Moving the Code to the Data – Dynamic Code Deployment using ActiveSpaces

Ciprian Docan and Manish Parashar
Rutgers University
Piscataway, New Jersey, USA
E-mail: [docan.parashar]@cac.rutgers.edu

Julian Cummings
California Institute of Technology
Pasadena, California, USA
E-mail: cummings@cac.caltech.edu

Scott Klasky
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
E-mail: sklasky@ornl.gov

Abstract—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 has to be extracted off the computing nodes and transported to consumer nodes so that it can be processed, analyzed, visualized, archived, etc. Several recent research efforts have addressed data-related challenges at different levels. One attractive approach is to offload expensive I/O operations to a smaller set of dedicated computing nodes known as a staging area. However, even using this approach, the data still has 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 rather than moving the data. Specifically, we present the ActiveSpaces framework, which provides (1) programming support for defining the data-processing routines to be downloaded to the staging area, and (2) run-time 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. We also present an experimental performance evaluation of ActiveSpaces using applications running on the Cray XTS 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 the sweet spots for each option.

Keywords—dynamic code deployment; in situ data processing; data-intensive application workflows; coupled simulations.

I. 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 due to 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 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 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 processes 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. As a result, several research efforts have explored techniques for high-throughput data transfers with low application overheads. These include, for example, buffering [4], multiple 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 overheads on the applications is to offload expensive I/O operations from the computing nodes to a smaller set of dedicated computing nodes known as the staging area. In our research on DataSpaces [9], 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, etc. 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 has 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 pre-defined data transformation operations in the staging area [10] 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] framework, which overlays the abstraction of an associative, virtual shared space on 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 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) run-time mechanisms for transporting the binary codes associated with these 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 the data kernels using all constructs of the native programming language (e.g., C). The run-time mechanism enables code offloading and remote execution at the data source for HPC applications. To the best of our knowledge, this is the first attempt for live code migration that addresses applications from the scientific community. Existing related approaches, from both enterprise and academia, either (1) pre-load the data transformations 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., 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 reduction operations. It can also reduce an application’s computation time by offloading computations, such as interpolation, redistribution, reformating, etc., which can be asynchronously executed in parallel on the staging area nodes. The ActiveSpaces approach can also be beneficial even 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 above), the data kernels can also better exploit data locality within the staging area 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 as part of the Center for Plasma Edge Simulation (CPES), a US Department of Energy (DoE) prototype Fusion Simulation Project (FSP). In this paper, we describe the design, implementation and operation of ActiveSpaces, and present an experimental performance evaluation using application coupling scenarios. 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 II presents a motivating application workflow scenario and an overview of DataSpaces. Section III presents the architecture of ActiveSpaces and describes its design. Section IV describes the implementation of ActiveSpaces, and Section V presents an experimental evaluation. Section VI presents related work, and Section VII concludes the paper.

II. BACKGROUND

A. Motivating Coupled Simulation Scenario

A motivating code-coupling scenario is the study of tokamak divertor heat load profiles in the presence of MHD instabilities being undertaken by CPES. 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 code-coupling 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 [3], is an extended MHD code that has been used to perform detailed studies of the evolution of so-called Edge Localized Modes, or ELMs. Since the interactions between the 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 vs. MHD instabilities), employ different computational models (particle-in-cell vs. finite element solver), and operate on somewhat different temporal scales (slow pedestal profile evolution vs. fast ELM 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. So this code coupling 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 ORNL, this means transferring roughly 70 MB of data perhaps every 10 minutes between two applications that are each running on several hundred computing nodes. In addition, the 3D field data that is decomposed across the computing nodes 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, since this quantity cannot be directly computed within the extended MHD code.

The initial implementation of this code-coupled simulation was done using DataSpaces [9]. 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 would write a new data set 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 would look for and read in this data set, use the index array to organize the data into global 3D arrays, and then use 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 exact 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) is 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 V using the ActiveSpaces framework.

B. Overview of DataSpaces

ActiveSpaces is based on the DataSpaces [9] 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 on the staging nodes that can be associatively accessed by applications and services using semantically meaningful descriptors. For example, the coupled simulation 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.

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 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 (SFC) to map the multi-dimensional applications domain to a linear index, and constructs a distributed hash table (DHT) using this index to manage objects meta-data. The data lookup service uses the DHT for quick data lookups. The query engine handles insert, retrieve and monitoring queries and uses the SFC 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] transfer layer. DART uses remote direct memory access (RDMA) 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 [11] of parallel file-systems [12] caused by concurrent accesses by multiple users of the system.
III. ACTIVESPACES ARCHITECTURE

A. ActiveSpaces Components Architecture

ActiveSpaces derives from DataSpaces and inherits and extends its main components. It has a stand-alone 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 run-time system, which executes the user-defined data kernels. Figure 2 presents a graphical representation of the architecture of ActiveSpaces. 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 services. 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, 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, e.g., data filters. This service is implemented by the run-time execution system (i.e., 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 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 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 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 binary code.

The space and the storage layer in particular, provides the context for the Rexec layer to execute the data kernels on. However, Rexec can be decoupled from the space and integrated with applications to allow live code migration directly between applications.

The ActiveSpaces client is a lightweight component that integrates with user applications and exposes the programming API to access and use the data services provided by the space. The client has different layers providing stub implementations that prepare and transfer application requests to the corresponding data services in the space.

The Rexec layer on the client prepares the user-defined data kernel codes for offloading and execution in the space. For example, it automatically determines the size of a kernel binary code and the necessary information for the calls to external routines that need to be linked with the space.

The client Rexec layer complements the functionality of the execution run-time 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 on 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 [13] and allows the user to program the space as an accelerator for an application by dynamically offloading pre-compiled data kernels to the space at runtime.

B. API Example for Code Loading

The ActiveSpaces client defines prototype signatures (presented in Listing 1) for the data kernel routines and their parameters. Every user-defined data kernel should have the same signature because it is internally used by the client to load the code in the space and by the run-time 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.

```c
typedef int
struct
  void
    
  void
    
  

  
```  

Listing 1. Prototype definitions for data kernels.

```c
void wrapper_space_min(<obj_descriptor>)
{
  /* Load and execute the min kernel code
   * on the space */
  dart_code_load(&rexec_min, <obj_descriptor>);
  /* Example of object descriptor:
   * flux:<10, 20, 5; 50, 40, 60> */
  min_reduce(partial_results);
}
```

Listing 2. Example application code to load and execute a data kernel on the space and reduce the partial results.

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 on 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 `ni`, `nj` and `nk` parameters are initialized by the Rexec layer at each server on 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. The kernel codes can apply transformations in place on the data from 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 the memory space for the result and assign it to the output parameters. The space automatically releases this memory after it returns the results to the application.

The details for data kernels code deployment and execution on the space are transparent to an application. The corresponding API (see Listing 2) call requires only a reference to the routine that implements the kernel and an object descriptor to select the data from the space.

IV. IMPLEMENTATION

A. Implementation of the Client

```c
int rexec_min(struct rexec_args *rargs)
{
  PLT_TAB;
  double (*A)[rargs->ni][rargs->nj][rargs->nk];
  int i, j, k;
  double min, *retval;
  int err = -ENOMEM;
  A = rargs->ptr_data_in;
  min = (*A)[0][0][0];
  for (i = 0; i < rargs->ni; i++)
    for (j = 0; j < rargs->nj; j++)
      for (k = 0; k < rargs->nk; k++)
        if (min > (*A)[i][j][k])
          min = (*A)[i][j][k];
  retval = malloc(sizeof(*retval));
  *retval = min;
  return 0;
}
```

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

The implementation of a data kernel can use all the constructs available in the native programming language (we use the C language in this paper), e.g., arithmetic and logical operations, conditional blocks, control loops, and all data types. Moreover, it can use custom data types defined within the applications that implement the kernel. These data types are not available in the space, but their layout is built into the kernel code. The space does not need to know the structure, or how to pack and unpack the data that it passes to a kernel. The applications that collaborate at runtime work on a common domain, and because the structure of the data that is stored in the space is known between applications, a kernel knows the data layout and can operate directly on the data. Listing 3 presents an example of a data kernel that finds the minimum value for a 3D matrix of type double. In this example, the kernel casts the generic data reference from the input parameters to the internal 3D matrix representation, which it then uses directly to find the minimum value.

To load a data kernel in the space, a user has to provide a reference to the code and an object descriptor (see Listing 2). The object descriptor is a data type that describes and identifies a data object within the space on which the data transformation
should be applied. To transfer data to the space, the client transport layer requires two parameters: a reference to the data and the size of the data. The client Rexec layer parses the binary code provided as a reference, and computes its length to determine the necessary linking information. It decodes each instruction and counts the number of bytes in the code. The current implementation of the parser decodes the complete x86-64 instruction set architecture, which contains variable-length instructions as defined in [14], [15]. The binary instruction parser has a simplified implementation \(^1\) which decodes and counts the length of each instruction, in order to detect the end of a binary kernel code when a return instruction is decoded.

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 size is large and the object is distributed. For distributed objects, the client component uses the data lookup service to determine the servers which store fragments of the data object, and then deploys the data kernel code to each server. For example, it transfers the same binary code, 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 a single and final answer. The reduction routine executes within the application and should be customized for each kernel operation.

This paper presents and implements customized reduction functions for each data kernel used, and wrapper routines that call the code deployment API (e.g., `dart_code_load()`) and the corresponding reduction operation (one such example is presented in Listing 2) automatically for an application. This approach can be easily extended to combine all the wrapper routines into a generic API that takes a reference to a user defined reduction function in addition to the reference to the data kernel. The reduction function can be defined and customized by the user for each data kernel implemented. Moreover, the reduction operation can also be offloaded to the space. The ActiveSpaces client can select a server from the space to collect the partial results for a data transformation, and deploy to the this server the reduction routine in addition to the data kernel.

### B. Implementation of the Server

The ActiveSpaces server provides the run-time execution mechanisms for the data kernels and uses other data services provided by the space to implement its functionality. 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 transfer and execution requests. Second, it uses DART to transfer the binary data kernel codes from an application to the space and send back the results to the application. DART uses asynchronous remote direct memory access (RDMA) mechanism to fetch the raw data objects as well as binary codes from the applications.

The Rexec layer manages the memory resources for binary code transfers. For example, it allocates and aligns the memory buffers for the transfers and sets the proper attributes for execution. The server stores the kernel binary codes on the heap, which is marked as non-executable as a defense measure against code injection attacks [16] on most systems. To be able to execute the binary code from the heap, the memory pages associated with the code must have the executable attribute set. Rexec can, in a controlled and algorithmic manner, set and clear the execution permissions of the memory buffers it manages. These permissions can be modified at the granularity of a memory page level; therefore, the buffers have to be aligned at a memory page boundary and their sizes have to be multiples of the memory page size. However, this does not impose any constraints on the size of a kernel code, which may fit in one or multiple pages.

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, i.e., the common region for 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 at the data storage layer at the server and does not require data movement between the servers in the space.

The run-time system transfers the kernel binary code to a local buffer, then uses the data query service provided by the space to quickly search the data storage layer and retrieve a reference to the local data object selected by the object descriptor argument. It then constructs the kernel arguments structure, sets the input parameters, and executes the kernel. It makes an indirect call to a `bin_code_fn_t` routine (see Listing 1) to execute the kernel using the address of the local buffer that stores the code. When the kernel execution completes, the run-time system uses the output parameters from the arguments structure to send the results back to the application and release the resources for both the results and the data kernel.

\(^1\)The initial implementation on which the parser is based can be found at [http://penberg.blogspot.com/2010/04/short-introduction-to-x86-instruction.html](http://penberg.blogspot.com/2010/04/short-introduction-to-x86-instruction.html). Date of access 08/20/2010.
The complete flow of a data kernel from the source code to the binary execution on the space is presented in Figure 3. A binary object file can contain multiple data kernels; however, the run-time execution system transfers upon request only the referenced kernel, and not all the kernels from the object file. The kernel code execution is not restricted to a single architecture, for example, for heterogeneous architectures the object file can be cross-compiled for the architecture that runs the space.

C. Code Linking

Based on the binary instructions, the run-time 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 (see 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 such a data kernel binary code 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 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. Since 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 the data exchanges in memory and avoid the temporary use of a file system, which 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 on the space would thus require overwriting of relative offsets, which is intrusive and difficult in the absence of additional metadata about the code.

The ActiveSpaces run-time execution system provides a more straightforward approach. It defines a local procedure linkage table (PLT) [17], 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 the references stored in the 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 entries registered in the same order. A user can transparently use the wrapper routines, and the underlying mechanism would translate each call to the proper address of the routine requested. An example of a call to a wrapper routine is presented on line 20 in Listing 3.

The approach described above 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 content can easily fit in a data cache line.

As mentioned above, 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 (see 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 run-time execution system uses this offset to overwrite the entries of the PLT with local addresses for the corresponding routines.

V. 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 applications 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 kernels, e.g., `min()`, `max()`, `sum()`, `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 test application codes. Each experiment consisted of two distinct cases. In the first case, the data was moved from the space to the consumer application and

| TABLE I
| SIZE OF THE DATA KERNELS. |
|---|---|---|---|---|---|---|---|
| size(B) | min | max | sum | avg | CA | peek | sort |
| 697 | 683 | 556 | 593 | 574 | 442 | 5583 |

764
the kernel code was applied locally at the 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.

A. 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 what were executed locally or deployed to the space. The first application ran on 16 computing nodes, the space ran on 4 staging nodes, and the second application was scaled from 64 to 1024 computing nodes. 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 computing node to be 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 (see Table I) that are 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 computing nodes because the total data size that needs to be processed increases. The increase in code offloading and execution time with the number of computing nodes 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 is transferred to the space. In fact, if the data is distributed across the staging nodes in the space, each computing node of the second application may send the code...
B. 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 computing nodes to insert data in the space, and scaled a second application from 64 to 1024 computing nodes to process the data from the space. It maintained a constant data size of 128 MB and 4 staging nodes for the space. We implemented data kernels that were first executed locally after the raw data was 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 move operation decreases when the number of computing nodes increases because the total data size is constant and the size of data per computing node becomes smaller. At the same time, offloading and executing the kernel codes on the space become more expensive, and the decreasing trend for time savings can be observed in Figure 8. As more computing nodes 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 data codes to the space becomes a more expensive operation than transferring the raw data from the space.

C. Data Scaling Experiment

This experiment used a constant number of computing nodes 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 on the space. Specifically, the experiment used 1 staging node to multiple nodes where the data is stored. This observation indicates an opportunity for further optimizations to reduce the number of code transfers.
to run the space, 1 computing node to run the application that inserted data into the space, and 1 computing node to run the application that processed the data retrieved from the space. The results are presented in Figure 9. For the kernel codes used in this evaluation, moving 10 kB of data to the application and applying the transformations locally takes about the same amount of time as transferring and executing the code on 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 on the space. In practice, applications exchange large volumes of data, and we expect moving the code using ActiveSpaces to 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 (see Table I) and the size of the result of 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 was 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.

D. 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 in the space. The XGC0 application retrieves the data from the space and locally applies data transformations, such as index mapping and coordinate interpolation, to prepare the data for its 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, which then deploys and executes this kernel on the space. Once the kernel executes on the space, XGC0 has to retrieve only 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 computing nodes and each computing node 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 on 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, which is 5.5 kB, and is much smaller than the size of the field data, which 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 execution time for the map operation. The average time savings is 0.14 seconds per processor, which may not seem significant. However, considering the case with 512 computing nodes, at the application level, the time savings becomes 1.2 minutes for each iteration, and for a 50-iteration run it becomes 1 hour of CPU time savings. The time saving metric refers to the CPU time by which an application is billed, and this is additive. These results clearly show that ActiveSpaces can improve application run times by offloading data reduction and/or computationally intensive operations.
VI. RELATED WORK

This section presents a summary of the 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 saving 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 shows 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 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 on which 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 the additional processing cores. This approach has demonstrated good performance benefits for a wide range of applications from gaming, to graphics rendering, 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 is 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, i.e., 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 with data processing at the 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 in 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 run-time system can migrate the operation execution to optimize the total execution time, but does not migrate the code itself, i.e., the binary code is statically compiled in the run-time system and called at each node on demand.

Dyinst [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 the 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 in the run-time system.

VII. 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 the staging area. This work is motivated by the data management challenges of large-scale simulations running on leadership-class resources, and explores an alternate approach to moving the data, i.e., moving the processing codes to the data. The ActiveSpaces approach can be applied to different classes of applications which include applications that filter data, e.g., application monitoring, visualization, biomedical imaging, or applications that pre-process data, e.g., interpolation, compression, scaling, inversion. We demonstrated the framework usage with real production codes from plasma fusion.

ActiveSpaces provides programming support for defining the data-processing routines to be downloaded to the staging area, using the native C programming 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 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.

ACKNOWLEDGMENTS

The research presented in this paper is supported in part by National Science Foundation via grant numbers IIP 0758566, CCF-0833039 and DMS-0835436, and by Department of Energy via the grant number DE-FG02-06ER54857, and by an IBM Faculty Award, and was conducted as part of the Center for Autonomic Computing at Rutgers University. This material was based on work supported by the National Science Foundation, while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material, are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES