AMRZone: A Runtime AMR Data Sharing Framework For Scientific Applications

Wenzhao Zhang^{1,3}, Houjun Tang^{1,3}, Steven Harenberg^{1,3}, Surendra Byna^{2}, Xiaocheng Zou^{1,3}, Dharshi Devendran^{2}, Daniel F. Martin^{2}, Kesheng Wu^{2}, Bin Dong^{2}, Scott Klasky^{3}, Nagiza F. Samatova^{1,3}

^{1}North Carolina State University, Raleigh, NC 27695, USA
^{2}Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
^{3}Oak Ridge National Laboratory, TN 37831, USA
*Corresponding author: samatova@csc.ncsu.edu

Abstract—**Frameworks that facilitate runtime data sharing across multiple applications are of great importance for scientific data analytics. Although existing frameworks work well over uniform mesh data, they can not effectively handle adaptive mesh refinement (AMR) data. Among the challenges to construct an AMR-capable framework include: (1) designing an architecture that facilitates online AMR data management; (2) achieving a load-balanced AMR data distribution for the data staging space at runtime; and (3) building an effective online index to support the unique spatial data retrieval requirements for AMR data. Towards addressing these challenges to support runtime AMR data sharing across scientific applications, we present the AMRZone framework. Experiments over real-world AMR datasets demonstrate AMRZone's effectiveness at achieving a balanced workload distribution, reading/writing large-scale datasets with thousands of parallel processes, and satisfying queries with spatial constraints. Moreover, AMRZone's performance and scalability are even comparable with existing state-of-the-art work when tested over uniform mesh data with up to 16384 cores; in the best case, our framework achieves a 46% performance improvement.**

I. INTRODUCTION

Scientific data analytics are often performed in a post-processing manner, as the data generated by a simulation is first written to the file system and then read for analytics, requiring substantial I/O time. Runtime data sharing across multiple applications is a promising alternative approach towards avoiding these increasingly severe I/O bottlenecks [30]. For instance, the data generated by a running simulation can be moved to the memory of a set of dedicated compute nodes where that data is then retrieved by various analytics applications, such as visualization and transformation [3], [14].

To facilitate this runtime data sharing, related methods typically organize a set of nodes to provide an in-memory data staging and management space on the server side. Through client side APIs, applications running on other nodes can efficiently write data to the space and retrieve data from it. Although these methods are effective at handling uniform mesh data, they currently do not support adaptive mesh refinement (AMR) data.

AMR represents a significant advance for large-scale scientific simulations [4]–[6]. By dynamically refining resolutions over time and space, AMR simulations generate hierarchical, multi-resolution, and non-uniform meshes. This kind of refinement provides sufficient precision for regions of interest at finer levels while avoiding unnecessary data generation for regions of non-interest [32]. In this paper, we focus on *block-structured* AMR, which consists of a collection of disjoint rectangular boxes (or regions) at each refinement level [32].

To the best of our knowledge, runtime AMR data sharing across applications has not been well explored. This is a non-trivial task due to the *dynamic characteristics* inherent to AMR data; namely, the numbers, sizes, and locations of the AMR boxes, are usually unpredictable before a simulation run and keep changing as the simulation progresses. These characteristics prevent existing methods from effectively handling AMR data. Moreover, flattening and unifying AMR boxes to make the data compatible with existing methods is not a viable solution as AMR's advantages would be lost and significant overhead would be introduced [32].

To create a framework that facilitates runtime AMR data sharing across multiple applications, there are three major challenges that should be addressed. First, the architecture of such a framework should enable efficient online AMR data management. However, runtime AMR metadata synchronization across distributed server processes would incur significant overhead, because of the dynamic characteristics inherent to AMR data and because the data being written to the staging space could arrive in any order.

Second, to retain high throughput, the framework should have a balanced workload distribution at runtime for the nodes on the server side. However, static data domain partitioning and distribution methods (e.g., space-filling curves [25]) typically fail to achieve this goal for AMR data. Due to the dynamic characteristics of AMR data, those methods can not determine a suitable partition size until most of the AMR boxes of a domain have been received and examined, which would produce a large runtime overhead.

Third, to support data retrieval of a specific spatial region, the framework needs an efficient online spatial index that can satisfy AMR's unique spatial data access patterns, which typically involve accessing multiple boxes across multiple levels [32]. However, existing spatial indices can not effectively catch the hierarchical and non-uniform structure of AMR data.

Moreover, due to the dynamic nature of AMR, how to build the spatial index efficiently at runtime while maintaining high data transmission performance poses another challenge.

In this paper, we propose AMRZone, a framework for facilitating runtime AMR data sharing across multiple scientific applications. In addition to addressing the above challenges, AMRZone even demonstrates comparable performance and scalability with existing state-of-the-art work when tested over uniform mesh data; in the best case, our framework achieves a 46% performance improvement. Specifically, towards addressing the above challenges, we make the following contributions through our framework:

- An architecture that facilitates AMR data management by dedicating some server processes to handle metadata exclusively (Section III-A).
- Online balanced AMR data distribution on the server side, by adopting an AMR boxes-based runtime workload assignment policy (Section III-B).
- A polytree-based online spatial index to facilitate spatially constrained AMR data retrieval (Section III-C).

II. Background

DataSpaces [16] is the current state-of-the-art framework for runtime data sharing across multiple scientific applications over uniform mesh data. In the following sections, we explain three major issues that prevent it from effectively supporting AMR data. Although other frameworks (described in Section V) can provide a distributed and in-memory data manipulating space, we select DataSpaces for comparison because it is the only framework that can build an explicit online index over the distributedly staged data as well as provide effective data access APIs, both of which are key features that facilitate runtime data sharing across applications.

At the heart of DataSpaces is a distributed hash index that enables efficient data retrieval from spatial regions of interest. The index is based on a Hilbert space-filling curve [23] that is used to partition a global data domain into sub-regions (or partitions) and then distribute these partitions evenly to the staging nodes. Although effective at handling uniform mesh data, there are several non-trivial issues that would arise if this Dataspaces were applied to AMR data. A 1GB 5-level dataset generated by BISICLES(a large-scale Antarctic ice sheet modeling code for climate simulation) [11] is used to illustrate the hierarchical and non-uniform structure of AMR data. Its visualization is shown in Figure 1.

A. Issue I: Architecture

DataSpaces has a client-server architecture. The server side is composed of a set of server processes (or servers) running on different nodes to form a virtual in-memory space. The client side is a collection of APIs used to interact with the space. Each server process is responsible for both maintaining metadata (the distribution of data sub-regions) and transporting data. The server side demands a pre-defined global domain size before any data is written or read. Therefore, once a partition

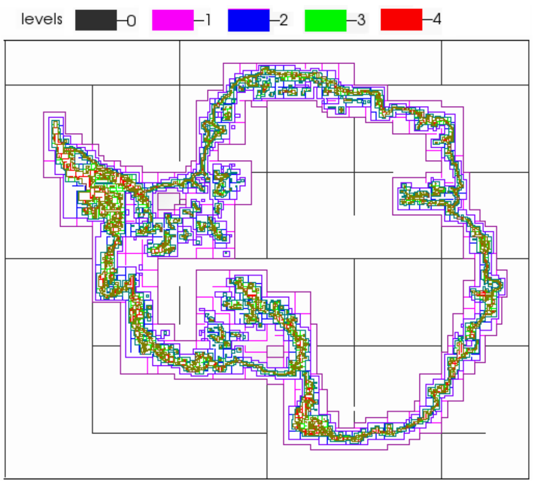

Fig. 1. The visualization graph of a 1GB block-structured AMR dataset generated by BISICLES [11]. The coarsest (or lowest) level (level 0) covers the entire global data domain. A finer (or higher) level is generated by refining a set of boxes on the adjacent coarser level, only covering some sub-regions of interest with higher resolution which is defined by a refinement ratio. The boxes at the finer levels represent spatial regions of more interest.

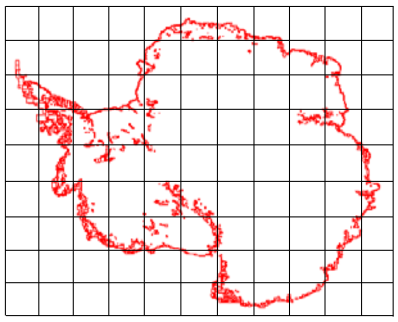

Fig. 2. A virtual bounding box over the finest level of the AMR data in Figure 1 and uniform partitions. These partitions would be distributed to the staging space according to a space-filling curve (e.g., Hilbert), leading to an unbalanced workload distribution, among other issues.

size is determined, it is easy to know the total number of sub-regions and how to map each one to the servers evenly, before any client connects to the servers. If a client needs to access a certain region of data, it can get the metadata by contacting any server, making the metadata management of this architecture very effective over uniform mesh data.

However, the dynamic nature of AMR data (namely, the numbers, sizes, and locations of the AMR boxes, are usually unpredictable before a simulation run and keep changing as the simulation progresses) makes it impossible for the framework to determine a balanced workload distribution before a simulation run (see Section II-B for details). Thus, in order to maintain consistent metadata between all the servers to monitor the overall workload status, it is necessary to frequently exchange metadata between all servers at runtime. This frequent communication between the servers results in a high runtime synchronization overhead.

B. *Issue II: Online Data Organization*

In the data staging space which is based on Hilbert curve, an unbalanced workload distribution could arise because the AMR boxes may not be evenly divided across the different partitions, as illustrated in Figure 2. Due to the dynamic nature of AMR data, it would be impossible for Dataspaces to determine a suitable virtual bounding box and partition sizes for all levels of all time-steps before a simulation run. Moreover, once the data has been partitioned and distributed to the server nodes, it would be too time consuming to dynamically optimize an unbalanced workload distribution, as that would require retrieving, repartitioning, and redistributing all of the staged data. On the other hand, attempting to achieve a balanced workload distribution at runtime would be costly because the majority of the AMR boxes must be received and evaluated first before a suitable partition size can be determined. Arguably, any partition methods that are based on space-filling curves (e.g., Z, Peano, Hilbert curves, etc. [25]) would not be effective at evenly distributing AMR data onto a set of nodes at runtime.

C. *Issue III: Online Spatial Index*

Analytics over AMR data are usually performed over the boxes from all levels that overlap with the specified spatial region [32], rather than just the boxes on a single level, as illustrated in Figure 4. However, to retrieve AMR boxes from all levels using Datapsace's hash index, where partitions are given by the Hilbert space-filling curve, it would be necessary to check the boxes in every partition that overlaps with the specified query region. This kind of linear checking is inefficient when faced with a large number of parallel queries. Moreover, it would appropriate CPU resources that could otherwise be used for processing other runtime tasks (e.g., data transportation), reducing parallelism.

III. Methods

AMRZone is designed to facilitate data sharing across multiple AMR-capable scientific applications. To address the three issues stated in Section II, AMRZone employs a centralized metadata management architecture design, an AMR boxes-based runtime workload assignment policy, and a polytree based spatial index.

A. *Architecture*

AMRZone consists of a distributed client-server architecture. The server side consists of a set of server processes (or servers), which run on a user-defined collection of compute nodes, providing a shared memory based virtual data staging and management space with public data access functions. The client side is a set of APIs that can be integrated by running applications (e.g., simulations or other data analytics programs) to access (e.g., write, retrieve, or update) the data in the space. The architecture of ARMZone is depicted in Figure 3.

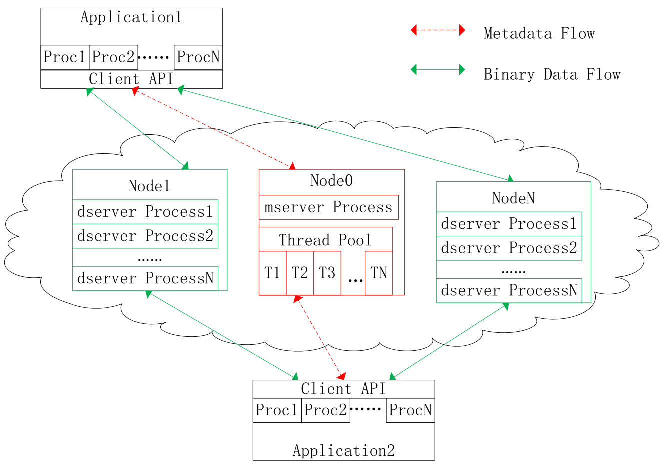

Fig. 3. The client-server architecture of AMRZone consists of two types of server processes: (1) *mservers* that only manage metadata, recording how AMR boxes are distributed across dservers and constructing spatial index; and (2) *dservers* that only manage the binary data. Note that `Application1` and `Application2` could be simulations or other data analytics programs. Also note that AMRzone does not limit the number of applications that can connect to the server side.

As described in Section II-A, the architecture design in which a server process is responsible for both maintaining metadata and transporting data introduces a significant performance overhead with AMR metadata management. To avoid this issue, AMRzone uses servers that exclusively handle either the metadata task or the data transportation task:

- *Mservers* - responsible for metadata, namely recording on which data-server an AMR box is placed and building a spatial index
- *Dservers* - manage the actual binary data of AMR boxes, such as data storage and transmission

The mservers act as coordinators between the clients and dservers. For example, when an application demands to write or read a certain region of data, it first contacts an mserver. The mserver updates or searches the metadata, and sends back the communication addresses of a certain set of dservers where the application can establish connections and perform the data transportation. Note that our framework does not limit the number of applications that connect to the data staging space.

With this architecture design, AMRzone is able to eliminate much metadata synchronization. To accomplish this, AMRZone uses a single mserver to handle all metadata associated with a single time-step of a simulation. This design eliminates the need to exchange metadata at runtime for the servers, preventing a significant performance overhead.

Concerning mservers and dservers placement on the compute nodes, each compute node contains only one mserver process. If a single node's memory is not sufficient, the metadata of different time-steps can be divided to multiple mservers on multiple compute nodes. Additionally, an mserver is able to set up a thread pool to handle incoming requests in parallel. Because threads can share the same memory space with a process, the threads inside the same processes do not need to exchange any metadata. Moreover, because the metadata-related message sizes are usually very small (a few dozens of bytes at most), this centralized metadata management design would not become a performance bottleneck to the framework. In contrast, multiple dserver processes run on a single node and there is no multi-threading inside a dserver process as there is no metadata sharing requirement for dservers. With this architecture design, AMRzone is able to achieve a high parallelism. In fact, as we show in Section IV-A, this architecture gives satisfactory performance when facing more than 10,000 writers/readers in parallel.

The write and read functions associated with the client APIs use AMR boxes as the atomic units. We make this design decision because AMR data is typically accessed by boxes. For example, in Chombo [9], a popular block-structured AMR data manipulation framework, a level's domain is represented by a collection of boxes. In a previous AMR data analytics method [32], analysis tasks are performed over a set of boxes. For more details of the initial prototype implementation of the framework, refer to Section III-D.

B. Online Data Organization

As stated in Section II-B, when space-filling curves are used to partition and distribute AMR data, it is difficult to avoid an unbalanced online workload placement at the staging space. To address this issue, the mservers of AMRzone do not perform a static partition of the data domain and, thus, do not require any global domain information. An mserver checks each received AMR box and determines its placement at runtime. To support this task, an mserver maintains a collection of workload tables to monitor how much data is stored on each dserver and each node.

Algorithm 1 gives the general procedure that the mservers use to decide on which dserver to place the binary data of an AMR box after a write-AMR-box request is received. First, it searches the dserver nodes workload table for a node with minimum workload (lines 1-7). Next, it finds the dserver process with the minimum workload on the specified node (line 9-14). By considering each of the AMR boxes received at runtime, the algorithm avoids any static and uniform data domain partition, thus realizing a far better data distribution balance across the staging space than space-filling curves. According to the results in Section IV-B, our framework's read performance over AMR data is comparable to the read performance over balanced uniform mesh data, demonstrating the effectiveness of this workload assignment policy.

To store the metadata for an individual AMR box, we employ a linear hash table because of its effective insertion and lookup operations. A cell of a hash table corresponds to an AMR box and a combination of a box's coordinates is used as the hash key. Inside each dserver process, there is also a collection of hash tables corresponding to all the levels of all time-steps. These are used to store and retrieve the boxes' binary data. When a dserver receives the binary data of a box, it only needs to insert the box to a hash table.

After a box is placed in the hash table, an mserver uses a simple pointer-based list to chain boxes at the same level of a time-step together. This operation would not compromise the performance of an mserver, as every insertion only involves the movement of a few lightweight pointers. These lists could be used for spatial query or constructing a polytree-based spatial index (III-C).

Algorithm 1: Algorithm of an mserver determines the placement for an AMR box at the staging space

```
Input: 1D array of the workload for all staging nodes:
       T1[]; the size of T1: N.
Input: 2D array of the workload for all dservers for each
       staging node: T2[N][]; the size of T2[N]: S[]
Result: A suitable dserver to place the box: $ds\_index$
1  /*Find the node with minimum workload:*/
2  $max\_wl = maximum\ possible\ workload\ for\ a\ node$
3  $node\_index = 0$
4  for $i = \{0,\ ...,\ N-1\}$ do
5      if $T1[i] < max\_wl$ then
6          $max\_wl = T1[i]$
7          $node\_index = i$
8  /*Find the dserver process with minimum workload:*/
9  $max\_wl = maximum\ possible\ workload\ for\ a\ dserver$
10 $ds\_index = 0$
11 for $i = \{0,\ ...,\ S[node\_index]-1\}$ do
12     if $T2[node\_index][i] < max\_wl$ then
13         $max\_wl = T2[node\_index][i]$
14         $ds\_index = i$
```

C. Online Spatial Index

To facilitate AMR data sharing across multiple applications, the framework needs to be able to effectively retrieve data from a specific spatial region of interest. As shown in a previous AMR data analytics work [32], spatial queries over AMR data usually request data from all levels, rather than a single level (as illustrated in Figure 4). To achieve this goal, an mserver's linear hash table, which is used to store the metadata of the AMR boxes (as described in III-B), is far from efficient. The hash table requires linearly checking all AMR boxes, reducing parallelism by competing with CPU resources from other runtime tasks, such as data transportation.

However, popular spatial indices (e.g., R-tree [21], Quadtree [26], UB-tree [2], etc.) are unsuitable for AMR data because they are not capable of capturing the hierarchical structure inherent to AMR data. The common idea of these indices is to organize a set of sub-spaces so that the selection space in the query can be efficiently narrowed down. A one-to-many relationship between sub-spaces is a necessary pre-condition to utilize most of those indices. However, relationships between AMR boxes at adjacent levels are many-to-many (in other words, one coarse box could cover multiple fine boxes and multiple fine boxes could cover one coarse box). Even if it is possible to divide a box into a set of

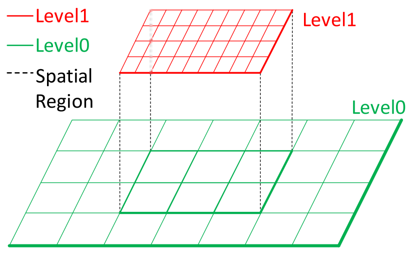

Fig. 4. A spatial query over AMR data. Typically, boxes are retrieved in multiple levels, rather than a single level. The boxes at level 1 are refined from bigger boxes at level 0.

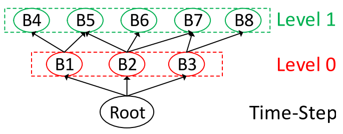

Fig. 5. The polytree based spatial index for AMR data. Tree nodes correspond to AMR boxes. Directed edges denote refinement relationship. It effectively represents the many-to-many refinement relationships of AMR boxes across different levels.

smaller boxes to reduce this many-to-many relationship into a one-to-many relationship, this approach may result in a large number of boxes and add more complicated runtime procedures to the framework. Not only could this compromise index construction and search performance, but it could also introduce network overhead due to more boxes write/read operations.

To overcome those issues, we propose a polytree [12] based spatial index for AMR data, as shown in Figure 5. The root of the polytree represents a single time-step of the simulation. Each level of the tree corresponds to an AMR level, and the nodes at the same level of the tree represent the AMR boxes of that level. Finally, the directed edges from coarser level nodes to finer level nodes represent refinement relationships. This polytree index is constructed over the boxes' metadata inside the mservers.

Given this index structure, the many-to-many AMR boxes' refinement relationships are well represented and spatial queries can be answered efficiently. After finding the AMR boxes at a level that overlaps with a given spatial region, the searching at the next finer level can be limited to the boxes which are refined from those found boxes at the coarser level. By performing a search in this manner, AMRZone avoids inefficiently checking all boxes at a certain level.

Algorithm 2 illustrates this complete depth-first search procedure. The algorithm starts at the coarsest level (level 0) to search for AMR boxes that overlap with the given spatial region (lines 3-7). When an overlapping box is found, a recursive function is invoked (lines 9-15) to check the boxes at the adjacent finer level that are refined from the found box. In the function, because we are dealing with boxes at finer levels, it is necessary to refine the given region by a refinement ratio (line 12) before checking if a box overlaps with the given spatial region. When the mserver finds a box satisfying the criteria, it first instructs the dserver that holds the box's binary data to transfer the data to the client, and then it updates the total number of found boxes (line 5-6 and 13-14). After the entire search concludes, the mserver sends the client the total number of found boxes. This number can be used as the condition variable used to terminate the querying API function.

Algorithm 2: Algorithm of searching the polytree based index to perform spatial query over one time-step AMR data

```
Input: Specified spatial region: R
Input: 1D array of refinement ratio for all levels: REF[]
Input: The built polytree based spatial index: Index
Input: 1D array of boxes at the coarsest level: Boxes0[]
Input: The size of Boxes0[]: N
Result: The metadata of all found AMR boxes
1  $num\_found\_box = 0$
2  /*Search the coarsest level(level 0) first:*/
3  for $i = \{0, ..., N-1\}$ do
4      if $region\_overlap(R, Boxes0[i]) == TRUE$ then
5          $num\_found\_box{+}{+}$
6          $process\_metadata(Boxes0[i])$
7          $spatialQuery(Boxes0[i], 0)$
8  /*Function to search box's(amr_box) refined boxes:*/
9  Procedure spatialQuery(box, lev)
10     $ref\_boxes = get\_refined\_boxes(box, Index)$
11     for $j = \{0, ..., size\_of\_ref\_boxes - 1\}$ do
12         if $region\_overlap(refine\_region(R, REF[lev]),$
               $ref\_boxes[j]) == TRUE$ then
13             $num\_found\_box{+}{+}$
14             $process\_metadata(ref\_boxes[j])$
15             $spatialQuery(ref\_boxes[j], lev+1)$
```

In order to build this polytree based spatial index, the mserver needs to iterate over boxes at each level. The iteration at each level is performed over the boxes lists, which is described in sub-section (III-B). For each box, it must then check all the boxes at the next finer level to see if there is refinement relationship. If so, a pointer to the box at finer level is added. This kind of linear iterations could be inefficient. To speed up this procedure, AMRZone's API can divide a level's index-building workload to multiple sub-tasks, which would be processed by different threads inside the mserver in parallel.

The next major issue is when to build this index, as the boxes generated by a running application could be sent to the framework in any order. Building the index while boxes are being received could introduce significant overhead, because it requires frequently checking the relationships of received boxes. Instead, after a box is received, an mserver only chains it to the boxes list at its corresponding level, as described in the above sub-section(III-B).

A client API function must be explicitly invoked to request the mserver to build the index. In this way, clients are given the freedom of choosing when to build the index. The client applications that use the API to transport data to the staging space usually have control over how to send the data and know when the transportation finishes. Therefore, the applications can schedule the data-write and index-build tasks in a disjoint manner to avoid unnecessary overhead. This practice is common in many data management fields, such as a DBMS where, after using SQL insertion statements to write data into the database, users can execute some index-build procedures to build a more complicated index over the data.

D. Implementation

To construct a prototype for AMRZone, one of the most important pieces is to implement a data transportation layer. We only have a few technical choices, such as TCP/IP based programming APIs, network native APIs, and MPI [20].

TCP/IP based APIs are not designed for high performance computing. Libraries that are based on network native APIs, such as DART [15] which is used to build DataSpaces, are not very portable. For example, DART uses complicated network native programming APIs to implement data transmission. In other words, there is a different implementation for the different types of network connections (e.g., InfiniBand [24], Gemini [8], etc.). As a result, it currently doesn't support newer high performance computing systems, such as Edison [18] at the Lawrence Berkeley National Laboratory (LBNL). Moreover, there may be a long transition period before it can be ported to next generation supercomputers, namely Summit [28] at the Oak Ridge National Laboratory (ORNL) and Cori [10] at LBNL. To the best of our knowledge, work that uses a network native API to perform data transmission face this portability issue, more or less.

Thus, to develop a portable prototype of AMRZone, we use MPI to implement the server processes and data transportation. Pthread [27] is used to implement the thread pool inside mservers. We leverage pthread-based mutex and reader/writer locks to protect the finely partitioned data structures to manage the metadata inside the mservers and maintain data consistency while achieving a high degree of parallelism. Finally, it is *important* to note that the methods described in the above three sub-sections are independent of any specific data transportation implementation.

IV. Results

Evaluations of AMRZone are driven by the goal to show its high performance compared with the existing state-of-the-art framework as well as its efficiency in sharing AMR data across multiple applications. Towards this end, we compare the scalability of write/read tasks between AMRZone and DataSpaces over uniform mesh data (Section IV-A). In addition, we evaluate AMRZone's performance of write/read actions and spatially constrained accesses over real AMR data (Section IV-B and IV-C, respectively).

The AMRZone prototype implementation is evaluated on Titan [29] at the Oak Ridge National Laboratory (ORNL). Titan is a Cray XK7 machine with a total of 18,688 compute nodes. Each node contains 16-core 2.2GHz AMD Opteron 6274 processors and 32GB memory. A pair of nodes share a Gemini [8] high-speed interconnect router. For each experiment, the total time of all writes/reads (seconds) is reported. Each experiment is repeated at least 15 times, and the run with the smallest write/read time is reported since it has the least influence from outliers with much larger values.

A. Scalability

To demonstrate that our architecture design could handle a large data transmissions with many parallel writers and readers, we compare the weak scalability between AMRZone and DataSpaces. Specifically, we use the officially distributed DataSpaces1.6 (the latest version) source code. The same compilation configuration as the DataSpaces1.6 module on Titan is adopted. Moreover, the code for testing the DataSpaces server program is unmodified and the same as the one used to build the module on Titan. We use the client APIs of both frameworks to develop our own client testing programs.

Since DataSpaces could not handle real AMR data, we use synthetic 3D double-precision uniform data. A time-step's domain is evenly partitioned to a set of sub-regions (or boxes) by four partition sizes respectively, which are assigned evenly to a collection of parallel processes. After launching, the set of parallel processes of a client testing program first write their assigned boxes of a time-step to the server space and then retrieve those written boxes from the space. In each experiment, the write/read operations are performed over 5 time-steps.

The configuration details of the experiments are summarized in Figure 6. Every row represents a sub-set of experiments where a box size is used to partition four different domains. As the domain size increases, so does the total number of boxes and parallel client/server processes. By default, we use the minimum number of Titan nodes to hold the client and server processes. However, because DART [15], on which DataSpaces is built, utilizes remote-direct-memory-access (RDMA) to transport data, and the memory available for RDMA on each Titan node is about 2GB by default, a large box size or high number of parallel processes means more nodes must be used to host the same number of processes, otherwise DataSpaces crashes. Also note that for AMRZone, Figure 6 only shows how many dserver instances are deployed. For all the experiment cases, we consistently use one mserver instance with 15 threads.

Figure 7 shows the results of the experiments. In total, there are 32 comparison scenarios (each one of the 16 cases in Figure 6 includes write/read senarios). In 22 of these scenarios, AMRZone performs better than DataSpaces. In the best case it achieves around a 46% improvement. The min, max, median and average improvement are around 0.9%, 46%, 22% and 24%, respectively. In the other 10 scenarios, AMRZone performs worse. In the worst case, it experiences around a 35% higher execution time. The min, max, median and average reduction are around 0.05%, 35%, 19% and 15%, respectively. Regarding scalability, AMRZone generally achieves a better result (less increased execution time while more compute re-

	16GB(1024*1024*2048) total 80GB	32GB(1024*1024*4096) total 160GB	64GB(2048*2048*2048) 320GB	128GB(2048*2048*4096) 640GB
256*256*256	128B 32C(4N) / 32S(4N)	256B 64C(8N) / 64S(8N)	512B 128C(16N) / 128S(16N)	1024B 256C(32N) / 256S(32N)
128*128*256	512B 128C(8N) / 128S(8N)	1024B 256C(16N) / 256S(16N)	2048B 512C(32N) / 512S(32N)	4096B 1024C(64N) / 1024S(64N)
64*64*256	2048B 512C(32N) / 512S(32N)	4096B 1024C(64N)/ 1024S(64N)	8192B 2048C(128N)/ 2048S(128N)	16384B 4096C(256N)/ 4096S(512N)
32*32*256	8192B 2048C(128N) / 2048S(128N)	16384B 4096C(256N) / 4096S(512N)	32768B 8192C(1024N) / 8192S(2048N)	65536B 16384C(2048N) / 8192S(2048N)

Fig. 6. Configuration details for the scalability comparison experiments. The row header denotes four domain sizes and the column header gives four partition sizes. The format, $B $C($N) / $S($N), denotes the total number of boxes(B), the total number of parallel client processes(C), the total number of client nodes(N), the total number of DataSpaces server or AMRZone dserver processes(S) and the total number of server nodes(N).

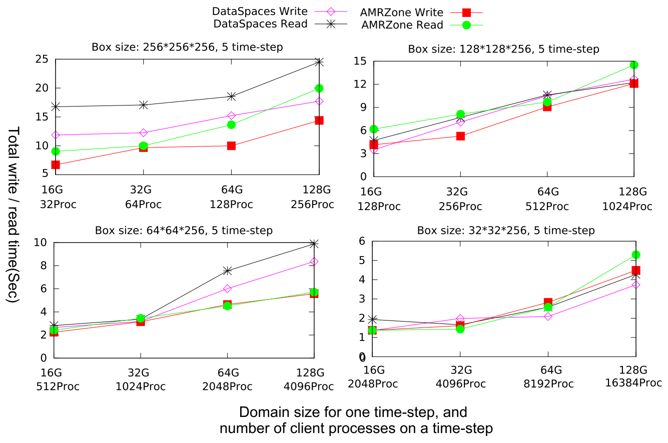

Fig. 7. Results of weak scalability comparison experiments between AMRZone and DataSpaces, over 5 time-steps. AMRZone generally performs better compared to DataSpaces (22 out of 32 cases). In the best case, it could achieve about 46% performance improvement, in the worst case there is about 35% performance reduction.

sources are devoted to process a larger domain size), compared to DataSpaces. Based on these results, we consider on average our framework's performance is comparable with DataSpaces.

B. Performance over AMR Data

Experiments in this section are aimed at evaluating the write/read performance and workload distribution of AMR boxes at the server space of our framework. First, we use the 2D AMR datasets generated by BISICLES [11], a large-scale Antarctic ice sheet modeling code for climate simulation. Then, to have a baseline for comparison, we also include experiments over 2D double-precision synthetic uniform data with similar configurations. The testing programs know the exact coordinates of boxes. Figure 8 gives the detailed information about the experiments over these two datasets.

The BISICLES-generated datasets consist of double-precision values and are 1GB in size. Each dataset has 5 levels, with a total of about 6,700 boxes. To create larger datasets for testing performance, different dataset sizes are created by expanding all boxes of a time-step 8, 16, 32 and 64 times, respectively. Specifically, we expand the size of each box, the total number of boxes in a time-step doesn't change.

BISICLES expanded AMR datasets, each one has about 6700 boxes			
8GB	16GB	32GB	64GB
512C(32N) / 512S(32N)	1024C(64N) / 1024S(64N)	2048C(128N) / 2048S(128N)	4096C(256N) / 4096S(256N)
~6700B / ~1.2MB	~6700B / ~2.4MB	~6700B / ~4.8MB	~6700B / ~9.7MB
Synthetic uniform datasets			
8GB(32768*32768)	16GB(32768*65536)	32GB(65536*65536)	64GB(65536*131072)
512C(32N) / 512S(32N)	1024C(64N) / 1024S(64N)	2048C(128N) / 2048S(128N)	4096C(256N) / 4096S(256N)
8192B / 1MB(256*512)	8192B / 2MB(512*512)	8192B / 4MB(512*1024)	8192B / 8MB(1024*1024)

Fig. 8. Configuration details for two sets of AMRZone experiments, one over expanded BISICLES AMR datasets, one over synthetic uniform datasets. Row 2, 6 give the dataset size(GB) for one time-step. For the synthetic datasets, it also gives global dimension size. Row 3, 7 give the total number of client processes(C), the total number of client nodes(N), the total number of dserver processes(S) and the total number of server nodes(N) for the corresponding time-step. Row 4, 8 give the total number of boxes(B) and box sizes(MB) in the corresponding time-step. For the synthetic datasets, it also gives the dimension size for a box. In row 2 and 4, the values for BISICLES datasets are average ones.

So an original 1GB dataset is expanded 8, 16, 32 and 64 times, respectively. During this procedure, we ensure that the relative position of the boxes to their adjacent levels does not change. In each experiment, 512, 1024, 2048 and 4096 parallel processes (based on the client APIs of AMRZone) are used to write/read 10 time-steps of data (recall, each write/read is based on one AMR box). On the server side, we consistently use 1 mserver process with 15 threads and the minimum number of nodes to host those client and dserver processes.

In each of these AMR data related experiments, the workload assignment policy makes each client process have a similar amount of data to write/read. This means that some processes may be assigned a few big boxes, while others may be assigned more boxes of smaller sizes. Although still not completely balanced, compared to assigning each process a similar number of boxes, this approach could achieve a more balanced workload distribution between client processes, thus improving performance.

It is important to point out that all of these experiments are not weak scaling. First, regarding the experiments over AMR data, although on average the per-process workload remains the same as the time-step domain size and number of processes increase, the actual workload for each process is not the same due to the highly irregular sizes of the AMR boxes. In fact, in one time-step of this BISICLES simulation, the biggest box size is 17 times larger than the smallest one. Second, for the experiments over synthetic data, although the actual overall workload for each process is consistent, the number of boxes and box size for each process are different among the experiments. A higher number of processes with bigger size data chunks could cause noticeably more network transmission overhead. Thus, when reviewing the results of the two sets of experiments, it is more appropriate to compare the two sets to each other, rather than comparing all experiments of a single set together.

Figure 9 shows the results of the experiments. As expected, the performance over AMR data is worse than the performance over the uniform synthetic data. This could be attributed to the unbalanced workload distribution for client processes when writing/reading AMR datasets. In the worst case (reading an 8GB dataset), AMR data related tasks demand more than 36%

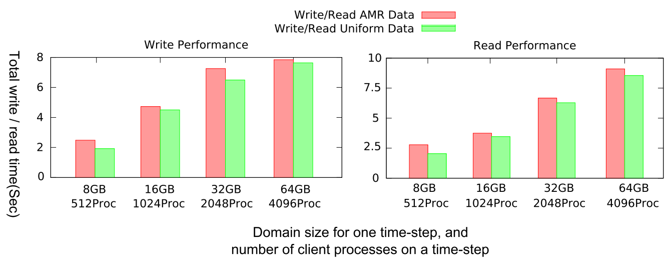

Fig. 9. Results of boxes write/read performance testing for AMRZone over real AMR datasets, with comparisons on synthetic uniform data, totally 10 time-steps. In the worst case, AMR data related task demands more than 36% additional execution time, in the best case it is 2% more. In 5 cases (out of 8), AMR data coupled write/read needs less than 10% more time. Note these are not weak scaling testings.

	8GB 512S	16GB 1024S	32GB 2048S	64GB 4096S
Min	154 MB	151 MB	145 MB	132 MB
Max	173 MB	187 MB	212 MB	300 MB
Avg	157 MB	157 MB	157 MB	157 MB
Median	157 MB	156 MB	156 MB	153 MB
Q1	156 MB	155 MB	153 MB	148 MB
Q3	158 MB	158 MB	159 MB	161 MB

Fig. 10. The statistics for the AMR data workload on dserver processes. The row header denotes the four domain sizes and total number of dserver processes respectively. The column header denotes the minimum, maximum, average, median, first quartile and third quartile for the workload on dservers for an experiment related to each domain size. Note, for each domain size, there are 10 time-steps of AMR data written to the server space.

additional execution time. In the best case (writing a 64GB dataset), they only need about 2% more additional execution time. There are three cases in which AMR data coupled tasks take more than 10% additional time: writing/reading an 8GB dataset (29%, 36%), and writing a 32GB dataset (11%). A possible explaination for the two 8GB AMR data related cases taking a noticeably higher percentage of additional time is that, the box size of the 8GB synthetic data domain is small, making write/read operations very efficient; therefore, the two perform comparatively worse. In all other cases (5 out of 8), AMR data coupled writes/reads require no more than 10% additional time. Considering the unbalanced AMR boxes distribution on the client processes (the biggest box size is 17 times larger than the smallest one), we believe AMRZone's performance over real AMR datasets to be comparable with its performance over uniform mesh, thus satisfactory.

Finally, the statistics for the AMR data workload on the dservers is shown in Figure 10. When the number of dservers is relatively small, the workloads are closer to each other; in contrast, as the number of dservers increases, the gaps between the workload also increase. This trend is expected because workload assignment would become more unbalanced for a higher number of processes. However, the min, avg, median, Q1, and Q3 are quite similar to each other for all dataset sizes, which indicates a good overall balance. Considering that, for the AMR dataset, the biggest box size is 17 times larger than the smallest one, we believe AMRZone's workload assignment policy on the server side produces satisfactory results.

C. Performance of Spatially Constrained Interaction Coupled with AMR Data

In this section, we evaluate the performance of AMRZone under a more complicated data sharing scenario: retrieving the data (or AMR boxes) of specific spatial regions of interest. The datasets are the same collection of BISICLES's 1GB time-steps used in the previous sub-section (Section IV-B). In this dataset, the regions that are covered by boxes at finer levels (for instance, level3 and level4) represent spatial areas of more interest, for example ice sheet grounding lines, calving fronts, and ice streams [11]. Figure 1 shows the visualization of one time-step. At the finest level, there are about 4,100 - 4,300 AMR boxes.

In each of the experiments, we expand all the boxes of a time-step by 0, 2, 4 and 8 times respectively, similar to what is done in Section IV-B. Moreover, we first write 10 time-steps of data to the server space, then use 256, 512, 1024 and 2048 parallel processes (based on the client APIs of AMRZone) to perform spatially constrained data retrievals over the staged time-steps one by one. The processes could represent potential data analytics applications. The coordinates of AMR boxes at the finest level are used by the client processes as the spatial query condition to perform the data retrieval. The boxes assignment policy assigns each process a similar number of spatial regions. On the server side, we consistently use 1 mserver process with 15 threads and a minimum number of nodes to host those client and dserver processes. Since in previous experiments, AMR data write performance has been evaluated, we only record the execution time of data retrieval in this section.

Before using the coordinates of an AMR box at the finest level for a spatial query, the coordinates need to be properly mapped to the domain of the coarsest level, according to the refinement ratios. Recall that, for AMR data spatial queries, boxes at all levels are retrieved rather than a single level(III-C), and more than one box could be refined from a single coarser-level box. So, the total amount of data retrieved may be much larger than the actual size of a time-step. At a single time-step of the 1GB BISICLES datasets, the above designed experiments would retrieve about 26,000 AMR boxes and 13GB data in total. So, for the time-steps that are expanded 2, 4 and 8 times respectively, the final retrieved amount of data is about 26GB, 52GB and 104GB.

To have a point of comparison, we also include AMR boxes read experiments over the BISICLES datasets (here the testing program knows the exact coordinates of each AMR box). We expand the boxes in the 1GB BISICLES datasets 13, 26, 52, 104 times, write them to the staging space and retrieve the boxes (similar to what is performed in IV-B, a read operation is provided the exact coordinates of a box, and the workload assignment policy assigns each process similar amount of data to read). Figure 11 gives detailed information about these experiments.

It is important to point out that neither of the two sets of experiments are weak scaling, because the actual workload for

Spatial constrained data retrieval over expanded BISICLES datasets			
~13GB	~26GB	~52GB	~104GB
256C(16N) / 256S(16N)	512C(32N) / 512S(32N)	1024C(64N) / 1024S(64N)	2048C(128N) / 2048S(128N)
~26,000Boxes / 0.5MB	~26,000Boxes / 1MB	~26,000Boxes / 2MB	~26,000Boxes / 4MB
Boxes retrieval over expanded BISICLES datasets			
~13GB	~26GB	~52GB	~104GB
256C(16N) / 256S(16N)	512C(32N) / 512S(32N)	1024C(64N) / 1024S(64N)	2048C(128N) / 2048S(128N)
~6,700Boxes / 2MB	~6,700Boxes / 4MB	~6,700Boxes / 8MB	~6,700Boxes / 16MB

Fig. 11. Configuration details for two sets of AMRZone experiments over expanded BISICLES datasets, one for spatial constrained data retrieval, one for AMR boxes retrieval. Row 2, 6 give the amount data(GB) retrieved for a time-step. Row 3, 7 give the total number of client processes(C), the total number of client nodes(N), the total number of dserver processes(S) and the total number of server nodes(N) for the corresponding time-step. Row 4, 8 give generally the total number of boxes(Boxes) and box sizes(MB) in the retrieved data for a time-step. The values in row 2, 4, 6 and 8 are average ones.

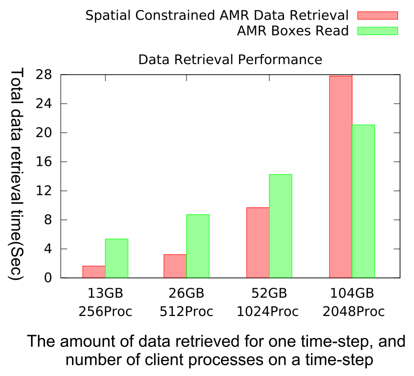

Fig. 12. Results of spatially constrained data retrieval performance testing for AMRZone over AMR datasests, with comparisons of AMR boxes read, totally 10 time-steps. Note these are not weak scaling testings.

each process doesn't remain the same while the dataset size and number of processes increase. Thus, when reviewing the results of the two sets of experiments, it is more appropriate to compare the two sets to each other, rather than comparing all experiments of a single set together.

Figure 12 shows the results. For the 13GB, 26GB and 52GB cases, the spatially constrained data retrieval use about 69%, 63%, and 32% less execution time compared to reading the AMR boxes. For the 104GB cases, spatially constrained data retrieval takes about 31% more execution time. An important fact which should be considered before explaining the results is that, the spatial access retrieves about 4 times more boxes than the boxes read (as described earlier). However, when the average box size is relatively small, transmitting an individual box is so efficient that even 4 times more transmissions could still be fast. In addition, a relatively smaller number of processes helps to achieve a more balanced workload distribution. Therefore, in the first three cases, the spatial queries have a better performance than reading the finest-level AMR boxes.

However, with an increasing average box size, 4 times more data transportations cause significant network overhead. Worse, more processes lead to a more unbalanced workload distribution, further compromising performance. In terms of the amount of data (MB) one process retrieves for one time-step spatial access, the ratio of maximum and minimum for the 2048 processes case is 143:37. So for the last case, spatial access endures a noticeable performance downgrade. However, considering the above factors, we believe the spatial AMR data retrieval performance of AMRZone is satisfactory overall.

Finally, it takes about 0.8 seconds for the mserver to build the polytree based spatial index for the 10 time-step data in these experiments. In fact, because expanding the boxes does not impact the box numbers and relative locations at each level of a time-step, whether the boxes are expanded or not does not influence the efficiency of the index construction. Considering the index is built once and read many times, we believe the result is satisfactory.

V. Related Works

In-situ and in-transit data analytics are widely used to avoid the high overhead related to file system I/O. In-transit refers to the approach of moving data from the compute nodes on which a simulation is running to a virtual in-memory space that is constructed by another collection of nodes, and performing various analytics tasks over the space. In-situ means the analytics tasks share the same compute resource as the running simulation. The term "analytics" can denote actions like writing data to storage, feature extraction, indexing, compression, transformation, visualization, etc [30]. Towards supporting these complicated tasks, a significant amount of researches have been conducted.

Works which do not involve file system I/O usually study how to efficiently move data among nodes and provide various functions to facilitate analytics tasks. EVPath [19] enables framework users to setup dataflows (or paths) among compute nodes through which fully-typed data (or events) can flow with assigned operators, filers, or routing logic. GLEAN [7] makes data movement topologically-aware and provides functionalities like data subfiling and compression. DataSpaces [16] builds a space-filling curve [23] based index over data in the virtual space, and provides efficient access functions to enable live data of any spatial region can be written to or read from the space. These important features of DataSpaces greatly facilitate runtime data sharing across applications, compared to manually implementing these complex communication behaviors by low level programming standards, such as MPI. Those data sharing scenarios typically consist of multiple heterogeneous and coupled simulation processes dynamically exchanging data on-the-fly [16].

Some other related researches are coupled with file system I/O and usually based on the ones that only focus on in-memory data management. Besides inheriting EVPath's data transportation, extending its functions and enhancing performance, FlexPath [13] is integrated into ADIOS [22] as a 'transport method'. Adopting the data transportation and manipulation methods of EVPath, DataStager [1] provides a phase-aware congestion avoidance data movement scheduler and compatible interface with ADIOS. PreDatA [31] is also based on EVPath and provides a few pluggable data analytics functions, such as sorting, plotting, and reducing/integrating with

ADIOS. The method in [3] combines DataSpaces, ADIOS, and other in-situ techniques to speed up scientific analysis tasks. Based on DataSpaces, ActiveSpaces [14] supports defining and executing data processing routines in the space. SDS [17] provides efficient scientific data management and query as services.

All above works are for uniform mesh data. Moreover, only DataSpaces can build an explicit online data index and provide a public data access API, which are indispensable features for supporting runtime data sharing across applications. To the best of our knowledge, runtime data sharing across AMR capable applications has not been studied before.

VI. Conclusion

In this paper, we first identify three major challenges for developing a framework to facilitate AMR data sharing across multiple scientific applications. We then propose a framework, AMRZone, which addresses these challenges. Besides addressing these issues, AMRZone's performance and scalability are even comparable with the existing state-of-the-art framework when tested over uniform mesh.

VII. Acknowledgment

We would like to thank the National Energy Research Scientific Computing Center and Oak Ridge National Laboratory for providing the computing resources. Oak Ridge National Laboratory is managed by UTBattelle for the LLC U.S. D.O.E. under Contract DE-AC05-00OR22725. Support for this work is provided by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs). Work at the Lawrence Berkeley National Laboratory 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.

References

[1] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[2] R. Bayer. *The universal B-Tree for multidimensional Indexing*. Mathematisches Institut und Institut für Informatik der Technischen Universität München, 1996.

[3] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, et al. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *SC*, pages 1–9. IEEE, 2012.

[4] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[5] M. J. Berger and A. Jameson. Automatic adaptive grid refinement for the euler equations. *AIAA journal*, 23(4):561–568, 1985.

[6] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512, 1984.

[7] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms. Scalable parallel i/o on a blue gene/q supercomputer using compression, topology-aware data aggregation, and subfiling. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 107–111. IEEE, 2014.

[8] R. D. Chamberlain, M. Franklin, C. S. Baw, et al. Gemini: An optical interconnection network for parallel processing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(10):1038–1055, 2002.

[9] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen. Chombo software package for amr applications-design document, 2000.

[10] Cori at the lawrence berkeley national laboratory. http://www.nersc.gov/users/computational-systems/cori/.

[11] S. L. Cornford, D. F. Martin, D. T. Graves, D. F. Ranken, A. M. Le Brocq, R. M. Gladstone, A. J. Payne, E. G. Ng, and W. H. Lipscomb. Adaptive mesh, finite volume modeling of marine ice sheets. *Journal of Computational Physics*, 232(1):529–549, 2013.

[12] S. Dasgupta. Learning polytrees. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 134–141. Morgan Kaufmann Publishers Inc., 1999.

[13] 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*, pages 246–255. IEEE, 2014.

[14] C. Docan, M. Parashar, J. Cummings, and S. Klasky. Moving the code to the data-dynamic code deployment using activespaces. In *IPDPS*, pages 758–769. IEEE, 2011.

[15] C. Docan, M. Parashar, and S. Klasky. Dart: a substrate for high speed asynchronous data io. In *HPDC*, pages 219–220. ACM, 2008.

[16] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. In *HPDC*, pages 25–36. ACM, 2010.

[17] B. Dong, S. Byna, and K. Wu. Parallel query evaluation as a scientific data service. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 194–202. IEEE, 2014.

[18] Edison at the lawrence berkeley national laboratory. http://www.nersc.gov/systems/edison-cray-xc30/.

[19] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan. Event-based systems: opportunities and challenges at exascale. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 2. ACM, 2009.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[21] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[22] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, et al. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[23] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124–141, 2001.

[24] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[25] H. Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[26] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics (TOG)*, 4(3):182–222, 1985.

[27] W. R. Stevens and S. A. Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

[28] Summit at the oak ridge national laboratory. https://www.olcf.ornl.gov/summit/.

[29] Titan at the oak ridge national laboratory. https://www.olcf.ornl.gov/support/system-user-guides/titan-user-guide/.

[30] F. Zheng, H. Abbasi, J. Cao, J. Dayal, K. Schwan, M. Wolf, S. Klasky, and N. Podhorszki. In-situ i/o processing: A case for location flexibility. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 37–42. ACM, 2011.

[31] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. Predata–preparatory data analytics on peta-scale machines. In *IPDPS*, pages 1–12. IEEE, 2010.

[32] X. Zou, K. Wu, D. A. B. II, D. F. Martin, S. Byna, H. Tang, K. Bansal, T. J. Ligocki, H. Johansen, and N. F. Samatova. Parallel in situ detection of connected components in adaptive mesh refinement data.