.

The DataSpaces indexing engine is also based on the application specific address space, and is designed to be distributed and efficient to enable online indexing of large amounts of data. For example, space-filling curve [5] indexing is used in cases where the address-spaces is based on a geometric discretization of the application domain. Indexing is typically done on a dynamic set of nodes representing a data staging area, and the resulting index space is distributed across these nodes. As a result, look-up and access operations based on this distributed index are decentralized, and allow the overall DataSpaces framework to be scalable, flexible and to handle large data volumes. Note that data capture in DataSpaces uses an asynchronous extraction model, which enables the application to make progress while data is being extracted.

Once indexed, the data is stored in-memory in the staging area and can be accessed using the DataSpaces query engine that provides flexible querying semantics. An application component can query any region in the overall address space. A key aspect of the querying engine is its ability to optimize for access locality in the application-specific address space. For example, in the case of an address space based on the geometric discretization of the application domain, application queries typically correspond to closed regions in the domain and as a result are localized to a subspace, and do not require global synchronization or information gathering.

Higher level data sharing and coupling abstractions are built on top of the DataSpaces query engine to support more complex and dynamic interaction and coupling patterns. For example, applications can query data regions on-demand, obtain continuous notifications of data availability, realize

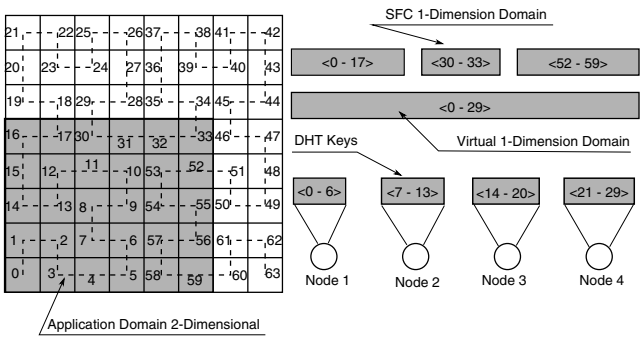

Figure 1: Distributed hash table (DHT) mapping.

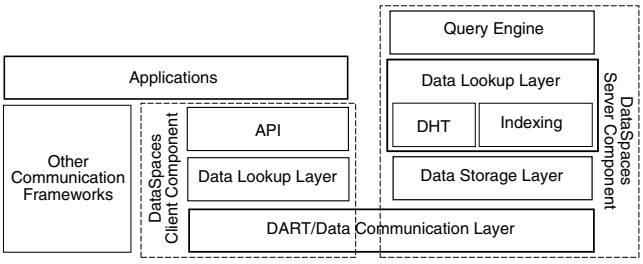

Figure 2: DataSpaces architecture.

the publisher-subscriber pattern, redistribute data, or monitor regions of interest with custom data filters, e.g., min, max, sum, avg, that may trigger application defined actions. DataSpaces automatically computes dynamic communication schedules based on the interactions and coupling requirements specified by the interacting applications components, and alleviates the burden of determining, maintaining, and updating these communication plans from the application programmer.

DataSpaces also supports coordination between application components using a model that is conceptually similar to the TupleSpaces [22] model. However, TupleSpaces defines a general shared space that is globally accessible, and a generic addressing mechanism, which makes high-performance and scalable implementations challenging. In contrast, DataSpaces is decentralized and scalable. Using DataSpaces, the applications components can interact/coordinate by sharing data tuples that can be inserted/retrieved in a decoupled manner using the asynchronous operators defined.

3. ARCHITECTURE

DataSpaces is a distributed framework constructed on a dynamic set of nodes. A key component of DataSpaces is a distributed hash table (DHT) that supports fast data lookup operations as well as fast metadata propagation for data stores. The index for this DHT is derived from the application domain, and corresponds to the address space used by the shared space abstraction to provide application access to the space. For example, for a multi-dimensional geometric domain, the Hilbert space-filling curve (SFC) [5] is used to generate the DHT index. An SFC is a mapping function from a n-dimensional domain to a 1-dimension domain. This function has important spacial properties. First, it passes through all the points in the n-dimensional domain only once in an algorithmic defined order, that defines a linear 1-dimension domain. The linear domain can be used as the keys index space to create and access the DHT. A data point from the geometric application domain can be uniquely identified using either the coordinates in the n-dimensional domain or the index in the 1-dimension domain. Similarly, a continuous data interval, e.g., array, matrix, cube, can be identified by a pair of lower an upper coordinates in the n-dimensional domain, or a set of pairs of lower and upper indices in the 1-dimension domain. Second, it preserves data locality, i.e., adjacent points in the 1-dimension domain are mapped from points that are close together in the n-dimensional space. This enables fast and local mapping of multi-dimensional application data into the index space without mapping every data point, but only the points corresponding to the lower and the upper coordinates.

DataSpaces constructs the DHT from the multi-dimensional application domain as follows. First, it maps the entire application domain to sets of intervals or index pairs defined by a lower and an upper index value in the 1-dimension index domain. Then, it assigns the set of intervals to the nodes in the staging area that will host the space. Depending on the geometry of the application domain, the corresponding interval in the index domain may not be contiguous, or the intervals may have different sizes, which can lead to a non-uniform distribution of keys to the nodes. The DataSpaces load-balancing mechanism addresses this problem. It performs an additional translation from the 1-dimension index space onto a virtual and continuous space and splits the virtual space equally to the nodes of the DHT. This method also compacts the intervals of the index space and allows DataSpaces to store only the start and end indices, and thus reduce the memory footprint of the index space. Each node of the DHT is responsible for a distinct and continuous interval in the virtual space, which is equivalent to a set of distinct intervals in the 1-dimension space and a distinct set of regions in the application data domain. Note that the DHT itself is only used to store the metadata that describes the data being shared, while the actual data is stored separately in a different layer.

DataSpaces has a layered architectures which contains a *communication layer*, a *data storage layer*, and a *data lookup layer*.

Communication Layer. The Data Communication Layer (DCL) defines custom communication primitives, which leverage advanced network interconnections that are commonly available in high-performance machines, for low-latency and high-throughput data transfers such as remote direct memory access (RDMA). It is independent of, but can complement existing communication frameworks for distributed memory machines such as message passing, e.g., MPI or shared memory library, e.g., ShMem.

DCL defines the communication protocol between the applications and DataSpaces and between distinct staging nodes hosting DataSpaces. These primitives are used for data capture, transport of data to/from the space as well as for send-

ing coordination and control messages among the nodes.

The DCL primitives enable efficient and asynchronous communications and data transport, and support better management of limited resources available on the host nodes, e.g., memory in case of RDMA memory-to-memory data transfers. Asynchronous data capture enable application data to be captured while the application is running without interrupting the computations.

Data Storage Layer. The Data Storage Layer (DSL) allocates in-memory storage space at each of the nodes in DataSpaces and manages the memory buffers to create the abstraction of a distributed virtual storage for application data. It stores the data captured from applications locally at the DataSpaces nodes, and complements the DHT, which stores the metadata (e.g., geometric descriptor) associated with the data. The application data and the corresponding index are stored in memory to enable faster access time as compared for example to a database that would store the information on disk, and thus incurs the associated latencies. Applications that interact through the space usually run on different number of processors and have different data distributions. For example, an application can distribute its domain so that each node is assigned a distinct region whose corresponding SFC-index mapping spans multiple nodes in the DataSpaces DHT. When such a node inserts its data in the space, the DSL stores the data for the entire region at a single node and the metadata for that region is propagated to the other nodes to which the corresponding span is mapped. DSL stores only a single copy of data inserted for each application region, and does not split the data to avoid extra communication overheads. Instead, data is split and assembled at the application level when it is queried and retrieved.

DSL can store multiple versions of data and provide quick access for servicing retrieve queries for a specific version of data. The number of distinct versions that can be stored in DSL is user defined as a configuration option for DataSpaces and is only limited by the size of physical memory available at each node. When an application queries the space for a specific version of data that is not yet available, DataSpaces returns an error code and allows the application to continue. The application can resubmit the query for the same data version at a later time, can query a different version of the data or a different region, or wait until data becomes available for its region of interest. The query options that DataSpaces provides can be viewed as building blocks that can be used to realize flexible interactions at the application level as required by the application computations without enforcing strict synchronization constraints. Note that different applications can access different data versions without requiring any synchronization, increasing the level of concurrency at the application level.

DSL serves as a back-end for the query engine for data manipulation. While the data for an insert query is localized to one node in the space, the data associated with a region specified by a retrieve query usually spans to multiple number of nodes. The DSL at each node involved in a retrieve query can service subsets of data that match or intersect the region in the query. The data retrieval process is independent of the application data distribution. It is DataSpaces that internally splits the region of a retrieve query into subregions and retrieves and services each piece returned by nodes in the space. These are then assembled and returned to the application. This query and assemble mechanism allows transparent data redistribution between applications, and enables applications with heterogeneous distributions to be simply coupled - an application does not have to be aware of the data distribution of the other application that it interacts with.

The property of data locality is preserved at both applications as the application domain is divided into contiguous regions that are assigned to the DataSpaces nodes using the SFC mapping. This enables localized communication, because the interaction between regions in the application domain involve a relatively small number of nodes in the space, which reduces the communication overhead.

Data Lookup Layer. The Data Lookup Layer (DLL) translates or hashes the geometric descriptors that describe application data regions into keys for the DHT , and routes the queries to the appropriate DataSpaces nodes that are responsible for the corresponding regions in the application domain.

DLL hashes the geometric descriptor of a data query to intervals in the 1-dimension domain and determines the nodes in the space to which these keys are assigned. For queries that insert data in the space, it propagates the geometric descriptor for the data region to all the nodes that have keys for that region. Similarly, for queries that retrieve data from the space, it routes the query to all the nodes that have keys in that region.

Computing the hash for a geometric descriptor is a local operation that does not require global knowledge or synchronization and is consistent across the nodes of the space. DLL is a distributed service that allows an application to join the space at any time and attach to any of the DataSpaces nodes, and to query any data region from the application domain.

DLL maintains the DHT entries for data and geometric descriptors and ensures coherency. For example a query can insert a data region to a node in the space which has no key in that region. DLL updates the local DHT entry at the node and forwards the geometric descriptor to all the nodes that have keys in that region and allows a retrieve query which hits the node to be serviced directly, or to be routed to the node holding the data through the nodes holding the keys.

The Query Engine. The query engine is the key components of DataSpaces and builds on the layers described above, i.e., the DLL layer for data look-up and query resolving, the DSL layer for data storage and manipulation, and on the DCL layer to accept and service data queries. It supports different query operators (see Table 1), such as operators for insertion, retrieval and registration of regions of interest, notification updates or monitoring with custom filters, etc. For example a typical query to retrieve a region of data from the space is specified using the *get()* operation, i.e., get("electrons", 100, 10, 0, 150, 20, 0, &electrons_tab).

Table 1: DataSpaces user interface

`ds_init`	Initializes the framework and registers a new application with the DataSpaces framework.
`ds_finalize`	Releases resources and unregisters an application from the DataSpaces framework.
`ds_get`	Queries the space to retrieve data specified by a geometric descriptor
`ds_put`	Queries the space to insert data specified by a geometric descriptor
`ds_filter`	Queries the space to retrieve data specified by the filter operation, e.g., *min()*, *max()*, *average()*, *sum()*
`ds_monitor`	Queries the space and registers a region of interest for which to receive updates continuously
`ds_lock_on_[read, write]`	Helper routines to obtain exclusive rights to access the space and ensure data coherency
`ds_unlock_on_[read, write]`	Helper routine to release the exclusive accessing rights to the space

The query engine is the main interface to DataSpaces and is used by the application to implement flexible, dynamic, and custom application interactions.

4. IMPLEMENTATION OVERVIEW

This section describes the prototype implementation of the DataSpaces framework. The implementation builds on the DART [13] communication and data transport substrate. In this section, we first present an overview of DART and then describe the implementation of DataSpaces.

4.1 Overview of DART

DART [13] is an asynchronous communication and data transport substrate for large-scale parallel applications. It builds on advanced communication capabilities provided by emerging interconnects, such as RDMA and one-sided communication, that provides mechanisms for extracting data from a running parallel application and transporting this data for external processing, storage, etc. For example, it allows applications to overlap communication and data transport with computation, and to reduce the overheads of I/O on the application by offloading the expensive I/O operations to dedicated staging nodes.

DART has a client-server architecture. DART clients are integrated with the application and provide the interfaces that allow the application to invoke DART services such as I/O offload, data transfers and streaming. DART servers typically run on a set of dedicated nodes that form the staging area, but may also run on general purpose login or service nodes. The servers support multiple data destinations such as streaming to remote services (e.g., monitoring or visualization), archiving to local or remote storage, and also support operations within the staging area such as data redistribution or processing for application coupling. The key services provided include node registration and discovery, node unregistration, data transfer and notification, and node coordination. These services form a lightweight communication layer that can be easily ported to different commodity systems.

The higher-level communication abstractions and mechanisms provided by DART are general and can be used to implement other communication/coordination patterns, and can complement existing parallel communication frameworks such as MPI. For example, DART provides an RPC-like abstraction with a plug-in interface that can be used to extend it to support higher level coordination and data services. The DART abstractions also hide the complexities of the underlying communication system. For example, in the case of RDMA, it hides issues such as memory management, i.e., buffers setup, registration, de-registration and cleanup, and management of asynchronous transfers, i.e., monitoring completion or enforcing flow control due to the limited memory resources.

A more detailed description of the DART framework, its architecture, implementation, sample applications, and evaluation can be found in [12].

4.2 DataSpaces Implementation

The DataSpaces implementation consists of two key components, a *DataSpaces Client* and a *DataSpaces Server*. These components provide data services that plug into the DART substrate and extend its functionality. The component implementations are single-threaded and the communications are non-blocking and re-entrant. While this approach makes the implementation of the framework more complex, it does have several advantages, such as eliminating overheads due to thread scheduling, synchronization, signaling or data locking, as well as offering better memory control for the communication buffers.

Although the components are single threaded, they allow the execution of concurrent data transfers with minimum system overhead. For example, a data transfer operation does not block the application until its completion because it is asynchronous, and the application can continue and even start a new data transfer. If the communication with a peer node blocks because, for example, there are not enough resources at the destination, DataSpaces can suspend the current transfer and initiate a different one with a different peer node, and later resume the suspended transfer when the blocking condition has cleared.

The client component is integrated with the user application and provides the interfaces for interacting with DataSpaces. It prepares, i.e., organizes for its internal representation, and submits data queries received from the application to the space. It also assigns a completion routine for each query, which is called automatically by DataSpaces after the completion of a retrieve or send operation, and releases resources or assembles the data. Note that the distribution of data differs between the application and DataSpaces. For example, a continuous region in the application domain is split by DataSpaces into smaller sub-regions, and continuous pieces of data associated with these sub-regions are located and stored at different nodes in the space. When a DataSpaces client requests such a region, it first connects to its corresponding node in the space, gathers the metadata and the location for all the pieces, and then requests the

data from each individual node that has a sub-region. To complete a retrieve, monitor, or filter query, the client component waits to retrieve all the data pieces, assembles them into the queried continuous data region, and then returns the result to the application. The client can also transform the data representation (e.g. column major to row major) before forwarding it to the application. Thus the conversion/redistribution of data exchanged between applications is handled transparently by the DataSpaces.

The client component dynamically registers or unregisters an application with/from DataSpaces, allowing the applications to dynamically join, interact-with, or leave a space at runtime. The nodes that an application runs on are typically assigned to it by the execution environment, and their identifiers are only known after it starts. In order to allow the applications to use the memory-to-memory data exchange capabilities provided by DART, the client component publishes its own node identifier to DataSpaces during the registration process, and discovers the identifiers of the other DataSpaces nodes when it joins the space. Similarly, when an application disconnects from DataSpaces, the client components notify DataSpaces so that the identifier references can be properly cleaned up.

The DataSpaces server component consists of multiple server instances that run independently of user applications, on a set of nodes in the staging area. The server component provides the communication interfaces for interactions with the client components and other server component instances, such as, for example, for metadata distribution, query routing, or application registration. It also allocates the memory buffers to provide the storage required for the data inserted into the space by the applications.

Similar to the client component, the server component is also single-threaded. It enables efficient data transfers with minimum overhead on the applications, and flexible and non-synchronized memory management of the storage and communication buffers. The server component follows the server initiated communication model and the data transfers to/from the space are initiated by the server component. This approach gives greater control over the communication buffers, and, since the amount of memory available at the servers is limited, the servers can protect their memory buffers from overflowing while handling multiple data transfers using a flow control mechanism. The server component monitors every message exchanged with a peer node in the system and updates a count of the available communication resources for that peer. If the resources are exhausted, communication with that peer is suspended until new resources become available. Because the communications are non-blocking and re-entrant, the server component can process other queries and control messages in the meantime.

The server component extends DART functionality with plugins for data services. The registration service handles applications, as well as other server component instances, and assigns a unique identifier for each node that is later used for communication. The query service handles data insert and retrieve operations. For a data insert operation, the server component retrieves and stores the data defined in the query from the nodes of the application. It then hashes the descriptor for that region and propagates the metadata to the nodes in the space, which are responsible for regions from the application domain that overlap with the region from the query. A data retrieve, filter or monitor is a two step operation. In the first step, the server component hashes the descriptor corresponding to the region from the query, and replies with the list of nodes in the space that have data which overlaps with that region. In the second step, the selected nodes reply with data that corresponds to the common sub-region.

A sample application code that uses the *DataSpaces Client* to interact with the DataSpaces is presented below.

Listing 1: Data sharing example using DataSpaces.

```
/* Application sample code to submit Insert
   queries to DataSpaces */
ds_init()
/* Application computation loop start */
/* Application computations */
ds_lock_on_write()
/* Example of <geometric descriptor>:
      <10, 20, 5; 50, 40, 60> */
ds_put(var name, version,
      <geometric descriptor>, var pointer)
ds_unlock_on_write()
/* [Application computations] */
/* Application computation loop end */
ds_finalize()
```

Listing 2: Data retrieval example using DataSpaces.

```
/* Application sample code to submit Retrieve,
   Notification or Monitor queries to DataSpaces
   with a custom filter */
ds_init();
ds_monitor(var name,
      geometric descriptor, var pointer)
/* Application computation loop start */
ds_lock_on_read()
/* "filter name" could be {min, max, avg ...} */
ds_filter(var name, version, "filter name",
         <geometric descriptor>, var pointer)
ds_get(var name, version,
         <geometric descriptor>, var pointer)
ds_unlock_on_read()
/* Application computation loop end */
ds_finalize()
```

5. EVALUATION

The DataSpaces prototype implementation was evaluated on a Cray XT4 platform at the Oak Ridge National Laboratory (ORNL) computing center, specifically the Jaguar [18] machine. Jaguar's XT4 partition has 7832 dedicated compute nodes, in addition to the private I/O nodes and the 8 public login nodes. Each compute node has a quad-core AMD Opteron CPU that runs at 2.1 GHz and 8GB of memory. The nodes are inter-connected in a 3-D torus topology by SeaStart2+ routers with a direct HyperTransport link to each node's memory, which has a peak performance of 6.4GB/s. The compute nodes run the Compute Node Linux

(CNL) micro-kernel, which is tuned for low latency communications and high performance computations.

The DataSpaces evaluation presented in this section is driven by interactions in the coupled fusion science application workflow that simulates the edge and core turbulence of a fusion reactor. The evaluation presented consists of multiple experiments. The first set of experiments aim to better understand the overall behavior of the DataSpaces framework and explore the impact of the system aspects e.g., the architecture of the XT4 platform, that influence communication performance. The second set of experiments analyze the performance and scalability of DataSpaces with increasing number of processors running the applications, and with the size of the data being shared through DataSpaces. The third set of experiments evaluate a dynamic interaction scenario using five heterogeneous applications that run on different number of processors and use different queries to interact with DataSpaces and with each other. The fourth set of experiments demonstrate and evaluate interactions in a distributed application workflow where application components runs on distributed resources.

Note that in the experiments below, rather than use the entire simulations codes described above, we use synthetic code that capture the interaction and coupling behaviors while removing the complexity the computational aspects.

5.1 Application Interaction in a Coupled Fusion Simulation Workflow

In plasma science, complex scientific workflows can describe applications interaction to enable physicists to study the dynamic interaction of kinetic effects that cause a buildup of the edge pedestal in plasma density and temperature profiles and large bootstrap currents with so-called ELMs that may limit pedestal growth and tokamak reactor performance [10].

In such a workflow, one application, e.g., XGC0 [8], calculates the kinetic edge pedestal buildup. At start-up, the application code reads the magnetic equilibrium data that describes the simulated fusion device and the physical experiment to produce and evolve a plasma profile. At the end of a simulation step, it writes the plasma profile data, which a different application, e.g., M3D-OMP [20], code then uses to update the equilibrium data based on the plasma density, temperature and bootstrap current.

XGC0 reads the updated equilibrium data to maintain self-consistency and to continue to evolve plasma particles. The linear stability of the updated equilibrium data is verified by a reference code, ELITE [25], which executes a parameter sweep over toroidal mode numbers using the plasma density profile data, and the magnetic equilibrium data.

The two simulation codes, e.g., XGC0 and M3D-OMP are loosely coupled, e.g., XGC0 may produce plasma profiles more often than M3D-OMP produces equilibrium updates. XGC0 can continue to evolve plasma particles using the current version of the equilibrium data and does not have to wait on M3D-OMP. This is valid because the magnetic equilibrium is not evolving rapidly.

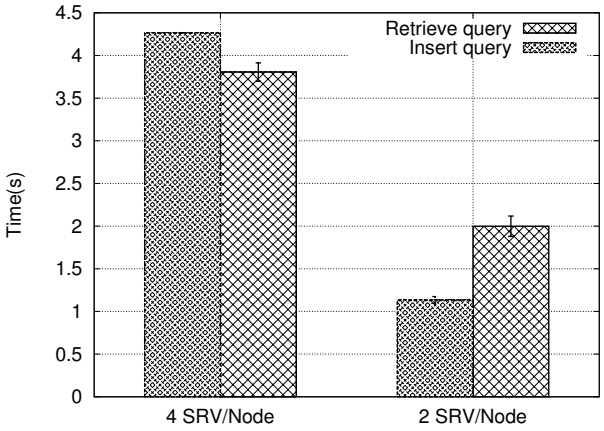

Figure 3: The effect of co-locating multiple DataSpaces server instances on a single compute node.

5.2 Experiment I: Impact of Environment Factors

The results from initial evaluation showed that the characteristics of the host platform, i.e., the XT4 architecture, can influence the communication behavior and performance of the DataSpaces framework. For example, on the Jaguar machine, the execution environment allows the user to control the number of MPI processes and the number of OpenMP threads for an application that can be placed on the same quad-core compute node.

This experiment used two test application components that interacted at runtime via DataSpaces using insert and retrieve queries. In the experiment, one application components ran on 32 processors (cores) that concurrently submitted insert queries to DataSpaces, and the other application component ran on 4 processors (cores) that concurrently submitted retrieve queries. DataSpaces ran on 4 processors (cores), and the 4 processes were mapped differently to the compute nodes during the experiment, i.e., two and four processes per compute node. The the total amount of data exchanged during one query was 3.5 GB and the experiment consisted of 100 queries from each application component.

The results of this experiment are presented in Figure 3, and show that the behavior of the framework depends on the mapping of DataSpaces server processes to compute nodes. When DataSpaces co-locates four server processes on the same compute node, the query times for both insert and retrieve operations are higher as compared with the case where only two server processes are mapped to a node. This is due to the increased contention at the node's memory system and the HyperTransport link being shared between the cores. While other server mapping schemes are possible, they do not present any practical advantages as they either waste resources, e.g., one server per compute node requires at least the double number of resources, or do not evenly match the ratio of application compute nodes to the server compute nodes, e.g., three servers per compute node may lead to data transfer imbalances. In the remaining experiments presented in this section, we will map two DataSpaces processes to each compute node.

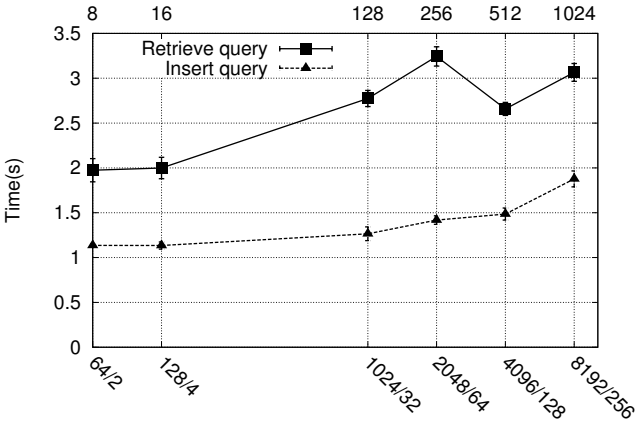

Figure 4: Execution time for *Insert* and *Retrieve* query types. On the X axis, bottom part represents the number of processors to submit insert queries / number of DataSpaces servers, and the top part represents the number of processors to submit retrieve queries.

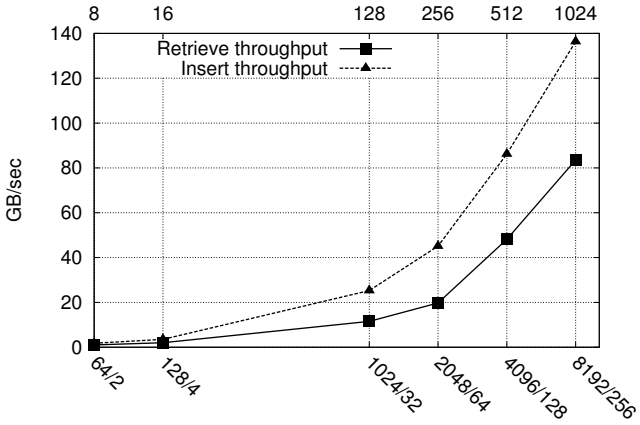

Figure 5: Application level query execution throughput. On the X axis, bottom part represents the number of processors to submit insert queries / number of DataSpaces servers, and the top part represents the number of processors to submit retrieve queries. For the query data size please see Table 2.

5.3 Experiment II: DataSpaces Scalability

Scientific applications simulating complex phenomena typically run on large numbers of processors and produce significant amounts of data. The scalability experiment evaluated the ability of the framework to respond to concurrent queries to insert or retrieve data to/from DataSpaces using a varying number of application processes and data sizes.

This experiment used two synthetic application codes that captured the interaction pattern of real physics simulation codes without all the computational intricacies, to enable us to focus the evaluation. The applications shared a common global domain, which they discretized and distributed differently as they were running on different numbers of processors. In this case, the global domain space was 3-dimensional and each application assigned a distinct number of processors for each dimension, i.e., $X \times Y \times Z$ (see Table 2). As it is usually the case with scientific simulations, each processor queried DataSpaces for the region of the application domain mapped to it, and each application queried the entire domain. One application inserted data and the other retrieved data.

This experiment measured weak scaling and varied the number of processors and consequently the number of concurrent insert queries from 64 to 8192; the number of processors and concurrent retrieve queries from 8 to 1024; and the number of DataSpaces servers from 2 to 256. Each DataSpaces server provided 1GB of memory for local storage and serviced 32 inserting queries, each inserting 32MB, and 4 retrieving queries, each retrieving 256MB. Each processor of the inserting application was assigned a region of size $128 \times 128 \times 256$ from the global domain space, and each processor of the retrieving application was assigned a region of $128 \times 512 \times 512$. The setup for each experiment in this set is presented in Table 2. The processor distribution represents the number of processors assigned for each dimension in the application domain. In a experiment, each application executed 100 queries to simulate 100 global data exchanges.

Figure 4 presents the time required by DataSpaces to service the *Insert* and *Retrieve* queries, and Figure 5 presents the application level throughput obtained by exchanging data using DataSpaces. After performing the experiment with 4 DataSpaces servers, we decided to test the framework with larger application sizes and skipped the two intermediary data points corresponding to 8 and 16 DataSpaces servers. The query service time was recorded at the application level and represents the total time to prepare, submit, resolve and transfer the data for an *Insert* query, and the total time to prepare, submit, resolve, transfer the data, and assemble the reply for a *Retrieve* query. The timing values on the graph in Figure 4 represent (1) the average over the number of processors for each query, and (2) the average and standard deviation over the number of application queries to better capture the global behavior of the framework.

The results in Figure 4 show good overall scalability with the number of application processors and data sizes. For retrieve queries, the time difference between the small and the large scale cases (i.e., first and last data point on the X axis) is a little over 1 second, while for the insert queries the difference in times is less than 1 second. At the same time, the difference in size of the data exchanged for the same two cases is 254GB (256GB in the last case, and 2GB in the first case). As the experiment used weak scaling, one would expect that the query time to be constant. However, considering the scale of the applications, the small increase is acceptable and it is due to sustained pressure on the shared communication links. Specifically, 256GB of data were continuously moved from one application to DataSpaces and from DataSpaces to the other application for 100 application queries, with a resulting flow of 512GB, (0.5TB) of data, per application interaction scenario at the DataSpaces level.

The difference in times between the insert and retrieve queries

Table 2: Application scenario setup for the evaluation of DataSpaces performance and scalability.

Number of parallel *Insert* queries	64	128	1024	2048	4096	8192
Number of parallel *Retrieve* queries	8	16	128	256	512	1024
Query data size	2GB	4GB	32GB	64GB	128GB	256GB
Processor distribution for *Insert* query	$4 \times 4 \times 4$	$4 \times 4 \times 8$	$8 \times 8 \times 16$	$8 \times 16 \times 16$	$8 \times 32 \times 16$	$16 \times 32 \times 16$
Processor distribution for *Retrieve* query	$4 \times 1 \times 2$	$4 \times 1 \times 4$	$8 \times 2 \times 8$	$8 \times 4 \times 8$	$8 \times 8 \times 8$	$16 \times 8 \times 8$

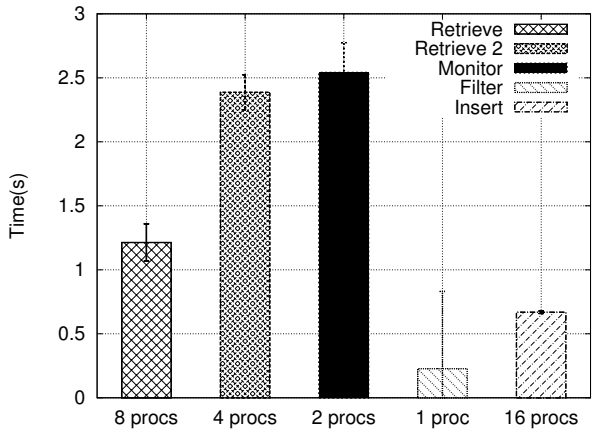

Figure 6: Dynamic interaction between multiple and distinct application components that run on different number of processors using DataSpaces.

is due to the difference in data sizes at each processor, i.e., 32MB per processor for the insert query and 256MB per processor for the retrieve query. We believe that the higher times for the retrieve queries corresponding to the experiments using 32 and 64 DataSpaces servers (data points 3 and 4 on the X axis of the graphs), are due to either (1) interference on the shared links between DataSpaces and the retrieving application with other applications that were running on the system at the same time, or (2) the distribution of data at the storage layer of DataSpaces, i.e., data was more spread out, and as a result, the retrieve query was decomposed into multiple smaller regions that spanned multiple DataSpaces servers and had to be assembled by the DataSpaces client on the application side. As shown in the figure, the time for the insert queries for the same cases were not affected.

Note that although the experiments used synthetic application codes, the scale in both number of processors and size of data exchanged in the experiments accurately represent the real codes.

5.4 Experiment III: Dynamic Interactions

Complex application workflows that connect multiple simulation codes generate dynamic coordinations and interactions patterns at runtime between heterogeneous application components that run independently on distinct resources and resource configurations, e.g., number of processors. DataSpaces can be used to enable these dynamic interactions. These experiments evaluate the performance of DataSpaces for such a dynamic interaction scenario.

The focus of the evaluation in these experiments is on application interactions and not on the computations, and the experiment used five distinct synthetic application components for this purpose. The first application ran on 16 processors and simulated the model of a solver component which produces partial solutions that need to be shared with other application components. Each solver processor inserted its associated data into DataSpaces. The second application component ran on 8 processors and simulated a helper application that retrieves partial solutions from the space. The third application component ran on 4 processors and simulated a solution validation code that retrieves partial solutions from the space. The fourth application component ran on 2 processors and simulated a monitoring code that continuously monitors the solutions inserted into DataSpaces. Finally, the fifth application component ran on 1 processor and simulated a data filtering service that retrieves only the minimum value of the partial solutions. The experimental results are presented in Figure 6 and show the query execution times for exchanging data using DataSpaces. The applications simulated 100 interaction scenarios and exchanged 2GB of data for each scenario. Once again, the results are an average over the number of processors and the average and standard deviation over the number of interaction scenarios. Each query exchanged (i.e., inserted or retrieved) the same amount of data except the case of the *Filter()* query, which retrieved only a single value from the space. The difference in times between the different queries is due to the number of processors that the corresponding application was running on, which internally determined the data distribution and the size of data exchanged for each query. For example, the queries for the applications running on a larger number of processors finished faster because the data size queried per processor was smaller.

5.5 Experiment IV: Distributed Interactions

Scientific application workflows can require interactions between components that run on distributed resources. For example a plasma simulation code may run on a large high-performance resource at a supercomputing center, while a visualization code or a code that checks the stability of partial solutions can run at a remote site on a smaller cluster and can retrieve partial solutions or simulation updates to refresh a visualization frame. This set of experiments demonstrates the ability of DataSpaces to support application workflows and the required interactions in a distributed setting, and

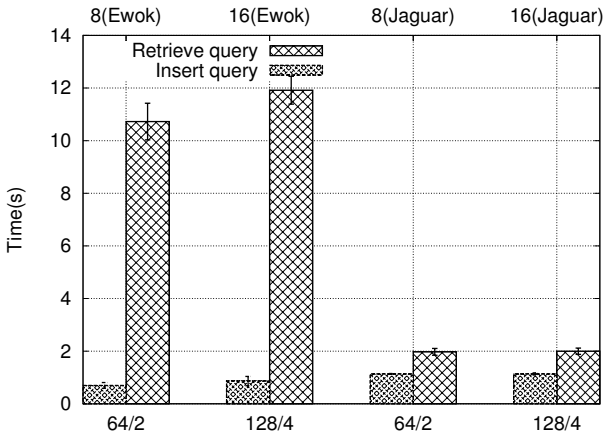

Figure 7: Enabling dynamic interactions for applications that run on distributed resources, i.e., Jaguar and Ewok, using DataSpaces. On the X axis, bottom part represents the number of processors to submit insert queries / number of DataSpaces servers, and the top part represents the number of processors to submit retrieve queries and their location.

evaluates its performance. In this experiment, a synthetic application code, based on the plasma simulation code, ran on the compute nodes of the Jaguar system, and used *Insert* queries to publish data to DataSpaces. DataSpaces ran on the login nodes of the Jaguar system, and another synthetic application component, which represented the visualization code, ran on the compute nodes of a remote cluster of smaller size (i.e., the Ewok system), and used *Retrieve* queries to fetch the data from DataSpaces. DataSpaces and the application running on Ewok used a different transport in DART, which is based on TCP sockets.

For this experiment, the applications simulated 100 time steps and for each time step they issued a set of queries to insert and retrieve data from the space. The total size of data exchanged in each query was 2GB when using 2 DataSpaces servers, and 4GB when using 4 DataSpaces servers. The results presented in Figure 7 show the average time and standard deviation to execute a complete application query from all the processors running the application components. For comparison, Figure 7 also plots the execution times for a similar setup in which both the applications and the DataSpaces framework ran on the compute nodes of the Jaguar machine. As expected, the time value for the *Retrieve* query is higher in the distributed scenario, because the data is moved across machines in this case, and the shared communication link between the machines, i.e., Gigabit Ethernet, is slower than the link between the compute nodes of the Jaguar machine. This experiment indicated the performance that can be expected if a coupled simulations were run in a distributed environment.

6. RELATED WORK

Efficient data transfers to support I/O, communication, application coordination, data sharing, and code coupling is an active field fueled by the requirements of emerging petascale systems and applications, and has been explored by multiple research efforts. For example, similar to our work, prior research has explored the approach of using a small and dedicated partition (i.e., a staging area) to offload I/O services. Previous efforts have successfully used such a staging area to support server initiated data transfer techniques with asynchronous completion. In [19] the authors try to maximize disk I/O by organizing the data so as to minimize disk seek latencies. A similar approach has been used in the Panda project [21] to optimize communication and collective operations, in [11] where the authors try to identify the I/O requirements for large scale applications producing massive amounts of data, and in the DataStager project [1] to minimize the data transfer overheads on a running application through custom scheduling schemes.

Data parallel languages, such as HPF [15] or Fortran-M [14] are well suited for SPMD parallel applications, but have limited or no support for interactions between heterogeneous applications. Opus [9] implements a coordination language to address the interactions between heterogeneous applications. It defines parallel data structures, i.e., SDAs, and methods that can exclusively operate on the data, based on input conditions or filters. The interactions between applications are supported through SDA method calls. However, the language does not define how the data is moved or redistributed between different applications.

CCA [2] is a group effort focused on creating a standard interface that enables interoperability between heterogeneous, high performance computing applications. Using this model, the applications are represented as components that implement *Provides* and *Uses* service Ports, which can be connected at runtime to create composed components and realize application-application interactions. However, the interface does not define how the data is transferred, redistributed, or how the communication schedules are computed, and this is a task left to the implementation. In fact, the DataSpaces framework can potentially be used to implement these functions.

There are multiple projects that implement the CCA specification. DCA [3] is a distributed framework that addresses the $M \times N$ data redistribution problem between distinct components. In DCA, communication schedules are implemented using parallel remote method invocations, which require global synchronization between application components and all processors to participate in the call. The MCT [16, 17] is another project that uses the CCA approach to address the problem of application coupling and data redistribution. It abstracts the applications as components whose interactions are coordinated through an external Coupler component. To achieve the coupling and data exchange, MCT constructs local to global data mappings for each application decomposition, collects and uses them in the Coupler component to create Router and Rearranger components that compute the communication schedules for data exchange between components. Before exchanging the data, it is the application's responsibility to organize data and copy it into special data structures defined by MCT for coupling. This makes the MCT toolkit very tightly integrated with the applications. In contrast, the DataSpaces framework allows applications to exchange custom application defined structures. Furthermore, DataSpaces

is loosely and asynchronously integrated with applications and it transparently computes and resolves communication schedules for data redistribution.

Data redistribution for exchange between heterogeneous applications is also explored in other projects, for example, DataCutter [4]. The DataCutter project attempts to optimize the movement of archived data from the storage system to the end consumer. The approach consists of executing filtering operations for data reduction at the storage, or in-transit to the data consumer. DataCutter is complementary to DataSpaces and can be used to consume data from DataSpaces. DataTurbine [23] is a similar project that orchestrates data-movement from a data acquisition source to the storage system. This project is also complementary to DataSpaces and can be used as a data source for DataSpaces.

An efficient data exchange framework is implemented in the GASNet project [6]. The project uses the Partitioned Global Addresses Spaces (PGAS) language and compiler to translate high level annotations in the code into efficient, RDMA-based data transfer routines. It has been ported to different implementations of RDMA for asynchronous and efficient direct memory-to-memory data transfers. However, the project is specialized for SPMD applications and requires a single memory image to be able to manipulate and translate pointer variables. In contrast DataSpaces can can be used to exchange data between heterogeneous application codes that use different parallelism models and data structures.

A related research effort that implements a distributed data sharing framework is the DDSS project [24]. However, the focus and underlying approach in DDSS are different from DataSpaces. DDSS aims to provide an abstraction of a virtual address space in which applications can explicitly allocate memory buffers, can lock/unlock to write and read and thus share information. DDSS is designed to enable web applications in data-centers to make use of modern interconnects, e.g., InfiniBand and *i*Warp.

DataSpaces is also related to our previous work with the Seine project [27, 28]. The abstraction provided by DataSpaces is similar to Seine - it also builds on TupleSpaces model [7] to provide an abstraction of a virtual shared space in which applications can put and get data tuples. However, Seine focused exclusively on code coupling, while DataSpaces provides a more general programming framework for data exchange and supports online data indexing, flexible asynchronous interaction patterns (e.g., publisher/subscriber/notification), custom data filtering, and multiple interacting applications that can join or leave dynamically. Moreover, the DataSpaces implementation can use advance communication systems, i.e., RDMA, and supports asynchronous data transfers to overlap computations with data communications.

7. CONCLUSIONS AND FUTURE WORK

This paper presented the architecture, design and implementation of the DataSpaces framework. DataSpaces is a novel data sharing framework that supports the complex interaction and coordination patterns required by coupled data-intensive application workflows. DataSpaces enables the asynchronous capture and online indexing of *live* data from applications and allow this data to be flexibly queried. This allow dynamic interactions and in-memory data exchanges between heterogeneous applications that run independently on different number of processors. Furthermore, DataSpaces addresses key challenges in coupled application workflows and data coupling, such as efficient exchange of large data sizes, transparent data redistribution and loose synchronization semantics for flexible application coupling.

The experimental evaluation presented in this paper, using both synthetic and real applications, demonstrated the performance of DataSpaces as well as its scalability with the number of processors on which the application executes and the size of the data exchanged, which meet the requirements of real coupled scientific applications. The evaluation also demonstrated the ability of DataSpaces to realize complex and dynamic interaction patterns between multiple applications running on parallel HPC resources at supercomputing centers as well as in distributed environments.

Our current work is focused on comparing the performance of DataSpaces with other relevant approaches, both in terms of performance and functionality. Additionally, we are working on integrating the DataSpaces framework with different applications and evaluating various end-to-end workflow scenarios. We are also working on porting DART and DataSpaces to other RDMA enabled platforms such as InfiniBand, DCMF, and *i*Warp.

8. ACKNOWLEDGEMENTS

The research presented in this paper is supported in part by National Science Foundation via grants numbers IIP 0758566, CCF-0833039, DMS-0835436, CNS 0426354, IIS 0430826, and CNS 0723594, by Department of Energy via the grant number DE-FG02-06ER54857, by The Extreme Scale Systems Center at ORNL and the Department of Defense, and by an IBM Faculty Award, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University.

9. REFERENCES

[1] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC'09)*, June 2009.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, August 1999.

[3] F. Bertrand and R. Bramley. DCA: A distributed CCA framework based on MPI. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, April 2004.

[4] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and

J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of Mass Storage Systems Conference*, March 2000.

[5] T. Bially. *A class of dimension changing mapping and its application to bandwidth compression*. PhD thesis, Polytechnic Institute of Brooklyn, June 1976.

[6] D. Bonachea, P. Hargrove, M. Welcome, and K. Yelick. Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT. In *Cray User Group (CUG'09)*, May 2009.

[7] N. Carriero and D. Gelernter. Linda in context. *Communications of ACM*, 32(4):444–458, 1989.

[8] C. S. Chang, S. Ku, and H. Weitzner. Numerical Study of Neoclassical Plasma Pedestal in a Tokamak Geometry. *Physics Plasmas*, 11(5):2649–2667, 2004.

[9] B. Chapman, H. Zima, M. Haines, P. Mehrotra, and J. V. Rosendale. Opus: A Coordination Language for Multidisciplinary Applications. *Journal of Scientific Programming*, 6(4):345–362, 1997.

[10] J. Cummings. Plasma Edge Kinetic-MHD Modeling in Tokamaks Using Kepler Workflow for Code Coupling, Data Management and Visualization. *Communications in Computational Physics*, 4:675–702, 2008.

[11] J. M. del Rosario and A. N. Choudhary. High-Performance I/O for Massively Parallel Computers: Problems and Prospects. *Computer*, 27(3):59–68, 1994.

[12] C. Docan, M. Parashar, and S. Klasky. Enabling High Speed Asynchronous Data Extraction and Transfer Using DART. *Concurrency and Computation: Practice and Experience*, DOI:10.1002/cpe.1567.

[13] C. Docan, M. Parashar, and S. Klasky. DART: a Substrate for High Speed Asynchronous Data IO. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC'08)*, June 2008.

[14] I. Foster and M. Chandy. Fortran M: a language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[15] HPF Language Specification, Version 2.0, January 1984. `http://www.netlib.org/hpf/hpf-v20-final.ps.gz`.

[16] R. Jacob, J. Larson, and E. Ong. M×N Communication and Parallel Interpolation in CCSM3 Using the Model Coupling Toolkit. *International Journal for High Performance Computing Applications*, 19(3):293–307, 2005.

[17] R. Jacob, J. Larson, and E. Ong. The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. *International Journal for High Performance Computing Applications*, 19(3):277–292, 2005.

[18] `http://info.nccs.gov/resources/jaguar`.

[19] D. Kotz. Disk-Directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, 1997.

[20] W. Park, E. V. Belova, G. Y. Fu, X. Z. Tang, H. R. Strauss, and L. E. Sugiyama. Plasma Simulation Studies Using Multilevel Physics Models. *Physics Plasmas*, 6(5):1796–1803, 1999.

[21] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Supercomputing Conference (SC'95)*, page 57, December 1995.

[22] H. D. Sterck, R. S. Markel, T. Pohl, and U. Rude. A Lightweight Java Taskspaces Framework for Scientific Computing on Computational Grids. In *Proceedings of the 18th Annual ACM Symposium on Applied Computing*, March 2003.

[23] S. Tilak, P. Hubbard, M. Miller, and T. Fountain. The Ring Buffer Network Bus (RBNB) DataTurbine Streaming Data Middleware for Environmental Observing Systems. In *International Conference on High Performance Computing(HiPC'07)*, December 2007.

[24] K. Vaidyanathan, S. Narravula, and D. K. Panda. DDSS: A Low-Overhead Distributed Data Sharing Substrate for Cluster-Based Data-Centers over modern interconnects. In *Int'l Symposium on High Performance Computing (HiPC'06)*, December 2006.

[25] H. R. Wilson, P. B. Snyder, G. T. A. Huysmans, and R. L. Miller. Numerical Studies of Edge Localized Instabilities in Tokamaks. *Physics Plasmas*, 9(4):1277–1286, 2002.

[26] M. Youssef, A. Yousif, N. El-Sheimy, and A. Noureldin. A Novel Earthquake Warning System Based on Virtual MIMO-Wireless Sensor Networks. In *Proceedings of Canadian Conference on Electrical and Computer Engineering (CCECE'07)*, April 2007.

[27] L. Zhang and M. Parashar. A Dynamic Geometry-based Shared Space Interaction Framework for Parallel Scientific Applications. In *Proceedings of the 11th Annual International Conference on High Performance Computing (HiPC'04)*, December 2004.

[28] L. Zhang and M. Parashar. Enabling Efficient and Flexible Coupling of Parallel Scientific Applications. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, April 2006.