Exploring Memory Hierarchy and Network Topology for Runtime AMR Data Sharing Across Scientific Applications

Wenzhao Zhang[1,3], **Houjun Tang**[2], **Stephen Ranshous**[1,3], **Surendra Byna**[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*—Runtime data sharing across applications is of great importance for avoiding high I/O overhead for scientific data analytics. Sharing data on a staging space running on a set of dedicated compute nodes is faster than writing data to a slow disk-based parallel file system (PFS) and then reading it back for post-processing. Originally, the staging space has been purely based on main memory (DRAM), and thus was several orders of magnitude faster than the PFS approach. However, storing all the data produced by large-scale simulations on DRAM is impractical. Moving data from memory to SSD-based burst buffers is a potential approach to address this issue. However, SSDs are about one order of magnitude slower than DRAM. To optimize data access performance over the staging space, methods such as prefetching data from SSDs according to detected spatial access patterns and distributing data across the network topology have been explored. Although these methods work well for uniform mesh data, which they were designed for, they are not well suited for adaptive mesh refinement (AMR) data.**

Two major issues must be addressed before constructing such a memory hierarchy and topology-aware runtime AMR data sharing framework: (1) spatial access pattern detection and prefetching for AMR data; (2) AMR data distribution across the network topology at runtime. We propose a framework that addresses these challenges and demonstrate its effectiveness with extensive experiments on AMR data. Our results show the framework's spatial access pattern detection and prefetching methods demonstrate about 26% performance improvement for client analytical processes. Moreover, the framework's topology-aware data placement can improve overall data access performance by up to 18%.

I. INTRODUCTION

Runtime data sharing is an effective approach for avoiding the high I/O latency incurred by post-processing methods [12], [21]. The principle idea of runtime data sharing is to assemble a dynamic-random-access-memory (DRAM) based *staging space* on a set of dedicated compute nodes. Client processes, which run over another set of nodes, can be simulations, producing data and writing to the staging space, or analytical applications, reading data from the staging space [2], [11]. Thus, simulations and analytics can be run concurrently, and accessing data from slow disk-based parallel file system (PFS) is replaced by accessing fast DRAM.

However, the volume of data being generated by scientific applications continues to grow, often exceeding the capacity of DRAM by 100% or more [17], necessitating the use of the PFS, thus increasing access latency. To address this capacity issue, solid state drives (SSDs) have been used as an overflow space for when DRAM fills [17]. SSDs are chosen as they are usually two orders of magnitude faster than hard disk drives, which the PFS uses, and provide more than ten times the capacity of DRAM [15]. Still, as SSDs are about one order of magnitude slower than DRAM [15], optimizations for data access performance over the staging space are desirable.

Currently, there are two independent lines of research for improving data access performance over the staging space. The first approach directly targets the speed gap between SSDs and DRAM. By detecting spatial patterns in the read requests of the clients, the staging space can prefetch data from the SSDs and bring it into DRAM, relieving the access latency of SSDs [17]. The second approach focuses on the topology of the network, aiming to decrease the distance between a requesting client and where the data is stored in the staging space [19]. However, both of these proposed methods were developed for uniform mesh data, and are thus hardly applicable to adaptive mesh refinment (AMR) data [3]–[5], given its multi-level and non-uniform structure, as well as the highly irregular box sizes.

To the best of our knowledge, a framework to support runtime AMR data sharing between scientific applications that employs more than the DRAM portion of the memory hierarchy and accounts for network topology has not been proposed. In the following, we summarize two major issues that should be addressed in order to develop such a framework.

First, similar to uniform mesh data, being able to identify spatial read patterns to facilitate prefetching data from the SSDs into DRAM would be helpful to reconcile the difference in access latency. Unfortunately, due to the multi-level, non-uniform structure of AMR data, the techniques used in [17] are scarcely applicable, requiring new methods be developed. To address this issue, our framework employs an AMR data-aware algorithm to effectively search all AMR levels to generate

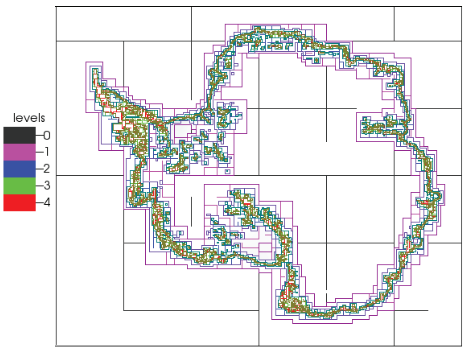

Fig. 1. The visualization graph of a 1GB block-structured AMR dataset generated by BISICLES [8]. BISICLES is a large scale simulation for modeling Antarctic ice-sheets.

spatial regions for prefetching.

Second, in order to further optimize data access performance, the framework needs to distribute data across the staging space properly to reduce network transmission latency *at runtime*. This is not as straightforward as it would be with uniform mesh data [19], as the sizes of AMR boxes are unknown *a priori* and highly irregular when generated [21]. To address this issue, we propose a multivariate cost model, considering AMR box sizes, staging space workload balance, and network topology for selecting the compute nodes for storing data.

In summary, we present a framework to facilitate runtime AMR data sharing between scientific applications. We specifically optimize its data access performance across multiple memory layers and the network topology. When tested over real AMR datasets, our framework's spatial access patterns detection and prefetching methods demonstrate about 26% performance improvement, and its runtime AMR data placement optimization can improve performance by up to 18%. Specifically, our framework makes the following contributions:

- Methods for AMR data spatial access patterns detection and prefeching (Section III-A).
- A model for optimizing runtime AMR data distribution across network topology (Section III-B).

II. Background

A. Block-structured AMR Data

Adaptive mesh refinement (AMR) data has been shown to be an important advancement for scientific applications [3]–[5]. In this paper, we focus on block-structured AMR data, which consists of a collection of disjoint rectangular boxes (or regions) at each refinement level [23]. Figure 1 [21] shows a visualization of a 1GB, 5-level block-structured AMR dataset produced by BISICLES [8], a program for modeling the ice-sheets in the Antarctic for climate simulation.

The first level, level 0, covers the entire domain and is the coarsest. Each higher level is a refinement of the coarser level below it, retaining only some spatial regions of interest, and storing them at a higher resolution. A refinement ratio determines how much finer the resolution is. The amount, sizes, and

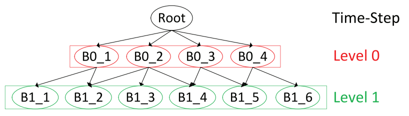

Fig. 2. The polytree-based spatial index for AMR data in AMRZone. Tree nodes correspond to AMR boxes. Directed edges represent the refinement relationship.

locations of the AMR boxes are usually unpredictable before a simulation run, and continually change as the simulation progresses [21]. Moreover, AMR simulations demonstrate several dynamic runtime behaviors [16], specifically largely and heterogeneously changing resource requirements (DRAM, CPU, and network bandwidth).

B. Overview of AMRZone

Research in this paper is based on our previous work AMRZone [21], which is a DRAM-based runtime AMR data sharing framework for scientific applications. AMRZone consists of a client-server architecture. The server is composed of a set of dedicated compute nodes, providing a virtual DRAM-based data staging space. The client is a set of APIs, which can be used by applications to access data in the space.

To facilitate runtime AMR data management, AMRZone consists of two types of server processes: mservers, dedicated for metadata, namely tracing AMR box distribution over the staging space and constructing a spatial index; and dservers, for handling binary data, such as storage and transfering. The mservers coordinate between dservers and clients. For instance, when a client needs to write some data, it first sends a request to an mserver. The mserver then finds a suitable dserver and sends back the dserver's communication address, which the client then connects to for data transmission. Inside an mserver, there is a thread pool to handle request in parallel. This design is proven to have comparable scalability with the state-of-the-art framework in the uniform mesh data domain when tested with up to 16,384 cores [21].

AMRZone achieves an overall balanced workload distribution over the staging space by (i) maintaining workload tables for every staging node and dserver, and (ii) employing an AMR-box-oriented runtime workload distribution policy. To facilitate spatial AMR data retrieval, it builds a polytree-based [9] online index, as illustrated in Figure 2 [21]. In the index, every box not in the highest level keeps a list of pointers to boxes in the next higher resolution level that are refined from it. This index represents the many-to-many relationships of AMR boxes well: a coarser level box can have multiple finer level boxes, and a finer level box can be refined from more than one coarser level boxes.

III. Methods

By greatly extending our previous work, AMRZone [21], this paper addresses the major issues stated in Section I. Since scientific simulations usually have sufficient and consistent computing time periods between generating two time-steps datasets, we would utilize those periods to move data to SSDs

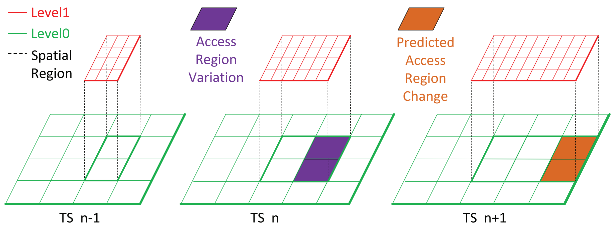

Fig. 3. The illustration of 2D AMR data coupled spatial access patterns detection for a client process. The accessed region over time-step n is compared with the one over time-step n-1. After finding the boundary variation (the right boundary moves one unit toward the right), the framework predicts this change will continue, and it generates a new predicted spatial access region by applying the same trend. This new predicted region is used for prefetching data of time-step n+1.

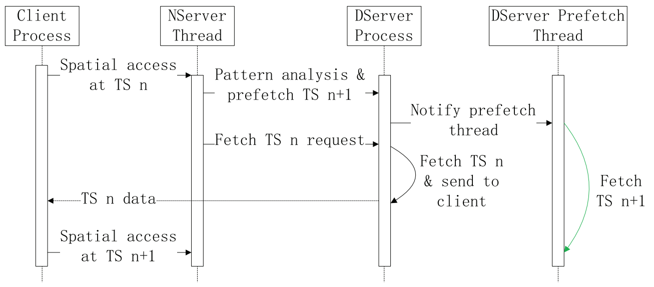

Fig. 4. The timing sequence diagram for prefetching, involving a client process, an mserver process's thread, a dserver processs and the dserver's dedicated prefetching thread. For a dserver, there is a time period between retrieving a time-step's data and receiving the data request of the next time-step, which can be leveraged to do prefetching for the next time-step.

if the DRAM space is insufficient. Thus, in the following sections we only discuss how to optimize data access performance of the the staging space by exploring the memory hierarchy and network topology of supercomputers.

A. Spatial Read Patterns Detection and Prefetching

In this section, we discuss how to detect common spatially constrained AMR data retrieval patterns and perform prefetching. We suppose the data to be accessed has been moved to SSDs, so prefetching it to DRAM can help to bridge the speed gap between DRAM and SSDs.

A spatial access region is represented by a bounding box over the coarsest level of an AMR dataset. The AMR boxes that overlap with the given bounding box are retrieved from *all* levels [21]–[23]. A selection region doesn't have to be aligned with the boundaries of AMR boxes.

The boundaries of an access pattern may change as the analytics progresses over a series of time-steps. However, the trends of the changes are usually predictable, and by catching the trends our framework can perform data prefetching. In this work, trends are identified by tracing each client process's accesses. For example, for a client process's access region over time-step n, our framework will compare the access boundaries with the one over time-step n-1; if there is no change, the same access region will be re-used for prefetching; otherwise, the framework will analyze the boundary variations, generate a new predicted access region by applying the detected boundary changes, and use the updated one for prefetching, as illustrated in Figure 3.

The mservers (server processes that only manage metadata) are responsible for tracing data access history for every client process, pattern detection, and sending prefetching messages to dservers (server processes that only manage binary data). Each dserver has a dedicated thread for prefetching, so the data retrieval for time-step n and the prefetching for time-step n+1 can be performed in parallel. While the dserver is transferring the requested data for time-step n to the client, the prefetched data for time-step n+1 begins to be brought into DRAM. However, as it is quicker to transfer data over the network than read it from SSDs, the dserver may not have all of the next time-step prefetched when the client requests it. As a result, the performance of the dserver with prefetching is bound above by the performance of the dserver if all the data was already in DRAM (please refer to Section IV-A for results). Figure 4 shows the time sequence diagram of prefetching.

Algorithm 1: Algorithm of an mserver searching the spatial index for prefetching data of the next time-step (TS)

```
Input: The predicted access region for prefetching: SR
Input: The array of refinement ratio for all levels: Ref[]
Input: The built spatial index: Idx
Result: Prefetching messages are sent
1  /*Search the coarsest level(level 0) of the next TS:*/
2  for box IN all_coarsest_level_boxes do
3      if box_not_prefetched AND
         $region\_overlap(SR, box) == TRUE$ then
4          record box as prefetched
5          $sendPrefetchMsg(box)$
6          $scPrefetch(box, 0)$
7  /*Function to search a box's refined boxes at the adjacent
   higher level:*/
8  Procedure scPrefetch(cbox, level)
9      $ref\_boxes = retrieve\_refined\_boxes(cbox, Idx)$
10     for box IN ref_boxes do
11         if box_not_prefetched AND
             $region\_overlap(refine\_region(SR, Ref[level]), box) == TRUE$ then
12             record box as prefetched
13             $sendPrefetchMsg(box)$
14             $scPrefetch(box, level + 1)$
```

Next we discuss how to search for AMR boxes which overlap with a predicted spatial access region for prefetching for the next time-step. It is necessary to search the predicted region rather than re-use the boxes' coordinates of the previous time-step, as the coordinates keep changing over time-steps [21]. For the coarsest level (level 0), the framework needs to linearly check each box to see if it overlaps with the given region. For higher level (more refined) boxes, it can take advantages of

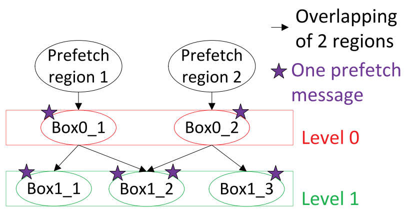

Fig. 5. Illustration of how redundant messages can be sent during prefetching. At level 0, the two spatial regions overlap with box0_1 and box0_2, respectively, and prefetching messages are sent for those two boxes. However, at level 1, box1_2 overlaps with both box0_1 and box0_2, so multiple messages are sent for box1_2.

AMRZone's spatial index. The boxes which are refined from a coarser level box also overlap with the coarser level box spatially [21]. So, after finding a box which overlaps with the given spatial access region, searching the next higher level can be limited to those boxes that are refined from the found one.

Algorithm 1 illustrates the depth-first procedure of how an mserver utilizes the spatial index to search for boxes in a given predicted access region. The algorithm first searches the coarsest level for boxes which overlap with the given spatial region (lines 2-6). After finding an overlapping box, a prefetching message will be sent (line 5) and a recursive function (lines 8-14) will be called to search boxes at the next higher level. In this function, it first retrieves those boxes at the next level which are refined from the found box (line 9). It then iterates over those finer level boxes to check if they overlap with the given region. Before this check, the refinement ratio must be applied to the given prefetch region to enable accurate overlap checks at the current refinement level.

Moreover, before the overlap check, the algorithm first makes sure a box has not already been prefetched; after the checking (lines 3, 11), it records the box as prefetched (lines 4, 12). Otherwise, given the many-to-many relationship of the boxes, redundant prefetching messages may be sent for a single box, as illustrated in Figure 5. Without this check, the performance improvement may be little under high parallelism scenarios, as too many prefetching messages can cause a bottleneck inside an mserver.

B. Runtime Data Placement Optimization over Topology

Our next optimization is network topology-aware data placement, or how to determine which staging node an AMR box should be stored on. Data transmission time is a nontrivial part of the overall data access time. However, the compute nodes allocated for a job can be distributed *physically far away* from each other over the entire topology, which can lead to two cases that incur undesirable network related overhead:

- Communication may have to go through many levels of switches/routers.
- Communication is more likely to be subject to contention from other users' tasks.

Figure 6 shows the throughput difference between nodes of different topology distances (to be defined later) at Cori [7]. As shown, the difference in throughput can be up to 34%

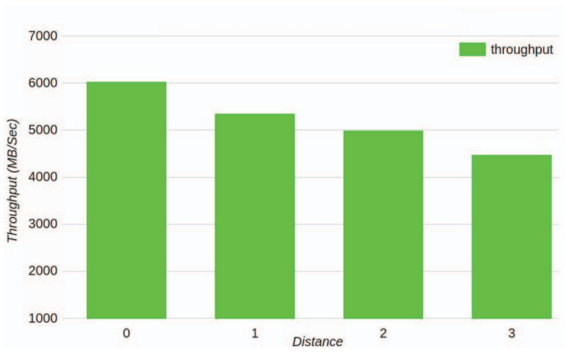

Fig. 6. The throughput difference between nodes of different topology distance on Cori [7] (the testbed for this work). For details of how the topology distances are calculated, please refer to Section III-C. The result is based on a micro benchmark, which sends 1 MB ping-pong messages between a pair of nodes, repeating the process 30,000 times. This benchmark setup (small message size and big number of messages) resembles AMR data, as each AMR box is usually not very large (from a few KBs to a dozen MBs), but the number of boxes in a time-step is high (from several thousands to tens of thousands). At any given time, only one pair of nodes are communicating with each other. Average experiment values are reported.

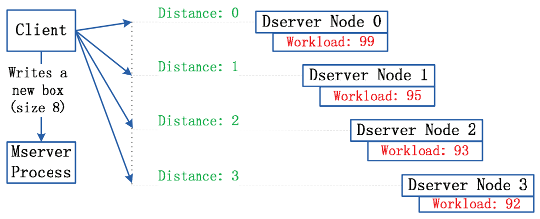

Fig. 7. An illustration of the runtime factors that our framework must consider when determining where to place an AMR box. The mserver must not only consider topology distance, but also the size of each AMR box, and the workload of all staging nodes which keep changing at runtime. For example, in the figure, in terms of topology distance node 0 should be chosen to place the incoming box, but in terms of workload, node 3 should be selected. The final choice should be appropriately balanced among all factors.

between different distances. Thus, matching two processes whose nodes are close together can help to improve data access performance.

For uniform mesh data, runtime topology-aware data placement is relatively straightforward. A time-step's domain is prepartitioned into unifom regions before the data is sent to the space [12], [21]. For each client process, the framework needs to determine a set of server processes which are the closest to the client. Then it maps the client's data to those selected server processes evenly [19].

For AMR data, ideally the framework would find the staging node which has the lowest workload and the shortest topology distance. However, this is usually unrealistic in real world scenarios, as the sizes of AMR boxes are highly irregular and unknown to servers [21], and the workload on each node is not absolutely balanced and is continuously changing during runtime. Hence, our framework must be able to consider multiple runtime factors, such as the size of received AMR boxes, the workload of all staging nodes, as well as the topology, as illustrated in Figure 7. The final placement should be a balanced choice that considers all of these factors.

In order to address this issue, we propose an experiment-based model to select a suitable staging node to place an AMR box on:

$$weight = (threshold - node\_workload) - \alpha * distance * (\frac{box\_size}{node\_avg\_box\_size})$$

The following is a brief explaination of each parameter.

- *weight*: Result of evaluating a node with the given model. The node with the highest weight is chosen to store the incoming box.
- *threshold*: A statistical value of all nodes' current workload, e.g., the first quartile.
- *node_workload*: How much binary data has been stored on a staging node.
- *distance*: Topology distance between a node and the client which sends a box.
- *box_size*: Size of the incoming box.
- *node_avg_box_size*: Average size of all boxes on the node.
- α: System specific variable (Section III-C).

For each incoming box, the model is evaluated for a set of selected nodes to determine the box's placement. The model considers workload balance the highest priority factor, as a large amount of data being imbalancely distributed across a small number of nodes/processes will not only reduce the parallelism of data access but also increase the likelihood of network congestion. Secondary is the topology distance between the client process and the staging node being considered. This helps achieve a balance of distance and workload distribution for staging node selection.

The minuend in the model, $threshold - node\_workload$, is directly related to workload. The *threshold* is first used to filter nodes before model calculation. Specifically, the staging node selection is only performed among the nodes whose workload is smaller than this *threshold*. The $threshold - node\_workload$ term is the *gap* between a node's workload and the threshold value. A higher gap value means lower workload on a node, which indicates this node as a more favorable choice for storing the box in consideration of workload balance. Through this setting, the model treats workload balance as first priority, because it only considers nodes whose current workload is not already high (smaller than the *threshold*), and because when facing a large amount of data this *workload gap* value will be much larger than a topology distance value, exerting a higher influence over the model result (Section III-C).

The subtrahend in the model, $\alpha * distance * (box\_size/node\_avg\_box\_size)$, is related to topology distance, where a smaller value indicates a more favorable choice. It is used to *offset* the influence of workload, i.e. rather than choosing a node that has the lowest workload but is *far away* from the client, it's better to select one whose workload is a little higher but distance to the client is shorter. The $box\_size/node\_avg\_box\_size$ term is used to reduce or increase the influence of the distance. Specifically, if this ratio is less than one then the incoming box is small relative to the others on the node, lessening the effect distance has, as transmission of a small box should be quite efficient. If the ratio is larger than one then the opposite is true, and distance plays a larger role, as the transmission is more likely to be subject to network contention.

Finally, the node with the highest weight is chosen, and the box is placed on the process whose workload is the lowest on the node.

C. Implementation

The implementation of this work is based on our previous work, AMRZone [21]. Based on AMRZone's prototype, we have implemented the performance optimization modules. In this section, we primarily discuss the network topology of one of our testbeds, Cori [7], and how the model described in Section III-B adapts to the topology.

How a topology is configured has great influence over how the topology distance should be calculated for a pair of nodes. For example, the Titan super computer [20] features a 3D torus topology. Every node on the machine is assigned an unique set of 3D coordinates, thus the topology distance between two nodes can be calculated as the distance between 2 points in a 3D space [19].

However, our testbed Cori has a more advanced Dragonfly topology [7], on which a 3D topological distance cannot be applied. In Cori phase I, nodes are physically organized in a 3 level architecture, cabinet-chassis-blade. At the top level, there are 12 cabinets; each cabinet contains 3 chassises; each chassis is composed of 12 blades; each blade has 4 compute nodes. Complicated connections are established between different levels [7], with nodes within the same blade being the most closely connected.

Given the topology, when calculating the distance between a pair of nodes, we only consider the highest level of difference. Specifically, if two nodes are within the same blade, they have a topology distance of 0; if they are only distributed in different blades, the distance is 1; if they are located in different chassises, the distance is 2; if they are in different cabinets, the distance is 3. Subsequent (dis)similarities don't matter, i.e. no matter how two nodes are located in two different cabinets, their distance is 3. This policy has proven to be able to represent significant distance differences in the topology while ignoring the ones which matter little. Figure 6 shows the difference in throughput for the 4 types of distances.

However, there is a special case in Cori. The 12 cabinets of Cori are divided to 6 pairs, 0-1, 2-3, 4-5, 6-7, 8-9, 10-11. Such a pair of cabinets are wired more closely together. Specifically, all the blades within a cabinet pair are also directly connected to the blade in the same position in each chassis of the cabinet pair. So, if a pair of nodes are located in two blades which are connected under to this case, their distance is set as 1.

Since the topology layout is fixed for the life time of Cori, we only need to compute the distance information once and store it in a file. During initialization, our framework just reads the pre-computed information and uses it for runtime data placement optimization.

Finally, as shown above, the topology distance values on Cori are small. If directly applied to our runtime optimization model, they would not effectively influence the outcome. Here is where the α in the model can help, by weighting the distance values appropriately. In our framework, we set α as 3% of the minuend in the model ($threshold - node\_workload$, which represents the workload factor). For example, if there is a pair of nodes whose distance is 3, multiplied by this α, the distance factor can account for approximately 10% of the workload factor. Our experiments show this choice of α produces reasonable results (Section IV-B).

IV. Results

Our experiments are designed to isolate and evaluate each of the proposed framework's new capabilities. Each experiment is run multiple times, and the average result values are reported.

The framework's prototype is primarily evaluated on Cori phase I [7] at the National Energy Research Scientific Computing Center (NERSC). Cori is a Cray XC40 machine with 1,630 compute nodes. Each node contains 32 cores and 128GB DRAM. Note that the actual amount of DRAM available on a node for applications is less than 128GB, due to the operating system and other runtime libraries usage. All nodes are connected via Cray Aries with a Dragonfly topology. Cori has 875TB of SSDs space, which is composed of a collection of dedicated SSDs nodes, which jobs gain access to by declaring their required space in the job submission file. The details of how the data is managed over the SSDs are transparent to a user.

The datasets used all come from BISICLES [8] simulations. Figure 1 shows a visualization of one dataset. Each dataset consists of double-precision values, is about 1GB in size, and is composed of 5 levels with 6,700-7,000 boxes. For the staging space, 1 mserver with 32 threads is used for all experiments. Client testing programs that connects to the staging space are based on our framework's APIs.

A. Spatially Constrained AMR Data Read Patterns Detection and Prefetching

Our first experiments evaluate our framework's effectiveness of spatially constrained AMR data access patterns detection and prefetching. To setup the experiment, we first write 100 1GB time-steps to the staging space, which is composed of 512 dservers.

The spatial regions the clients access are based on 11 major Antarctic ice shelves represented in the BISICLES datasets, as illustrated by Figure 8. Totally 11 client processes are used, each accesses a region. For every 10 time-steps, a spatial region expands 5% in a certain fixed direction, as illustrated in Figure 3.

For the server side of our framework, we include three cases. First, the data is entirely in the staging space (DRAM), which should produce the highest throughput. Second, the data is moved to the SSDs, and the spatial access patterns detection and prefetching are *enabled* in the servers. Third, the data is moved to the SSDs, and the spatial access patterns detection and prefetching are *disabled* in the servers.

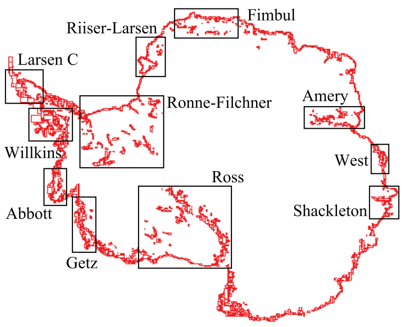

Fig. 8. Illustration of the 11 major Antarctic ice shelves. The spatial regions of those ice shelves are used as spatial access constraints over the BISICLES datasets.

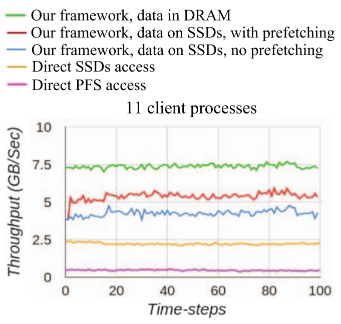

Fig. 9. The effectiveness of our framework performing prefetching for spatially constrained access, compared to direct SSDs and PFS access (not involving our framework's staging space). The spatial access patterns of client processes are based on the 11 major Antarctic ice shelves on a BISICLES dataset, as illustrated in Figure 8. Each client accesses one such region. For our framework with patterns detection and prefetching, the average and median performance improvement are 26.47% and 26.03% respectively.

We also include experiments of direct SSDs and PFS access for baseline comparison. The datasets are placed on SSDs and PFS in HDF5 format [14]. As we are directly accessing the file system, there is no spatial index that we can leverage. When retrieving a box, the program must search all the metadata (all boxes' coordinates) to find which ones overlap with the given spatial region. Once found, the binary data is read back into the testing program's DRAM.

The full results are shown in Figure 9. Read throughput is the highest when all of the data fits entirely into DRAM, which is the upper bound scenario. Pattern detection and prefetching make a noticeable difference in throughput when the data is on the SSDs. Compared to no prefetching, the average throughput is 26.47% higher with a median of 26.03%. Moreover, the throughput improvement is consistent throughout all time-steps, indicating our framework can catch most of the spatial patterns changes and perform prefetching with high precision. However, even with prefetching from the SSDs, the throughput cannot reach the upper bound. This is due to the fact that the time required to transmit the current time-step data from the staging space to the client process is smaller than the time

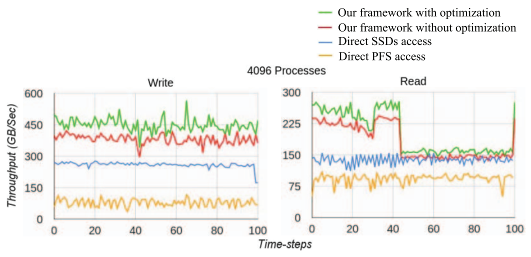

Fig. 10. On Cori, the effectiveness of our framework's topology-aware runtime AMR data placement optimization, compared to direct SSDs and PFS access (not involving our framework's staging space).

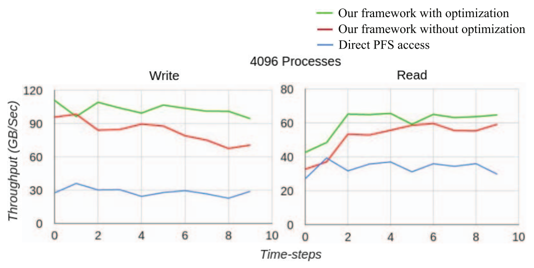

Fig. 11. On Titan, the effectiveness of our framework's topology-aware runtime AMR data placement optimization, compared to direct PFS access (not involving our framework's staging space). Titan does not have SSDs, so no comparison to direct SSDs access is possible.

required to prefetch the predicted data entirely from SSDs to DRAM, as discussed in Section III-A.

When directly accessing the SSDs and PFS, the boxes' binary data retrieval is random, as the program has to search all metadata. For PFS, which is based on spinning hard drives, random data access is very slow, thus the throughput is commensurately low. For SSDs, as it is insensitive to random data access, its performance is about 400% times better than that of PFS, but it is still at least 50% below our framework, because there is no spatial index to utilize.

B. Topology-aware Runtime AMR Data Placement Optimization

We now look at the next optimization, topology-aware data placement. In addition to testing on Cori, we also test this optimization on the Titan supercomputer at Oak Ridge National Lab [20]. Each dataset is expanded by a factor of 256, in addition to scaling the client and dserver processes up to 4096 each. We only expand box sizes, the total number of boxes and relative boxes positions in a dataset don't change.

In this experiment we first write 100 time-steps to the staging space, then read all 100 time-steps back from the staging space. We have an idle time inserted between writing time-steps, mimicking a process's compute time. Experiments are run on server processes both with the optimization and without. We also include experiments of directly accessing SSDs and PFS (not involving our framework's staging space) for comparison. All data in a time-step is used, thus no spatial access patterns are involved.

Figure 10 shows the results on Cori. For writing, the average and median improvement of topology-aware optimization are 18.08% and 18.73%, respectively. The throughput is consistently high, as our program utilizes the compute time period to move data to SSDs. When reading the time-steps back, the throughput for the first 42 time-steps is high, as the data is read from DRAM. During this period the average and median improvements are 12.89% and 12.84%. For the following time-steps, the throughput goes down sharply, as data is read from SSDs (at about time-step 43 the DRAM is becoming full, hence the following time-steps have to be moved onto SSDs). During this period the average improvement was 8.77%, and the median improvement was 8.85%, which are lower than the writing case, because SSDs access latency becomes dominant, thus shadows the effectiveness of topoloy optimization. Overall, the average and median improvements for reading are 10.57% and 10.49%, respectively. For both writing and reading, our framework performs consistently better than directly accessing SSDs or the PFS.

We run a similar set of experiments on Titan at ORNL. The only configuration difference from the ones on Cori is that only 10 time-steps are used, as each Titan node only has 32GB DRAM. As Titan has no SSDs, if too much data is written to the staging space the PFS would have to be used, which would drastically decrease throughput. Figure 11 shows the results. For writing, our topology-aware optimization improves the throughput by an average of 24.85%, and a median of 26.39%. For reading, the average and median improvements are 17.21% and 16.43%. We speculate that the improvements on Titan are better than on Cori as a result of Titan having a less closely connected topology, meaning it is more likely to allocate nodes that are "far away" from each other, increasing the need for a topological optimization. Overall, experiments on both machines prove the effectiveness of our method.

V. Related Work

Runtime data sharing across applications has been widely adopted in an effort to avoid slow PFS access. Processes producing data can write it to a virtual DRAM-based staging space, from which analytical processes can read the data, reducing PFS reads.

An integral component of runtime data sharing is the DRAM staging space, as it is the medium through which data is exchanged. Still, DRAM staging spaces are not unique to runtime data sharing, and can be constructed and configured in different ways for different purposes. Two examples are EVPath [13], for setting up data flow pipelines across compute nodes, and GLEAN [6], for performing topology-aware data processing, such as compression and subfiling. The staging spaces utilized in these methods lack two key features to be suitable for runtime data sharing: a distributed index over the staging space, and an efficient API for reading and writing.

One such framework that incorporates a staging space with these features is DataSpaces [12], for runtime sharing of uniform mesh data. As we have shown in our previous work [21], as DataSpaces was designed for uniform mesh data, applying it to AMR data is not straightforward. This is largely due to its architecture and space-filling curve based index [18]. Using DataSpaces as a foundation, a topology-aware data placement strategy has been proposed [19] to further reduce data access times by better matching staging space nodes and client processes. However, as it is built upon DataSpaces, it also is not directly applicable to AMR data.

As file systems are integrated into the staging space, to act as an overflow for the insufficient space of DRAM, how to move data between the two memory layers efficiently becomes an important question. DataStager [1] acts as a channel between the process generating data and the PFS, helping to hide the access latency of the PFS. FlexPath [10], based on EVPath, introduces using the PFS as part of the data flow process. However, neither of these methods try to optimize the way which the data is stored or retrieved. One approach to optimizing the data movement is to prefetch the data from SSD into DRAM before the client needs it. A pattern detection and prefetching strategy [17] has been implemented on top of DataSpaces, and was shown to considerably improve throughput for uniform mesh data. We propose a method for performing a similar task, except designed to account for the qualities inherent to AMR data instead of uniform mesh data.

All of the above works either are not designed for processing AMR data, or do not support runtime data sharing in their staging space. To the best of our knowledge, memory-hierarchy aware and topology-aware runtime AMR data sharing across scientific applications has not been studied before.

VI. Conclusion

In this paper, in order to develop a memory-hierarchy aware and topology aware framework for runtime AMR data sharing, we first identify two major unaddressed challenges. We then present our framework to address those challenges. Our framework can achieve about 26% performance improvement by spatial access pattern detection and prefetching, and up to 18% performance improvement by topology-aware runtime AMR data placement.

VII. Acknowledgement

This work is supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under contracts DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory and DE-AC05-00OR22725 at Oak Ridge National Laboratory, and by the U.S. National Science Foundation (Expeditions in Com- puting and EAGER program). This research uses resources from the National Energy Research Scientific Computing Center and Oak Ridge Leadership Computing Facility.

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] 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, 2012 International Conference for*, pages 1–9. IEEE, 2012.

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

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

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

[6] 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.

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

[8] 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.

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

[10] 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 *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 246–255. IEEE, 2014.

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

[12] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[13] 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.

[14] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[15] J. He, J. Bennett, and A. Snavely. Dash-io: an empirical study of flash-based io for hpc. In *Proceedings of the 2010 TeraGrid Conference*, page 10. ACM, 2010.

[16] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi. Using cross-layer adaptations for dynamic data management in large scale coupled scientific workflows. In *SC*, page 74. ACM, 2013.

[17] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, et al. Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows. In *IPDPS, 2015 IEEE International*, pages 1033–1042. IEEE, 2015.

[18] 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.

[19] Q. Sun, T. Jin, M. Romanus, H. Bui, F. Zhang, H. Yu, H. Kolla, S. Klasky, J. Chen, and M. Parashar. Adaptive data placement for staging-based coupled scientific workflows. In *SC*, page 65. ACM, 2015.

[20] Titan at the oak ridge national lab. https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/.

[21] W. Zhang, H. Tang, S. Harenberg, S. Byna, X. Zou, D. Devendran, D. F. Martin, K. Wu, B. Dong, S. Klasky, et al. Amrzone: A runtime amr data sharing framework for scientific applications. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 116–125. IEEE, 2016.

[22] X. Zou, D. A. Boyuka II, D. Desai, D. F. Martin, S. Byna, and K. Wu. Amr-aware in situ indexing and scalable querying.

[23] X. Zou, K. Wu, D. A. Boyuka, 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. In *CCGrid, 2015 15th IEEE/ACM International Symposium on*, pages 302–312. IEEE, 2015.