CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3407

ActiveSpaces: Exploring dynamic code deployment for extreme scale data processing

Ciprian Docan[1], Fan Zhang[1], Tong Jin[1], Hoang Bui[1], Qian Sun[1], Julian Cummings[2], Norbert Podhorszki[3], Scott Klasky[3] and Manish Parashar[1,*,†]

[1]*NSF Cloud and Autonomic Computing Center, Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ, USA*
[2]*Department of Computer Science, California Institute of Technology, Pasadena, CA, USA*
[3]*Oak Ridge National Laboratory, Tennessee, TN, USA*

SUMMARY

Managing the large volumes of data produced by emerging scientific and engineering simulations running on leadership-class resources has become a critical challenge. The data have to be extracted off the computing nodes and transported to consumer nodes so that it can be processed, analyzed, visualized, archived, and so on. Several recent research efforts have addressed data-related challenges at different levels. One attractive approach is to offload expensive input/output operations to a smaller set of dedicated computing nodes known as a *staging area*. However, even using this approach, the data still have to be moved from the staging area to consumer nodes for processing, which continues to be a bottleneck. In this paper, we investigate an alternate approach, namely moving the data-processing code to the staging area instead of moving the data to the data-processing code. Specifically, we describe the ActiveSpaces framework, which provides (1) programming support for defining the data-processing routines to be downloaded to the staging area and (2) runtime mechanisms for transporting codes associated with these routines to the staging area, executing the routines on the nodes that are part of the staging area, and returning the results. We also present an experimental performance evaluation of ActiveSpaces using applications running on the Cray XT5 at Oak Ridge National Laboratory. Finally, we use a coupled fusion application workflow to explore the trade-offs between transporting data and transporting the code required for data processing during coupling, and we characterize sweet spots for each option. Copyright © 2014 John Wiley & Sons, Ltd.

Received 29 March 2014; Revised 18 August 2014; Accepted 25 August 2014

KEY WORDS: dynamic code deployment; *in situ* data processing; data-intensive application workflows; coupled simulations

1. INTRODUCTION

Emerging scientific and engineering simulations running at scale on leadership-class resources have the potential for enabling simulations with unprecedented levels of accuracy and providing dramatic insights into complex phenomena. At the same time, these simulations present new challenges because of their scales and overall complexities. A critical challenge is due to the rate and volume of data generated by these simulations, and the overheads associated with extracting this data from the computing nodes and transporting it to consumers so that it can be processed (e.g., transformed, analyzed, visualized, archived, etc.) and/or coupled as part of end-to-end application workflows.

For example, the coupled kinetic-MHD [1] plasma simulation consists of parallel codes that simulate neoclassical particle dynamics and fluid instabilities in the edge region of a tokamak fusion reactor, run concurrently on thousands of computing nodes, and are coupled through an

*Correspondence to: Manish Parashar, NSF Cloud and Autonomic Computing Center, Rutgers University, Piscataway, NJ, USA.
†E-mail: parashar@rutgers.edu

integrated, predictive plasma modeling workflow. These codes generate large volumes of data that must be exchanged during coupling, as well as transported to other support services required to ensure simulation progress and the correctness of results, such as data visualization and analysis or execution/completion monitoring. These processes often run independently, on demand, and on distinct and distributed resources resulting in interaction and coupling patterns that are complex, data-intensive, and highly dynamic.

The data transport and processing costs associated with these interactions and couplings are increasingly dominating the overall application execution costs and are becoming a growing concern [2]. As a result, several research efforts have explored techniques for high-throughput data transfers with low application overheads. These include, for example, buffering [3, 4], multiple input/output (I/O) phases [5], asynchronous I/O [6], overlapping communications with computations for latency hiding [7], and active messages [8]. One attractive approach for reducing the I/O overhead on the applications is to offload expensive I/O operations from the computing nodes to a smaller set of dedicated nodes known as the *staging area*. In our research on DataSpaces [9, 10], we have used this approach to asynchronously move data from the computing nodes to the staging area, where it can be queried by consumer nodes for coupling, analysis, visualization, archiving, and so on. However, the costs associated with moving the data off the staging area, coupled with limited resources at these staging nodes, make this only a partial solution.

In data-intensive application workflows, data typically have to be transformed and often reduced before it can be processed by consumer applications or services. For example, application coupling may only require subsets of data that are sorted and processed to match the data representation at the consumer. Similarly, visualization and monitoring applications may only require discrete values, such as the maximum, minimum, or average value of a variable over a region of interest. Processing the data before transporting it can be advantageous in these scenarios. For example, in our research, we have explored embedding predefined data transformation operations in the staging area [11] so that CPU resources at the staging area can be utilized to transform the data before it is shipped to the consumer. This approach requires *a priori* knowledge of the processing, as well as the data structures and data representation. However, the dynamic nature of the overall workflow, especially in terms of the amount of data and the processing requirements of the monitoring, analytics and visualization consumers, warrants a more general approach, where application developers can programmatically define data-processing routines, which are dynamically deployed and executed in the staging area at runtime.

In this paper, we present ActiveSpaces, a data-management framework that explores this alternate paradigm, namely moving the processing code to the data rather than the data to the processing code. ActiveSpaces builds on the concept of a staging area, and specifically on the DataSpaces [9, 10] framework, which overlays the abstraction of an associative, virtual shared space over the staging area. Applications, which may run on remote and heterogeneous systems, can insert and retrieve data objects at runtime using semantically meaningful descriptors (e.g., geometric regions in a discretized application domain).

The key contributions of the ActiveSpaces framework include (1) programming support for defining the data-processing routines, called *data kernels*, to be executed on data objects in the staging area and (2) runtime mechanisms for transporting the user-defined data kernels to the staging area and executing them in parallel on the staging nodes that host the specified data objects. The programming abstractions provided allow the user to define and implement data kernels using all constructs of the native programming language (e.g., C), or using the Lua scripting language. The runtime mechanisms enable code offloading and remote execution at the data source for HPC applications. To the best of out knowledge, this is the first attempt at supporting live code migration that addresses applications from the scientific community. Existing related approaches, from both enterprise and academia, either (1) preload the data transformation codes at the data source ahead of time and execute them at runtime, (2) transfer the source code and compile it on the spot at the data source, or (3) transfer the binary code (e.g., Graphics processing unit (GPU) and the CellBE) but execute it on dedicated hardware platforms.

The ActiveSpaces approach presents several advantages. It can reduce the amount of data that needs to be transferred over the network for data-processing operations. It can also reduce an

DOI: 10.1002/cpe

application's computation time by offloading computations, such as interpolation, redistribution, reformatting, and so on, which can be asynchronously executed in parallel at the staging nodes. The ActiveSpaces approach can also be beneficial in more constrained cases where execution of the data kernels is synchronous, and an application has to wait for data to be processed before it can continue. In addition to reducing the amount of data transported (as mentioned earlier), the data kernels can also better exploit data locality within the staging nodes, because the number of nodes hosting the staging area is much smaller than the number of nodes running the application. ActiveSpaces does not require any data marshalling/unmarshalling—the applications defining the kernels already know the structure, representation, and layout of the data, and the code running in the staging area can access and process the data directly.

We have developed and deployed the ActiveSpaces framework on multiple platforms and are using it to support fusion simulation workflows developed as part of the Center for Plasma Edge Simulation, a US Department of Energy prototype Fusion Simulation Project. In this paper, we describe the design, implementation, and operation of ActiveSpaces, and present an experimental performance evaluation using application coupling scenarios. In addition, we present a coupled fusion simulation workflow using ActiveSpaces, explore trade-offs between transporting data and transporting code for data transformations required by these simulations, and characterize sweet spots for each option.

The rest of the paper is structured as follows. Section 2 presents a motivating application workflow scenario and an overview of DataSpaces. Section 3 presents the architecture of ActiveSpaces and describes its design. Section 4 presents the ActiveSpaces programming interface. Section 5 describes the implementation of ActiveSpaces, and Section 6 presents an experimental evaluation. Section 7 presents related work, and Section 8 concludes the paper.

2. BACKGROUND

2.1. Motivating coupled simulation scenario

A motivating application scenario used is the study of tokamak divertor heat load profiles in the presence of MHD instabilities being undertaken by Center for Plasma Edge Simulation. The purpose of the study is to understand the potential effect of rapid electromagnetic fluctuations on the heat loads borne by critical tokamak fusion reactor components in order to better optimize their design.

The coupled simulation workflow used in this scenario involves two separate parallel application codes (XGC0 and M3D-MPP), along with auxiliary services for post-processing, diagnostics, and visualization, as shown in Figure 1. XGC0, which was developed from the original ion guiding-center code XGC [1], is a kinetic code that follows the neoclassical ion-electron-neutral dynamics and computes plasma equilibrium evolution in the edge region of tokamak plasmas. M3D-MPP, the parallel version of M3D [12], is an extended MHD code that has been used to perform detailed studies of the evolution of so-called edge localized modes. Because the interactions between the

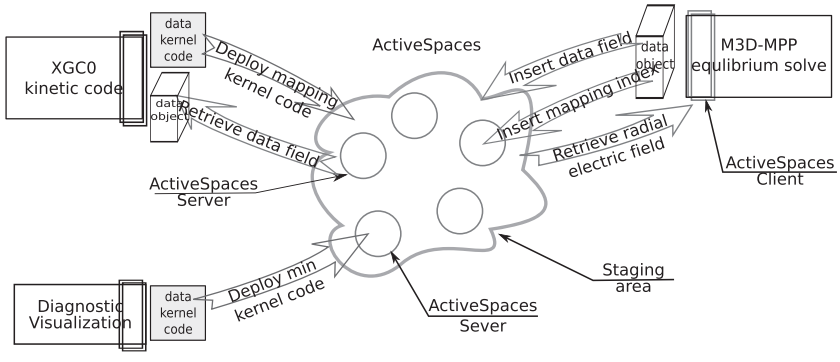

Figure 1. Application coupling scenario. Heterogeneous components exchange data objects and deploy data kernel codes using ActiveSpaces.

DOI: 10.1002/cpe

two codes and the auxiliary services are self-explanatory (e.g., monitoring, visualizing the results of a timestep, or storing the results as a checkpoint), the description below focuses on the coupling aspects between the two codes.

The two applications codes, XGC0 and M3D-MPP, embody different physics (neoclassical transport versus MHD instabilities), employ different computational models (particle-in-cell versus finite element solver), and operate on somewhat different temporal scales (slow pedestal profile evolution versus fast edge localized mode evolution). Nevertheless, it is of great current interest to examine the effect on divertor heat loads as measured in the XGC0 kinetic code when rapid changes in perturbed electromagnetic fields as computed by the M3D-MPP extended MHD code are introduced. As a result, this code coupling scenario requires relatively frequent transfers of several fields of 3D data representing the electrostatic potential and the components of the magnetic vector field from M3D-MPP over to XGC0. For a typical coupled simulation of modest size on the Cray XT5 at Oak Ridge National Laboratory (ORNL), both applications run on several hundred processor cores, and each XGC0 processor core is retrieving roughly 70 MB of data. In addition, the 3D field data that are decomposed across the processor cores within M3D-MPP must be properly recomposed as coherent 3D arrays and then interpolated from the finite element mesh of M3D-MPP to the rectilinear mesh and cylindrical grid coordinates of the XGC0 code. Finally, in order to make the coupling scenario fully self-consistent, it is preferable to send back from XGC0 to M3D-MPP a data set describing the radial electric field, because this quantity cannot be directly computed within the extended MHD code.

The initial implementation of this code-coupled simulation was performed using DataSpaces [9, 10]. Each simulation was augmented with a DataSpaces client, and these two simulations were launched concurrently along with the DataSpaces server. Periodically, the M3D-MPP code writes a new dataset into the shared space, which contains the raw data values of the electrostatic potential and magnetic vector field components at each finite element mesh point, cylindrical grid coordinates for each of those points, and an index array for mapping the decomposed local chunks of the field data from each M3D-MPP process into global arrays for each full 3D field. XGC0 looks for and reads in this data set, uses the index array to organize the data into global 3D arrays, and then uses the grid coordinate data to interpolate the field data onto the XGC0 rectilinear mesh. DataSpaces supports data set versioning, which allows for multiple data sets to be stored within the shared space and alleviates the need for strict synchronization between the two applications. Additional data beyond the raw field values (namely, the index array and the cylindrical grid coordinate data for the finite element mesh) are being transferred from M3D-MPP to XGC0 so that the latter code can properly transform the field data for its use. Thus, it may be possible to improve upon the performance of this implementation by having XGC0 provide data kernels that could perform the requisite mapping and interpolation of the field data within the shared space and then simply read out the transformed field data. This approach is explored in more detail in Section 6 using the ActiveSpaces framework.

2.2. Overview of dataspaces

ActiveSpaces is based on the DataSpaces [9, 10] framework, which provides a virtual, distributed shared space abstraction that can be asynchronously accessed by multiple applications. In the context of the coupled simulation workflows, it overlays a virtual shared space abstraction over the staging nodes that can be associatively accessed by applications and services using semantically meaningful descriptors. For example, the coupled applications can interact by asynchronously storing and retrieving data objects using descriptors derived from the discretization of the application domain. The coupled simulations may run on different systems, at different scales, and may have different data decompositions. The simple *put()* and *get()* operators provided by DataSpaces are agnostic to the location or distribution of the data, and data redistribution is implicitly handled. Similarly, other application components and services, such as those for monitoring, visualization, or verification, can access these data objects by dynamically connecting to DataSpaces and querying for data objects of interest using semantic descriptors.

DataSpaces has two components, a DataSpaces server, which runs on the staging area, and a DataSpaces client, which is integrated with the applications on the computing nodes. The

DOI: 10.1002/cpe

DataSpaces servers create a distributed *data storage* layer across the staging area nodes to store and maintain data objects from user applications. They implement data services such as the *query engine* and the *data lookup* to access and manipulate the data objects. The server uses a Hilbert space-filling curve to map the multi-dimensional applications domain to a linear index, and constructs a distributed hash table using this index to manage objects metadata. The *data lookup* service uses the distributed hash table for quick data lookups. The *query engine* handles insert, retrieve, and monitoring queries and uses the space-filling curve to index the data objects for fast data accesses.

The DataSpaces client is a lightweight component designed to have minimal overhead on the applications. It essentially complements the data services provided by the server and exposes an API for accessing DataSpaces at the application level. For example, it allows an application to submit simple data queries to a DataSpaces server and obtain the results. In the case of more complex queries, such as retrieving data objects that spans multiple server nodes, it transparently splits the original query into multiple simple queries, then forwards them to DataSpaces servers and assembles the final results. The *data communication* in DataSpaces uses the DART [7] data transport layer. DART uses remote direct memory access to provide asynchronous memory to memory data transfers and enables overlapping of computations with communications to hide data transfer latencies.

The DataSpaces in-memory exchange mechanism enables faster data transfers with less performance variability than using a persistent storage system. This is particularly important for applications that exchange data frequently because it avoids the latency and variability [13] of parallel file systems [14] caused by concurrent accesses by multiple users of the system.

3. ACTIVESPACES ARCHITECTURE

ActiveSpaces derives from DataSpaces and inherits and extends its main components. It has a standalone ActiveSpaces server component, which runs on the staging area and provides data services to user applications, and an ActiveSpaces client component, which integrates with user applications and runs on the computing nodes. The two components implement the programming API, which is exposed at the application level, and the runtime system, which executes the user-defined data kernels. Figure 2 presents a schematic overview of the ActiveSpaces architecture. In the rest of this paper, we refer to the ActiveSpaces/DataSpaces server components running on the staging nodes as the *space*.

The ActiveSpaces server is a distributed component based on DataSpaces servers. It constructs a temporary storage space in-memory and provides data services for interacting with this space, such as inserting and retrieving data objects, monitoring data of interest, and data lookup and indexing. In addition, it provides a new data service that applies transformations to the data in the space or to the results of a data request, for example, data filters. This service is implemented by the *runtime execution* (Rexec) layer.

The Rexec layer extends the plug-in architecture of DART and integrates with other services provided by the server. Its main function is to transfer the binary code or Lua scripts code corresponding to a user-defined data kernel from an application to the space, execute the code on data objects selected by the kernel from the space, and return the results back to the application. The Rexec layer handles the execution of self-contained data kernels, which describe transformations that do not rely

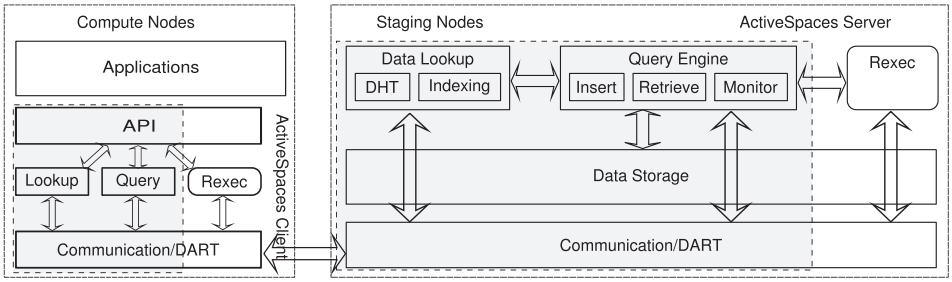

Figure 2. ActiveSpaces framework architecture (the shaded area corresponds to DataSpaces).

DOI: 10.1002/cpe

on external libraries. It also handles more complex data kernels that rely on other helper routines, which require external calls to libraries or other services from the space. For example, a data kernel implemented using C language may allocate or release dynamic memory and have to call the *malloc* and *free* routines from the system *libc* library, it may print debug messages and have to call the *printf* routine, or it may look up and retrieve data objects from the space and have to call other services from the space. In the latter case, Rexec first links the kernel code with the server, so that the external calls to library routines are resolved locally in the context of the server, and then executes the compiled binary code .

The ActiveSpaces client is a lightweight component that integrates with user applications and exposes APIs to access and use the data services provided by the space. The Rexec layer at the client prepares user-defined data kernel codes for offloading and execution within the space. The client Rexec layer complements the functionality of the runtime system of the space and coordinates the execution of kernel codes on data objects that may be distributed over multiple servers. It uses other data services from the client to determine the distribution and the location of different pieces of a data object in the space, and it offloads the kernel code only to those servers that host the data. A data kernel that is distributed in this manner can produce multiple partial results that need to be combined in order to construct the final answer. Using the provided API, the client Rexec layer allows each data kernel to be paired with a user-defined routine that implements the reduction operation. The reduction operation executes on the client after all the partial results from the kernels executing on the space have been gathered. This approach is similar to a map-reduce operation [15] and allows the user to program the space as an accelerator for an application by dynamically offloading customized data kernels to the space at runtime.

4. ACTIVESPACES PROGRAMMING APIS

4.1. Programming data kernels in C language

The ActiveSpaces client defines prototype signatures (presented in Listing 1) for the data kernel routines implemented in C language. Every user-defined data kernel should have the same signature because it is internally used by the client to load the code into the space, and by the runtime system on the space to decode and execute the kernel. A data kernel accepts as input a reference to the arguments structure, which contains fields for both input and output parameters.

The input *ptr_data_in* field parameter is a generic reference to a data object from the space, on which the kernel will operate. This reference is initialized within the space by the runtime system before the kernel is executed. The kernel code can internally cast this generic reference to the appropriate data type known by the application and perform the data transformations. The n_i, n_j and n_k parameters are initialized by the Rexec layer at each server in the space where the kernel executes, exist in every *rexec_args* structure, and represent the size of the local fragment of data at each server that intersects with the object descriptor specified in the kernel argument.

The output *ptr_data_out* field parameter is a generic reference to the result produced by the execution of the kernel code, and the *size_res* parameter defines the size of the result. These two parameters are used by the space to send the results back to the application that initiated the remote call. The *rc* output parameter defines a return code for the execution of the data kernel, which is sent to the application together with the results.

4.2. Programming data kernels in the Lua scripting language

The ActiveSpaces defines a set of C functions for the Lua data kernel to access the I/O data buffers. At runtime, the server registers the C functions inside a global table, and creates an auxiliary Lua library (named 'DSpaceAux' in our implementation). Each function in the Lua library actually binds to the C implementation. As a result, the user-defined Lua script can directly write/read the I/O memory buffers allocated on the server by calling functions in the auxiliary Lua library. Unlike the C data kernel, which provides more generic references to I/O memory buffers and can cast the references to any appropriate data types, the auxiliary library for Lua data kernels only abstracts the

DOI: 10.1002/cpe

```
struct rexec_args {
        /* Input arguments. */
        void *ptr_data_in;
        int ni, nj, nk;

        /* Output arguments. */
        void *ptr_data_out;
        size_t size_res;
        int rc;
};

typedef int (*bin_code_fn_t)(struct rexec_args *);
```

Listing 1. Prototype definitions for C language data kernels.

```
static const luaL_Reg DSpaceAux_methods[] = {
        /* Get the input buffer */
        {''get_input_obj'', DSpaceAux_input},
        /* Get the output buffer */
        {''get_output_obj'', DSpaceAux_output},
        /* get_dim_size(i) returns the size of dimension i of
            the input/output buffer */
        {''get_dim_size'', DSpaceAux_get_dim_size},
        /* get_val(i, j, k) returns the value at (i,j,k) */
        {''get_val'', DSpaceAux_getval},
        /* set_val(i, j, k, val) sets the value at (i,j,k) */
        {''set_val'', DSpaceAux_setval},
        {0, 0} // End of the mapping
};
```

Listing 2. Registration of Lua auxiliary functions.

I/O memory buffers as N-dimensional double precision floating point arrays with row-major layout. Listing 2 presents the mapping from the registered Lua function names to the C functions.

User-defined Lua scripts obtain the reference to the input data buffer of a kernel execution by calling the *get_input_obj* auxiliary function. Similarly, the *get_output_obj* function is invoked to obtain the reference to the output data buffer. Both data buffers are initialized within the space by the *Rexec* layer before the data kernel code is executed. User Lua scripts can call the *get_dim_size* function to obtain the size of the local N-dimensional array at each server, and call the *get_val* and *set_val* functions to get/set the local array element specified by the cartesian coordinate-based index *(i, j, k)*.

The kernel codes (both C and Lua) can apply transformations in place on the data within the space, which does not create additional data, or can create new data as a result of the execution. In the latter case, the kernels have to allocate memory for the results. This allocated memory is automatically released by the space after it returns the results to the application.

4.3. Examples of data kernel codes

ActiveSpace data kernels can use all the constructs available in the supported programming language (we use the C and Lua language as examples in this paper), for example, arithmetic and logical operations, conditional blocks, control loops, and all data types. Listing 3 and Listing 5 present examples of data kernels that finds the minimum value for a 3D matrix.

To load a data kernel in the space, a user has to provide a reference to the code and an object descriptor (see Listing 4 and 6). The object descriptor is a data type that describes and identifies a

DOI: 10.1002/cpe

```
1  int rexec_min(struct rexec_args *rargs)
2  {
3          PLT_TAB;
4
5          double (*A)[rargs->ni][rargs->nj][rargs->nk];
6          int i, j, k;
7          double min, *retval;
8          int err = -ENOMEM;
9
10         A = rargs->ptr_data_in;
11         min = (*A)[0][0][0];
12
13         for (i = 0; i < rargs->ni; i++)
14               for (j = 0; j < rargs->nj; j++)
15                     for (k = 0; k < rargs->nk; k++)
16                           if (min > (*A)[i][j][k])
17                                 min = (*A)[i][j][k];
18
19         rargs->ptr_data_out =
20         retval = MALLOC(sizeof(*retval));
21         rargs->size_res = sizeof(min);
22         if (!retval)
23                   return err;
24         *retval = min;
25         return 0;
26 }
```

Listing 3. Example C language kernel code to find the min value of a 3D matrix in the space.

```
1  void wrapper_space_min_binary(<obj_descriptor>)
2  {
3          /* Load and execute the min kernel code
4              on the space */
5          dart_code_load_binary(&rexec_min, <obj_descriptor>);
6          /* Example of object descriptor:
7                    flux:<10, 20, 5; 50, 40, 60> */
8          min_reduce(partial_results);
9  }
```

Listing 4. Example application code to load and execute a C language data kernel in the space and reduce the partial results.

data object within the space on which the data transformation should be applied. The object descriptor parameter can describe data objects of arbitrary sizes, which can be stored at a single server in the space or across multiple servers if the object is distributed. For distributed objects, the client component uses the data lookup service to determine the servers that store fragments of the data object, and then deploys the data kernel code to each server. For example, it transfers the same data kernel code to the servers, and each server executes the data kernel on its local fragment of the data object and returns a partial result to the client. The kernel code deployment process is handled transparently by the ActiveSpaces client, but the partial results are returned to the application. In turn, the application should pair each data kernel with an appropriate reduction routine that combines all the partial results to produce the final result. The reduction routine executes within the application and may need to be customized for each kernel operation.

DOI: 10.1002/cpe

```
input_obj = DSpaceObj.get_input_obj(0)
output_obj = DSpaceObj.get_output_obj(0)

ni = input_obj:get_dim_size(0)
nj = input_obj:get_dim_size(1)
nk = input_obj:get_dim_size(2)
min = input_obj:getval(0, 0, 0)

for i = 0, ni−1 do
    for j = 0, nj−1 do
        for k = 0, nk−1 do
            val = input_obj:getval(i, j, k)
            if val < min then
                min = val
            end
        end
    end
end

output_obj:setval(0, 0, 0, min)
return 0
```

Listing 5. Example Lua language kernel code to find the min value of a 3D matrix in the space.

```
void wrapper_space_min_lua(<obj_descriptor>)
{
    /* Load and execute the lua script file on the sapce */
    dart_code_load_lua(''compute_min.lua'', <obj_descriptor>);
    min_reduce(partial_results);
}
```

Listing 6. Example application code to load and execute a Lua data kernel in the space and reduce the partial results.

5. IMPLEMENTATION OF RUNTIME EXECUTION

5.1. Overview

The complete flow of a data kernel from source code to its execution in the space is presented in Figure 3. A batch execution system runs the application executable on the computing nodes. The running application uses the client API to load the kernel code (as presented in Listing 4 and 6), and the runtime system transfers the kernel code to the space, executes it on the space, and returns the results to the application. For data kernels implemented in C, the source code needs to be compiled and linked with the application's object files and libraries to create the application executable. For data kernel implemented in Lua scripting language, the source code file is directly read into memory by the client component at runtime, and transferred to the space for dynamic execution.

The ActiveSpaces server provides the runtime execution mechanisms for data kernels and uses other data services provided by the space to implement these mechanisms. First, it registers the remote execution data service with the data communication service (i.e., DART) to enable it to receive and handle the incoming kernel transfer and execution requests. Second, it uses DART to transfer the data kernel codes to the space for execution and send back the results to the application. DART uses asynchronous remote direct memory access mechanism to fetch the raw data objects as well as data kernel codes from the applications.

DOI: 10.1002/cpe

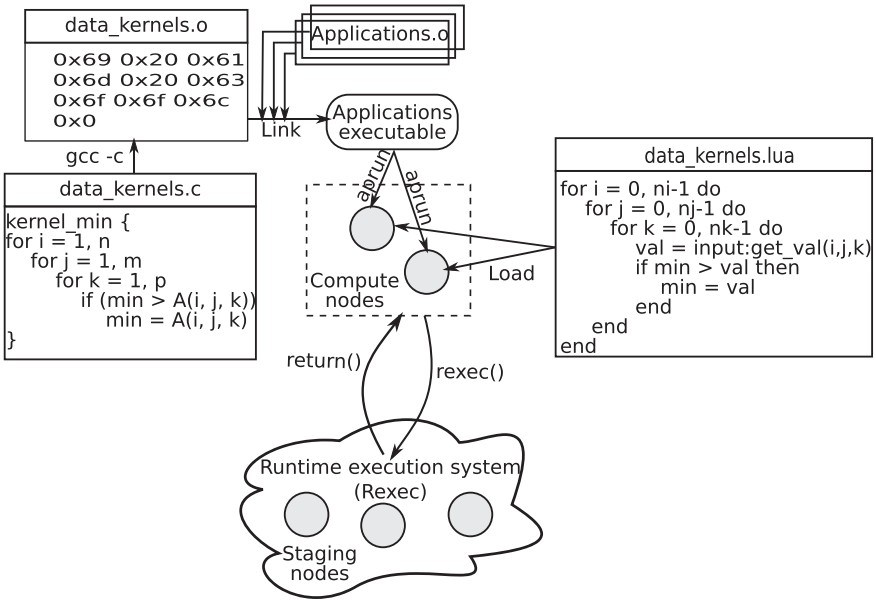

Figure 3. Data kernel code offloading and execution.

Each request for a data kernel transfer contains an object descriptor that identifies the data object on which the kernel should execute. The request may select a fragment or the entire data object, that is, the overlap between the region specified by the request and the data object that is stored locally at the server. To be effective, a data kernel operates on the data that is locally available within the data storage layer at the server and does not require data movement between the servers in the space.

5.2. Code linking for C kernels

Based on the binary instructions, the runtime execution system distinguishes between two types of kernel codes. A simple and self-contained kernel code whose execution is confined to the scope of its binary code, and a more complex kernel code whose execution can jump to external addresses by means of function calls. In the latter case, the linking process (Figure 3) resolves the addresses of the functions called in the scope and address space of the application; thus, the calls are meaningful only to that application. When the binary code associated with such a complex data kernel is transferred to the space, these addresses have to be re-computed to have the same meaning in the new address space for the kernel in order to execute correctly.

The standard solution available on POSIX-compliant systems of using the *dlopen()* and *dlsym()* calls for resolving a symbol address does not work for our environment. Because the environment on the computing nodes does not have a local file system to store custom shared libraries or object files, it requires the applications to be statically linked, and it may not always provide a dynamic linker for the external routines. Moreover, the kernels object files are not directly available to the servers, and transferring them to the space is not a solution because the call to *dlopen* requires a full path to a library and cannot load the object from an address in memory. ActiveSpaces is designed to support data exchanges in memory and avoid the temporary use of a file system that would be required to store the object file for a data kernel should it use the *dlsym* call.

The API to load a kernel to the space requires only a reference to the routine that implements that kernel, and no other metadata information. A function call is encoded in a binary code as a jump instruction with a relative offset. For example, the offset for a call to *malloc* is the difference between the memory address of the current instruction and the memory address of the *malloc* routine. An external routine may be called multiple times by a data kernel, and each instance of the call has a different offset because it is called from different places. Executing a binary code in the space would thus require overwriting relative offsets, which is intrusive and difficult in the absence of additional metadata about the code.

DOI: 10.1002/cpe

The ActiveSpaces runtime execution system provides a more straightforward approach. It defines a local procedure linkage table (PLT) [16], registers the addresses of each external routine used by the kernel codes at unique entries in this table, and replaces the direct calls to the external routines with indirect calls through references stored in this table. The API provides wrapper routines with the same names and signatures, but using capital letters, for all the external routines invoked by a kernel code. A local PLT table is maintained by both the client and the server preserving the order of the entries registered. A user can transparently use the wrapper routines, and the underlying mechanism translates each call to the proper address for the routine requested. An example of a call to a wrapper routine is presented on line 20 in Listing 3.

The approach described earlier presents several advantages. It eliminates the need for code overwriting, and specifically the relative offsets used for function calls. This is because the new routine calling scheme uses constant entries in the local PLTs, whose offsets are the same on both the client and the server. Moreover, multiple calls to the same external routine no longer require different relative offsets, because they use the same PLT entry, and thus the same reference to the external routine. The overhead introduced by this approach consists of two additional *load* instructions, one for the address of the PLT and the other for the address of the actual routine. However, the performance impact is minimal because the PLT table is small and compact, and its contents can easily fit in a data cache line.

As mentioned earlier, both the client and the server maintain a local PLT table, which is stored at different memory addresses on the client and the server, respectively (Figure 4). To match these addresses, ActiveSpaces instantiates the PLT tables on the stack frame of each kernel code (see line 3 in Listing 3) and transfers the PLT to the space as part of the binary kernel code. The client routine that parses the binary code to determine its size also detects the offset where the PLT is stored on the stack frame. The server runtime execution system uses this offset to overwrite the entries of the PLT with local addresses for the corresponding routines.

5.3. Just-in-time compilation and execution for Lua kernels

Lua is an extension programming language and intended to be used as a cross-platform, lightweight, embeddable scripting language. Being an extension language, Lua only works when embedded in a host program. The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by the Lua code. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection.

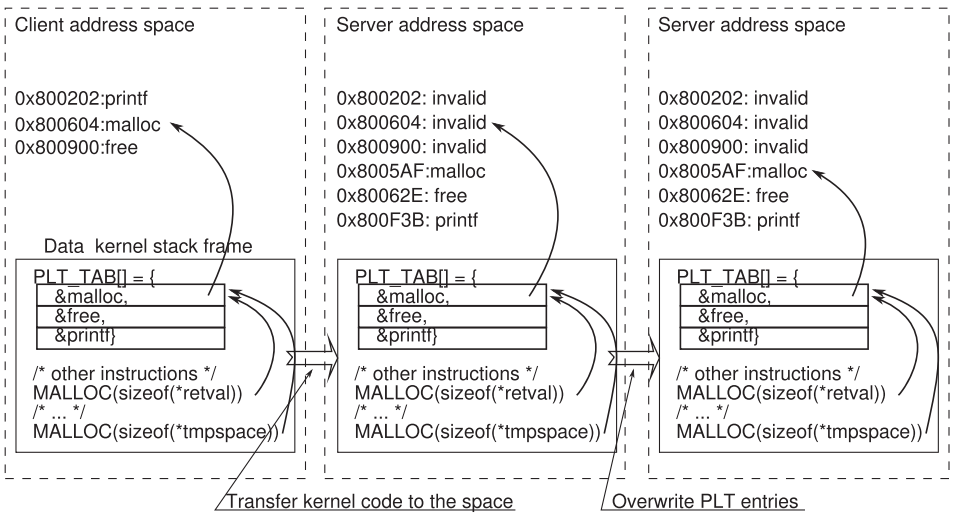

Figure 4. The linking process for the data kernel code with tine runtime execution system.

DOI: 10.1002/cpe

Supporting Lua data kernels in ActiveSpaces presents several advantages. First, Lua is portable to all platforms that have a standard C compiler. Second, Lua is embeddable and can be easily embedded into a host program written in other language such as C/C++, Java. Third, Lua is fast and is one of the fastest language in the realm of interpreted scripting languages.

ActiveSpaces uses LuaJIT [17], one of the fastest Lua implementation, to decrease the performance gap between data kernels written in the Lua dynamic language and data kernels written in a compiled static language (e.g. C). LuaJIT combines a high-speed interpreter written in assembler, with a state-of-the-art just-in-time (JIT) compiler.

The LuaJIT VM, interpreter, and JIT compiler are built into the ActiveSpaces server component, and are used to evaluate the Lua data kernel code transferred from the client applications. Like the C data kernel, the Lua kernel code can also be of two types. A simple kernel code that is implemented only using Lua built-in data types, operations, and functions, and a complex kernel code that may use data structure or invoke functions defined in external libraries. For the latter case, successful in-space evaluation and execution of the data kernel requires that the ActiveSpaces server component preload all the necessary external Lua modules/libraries.

6. EVALUATION

This section presents the experimental evaluation of the ActiveSpaces framework conducted on the *Jaguar* Cray XT5 system at Oak Ridge National Laboratory. The experiments used a coupled application scenario in which one application inserts data into the space running in the staging area, and another application retrieves the data for its local computations. In these experiments, we implemented data reduction kernel such as *min()*, *max()*, *sum()*, and *avg()* and count data objects with values above a given threshold (*ca()*), as well as data transformation kernels such as *data field map()*. We integrated these data kernels with the application codes. Each experiment consisted of two distinct cases. In the first case, the data were moved from the space to the consumer application and the kernel code was applied locally by the consumer application. In the second case, the data kernel was transferred to the space by the consumer application, executed on the data in the space, and the result returned back to the consumer application. The evaluation explored the scalability of ActiveSpaces under different scenarios and the trade-offs between the two cases. Note that the experiments presented in Sections 6.1–6.4 deployed C language kernel codes, and those presented in Section 6.5 deployed Lua script kernel codes.

6.1. Scalability

6.1.1. Weak scaling experiment. This experiment evaluates the behavior and performance of dynamic code offloading using ActiveSpaces for a weak scaling scenario. We used two applications to insert and retrieve data from the space, and implemented data reduction kernels that were executed locally or deployed to the space. The first application ran on 16 processor cores, the space ran on four processor cores, and the second application was scaled from 64 to 1024 processor cores. The experiment also scaled the data size that was exchanged through the space from 64 MB to 1 GB to keep a constant ratio of 1 MB per core processed by the second application.

The results presented in Figure 5 show that offloading the code to the space is faster in all the cases considered. The size of the data kernels (Table I) that is transferred to the space and the size of the results returned to the application are smaller than the size of the raw data, which results in faster transfer times. The time savings for each case is presented in Figure 6, and it increases with the number of cores because the total data size that needs to be processed increases. The increase in code offloading and execution time with the number of cores is interesting considering that the size of the data on which they operate is constant (1 MB). This is caused by the increase in the number of kernel code instances that are transferred to the space. In fact, if the data are distributed across the staging servers in the space, each processor core of the second application may send the code to multiple servers where the data are stored. This observation indicates an opportunity for further optimizations to reduce the number of code transfers.

DOI: 10.1002/cpe

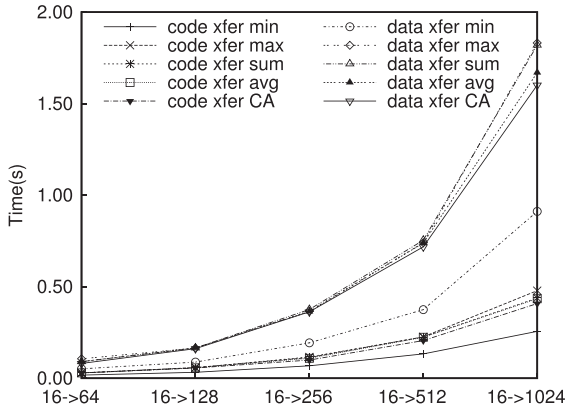

Figure 5. Time for data transfer and kernel code offloading cases for a weak scaling scenario.

Table I. Size of the C data kernels.

	Min	Max	Sum	Avg	CA	Peek	Sort
size(B)	697	683	556	593	574	442	5583

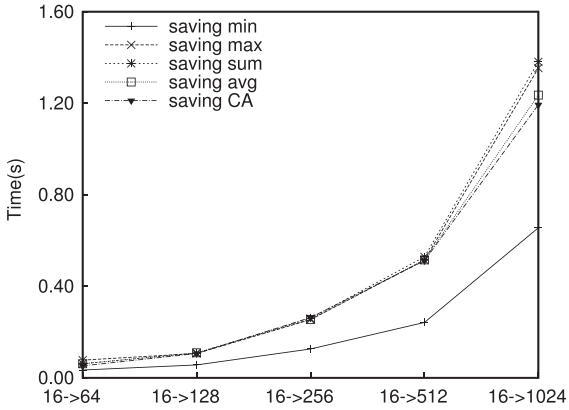

Figure 6. Time savings due to kernel code offloading for a weak scaling scenario.

6.1.2. Strong scaling experiment. This experiment presents an evaluation of the time savings resulting from code offloading using ActiveSpaces under a strong scaling scenario. The experiment ran one application on 16 processor cores that inserted data into the space, and scaled a second application from 64 to 1024 processor cores that read data from the space and process it. The data size as maintained constant at 128 MB and four staging servers were used for the space. We implemented data kernels that were first executed locally after the data were transferred from the space and then were deployed and executed in the space. The time for data transfers and remote kernel executions is presented in Figure 7. The results show that the time for the data movement decreases when the number of cores increases because the total data size is constant and the size of data per core becomes smaller. At the same time, offloading and executing the kernel codes on the space become more expensive, and the time savings decrease as the number of cores increase as seen in Figure 8. As more processor cores offload data codes to the space, the cost of the transfer becomes an expensive operation compared to the amount of data they need to retrieve. The next experiment shows a crossover point at which moving the kernel codes to the space becomes a more expensive operation than transferring the data from the space.

DOI: 10.1002/cpe

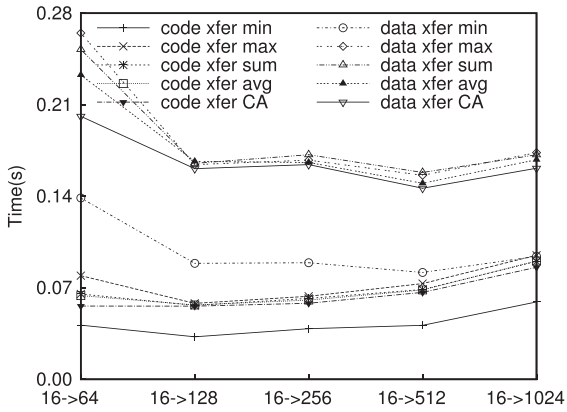

Figure 7. Time for data transfer and kernel code offloading cases for a strong scaling scenario.

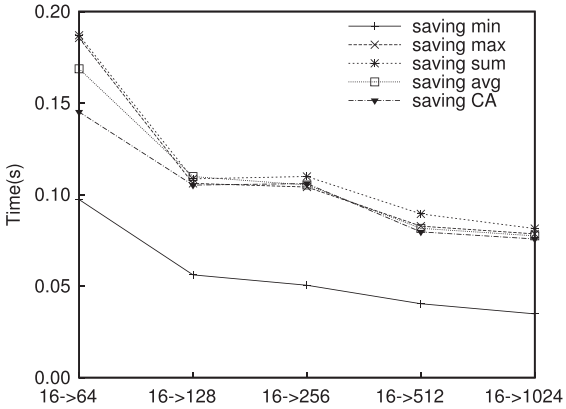

Figure 8. Time savings due to kernel code offloading for a strong scaling scenario.

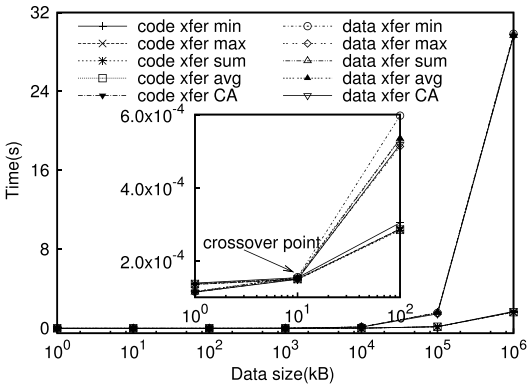

Figure 9. Time for raw data transfer and kernel code offloading cases when data scales from 1 kB to 1 GB.

6.2. Data scaling experiment

This experiment used a constant number of cores to run the space and the applications, and scaled the size of the data exchanged from 1 kB to 1 GB to investigate the relation between the time required to move the data and the time required to deploy and execute custom kernel codes in the space. Specifically, the experiment used one staging server to run the space, one processor core to run the application that inserted data into the space, and one processor core to run the application that processed the data retrieved from the space. The results are presented in Figure 9. For the

DOI: 10.1002/cpe

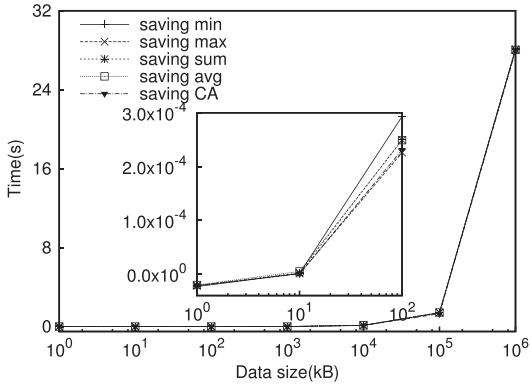

Figure 10. Time savings due to kernel code offloading when data scales from 1 kB to 1 GB.

kernel codes used in this evaluation, moving 10 kB of data to the application and applying the transformations locally take about the same amount of time as transferring and executing the code in the space. For data sizes smaller than 10 kB, it is more efficient to move the data, while for data sizes larger than 10 kB, it is more efficient to deploy the code in the space. In practice, applications exchange large volumes of data, and we expect that moving the code using ActiveSpaces will be more efficient.

The corresponding time savings are presented in Figure 10, as the difference between the time to move the data and the time to deploy the code (the negative value of the first data point represents a penalty). As expected, the time savings increase with the data size because the size of a kernel code (Table I) and the size of the result generated by its execution are much smaller than the size of the data. The two applications used in the experiment had different data representations: row major for the application that inserted the data and column major for the application that retrieved it. The data were rearranged in the space to match the representation at the destination application, and this also contributed to the total data transfer time. However, executing the kernels in the space did not require rearranging the data representation, and so their execution was not affected.

6.3. ActiveSpaces for applications coupling

This experiment used the XGC0 and M3D-MPP applications to analyze the conditions under which a real coupled application scenario would benefit from using ActiveSpaces to offload computations to the space. In this scenario, the M3D-MPP application generates and inserts new field data to the space. The XGC0 application then retrieves the data from the space and locally applies data transformations, such as index mapping and coordinate interpolation, to prepare the data for computations. To evaluate the benefits of code offloading, we implemented a data kernel to map the decomposed field data into global field arrays and integrated it with the XGC0 code—XGC0 then deployed and executes this kernel in the space. Once the kernel executes in the space, XGC0 has to only retrieve the global data arrays from the space, and no longer requires the index array used for the mapping operation.

In this experiment, the XGC0 test application was scaled from 1 to 512 processor cores, and each core retrieved a 72 MB copy of the field data from the space. The mapping operation used an index array to arrange the field data in the proper order in place in the space. This is an example of data transformation that does not result in a reduction in data volume. Figure 11 presents the results of this experiment. The results show that the field transfer to the application is the dominant operation, while the kernel execution represents only a small fraction of the execution time. In this case, the data reduction is minimal and equals the size of the index array, which is 76 kB, minus the size of the data kernel, that is, is 5.5 kB, and is much smaller than the size of the field data, that is, is 72 MB. Nevertheless, offloading the map operation still offers some benefit as seen in Figure 12, as the time difference between the transfer of the raw data and the transfer of pre-processed data. There are two sources contributing to the time savings: first is the (small) data reduction and second is the

DOI: 10.1002/cpe

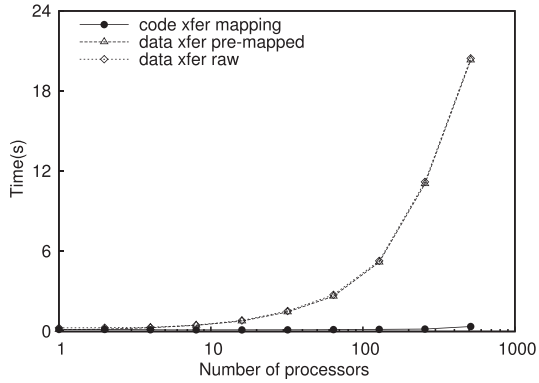

Figure 11. Execution time for the mapping data kernel and data transfer time for raw and pre-processed data for the XGC0–M3D-MPP coupled simulation scenario.

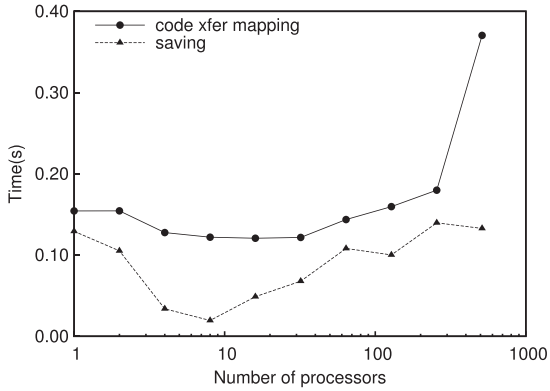

Figure 12. Time savings due to offloading map operations to the space for the XGC0–M3D-MPP coupled simulation scenario.

execution time for the map operation. The average time savings is 0.14 s per processor, which may not seem significant. However, considering the case with 512 processor cores, at the application level, the time savings become 1.2 min for each iteration, and for a 50-iteration run, it becomes 1 h of CPU time savings. Given that CPU time is the metric used for billing applications, this saving can be significant. Furthermore, this saving is additive.

These results clearly show that ActiveSpaces can improve application run times by offloading data reduction and/or computationally intensive operations.

6.4. ActiveSpaces in distributed environments

Processing data using data kernels allows applications to run not only on a single architecture or machine but also on distributed resources located at physically separated geographic locations. This subsection presents an evaluation of ActiveSpaces using a distributed scenario involving two sites, ORNL and Rutgers University (RU) interconnected over the WAN. The first experiment presents the performance evaluation of ActiveSpaces across the two sites and the trade-offs of moving data versus code for data debugging and analysis over the WAN. The second experiment illustrates the use of ActiveSpaces to deploy data kernels into the space for remote data monitoring and visualization.

6.4.1. Application setup. The setup of the application involves distinct components that run at the two sites. The main simulation workflow and the space execute at the ORNL on the Jaguar Cray system, while a remote client executes at RU and monitors the evolution of the simulation and visualizes data values of interest (Figure 13). The simulations and the space execute on the compute nodes of Jaguar, which are not directly accessible by local users or by remote machines. To make

DOI: 10.1002/cpe

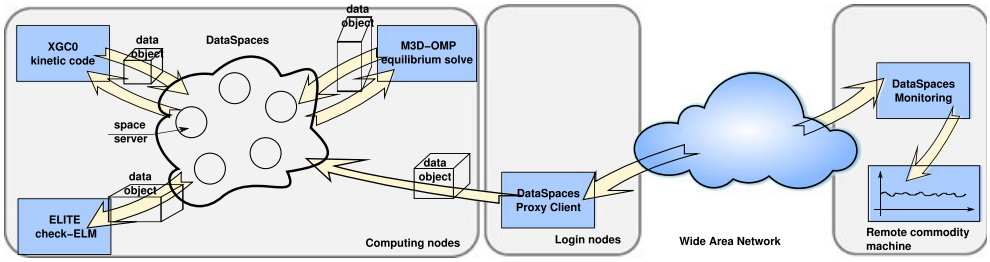

Figure 13. Wide application network application setup.

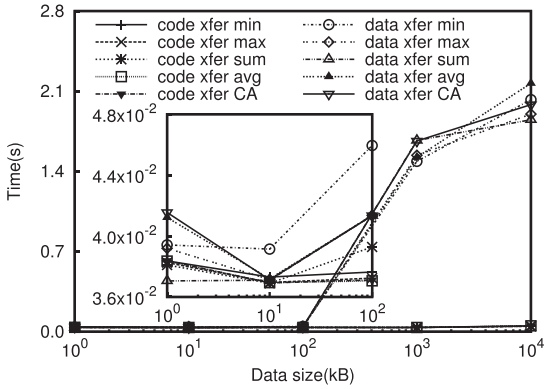

Figure 14. Wide application network data transfers results.

the data in the space accessible to remote users, an ActiveSpaces proxy client runs on the login nodes and connects to the space over the internal network using DART, and to remote clients over the WAN using the TCP transport. The proxy client relays data queries from clients at remote sites to the space and returns the results of the queries from the space to the remote clients. It presents a virtual interface to the space that is accessible by the remote clients. A remote ActiveSpaces client (e.g., monitoring and visualization) runs on commodity machines at RU, sends data queries or deploys data kernels to the space, and interfaces with a front-end visualization module to display and monitor data values of interest.

The focus of these experiments is on data extraction from a running application using ActiveSpaces. Nevertheless, the reverse is also possible, that is, data injection into a running application, for example, for computational steering. Therefore, for the purpose of the experiments, we represented the simulation workflow with one application, which produces and inserts data into the space.

6.4.2. Data scaling over wide area network. This experiment evaluates the performance of the ActiveSpaces framework in a distributed environments over the WAN as a function of the data size being exchanged, and analyzes the trade-offs between transferring data and deploying data-processing kernels.

The data inserted in the space, which was scaled from 1 kB to 10 MB in size, were processed or retrieved by the remote client during each simulation iteration for a total of 50 iterations. The remote client running at RU sent direct data queries and transferred the data to local storage and then applied data transformations locally. The remote client also deployed data-processing kernels into the space and retrieved only the result. The experiment measured the total time required for each request to complete in the two cases, and computed the average over the number of simulation iterations.

Figure 14 presents the average completion time for data transformation operations for the two cases, that is, data transfer with local data manipulation and kernel deployment and remote execution. For data sizes smaller than 10 kB, some of the direct data transfers operations were faster than the corresponding kernel deployment because the size of each direct transfer request is smaller

DOI: 10.1002/cpe

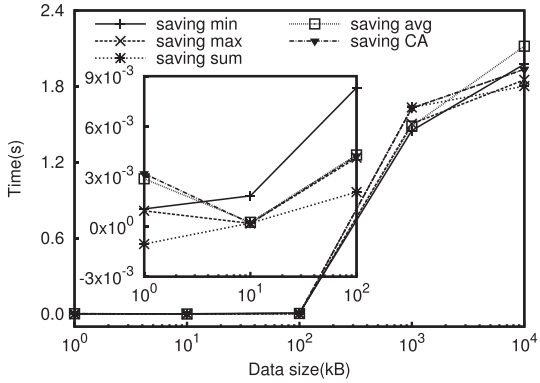

Figure 15. Wide application network data transfer savings.

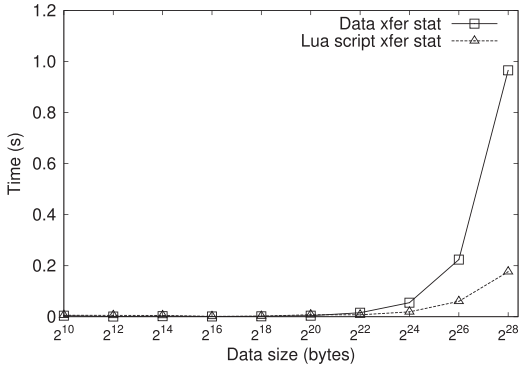

Figure 16. Time for raw data transfer and Lua script offloading cases when data scales from 1 kB to 512 MB.

than the size of the kernel that was transferred from RU to ORNL. The 10 kB data size represents a crossover point at which both direct transfers and kernel deployment operations have similar completion times. For data sizes larger than 10 kB, kernel deployment is clearly faster, and the difference increases with the data size. The amount of time saved to complete these operations per simulation iteration is presented in Figure 15.

6.5. Dynamic deployment of Lua scripts

This subsection presents a data scaling experiment for the dynamic deployment of Lua kernel code. Specifically, the experiment used one staging server to run the space, one processor core to run the data inserting application, and one processor core to run the data retrieving and processing application. The size of data was scaled from 1 KB to 512 MB to investigate the relation between the time required to move the data and the time required to deploy and execute kernel codes in the space. A *stat()* data kernel was used in the evaluation, which computes simple statistics including the maximum, minimum, sum, and average value of an input array data. The experiment compares the performance of remote execution that deploys Lua kernel code into the space, and the performance of local execution that fetches data from the space and executes the kernel code locally. Figure 16 presents the execution time, and Figure 17 presents the corresponding time savings. The results show the crossover point at 256 KB. For data sizes smaller than 256 KB, it is more efficient to move the data, while for data sizes larger than 256 KB, it is more efficient to offload Lua script data kernel on the space. As compared with the data scaling results in 6.2, the data size of the crossover point for Lua data kernel is much larger than the C data kernel. Because of the runtime overhead of Lua interpreter and virtual machines, there exists a performance gap between Lua script execution and compiled C binary code execution. As a result, it needs larger data sizes to reach the crossover point where the overhead of Lua execution is overshadowed by the overhead of network data transfers.

DOI: 10.1002/cpe

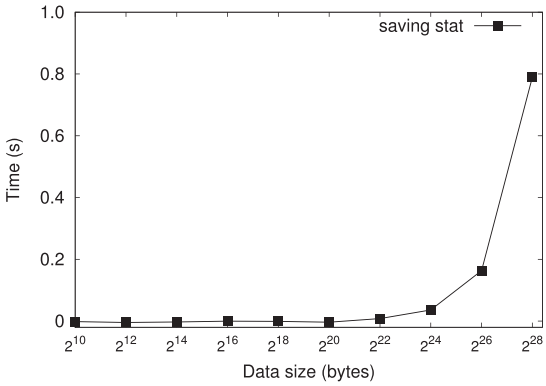

Figure 17. Time savings due to Lua script offloading when data scales from 1 kB to 512 MB.

7. RELATED WORK

This section presents a summary of related research efforts that support out-of-core data-processing operations to improve overall application runtime, to reduce the size of data transferred over the network and/or to better utilize available resources. It also discusses related research on dynamic code deployment and execution at runtime.

The ActiveDisks [18, 19] project implements data-processing operations in the data storage path. It uses lower-level resources such as disk controllers to deploy and apply simple user-defined operations on the data before it is stored or retrieved from the disk. This approach demonstrates good improvements in performance for data-mining applications, where pattern searches and data filtering can reduce the total traffic on the common interconnection bus. Data processing is limited to a low number of operations per byte and is constrained by the processing power and available memory at the controller.

The ActiveStorage [20] project leverages the concepts of ActiveDisks and moves the data-processing operations to the file system level. It overcomes the processing limitations of a disk controller by executing the data operations in a separate process in user space on behalf of the user. This approach is mainly focused on file operations where data transformations can be applied to reduce the network traffic from storage to the requesting clients. It requires external and independent data transformation applications (independent from the consumer or the producer of the data) that implement the processing operations.

Accelerator platforms can offload expensive data-processing operations to dedicated units to improve the overall application runtime. For example, the GPU [21] and the Cell [22] architectures provide hardware support and programming models to implement and deploy user-defined operations on additional processing cores. This approach has demonstrated good performance benefits for a wide range of applications from gaming to graphics rendering, and to scientific computing. However, the accelerator programming approach is different from the ActiveSpaces model because in this case, the user has to deploy both the code and the data on the accelerator cores, while in ActiveSpaces, only the code is deployed.

A related software effort focused on applying data transformations at the data storage is the DataCutter [23] project. It applies data-processing operations such as aggregation and transformations at the storage server before the data are retrieved by data analysis applications that run on distributed clients. The goal is to reduce the volume of data that is transported over the network. DataCutter also supports the application of data filters at intermediate nodes while the data is in transit from storage to the clients. These filters must have predictable resource requirements and have to be provisioned at the intermediate nodes in advance, that is, cannot be loaded dynamically at runtime.

A similar approach that applies data transformations while data is in transit is presented in [24]. The focus of this project is to meet end-to-end data transfer and processing requirements in congested commodity networks. The authors try to compensate for the latency of the network links by

DOI: 10.1002/cpe

processing data at intermediate nodes in a congested data path between data source and destination. In this approach, once again, the data-processing operations have to be predefined and pre-installed at the intermediate nodes within the data path.

The Abacus [25] system supports dynamic function placement for data-processing operations. It implements mobile objects whose state can be serialized and restored, as well as operations that can be executed independently on these objects. The runtime system can migrate the execution of the operation to optimize the total execution time, but does not migrate the code itself, that is, the binary code is statically compiled by the runtime system and called at each node on demand.

Dyninst [26] is a framework that provides programming support to insert user-defined codes into a running program. The primary function of the inserted code is for instrumentation, debugging, or profiling. The framework focuses on local use, and does not explore binary code transfers over the network.

The ActiveStreams [27] project presents a different approach of applying data transformations in the data path. The proposed solution is to move the source code implementing the data transformations to intermediate nodes and compile it on demand. This approach has the advantage of being portable. However, it can only support a limited set of data transformation operations because the compiler and the runtime system are separate from the applications and lack valuable information such as custom application defined data types. Moreover, it adds the overhead of compiling the source code and requires compiler availability within the runtime system.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented the design and implementation of the ActiveSpaces framework, which supports the dynamic deployment and execution of data-processing routines on nodes that are part of the data staging area. This work is motivated by the data management challenges of large-scale coupled simulation workflows running on leadership-class resources, and explores an alternate approach to moving the data for coupling, processing, or analysis, that is, moving the processing codes to the data. The ActiveSpaces approach can be applied to different classes of applications including applications that filter data, for example, application monitoring, visualization, biomedical imaging, or applications that pre-process data, for example, interpolation, compression, scaling, and inversion. We demonstrated the framework usage using real production codes that are part of a coupled plasma fusion application.

ActiveSpaces provides programming support for defining the data-processing routines that are downloaded to the staging area, using the native C programming language or the Lua scripting language, and runtime mechanisms for transporting binary codes associated with these routines to the staging area, executing the routines on the nodes of the staging area, and returning the results to the applications.

We also presented an experimental evaluation of ActiveSpaces on the Cray XT5 system, and investigated, in the context of real applications, the trade-offs between binary code deployment and remote data processing at the staging area, and data movement and local data processing at the data consumer.

The experiments presented used various data reduction kernels for analytics and visualization, as well as data transformation kernels for code coupling. The experimental results demonstrated that ActiveSpaces can achieve significant reduction of the overall data-processing time for data reduction operations. In the case of data transformations that do not reduce the size of the data, ActiveSpaces still provides some performance benefits as the processing is offloaded to the staging area nodes. The results also demonstrated that the decision of whether to transport the code or the data for a particular system configuration depends on multiple factors including the size of the data, the type of processing, and the possibility of asynchronous execution.

Our future work includes further optimizations for binary code deployment, such as offloading the binary code once and executing it on demand, which would reduce the total code transfer time, especially at large scales. We will also explore alternate ways for offloading code to the staging area and for conditional processing. For example, the code deployed may be guarded by a filter

DOI: 10.1002/cpe

condition and it could be applied only to those data objects that satisfy the filter condition. We will investigate the use of ActiveSpaces for remote code monitoring and debugging over commodity networks, where efficient data transfers are critical.

ACKNOWLEDGEMENTS

The research presented in this work is supported in part by the US National Science Foundation (NSF) via grant numbers ACI 1339036, ACI 1310283, DMS 1228203, and IIP 0758566; by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the US Department of Energy through the Scientific Discovery through Advanced Computing (SciDAC) Institute of Scalable Data Management, Analysis and Visualization (SDAV) under award number DE-SC0007455; by the Advanced Scientific Computing Research and Fusion Energy Sciences Partnership for Edge Physics Simulations (EPSI) under award number DE-FG02-06ER54857; by the ExaCT Combustion Co-Design Center via subcontract number 4000110839 from UT Battelle; by the RSVP grant via subcontract number 4000126989 from UT Battelle; and by an IBM Faculty Award. The research was conducted as part of the NSF Cloud and Autonomic Computing (CAC) Center at Rutgers University and the Rutgers Discovery Informatics Institute (RDI2).

REFERENCES

1. Chang CS, Ku S, Weitzner H. Numerical study of neoclassical plasma pedestal in a tokamak geometry. *Physics of Plasmas* 2004; **11**(5):2649–2667.
2. Parashar M. Addressing the petascale data challenge using in-situ analytics. In *Proceedings of the 2nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, PDAC '11. ACM: New York, NY, USA, 2011; 35–36. DOI: 10.1145/2110205.2110212.
3. Liu Q, Logan J, Tian Y, Abbasi H, Podhorszki N, Choi JY, Klasky S, Tchoua R, Lofstead J, Oldfield R, Parashar M, Samatova N, Schwan K, Shoshani A, Wolf M, Wu K, Yu W. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience* 2014; **26**(7):1453–1473. DOI: 10.1002/cpe.3125.
4. ADIOS: ADaptable I/O System. (Available from: https://www.olcf.ornl.gov/center-projects/adios/.) [Accessed on January 2014].
5. Coloma K, Ching A, Choudhary A, keng Liao W, Ross R, Thakur R, Ward L. A new flexible MPI collective I/O implementation. *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006.
6. Abbasi H, Wolf M, Eisenhauer G, Zheng F, Schwan K, Klasky S. DataStager: scalable data staging services for petascale applications. *Proceedings of 18th International Symposium on High Performance Distributed Computing (HPDC'09)*, Munich, Germany, 2009.
7. Docan C, Parashar M, Klasky S. Enabling high speed asynchronous data extraction and transfer using DART. *Concurrency and Computation: Practice and Experience* 2010; **22**(9):1181 –1204.
8. Lumetta SS, Culler DE. Managing concurrent access for shared memory active messages. *Proceedings of 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)*, Orlando, Florida USA, 1998.
9. Docan C, Parashar M, Klasky S. DataSpaces: an interaction and coordination framework for coupled simulation workflows. *Proceedings of 19th International Symposium on High Performance Distributed Computing (HPDC'10)*, Chicago, Illinois USA, 2010.
10. Docan C, Parashar M, Klasky S. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing* 2012; **15**(2):163–181.
11. Zheng F, Abbasi H, Docan C, Lofstead J, Liu Q, Klasky S, Parashar M, Podhorszki N, Schwan K, Wolf M. PreDatA: preparatory data analytics on peta-scale machines. *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, Georgia USA, 2010.
12. Park W, Belova EV, Fu GY, Tang XZ, Strauss HR, Sugiyama LE. Plasma simulation studies using multilevel physics models. *Physics of Plasmas* 1999; **6**(5):1796 –1803.
13. Lofstead J, Zheng F, Liu Q, Klasky S, Oldfield R, Kordenbrock T, Schwan K, Wolf M. Managing variability in the io performance of petascale storage systems. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'10)*, New Orleans, LA USA, 2010. To appear.
14. Braam PJ. Lustre: a scalable high performance file system, 2002. (Available from: http://www.lustre.org/docs/whitepaper.pdf.) [Accessed on December 2011].
15. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, San Francisco, CA USA, 2004.
16. Drepper U. *How To Write Shared Libraries*, 2002. (Available from: http://www.akkadia.org/drepper/dsohowto.pdf.) [Accessed on December 2011].
17. luajit, 2013. (Available from: http://luajit.org/luajit.html.) [Accessed on January 2014].
18. Riedel E, Faloutsos C, Gibson GA, Nagle D. Active disks for large-scale data processing. *IEEE Computer* 2001; **34**(6):68 –74.

19. Riedel E. Active disks - remote execution for network-attached storage. *Ph.D. Thesis*, Carnegie Mellon University, Pittsburgh, PA USA, 1999.
20. Piernas J, Nieplocha J, Felix EJ. Evaluation of active storage strategies for the Lustre parallel file system. *Proceedings of Super Computing Conference (SC'07)*, Reno, NV USA, 2007.
21. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC. GPU Computing. *Proceedings of the IEEE* 2008; **96**(5):879 –899.
22. Bellens P, Perez JM, Badia RM, Labarta J. CellSs: a programming model for the cell BE architecture. *Proceedings of Super Computing Conference (SC'06)*, Tampa, FL USA, 2006.
23. Beynon M, Ferreira R, Kurc T, Sussman A, Saltz J. DataCutter: middleware for filtering very large scientific datasets on archival storage systems. *Proceedings of 17th IEEE Symposium on Mass Storage Systems*, College Park, MA USA, 2000.
24. Bhat V, Parashar M, Klasky S. Experiments with in-transit processing for data intensive grid workflows. *Proceedings of the 8th IEEE International Conference on Grid Computing (Grid'07)*, Austin, Texas USA, 2007.
25. Amiri K, Petrou D, Ganger GR, Gibson GA. Dynamic function placement for data-intensive cluster computing. *Proceedings of USENIX Annual Technical Conference (USENIX'00)*, San Diego, CA USA, 2000.
26. Buck B, Hollingsworth JK. An API for runtime code patching. *International Journal of High Performance Computing Applications* 2000; **14**(4):317 –329.
27. Bustamante FE. The active streams approach to adaptive distributed applications and services. *Ph.D. Thesis*, Georgia Institute of Technology, Atlanta, GA USA, 2001.

DOI: 10.1002/cpe