Analysis in the Data Path of an Object-centric Data Management System

Richard Warren*, Jerome Soumagne*, Jingqing Mu*, Houjun Tang†, Suren Byna†, Bin Dong†, Quincey Koziol†
*The HDF Group, Champaign, IL
†Lawrence Berkeley National Laboratory, Berkeley, CA
Email: *{kmu, jsoumagne, richard.warren}@hdfgroup.org, †{htang4, sbyna, dbin, koziol}@lbl.gov

***Abstract*—Emerging high performance computing (HPC) systems are expected to be deployed with an unprecedented level of complexity due to a deep system memory and storage hierarchy. Efficient and scalable methods of data management and movement through the multi-level storage hierarchy of upcoming HPC systems will be critical for scientific applications at exascale. In this paper, we propose *in locus analysis* that allows registering user-defined functions (UDFs) and running those functions automatically while the data is moving between levels of a storage hierarchy. We implement this analysis in the data path approach in our object-centric data management system, called Proactive Data Containers (PDC). The transparent invocation of analysis functions as part of PDC object mapping is an optimized approach to minimize latency to access data as it moves within the storage hierarchy. Because a user defined analysis or transform function will be invoked automatically by the PDC runtime, the user simply registers their functions for PDC to identify the function name as well as the required list of actual parameters. To demonstrate the validity and flexibility of this analysis approach, we have implemented several scientific analysis kernels to compare against other HPC analysis-oriented approaches.**

I. Introduction

The computing trend in high-performance computing (HPC) that has gained significance is the application of *in situ* processing for data analysis[1]. In situ processing in recent years has been shown to be advantageous in many domains. Often, as was the case in the initial forays into computer animations, the amount of data that can be generated by a computer can easily outstrip the ability of the system to store that data. Similarly, long running simulations of real world phenomena can take large amounts of computer time and storage. To better optimize computing resources and lower costs, data sampling of such simulations can provide users with timely feedback to allow human intervention to restart or re-calibrate a simulation gone wrong. In situ processing within HPC has been used for reducing I/O latency, data size, and analyzing data while it resides in the memory of compute nodes. For instance, in situ processing has been used to gather descriptive statistics for future data indexing and querying (see DIRAC [1]); or to utilize data sampling for graphical display. Additional examples include data compression [2], I/O systems such as DataSpaces [3], FlexPath [4], and Glean [5] support in situ analysis while the application data is in memory. Among these, DataSpaces and FlexPath are integrated into the ADIOS framework [6] for performing I/O and data analytics. However, user involvement is needed in these frameworks either by modifying simulation codes to infuse the analysis functions or by scheduling analysis codes to run concurrently with the simulation. ArrayUDF [7] executes user-defined functions (UDF) on array data structures using the stencil abstraction. ArrayUDF has been primarily designed for post-processing analysis and execution of in situ tasks but requires user involvement in allocating resources.

In this paper, we propose an automatic analysis mechanism called ***in locus analysis***, which expands on the concept of in situ processing. A 'locus' refers to a storage location in a multi-layer storage hierarchy, such as memory or storage on a compute node, or storage on burst buffer, etc. Automating the process of expressing arbitrary analysis functions on data while the data is in flight between different storage layers and running the functions without any user involvement are challenging tasks. Towards this overarching goal, we provide users with a capability to register analysis functions for a system to schedule running the functions while the data is in flight. We provide analysis function registration APIs and execution semantics on scheduling to run those functions.

We have implemented the in locus analysis framework in our recently developed Proactive Data Containers (PDC) framework [8]–[11]. PDC provides an object-centric abstraction for data. It allows transparent and asynchronous data movement across deep storage hierarchy using user-level computation resources. As HPC systems are equipped with multi-level storage hierarchy, moving the data transparently across these levels removes the burden of data management on users. PDC uses a client-server architecture and the object management that runs data management services in user space to invoke user defined functions on clients or servers, which is nearest to the data of interest. This automated in locus analysis approach frees the application authors to concentrate on developing their analysis codes rather than on the mechanics of accessing data and scheduling analysis while the data of interest is moving through the system.

[1]The earliest in situ applications were from the 1960's, e.g. Zajac, direct-to-film animations.

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00020

IEEE computer society

The principal software components of the PDC framework include a metadata service to manipulate (create, update, and delete) rich sets of metadata and provenance of data objects, along with a data service that moves the data within the storage hierarchy. More details of the PDC system can be found in [8] and [10]. By providing the APIs to allow users to register analysis functions on data objects and to express when to perform analysis, one can easily express the resources to execute analysis functions and also define a workflow of data transformations. The PDC system uses an integrated approach in which the user-provided input gives guidance for the execution of analysis in the data path to be optimized. The generated results are automatically managed as objects and handles are provided back to the user. The metadata related to analysis output is subsequently stored back in the original data objects to capture provenance.

In summary, the contributions of this work are:

1) Introduction of the concept of in locus analysis in the data path for automated execution of analysis functions while the data is moving between storage layers.
2) Definition of an API for applications to register analysis and transformations on data objects;
3) Design and development of an analysis framework in the PDC object-centric data management system.

The remainder of this paper is organized as follows. Because we used the PDC framework for implementing in locus analysis, Section II provides the basics of the PDC system. We then present details of our proposed in locus analysis and transformation framework in Section III. We evaluate the analysis framework in Section V using various analysis and transformation functions at different scales. We conclude the paper in Section VII with a brief discussion of future work.

II. Background: Proactive Data Containers

In this section, we provide a more detailed overview of the PDC framework and its data management interface.

Proactive Data Containers (PDC) [8]–[11] provides transparent and asynchronous management of data movement in hierarchical storage systems. The principal functional components of PDC as shown in Figure 1, are a collection of distributed clients which interact with a collection of servers that provide the data and metadata management services. This figure describes *compute nodes* and *burst buffer* nodes; these are similar but distinguished by the fact that the compute nodes do not have access to the burst-buffer hardware. In most other respects, nodes are simply a collection of some number of physical CPUs and will either run a PDC server instance or a PDC client application. Servers and client applications all run as user processes and are started and run independently. Servers subdivide the data and meta-data management tasks and communicate between themselves and clients utilizing the

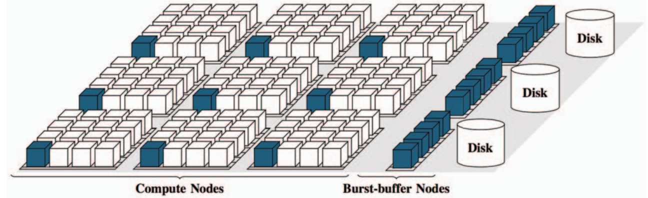

Figure 1: A simplified representation of a pre-exascale system with multi-tier storage [8]. PDC running services are highlighted in blue. In that model, PDC services are either co-located within the same node or eventually distributed over remote burst buffer nodes.

Mercury [12] remote procedure call (RPC) library. As a collection this coordinated runtime environment provides PDC with efficient, asynchronous data movement operations, while simultaneously enabling *in-transit* data transforms and *in locus* analysis operations.

Object-centric abstractions in PDC are the data constructs of *Containers*, *Objects*, and *Regions*, all of which have metadata attributes or *Properties*. A PDC *Object* is the generic term used to describe a byte stream and can represent either a data variable or an application object.

A *Container* is a collection of PDC *Objects* which share similar user-defined attributes. An example of this conceptual usage might be that all data variables produced by a simulation or experiment would logically be added to single PDC container. Objects are managed globally by data and metadata services and can be placed at any level of the storage hierarchy (e.g., NVRAM, burst-buffer, disk, etc.). This flexible approach allows data, objects, and containers to be spread over different types of storage media or *storage loci*, which in turn can be thought of as differing levels of data caching. Regions are the lowest level containers in PDC and hold user data, associated metadata, and are the basic units of management for data movement operations within PDC.

Traditional I/O libraries include *explicit* data movement operations such as read and write functions. PDC introduces an alternative concept of *object mapping* to define object relationships which when used in concert with a PDC data consistency framework, enable *implicit* IO operations, e.g. if a mapping between region A and region B is defined, then when modifications to A are made, those changes will be reflected automatically into B as a consequence of a region unlock operation (Section II-A).

A. Object Mapping and Data Consistency

An object mapping primitive allows users to define an association between a region within an application's memory and that of a similar region within a global PDC object. Mapping operations are defined on a per-region basis and can be thought of as a publish and subscribe mechanism so that once a mapping is established and a region is published,

data movement can occur to keep updates globally visible. When defining a mapping, an application provides property information (metadata) about the mapped region. This is the key information which allows PDC services to keep track of all mappings and prevent potential overlaps and conflicts.

To keep data consistency between the application's memory and that of the global PDC object, Read and Write locking semantics have been defined for PDC objects (with a region granularity to allow multiple regions within an object to be concurrently updated). Assuming the mapping from memory to object has already been established, a user will express the intent to read or modify the application's memory region by issuing a *lock request*. Acquiring a read lock enables the client view of region data to be updated from the global version if modifications have been made. The process of *releasing a lock* re-enables the global object to be updated with data from the client. Data movement is scheduled and will occur asynchronously and transparently to the user application.

III. In locus Analysis and Transformations in Data Path

The approach that PDC has initially adopted is to utilize the locus information in PDC object internals to determine where computation should take place. Computation, in the form of user-defined functions is enabled by registering functions with the runtime system. These registration APIs further identify which PDC objects and regions will be utilized as inputs and outputs when user-defined functions are invoked. In reality, the registrations provide the necessary metadata information to the PDC runtime and allow the identified functions to be invoked and run on nodes which are physically close to the data. This approach is intended to minimize data movement and to provide a level of asynchronous execution when coupled with PDC object mapping and the region update methodology that has been discussed previously (See Section II-A).

```
typedef enum {
    UNKNOWN =0,
    SERVER_MEMORY =1,
    CLIENT_MEMORY =2,
    FLASH =3,
    DISK =4,
    FILESYSTEM =5,
    TAPE =6
} PDC_loci;
```

Figure 2: PDC Locus values

PDC defines locus values as shown in figure 2. At present, not all definitions are utilized. In the context of the current discussion however, the SERVER_MEMORY and CLIENT_MEMORY locus values provide guidance as to the locale to run a registered analysis function. Because data is allowed to migrate somewhat autonomously, analysis registration metadata is copied and utilized by both the SERVER and CLIENT runtime systems. As the PDC platform matures, we envisage both user controlled data migration, e.g. from SERVER_MEMORY to CLIENT_MEMORY or vice-versa to be enabled by introducing a *Redistribute* API. Some machine learning algorithms might also be introduced to predict data movement as a function of computation and thus pre-fetch input data into low latency storage using these same redistribution interfaces.

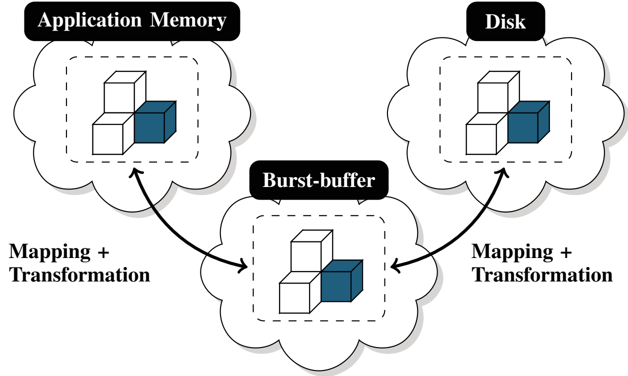

Figure 3: Abstracted data can reside at any level of the storage hierarchy [8]. While the data objects are mapped between different storage layers and are in movement, if transform or analysis functions have been registered for those objects, the defined data transformations or analysis functions will be executed at the designated locus. These transformations and analysis are user-defined functions and are executed asynchronously by the PDC runtime system.

To achieve maximum benefit, analysis functions need to be applied at the locus where data currently resides (see figure 3). PDC is coded in 'C' and thus presents users with a native API which is consistent with that language, i.e. the storage order for arrays is row-major and user programs and analysis functions are generally expressed in this language. PDC *transforms* and *analysis* registration APIs are defined as 'C' APIs. While PDC transforms and PDC analysis functions are similar, the intent of each API is distinct. The majority of PDC transforms should be packaged as a standard library to affect automatic data operations such as array transpose operations or simple datatype conversions. In addition to prepackaged transforms, *user-defined functions* can of course be utilized as we show in one specific example in which the Blosc [13] compression library is employed to affect higher performance data movement and data storage.

A final differentiation factor between analysis and transforms is the idea that a PDC *transform* produces state changes, (e.g. compressed, transposed, converted datatype, etc), and that transform outputs are intended as temporary containers in a potential chain of computation or storage

```
...
numparticles = NPARTICLES;
dims[0] = numparticles;
x = (float *)malloc(numparticles*sizeof(float));
...
mysize[0] = numparticles;
obj_xx = PDCobj_create_mpi(cont_id,
"obj-var-xx", obj_prop_xx, 0, MPI_COMM_WORLD);
...
region_x = PDCregion_create(ndim, offset, mysize);
...
region_xx = PDCregion_create(ndim,
                             offset_remote, mysize);
...
/* Register COMPRESSION function on outgoing data */
PDCbuf_map_transform_register("pdc_transform_compress",
      &x[0], region_x, obj_xx, region_xx,
      0, INCR_STATE, DATA_OUT);
/* Register UN-Compress on server incoming data */
PDCbuf_map_transform_register("pdc_transform_decompress",
      NULL, region_x, obj_xx, region_xx,
      1, DECR_STATE, DATA_IN);
...
```

Figure 4: Register PDC Transforms

operations. A common PDC example might be the coupling of a transpose operation to take native row-ordered data and output column-order in order to invoke a FORTRAN based analysis function. In the context of staging these buffers to persistent storage, the temporary nature of this data needs to be recorded as a new state along with the size meta-data and an associated registered transform to allow subsequent restoration of the data for application use.

A. Enabling Analysis and Transform Relocation

Analysis and transform registration APIs define the desired function entry points symbolically rather than as a native pointer to function. This mechanism enables the actual function resolution and loading to be delayed until the user defined function is required. This dynamic loading capability allows functions to be executed either within the client program or by the PDC server as part of data movement operations, e.g. PDC buffer or region mapping, or even both client and server as might be required in a multistep data transform.

Analysis and transform function dynamic loading by a client or server, is accomplished by utilizing the `dlopen` and `dlsym` functions to resolve the desired functions from shared libraries or from the application itself. We have implemented default library names which are searched to resolve the desired function name or alternately the function naming syntax allows either a *DLL* or executable name to be specified directly by the user, e.g. `functionName[:libraryName]`.

Simple code examples will illustrate a few details of the PDC transform and analysis APIs. In figure 4 `PDCbuf_map_transform_register()`, the programmer expresses not only the function to invoke, but the conditions under which the call to the transform is made. In the first registration call, the last three input arguments provide a *state*, *state_transition_action*, and *relative location*. We understand from the API name, that the desired function will occur as part of the *buf_map* operation and further, that on the outgoing side of a data transfer (in this case, the client), the function will be called and upon successful completion will increment the internal state of the data. Secondly, there is a matching call to `pdc_transform_decompress()` on the receiving (server) side of the map operation. If that completes successfully, the data state is decremented. This state transition mechanism allows multiple transforms within a single data movement operation which can obviously be ordered by matching the transform with the data state. In this example, data is compressed on the client, then sent to the server where it is uncompressed.

```
...
obj1 = PDCobj_create_mpi(cont_id, "obj-var-array1",
 obj1_prop, 0, comm);
...
obj2 = PDCobj_create_mpi(cont_id, "obj-var-result1",
 obj2_prop, 0, comm);

// create regions
r1 = PDCregion_create(2, offset0, myTestArray_dims);
r2 = PDCregion_create(2, offset0, myTestArray_dims);

input1_iter = PDCobj_data_iter_create(obj1, r1);
result1_iter = PDCobj_data_iter_create(obj2, r2);

PDCobj_analysis_register(
      "arrayudf_stencils:arrayudf_example",
      input1_iter, result1_iter);

PDCbuf_obj_map(myTestArray, PDC_FLOAT,
      r1, obj1, r2);
```

Figure 5: Register PDC Analysis

The second example (figure 5) introduces an API that we have not yet introduced. The `PDCobj_data_iter_create()` call returns a PDC identifier that is associated with a data *iterator*. A PDC iterator is similar to data iterators in other languages in that it provides an abstraction to allow multiple calls to `PDCobj_data_getNextBlock()` to return successive slices of data for use in the defined analysis functions. This abstraction attempts to limit the amount of data copying that might otherwise be necessary. In our PDC implementation, the basic iterator form returns a slice of data which contains one dimension fewer than the object region to which it refers. Example: if a region has 2 dimensions, then the iterator will return a 1D data slice (rows or columns depending on storage order); if the referenced region has 3 dimensions, then our iterator will return a 2D plane of data having the same dimensions as the original rows and columns. Another important point with regards to the PDC data iterator is the fact that we generally return data as contiguous data slices. This enables users to code high-performance algorithms which can take

advantage of compiler technology to produce vectorized code and/or to apply other optimization techniques that are currently available. At a minimum, the approach is meant to be *hardware cache friendly* while maintaining a simplified programming approach. We will look at example stencil applications and some performance comparisons later in this paper (section V).

B. A Simple PDC Analysis Example

With the registration process of a PDC analysis functions having been introduced, we can examine the various elements of coding a stencil analysis application for use in a benchmarking exercise.

In this endeavor, we borrowed the basic algorithms that are provided as the example application codes described in the ArrayUDF project papers. They provide a base implementation and the series of benchmarks against which we were able to validate our PDC results for similar analysis calculations. Additionally, having a central focus on user defined functions (UDFs) as the vehicle for scientific analysis provided an interesting match to the challenge of how to incorporate analysis and automated data transformations into an in locus computational design that is enabled within PDC. It differs significantly from PDC however, in that ArrayUDF applications are MPI based parallel applications. The programming environment provides C++ classes to address the data management tasks such as the reading and writing of parallel HDF5 files, while giving users the ability to express complex algorithms using structural locality. Access to HDF5 file data is relatively transparent, but beyond reading and writing of file data there is no additional data movement, e.g. between a client and server; nor is ArrayUDF an embedded part of a more general runtime environment.

We can see from Figures 6 and 7, that like ArrayUDF, it is not difficult to construct stencil algorithms in PDC which perform fairly efficiently. Each of the examples described the following sections are run on one or more data servers as the result of an object mapping update operation. As a result, these analysis functions are run asychronously on a PDC server once the input array has been filled by reading an HDF5 input file and followed by a PDC write unlock operation.

The stencil construction at the heart of the moving average of a time series benchmark example, is integrated into a loop construct and is invoked N times (N being the size of the 3rd dimension and representing the historical information for each geographic location in the survey). In the CoRTAD example, the version 4 sectional map of the database has an overall size of `1522 x 540 x 540`, thus producing a loop of 1522 iterations. Initialization of the stencil values is one *edge condition* that must be taken care of. In this example, the first plane returned by `getNextBlock()` is replicated 3 times into the stencil. For each loop, the stencil entries are moved by one position with the last value being replaced by the data pointer returned from `getNextBlock()`.

```
/* PDC analysis entrypoint:
 * This wrapper calls the CoRTAD running average function
 * on each PDC array slice as returned from the
 * PDCobj_data_getNextBlock();
 *
 * We treat each 2D array slice as a contiguous vector
 * which is then presented as a series of 4 historical
 * datapoints. In other words, this is a PDC version
 * of a stencil implementation.
 */
size_t neon_stencil(pdcid_t iterIn, pdcid_t iterOut,
                    iterator_cbs_t *cbs)
{
    short *dataIn = NULL;
    short *stencil[4] = {NULL, NULL, NULL, NULL};

    float *dataOut = NULL;
    size_t dimsIn[3] = {0,};
    size_t dimsOut[3] = {0,};
    size_t k, blockLengthOut, blockLengthIn;
    size_t number_of_slices;
    size_t result = 0;
    ...
    number_of_slices = (*cbs->getSliceCount)(iterOut);
    ...
    if ((blockLengthIn = (*cbss->getNextBlock)(
            iterIn, (void **)&dataIn, dimsIn)) == 0)
        printf("neon_stencil: Empty Input!\n");
    else {
        stencil[0] = stencil[1] =
        stencil[2] = stencil[3] = dataIn;
        for (k=0; k< number_of_slices; k++) {
            if ((blockLengthOut = (*cbs->getNextBlock)(
                 iterOut, (void **)&dataOut, dimsOut)) >
                     0) {
                if (cortad_avg_func(stencil, dataOut,
                    blockLengthIn) == 0) {
                    /* move stencil values */
                    stencil[0] = stencil[1];
                    stencil[1] = stencil[2];
                    stencil[2] = stencil[3];
                }
                /* Get the next data block and insert
                 * it into the stencil
                 */
                blockLengthIn = (*cbs->getNextBlock)(
                    iterIn, (void **)&dataIn, NULL);
                stencil[3] = dataIn;
            }
        }
    }
```

Figure 6: PDC Analysis Loop

IV. Experimental Setup

The evaluation of the PDC analysis infrastructure was accomplished in two steps. The first step was to utilize small HDF5 datasets to validate the PDC implementation of the analysis algorithms against the results generated by ArrayUDF for the same problems on the same datasets. This validation step was accomplished on a generic Linux workstation consisting of four i5-3570 cores with a total of 16GB of main memory. Subsequent to that correctness validation, an initial set of performance comparisons at this small scale was done.

```
/* -------------------------------------------------
 * ArrayUDF:: neon/ running average application
 * For each point, create the average of the current
 * point value with the 3 previous values.
 *
 inline float myfunc1(const SDSArrayCell<float> &c) {
    return (c(0,0,0)+c(-1,0,0)+c(-2,0, 0)+c(-3,0,0))/4;
 }
*/
/* -------------------------------------------------
 * Here is the PDC version that is called by the
 * PDC Analysis loop (see the previous figure)
 */
size_t cortad_avg_func(short *stencil[4],
                       float *out,
                       size_t elements)
{
    int k;
    for(k = 0; k < elements; k++) {
        out[k] = (float)((stencil[0][k] +
                          stencil[1][k] +
                          stencil[2][k] +
                          stencil[3][k]) / 4.0);
    }
    return 0;
}
```

Figure 7: CoRTAD Average Function

The next step addressed performance of the analysis algorithms on larger scale datasets which required a larger number of cores. This second stage performance evaluation took place on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC), which is a Cray XC40 supercomputer with 1630 Intel Xeon Haswell nodes. Each node consists of 32 cores and 128GB of memory.

PDC is a client-server system which can be deployed in two slightly different configurations. A *shared mode* configuration, has one PDC server per node, which utilizes single core, leaving the remaining 31 cores for user application execution. A *dedicated mode* where the PDC servers and user's application are on separate nodes. All PDC analysis experiments were run using the shared mode configuration. The advantage of this decision for analysis and transforms is that communications between co-located clients and server takes place via shared memory. Additionally, the Server is typically configured to utilize multi-threading. This should result in a performance advantage over the non-multi-threaded approach since computational tasks can be offloaded to dedicated threads for completion.

Though PDC utilizes `srun` to launch parallel application instances, it does not generally use MPI for communication. Instead, we relied on the Mercury [12] RPC library, an HPC-optimized C library for Remote Procedure Calls, as the communication mechanism. In our experiments, we configured Mercury to utilize the communication protocols of the libfabric [14] plugin with TCP. Other options such as Cray GNI are available but were not used in these experiments.

V. Performance Evaluation

Though we utilized ArrayUDF as a baseline to validate the PDC ability to address scientific data analysis, our performance report is intended to show that we believe PDC is currently able to achieve an acceptable level of performance while adding a significant level of new functionality with the incorporation of in locus computation. These new capabilities can provide valuable insight into the potential benefits of intelligent storage devices as computer architecture evolves in this new direction.

ArrayUDF [7] reported performance comparisons of their implementation against SciDB [15], RasDaMan [16], and Spark [17] on several common science applications:

- Moving average based smoothing for a time series.
- Vorticity computation
- Peak detection
- Trilinear interpolation

Rather than reiterating those ArrayUDF comparisons, we took the opportunity to re-implement most of these benchmark codes in PDC. In all cases, PDC has been able to meet or exceed the observed performance of the ArrayUDF examples. Previously mentioned and shown as an example in figures 6 and 7, the CoRTAD example is a moving average based smoothing application.

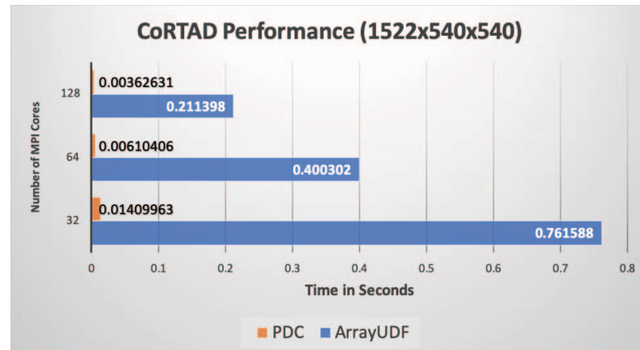

Figure 8: Performance of CoRTAD

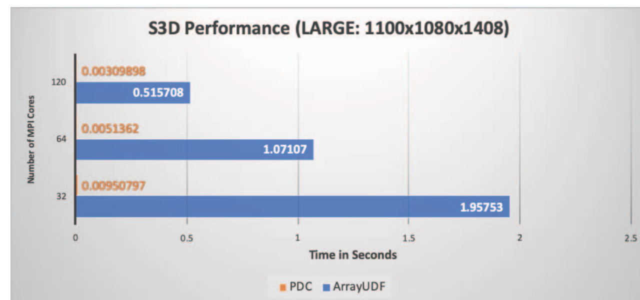

Figure 9: Performance of S3D: A 3 point stencil

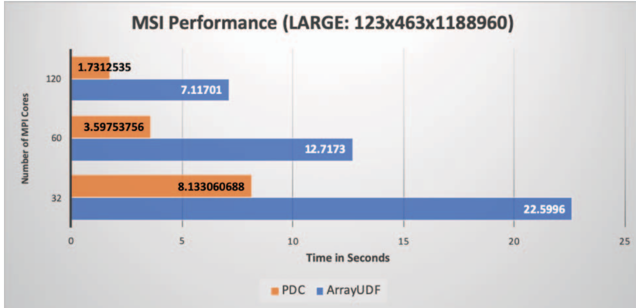

Figure 10: Performance of OpenMSI: A 6 point stencil

The S3D test (vorticity computation) is a three point stencil code and is programmed similarly to the CoRTAD example shown previously in figures 6 and 7. The fact that S3D was the simplest code and exhibits the best cache behavior of all the test examples, allows the PDC implementation to significantly outperform the arrayUDF implementation (see figure 9).

The most complex test in the collection that we used for performance comparisons is the MSI application. This is the core code of a Laplacian calculator and is programmed as a 6 point stencil. The complexity is derived from the necessity to detect and manage edge conditions for the five surrounding points of the stencil calculation and the overall performance relative to ArrayUDF is the smallest that we observed (see figure 10).

For purposes of leveling the performance comparisons as seen in Figures 8, 9, and 10, we utilized the reported *average UDF times* running on the same number of cores.

Our PDC experiments rely on programming in HDF5 to read datasets and to utilize the scaleout capabilities of the parallel HDF5 library to reduce the local memory footprint on each PDC client process.

A. Explaining the Analysis Performance

It is worth stating once again, that unlike PDC which utilizes object mapping to invoke the in situ analysis automatically on a server (in this case) as a result of a buffer map operation, the ArrayUDF system provides a parallel programming approach for HPC which addresses the analysis of data with a strong focus on ease of programming. Timings for the PDC analysis are provided by timing the actual code that runs on the PDC server. At the point where a PDC analysis function is invoked, data is already local to the server and the execution thread or process (depending on how the server is configured) is exactly similar to ArrayUDF or any other computing engine. The actual performance differences between PDC and ArrayUDF are partially explained by the different computational approaches. The ArrayUDF C++ framework provides users with *structural locality* and ease of use through a C++ *Array* class and the `Apply` method which implements *Stencil* and other analysis optimizations such as data *Chunking* to minimize memory consumption when working on large scientific data sets.. This approach is quite nice, but comes at the overhead cost of imposing method calls to implement the relative indexing for each element access in the stencil computations. In comparison, PDC appears to benefit significantly from having relatively high cache locality and straight-forward direct element access via caching and/or compiler vectorization within the analysis loops. Much work remains to be accomplished however. In particular, there is a need for PDC to implement additional APIs to allow more flexible input and output arguments both in number and iterator types.

B. Performance Choices using PDC Transforms

As previously described in section III-A, we introduced specific PDC APIs which give users some flexibility with regards to when and where transformations should take place. The more general thought behind transforms however, is that many such actions can be undertaken automatically by the data management system.

A simple type casting experiment was created to test the impact on the perceived performance of the operation as observed by the client. The experiment consists of mapping parallel client regions containing a range of 1 to 32 Mega elements (2^{25}) of double precision data per client to a target region of similar shape per slice but defined as containing 32 bit integers.

Once the PDC data consistency operation (region lock release) completes for the mapped target, the region buffer will contain the transformed double precision data of the source into 32 bit integers in the target. The summary of release times shown in Table I reflect the sum of the transform execution time plus the data transfer time and lock release as measured by the client.

Intellectually, this experiment is similar to the one mentioned previously involving the BLOSC compression library. If the type conversion takes place on the client as part of the mapping operation between client and server, we might expect that since the data size to be transferred is reduced by a factor of 2, that the overall performance will be optimal. By changing the selection of the relative location in the transform registration in a subsequent test, i.e. to DATA_IN, to specify that the type conversion operation will be run on a server, we attempt to validate our expected results.

The observed performance shown in Table I validates our initial expectations, i.e. that the client based transform for this example should always outperform the server based version. The third column shows the best case scenario, i.e. there is a dedicated server per client and thus there should never be any resource contentions to negatively affect performance. In contrast, the fourth column of the table captures the region release times for a *shared server* which provides services to 31 clients. The performance differences are quite evident and are due to the apparent over-subscription scenario in which each of the 31 near

Size/proc	Locus	Release time*	Release time**
1 MB	Client	3.89ms	117.65ms
	Server	18.56ms	411.30ms
4 MB	Client	22.27ms	250.60ms
	Server	27.07ms	476.44ms
16 MB[2]	Client	47.82ms	173.55ms
	Server	63.19ms	167.15ms
32 MB[2]	Client	95.35ms	304.84ms
	Server	120.02ms	396.50ms

Table I: Region release times for mapped arrays with Transforms enabled. Release time(*) refers to a single client per server. Release time(**) column shows the multi-threaded shared Server results per 31 clients. The 16 and 32 MB element timing entries were run with the 31 clients but these are distributed over 2 nodes with two servers, effectively removing the dynamic over-subscription scenario during server based transforms.

simultaneous client requests are handled by 31 server threads which run more or less in competition with the clients. The 16MB and 32MB array element entries in the table show a server configuration that is slightly different than that used with the 1MB and 4MB experiment. In these final two experiments, there are two (2) servers per 31 clients and these are split more or less equally over two nodes. The forth column reflects the improvement in the server response time for this configuration (note the 16MB entries vs. those of the 4MB experiment).

Size/proc	Locus	Best Case*	Shared Servers**
1 MB	Client	1	30.24
	Server	4.77	22.16
4 MB	Client	1	11.25
	Server	1.21	17.60
16 MB[2]	Client	1	3.62
	Server	1.32	2.64
32 MB[2]	Client	1	3.19
	Server	1.25	3.30

Table II: Region release times expressed as ratios. Best Case(*) ratios are measured relative to client based transforms and a dedicated server. Shared Servers(**) column shows the multi-threaded shared Server results per 31 clients.

For applications which might require heavy utilization of transforms (or in locus analysis), a suggested approach might be to reduce the dynamic over-subscription that we describe above, by allocating fewer clients per PDC server. Table II shows the dynamic oversubscription effects of the shared server configuration under a worst case scenario, e.g. all 31 clients are requesting the services of a single multi-threaded server. As in the Table I, the 16MB and 32 MB entries in Shared Servers column reflect the improved performance of halving the server load by adding an additional server to the PDC configuration.

VI. Related Work

Existing in situ infrastructures (e.g., ADIOS [6], ParaView Catalyst [18], VisIt Libsim [19], SENSEI [20], and Ascent [21]) provide APIs to allow Python to be employed as an Analysis engine. Through an embedded Python interpreter, users have access to a plethora of pre-packaged analysis tools and in most cases these will have direct access to simulation data. What is typically missing however, is the ability to relocate those analysis functions automatically and repeatedly as data moves through an analysis pipeline.

ALPINE [22] is another in situ infrastructure that grew out of an earlier effort called *Strawman* [23], and focuses for the most part on visualization tasks. It relies on *Conduit* and a parallel hybrid version of VTK-m [24] called VTK-h. It is tightly coupled with the *Ascent* [25] runtime that provides a convenient visualization programming feature called *Pipelines*. This construct allows explicit collections of filters to be constructed, managed, and applied to create data visualization *Scenes and Extracts*. This approximates the workings of PDC transforms and in locus analysis functions. In PDC, however, the functional abstractions of Scenes, Extracts, and Pipelines would be replaced by PDC Objects and Regions, which are managed automatically by the PDC runtime.

DataSpaces [3] addresses the problems of data movement within large scale scientific simulations. In particular, there is an attempt to utilize an in situ analysis approach to address the increasing performance gap between simulations which can produce data at rates that outstrip the storage system capability to store those results.

Glean [26] adds topology awareness and IO acceleration in addition to the in situ analysis framework in an attempt to address the growing issue of simulation data production vs. IO capacity.

Flexpath [4] messaging implements parallel pipelines and *active messages* as yet another tool to provide an efficient staging context for analytics and visualization in the current and next generation HPC computing environments.

Lastly, we mention once again the ArrayUDF [7] framework, which figured prominently in the PDC analysis design. This framework has focused primarily on the ease of programming parallel codes for post-processing data analysis.

VII. Conclusions and Future Work

As emerging high-performance computing (HPC) systems get deployed with an unprecedented level of complexity due to a deep system memory and a storage hierarchy with integrated processing, there will be an ever increasing requirement to provide software tools which give users access to this distributed intelligence. In locus execution will introduce many opportunities for significant performance boosts for improving big data operations. It accomplishes this by adding computation capabilities at various points

while data is moving through the system. Ultimately this approach reduces the total amount of communication required to achieve some of the important tasks such as data indexing or querying. Despite these clear benefits, in locus processing is not commonly available.

We have demonstrated in this paper that Proactive Data Containers (PDCs) provides the necessary infrastructure to enable high performance data analysis solutions for current and next generation HPC systems. The use of dynamically loaded functions from shared libraries or application executables is an efficient approach which enables the runtime relocation of functions by PDC to a processor which is physically close the data of interest. Rather than rely on pre-packaged functionality, PDC instead allows user-defined functions to be specified via a flexible programming API. The PDC runtime then loads and executes the specified function either as a transform or analysis function as a result of data mapping operations. This offers unprecedented functionality and because basic operation entry-points are written in 'C', the performance has been shown to be excellent. PDC is actually achieving superior performance in all experiments which we have run to date.

The PDC framework will continue to expand upon the already defined capabilities and look to include pre-defined transforms which can automatically be invoked by the PDC runtime. The primary focus will likely be an expansion of the PDC data analysis capabilities in two major categories of Data augmentation: Index generation; and Data Analysis (e.g. Data Max, Data Min, etc). Lastly, we expect to further improve the number of arguments and Data iterator types which can be utilized when calling PDC analysis functions.

Acknowledgment

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and DE-SC0016454 (Project: Proactive Data Containers, Program manager: Dr. Laura Biven). This research used resources of the National Energy Research Scientific Computing Center, which is a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] A. Tsaregorodtsev, M. Bargiotti, N. Brook, A. C. Ramo, G. Castellani, P. Charpentier, C. Cioffi, J. Closier, R. G. Diaz, G. Kuznetsov, Y. Y. Li, R. Nandakumar, S. Paterson, R. Santinelli, A. C. Smith, M. S. Miguelez, and S. G. Jimenez, "DIRAC: a community grid solution," in *Journal of Physics Conference Series*, ser. Journal of Physics Conference Series, vol. 119, Jul. 2008, p. 062048.

[2] M. Howison, "High-Throughput Compression of FASTQ Data with SeqDB," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 10, no. 1, pp. 213–218, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1109/TCBB.2012.160

[3] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851481

[4] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/-subscribe system for large-scale science analytics," in *CCGrid '14*, May 2014, pp. 246–255.

[5] V. Vishwanath, M. Hereld, V. A. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Supercomputing*. ACM, 2011.

[6] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS)," in *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, ser. CLADE '08. New York, NY, USA: ACM, 2008, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/1383529.1383533

[7] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu, "ArrayUDF: User-Defined Scientific Data Analysis on Arrays," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: ACM, 2017, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/3078597.3078599

[8] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu, and R. Warren, "Toward Scalable and Asynchronous Object-Centric Data Management for HPC," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2018, pp. 113–122.

[9] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable Object-centric Metadata Management for High Performance Computing," in *CLUSTER*, 2017, pp. 359–369.

[10] J. Mu, J. Soumagne, H. Tang, S. Byna, Q. Koziol, and R. Warren, "A Transparent Server-Managed Object Storage System for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 477–481.

[11] J. Mu, J. Soumagne, S. Byna, Q. Koziol, H. Tang, , and R. Warren, ""interfacing hdf5 with a scalable object-centric storage system on hierarchical storage"," in *"2019 Cray User Group (CUG) meeting"*, May 2019.

[12] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling Remote Procedure Call for High-Performance Computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2013, pp. 1–8.

[13] F. Alted, “Blosc,” 2009, http://blosc.pytables.org/.

[14] Libfabric. [Online]. Available: https://ofiwg.github.io/libfabric/

[15] P. G. Brown, “Overview of SciDB: Large Scale Array Storage, Processing and Analysis,” in *SIGMOD*, 2010, pp. 963–968.

[16] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, “The Multidimensional Database System RasDaMan,” *SIGMOD Rec.*, vol. 27, no. 2, pp. 575–577, Jun. 1998. [Online]. Available: http://doi.acm.org/10.1145/276305.276386

[17] G. Gousios, “Big Data Software Analytics with Apache Spark,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE ’18. New York, NY, USA: ACM, 2018, pp. 542–543. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183458

[18] U. Ayachit, A. Bauer, B. Geveci, P. O’Leary, K. Moreland, N. Fabian, and J. Mauldin, “Paraview catalyst: Enabling in situ data analysis and visualization,” in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV2015. New York, NY, USA: ACM, 2015, pp. 25–29. [Online]. Available: http://doi.acm.org/10.1145/2828612.2828624

[19] J. Kress, D. Pugmire, S. Klasky, and H. Childs, “Visualization and analysis requirements for in situ processing for a large-scale fusion simulation code,” in *Proceedings of the 2Nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, ser. ISAV ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 45–50. [Online]. Available: https://doi.org/10.1109/ISAV.2016.14

[20] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel, “The SENSEI generic in situ interface,” in *Proceedings of the 2Nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, ser. ISAV ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 40–44. [Online]. Available: https://doi.org/10.1109/ISAV.2016.13

[21] M. Larsen, A. Woods, N. Marsaglia, A. Biswas, S. Dutta, C. Harrison, and H. Childs, “A flexible system for in situ triggers,” in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV ’18. New York, NY, USA: ACM, 2018, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/3281464.3281468

[22] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, “The alpine in situ infrastructure: Ascending from the ashes of strawman,” in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV’17. New York, NY, USA: ACM, 2017, pp. 42–46. [Online]. Available: http://doi.acm.org/10.1145/3144769.3144778

[23] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison, “Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes,” in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV2015. New York, NY, USA: ACM, 2015, pp. 30–35. [Online]. Available: http://doi.acm.org/10.1145/2828612.2828625

[24] D. Thompson, S. Jourdain, A. Bauer, B. Geveci, R. Maynard, R. R. Vatsavai, and P. O’Leary, “In situ summarization with vtk-m,” in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV’17. New York, NY, USA: ACM, 2017, pp. 32–36. [Online]. Available: http://doi.acm.org/10.1145/3144769.3144777

[25] L. L. N. Laboratory, “Ascent documentation web page,” https://ascent.readthedocs.io/en/latest/index.html, 2015-2018.

[26] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, “Topology-aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11. New York, NY, USA: ACM, 2011, pp. 19:1–19:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063409