Assembling Portable In-Situ Workflow from Heterogeneous Components using Data Reorganization

Bo Zhang*, Pradeep Subedi†, Philip E Davis*, Francesco Rizzi‡, Keita Teranishi§ and Manish Parashar*
*Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, Utah 84112
†Samsung Semiconductor Inc., San Jose, California, 95134
‡NexGen Analytics, Sheridan, Wyoming 82801
§Sandia National Laboratories, Livermore, California 94551

2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-6654-9956-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCGRID54584.2022.00013

***Abstract*—Heterogeneous computing is becoming common in the HPC world. The fast-changing hardware landscape is pushing programmers and developers to rely on performance-portable programming models to rewrite old and legacy applications and develop new ones. While this approach is suitable for individual applications, outstanding challenges still remain when multiple applications are combined into complex workflows. One critical difficulty is the exchange of data between communicating applications where performance constraints imposed by heterogeneous hardware advantage different data layouts. We attempt to solve this problem by exploring asynchronous data layout conversions for applications requiring different memory access patterns for shared data. We implement the proposed solution within the DataSpaces data staging service, extending it to support heterogeneous application workflows across a broad spectrum of programming models. In addition, we integrate heterogeneous DataSpaces with the Kokkos programming model and propose the Kokkos Staging Space as an extension of the Kokkos data abstraction. This new abstraction enables us to express data on a virtual shared space for multiple Kokkos applications, thus guaranteeing the portability of each application when assembling them into an efficient heterogeneous workflow. We present performance results for the Kokkos Staging Space using a synthetic workflow emulator and three different scenarios representing access frequency and use patterns in shared data. The results show that the Kokkos Staging Space is a superior solution in terms of time-to-solution and scalability compared to existing file-based Kokkos data abstractions for inter-application data exchange.**

I. Introduction

High performance computing has stepped into a heterogeneous era as various computing devices are integrated into new supercomputers. As of June 2021, seven out of the top ten systems on the TOP500 list [1] are built with GPUs. This increases the computing capability of these systems, but also increases the complexity of the applications deployed on them. Scientific simulations are capable of running at higher fidelity by utilizing the computing power of heterogeneous machines [2]. A variety of applications, such as LULESH [3], LAMMPS [4] and GTC-P [5], have been ported to GPUs to improve their capability.

Although these applications obtain great benefits from being ported to new hardware, the cost of refactoring legacy code for new architectures and platforms is high. It requires that application programmers understand the performance characteristics of the target computing platform as well as the application program in detail. Further, assembling heterogeneous simulations and analyses into a scalable in-situ workflow becomes even more challenging than porting individual applications. Either refactoring all the components in the workflow to an identical programming model or designing ad-hoc data transformations between components with mismatching data layouts renders the entire porting process time-consuming and complicated.

Performance portable programming frameworks, such as Kokkos [6] [7] and RAJA [8], are widely adopted as productive solutions to write applications targeted at all major HPC platforms. Applications adapted to the high-level abstractions provided by such frameworks allow users to simply choose a particular execution platform at compile time. While these programming frameworks consider the performance portability of single applications, to the best of our knowledge, none of them have the capability to link multiple heterogeneous applications into a complex workflow using similar performance portable abstractions. Workflow coupling middleware, such as DataSpaces [9] and ADIOS [10] [11], provide application-level data exchange abstractions for efficient code coupling. However, they do not consider the heterogeneity between components and various data representations associated with these components, which is usually represented as the requirement for the same data but in different memory layouts.

Our previous work [12] built a workflow for projection-based reduced-order models (pROMs) [13] in Kokkos. We observed that while using a uniform programming model in a single application is intuitive, coupling multiple components implemented with optimal data layout for underlying hardware requires data reorganization between them. Existing programming interfaces and semantics are inflexible from the workflow-level perspective, which forces the data reorganization to be offloaded to the application implementation. Staging-based coupling tasks are typically I/O-bound, which enables us to utilize compute resources in the staging area for data reorganization. Building on these insights, we explore the effects of several data reorganization methods based on

DOI 10.1109/CCGrid54584.2022.00013

data access patterns and propose a Self-Adaptive Hybrid Reorganization (SAHR) method. We implement this method within the DataSpaces data staging service, and find that we are able to leverage workflow data access characteristics to improve heterogeneous I/O performance.

In addition, we integrate our solution into the Kokkos programming model and propose the Kokkos Staging Space as an extension of the Kokkos data abstraction, improving the portability of individual applications within a heterogeneous workflow by enabling asynchronous data layout conversions. Ultimately, this achieves performance portability of both individual applications and combined workflows of these applications.

The Kokkos Staging Space extends the data copy semantics of the Kokkos *view*, a platform-agnostic representation of a multi-dimensional array, to heterogeneous application workflow settings. Our data exchange API for heterogeneous views requires only one additional parameter at initialization.

Our main contributions can be summarized:

- We explore the trade-offs between three data reorganization methods within the DataSpaces data staging service with respect to the available resources and features of the workflow, then propose a Self-Adaptive Hybrid Reorganization (SAHR) method which reduces resource consumption by collecting data access pattern information.
- We design the Kokkos Staging Space, a prototype of a portable application coupling framework, implemented based on Heterogeneous DataSpaces as an extension of the Kokkos data abstraction, which enables asynchronous data layout conversions for heterogeneous applications.
- We evaluate the Kokkos Staging Space on current leadership computing systems using a synthetic workflow running up to ∼5K cores and demonstrate that it can reduce I/O time by up to 98.7% in comparison to C++ standard I/O and HDF5.

The rest of the paper is organized as follows: Section II provides background and related work. Section III describes the design and implementation details of the data reorganization methods. We then present the architecture and a usage example of the Kokkos Staging Space in Section IV. In Section V, we evaluate the trade-offs between the data reorganization methods and compare the Kokkos Staging Space with two other current methods of inter-application data exchange based on a file used by Kokkos. We present our conclusions and future work in Section VI.

II. Background and Related Work

Coupled workflows still do not benefit from the latest hardware equipped in HPC clusters due to porting difficulties [14]. Often, heterogeneous hardware imposes constraints on which data layouts are performant that are stronger than the constraints on homogeneous systems [15] [16]. This may necessitate data transformations when moving data between execution platforms in order to maintain overall application performance. In addition, heterogeneity is found not only within hardware but also in the software stack. Even if all of

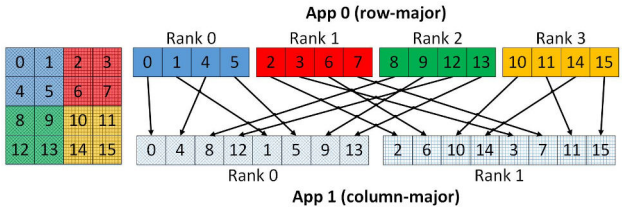


Fig. 1. Data layout mismatch between heterogeneous applications. Left: Application 0 partitions a 4x4 2D array into 4 processors. Application 1 partitions it into 2 processors. Above: Row-major data layout in each processors memory. Below: Column-major data layout in each processor's memory. Arrows show required data movement.

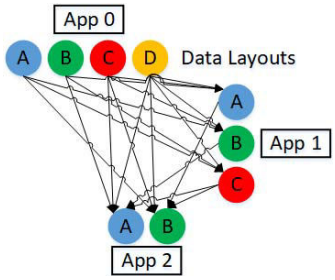


Fig. 2. Complex workflow requires combinatorial numbers of ad-hoc data layout transformations for polymorphic applications.

the applications are running on CPUs, programming languages and underlying libraries exhibit different data layout requirements. Kokkos, as a heterogeneous programming framework, accommodates bindings to include Python and Fortran applications into its ecosystem [17] [18]. Adding these applications to coupled workflows or reusing math kernels further introduces heterogeneity issues. Figure 1 illustrates a simple workflow consisting of two heterogeneous applications with four and two processes respectively. The first application serializes data in row-major format with its programming abstraction while the second requires column-major data to reach peak performance in the example platform. Such reorganizations are usually achieved by ad-hoc transformations after the I/O process, which incurs extra overhead and coding complexity. With every workflow component also targeting several optional layout models, extreme scale workflows make porting a combinatorial problem, as shown in Figure 2. Refactoring all of the components to an identical programming model and implementing ad-hoc data transformations between all components of a workflow makes such a task an insurmountable challenge.

There are several popular heterogeneous programming frameworks targeted at simplifying the application porting process to a specific platform. Kokkos, RAJA, and SYCL [19] provide high-level programming abstractions where users are able to specify a parallel execution policy, manage multi-dimensional data, and execute collective operations in a flexible manner. Applications written in the provided programming abstraction can be configured to support platforms at compile time. A cornerstone of this portability is the multi-dimensional data abstraction, which is optimized for each

architecture to minimize data access penalties. Both Kokkos and RAJA refer the multi-dimensional data abstraction as the *view*, while SYCL calls it a *buffer*. These programming frameworks have also been extended across nodes by applying the MPI+X model to improve performance through node-level parallelism [20] [21]. Instead of focusing on the portability and heterogeneity of only a single application, our work strives to assemble multiple heterogeneous applications into a workflow while maintaining their portability.

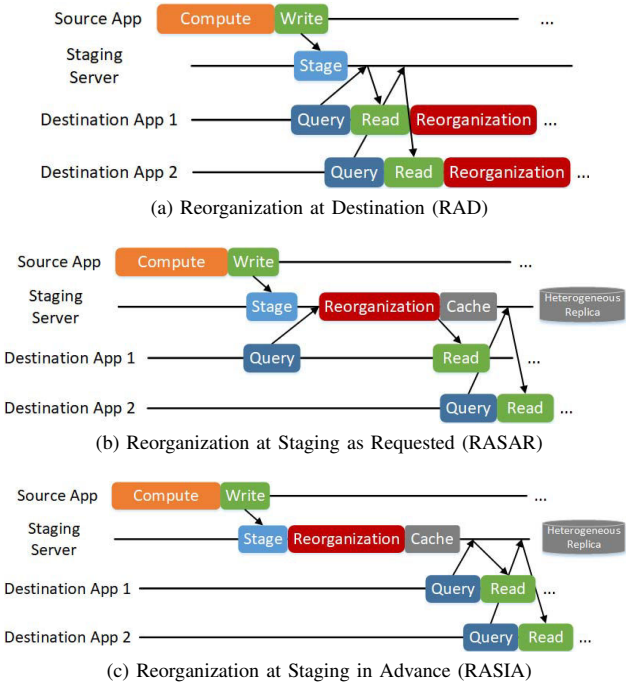
To enhance I/O performance, [22] [23] [24] apply efficient layout reorganization mechanisms to parallel I/O systems by identifying data access patterns. However, they do not consider heterogeneity requirements from a workflow perspective. Apache Arrow [25] defines a language-independent columnar memory format to enable data reorganization for heterogeneity, but it has not been applied to HPC settings. Coupling frameworks for scientific workflows such as DataSpaces [9] and ADIOS [11] define coupling semantics between components in a workflow. Unfortunately, they only operate as a pipeline between applications and offload the data transformation to the workflow components. While these basic coupling semantics together with application-specific data transformations work to create a consistent multi-component workflow, this approach lacks flexibility and is labor-intensive when components can be configured to any given platform at compile time. In contrast, our work integrates data reorganization inside the coupling and adds heterogeneity semantics, enabling flexible coupling between polymorphic components.

III. Heterogeneous Data Reorganization Methods

Data producer applications in extreme-scale in-situ workflows using a staging-based approach are computationally intensive, and therefore extra overhead generated by heterogeneity should not be handled by these applications. We thus propose four data reorganization methods, reorganization at reorganization at destination (RAD), reorganization at staging as requested (RASAR), reorganization at staging in advance (RASIA), and self-adaptive hybrid reorganization (SAHR), and present the details of these four methods.

A. Reorganization at Destination (RAD)

To prepare data for a different layout than its original format, the most straightforward approach is to get the original data and reorganize it right before usage. We implement this approach by adding a generic function for data reorganization inside the data get API call. The data reorganization is triggered after the original data is moved from the DataSpaces server to the destination application/client. This straightforward design has obvious advantages. The embarrassingly parallel data reorganization task displays consistent performance and is easy to scale out with the destination application. The weakness of this approach becomes apparent when workflow is complex. As shown in Figure 3a, if multiple applications request the data the same layout which is different from the original layout, every application has to reorganize the data



(a) Reorganization at Destination (RAD)

(b) Reorganization at Staging as Requested (RASAR)

(c) Reorganization at Staging in Advance (RASIA)

Fig. 3. A schematic illustration of data reorganization methods.

on its own, which is a waste of both computational resources and time to solution.

B. Reorganization at Staging as Requested (RASAR)

Placing the data reorganization in the staging server is another viable solution. Figure 3b illustrates data reorganization requested at the staging server. The first get request for the data in a different layout from the original layout will invoke the reorganization process at the server. A lock mechanism is applied here to avoid repetitive reorganization overheads as well as duplicated heterogeneous replica storage. The first request for a data object in the heterogeneous layout will acquire the lock so that other concurrent requests will be halted until the first one finishes. Then, the staging server will send the reorganized data to the proper destination according to the requests and add the reorganized data object into its storage at almost the same time. Since the staging server keeps the replicated data in the new layout, subsequent get requests for the data in this layout can reuse the existing one, which saves I/O time as well. This design leverages the idle computing resources at the staging server as data staging is an I/O-bounded operation. However, placing reorganization in the middle of the data request and transfer still leads to elongated I/O time for the first request for each data object. If the staging server is extremely limited in scale, server-side reorganization may also consume more time.

C. Reorganization at Staging in Advance (RASIA)

In order to hide the data reorganization overhead and make it transparent to the destination applications, overlapping the reorganization time in the staging server with the processing

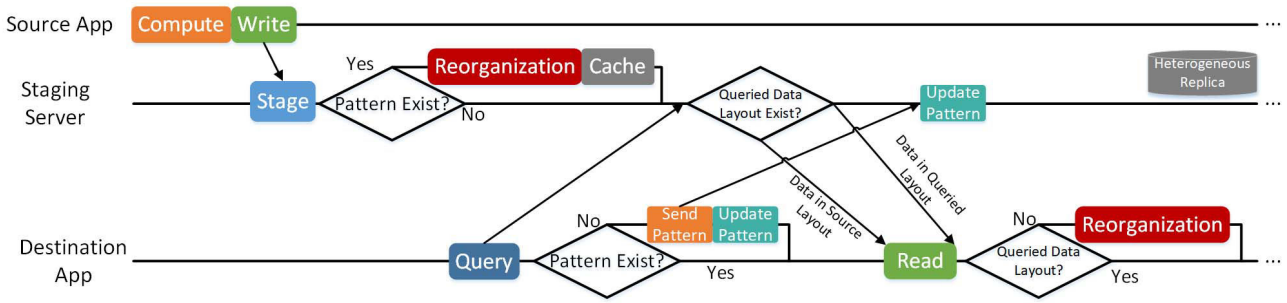


Fig. 4. A schematic illustration of Self-Adaptive Hybrid Reorganization (SAHR) method.

time in the destination applications is essential. We achieve this goal by reorganizing the data to all layouts at the staging server in advance. As shown in Figure 3c, the staging server starts to reorganize the entire data domain immediately after receiving it from the source applications. In this approach, all subsequent get requests do not have to spend extra time on reorganization, since the staging server has a heterogeneous data replica for all possible layouts. However, a major weakness of this design is the huge memory requirement in the staging server to support multiple data layouts for every single data object exchanged in the workflow. This very likely results in waste, if a given workflow does not use all layout types.

D. Self-Adaptive Hybrid Reorganization (SAHR)

Although a dynamic decision between RAD, RASAR, and RASIA based on the workflow features maximizes the advantages of each method, the decision synchronization between servers and clients, as well as server instances, becomes a bottleneck when the scale increases. Thus, we propose a self-adaptive hybrid reorganization method based on two assumptions:

- A particular numerical application is interested in a fixed set of data objects with iterative values.
- The particular numerical application only requests one specific layout for each data object.

This method combines the three methods above by adding a *data access pattern collection module*. The *data access pattern collection module* extracts a pattern for each requested variable from the query by collecting its layout, domain index descriptor(bounding box), and request frequency. While each individual destination application holds a pattern record for itself, the staging server keeps the pattern record for all components in the workflow. As shown in Figure 4, when a data get request is initiated, the module will check local records to determine if the request exhibits a new data object get pattern that should be collected asynchronously and sent to the server. Algorithm 1 describes the details of the pattern update process at both the destination application and the staging server. For those data objects which intersect with existing get patterns from other applications in the workflow, the server will calculate a superset to prepare the data for all applications. When data is put to the server, it will check the get pattern list for the intersected record and transform the data object to the destination layout in advance. The pattern collection module predicts what the applications will require after their first data get request, which makes reorganization at staging efficient and accurate. When a data query is received by the server, it first searches the data objects in the requested layouts. If the requested data object exists, then the client does not reorganize it after the bulk transfer is complete. Otherwise, the server transfers the data object in the source layout and lets the client reorganize it at the destination. In this design, requested data objects are reorganized in advance except for the initial get, and a missing search implicitly indicates that the server is busy. Thus, reorganizations are then placed to destination clients to reduce the overhead load on the server.

Algorithm 1 Get Pattern Update

```
qodsc ←MetaData for the queried data object {Contains varname, bounding box descriptor, version, layout, src_layout, etc..}
query ← Query(qodsc) {Request for a specified data object}
pattern ← ExtractPattern(query)
record_list ←SearchGetPattern(pattern.varname, pattern.layout)
if record_list != NULL then
  for all record in record_list do
    if CheckGridIntersection(pattern.bbox, record.bbox) then
      pattern ←CalculateSuperSet(pattern, record)
    end if
  end for
end if
record_list ←UpdateGetPattern(pattern)
```

IV. Implementation

Our implementation aims to provide a concise coupling tool that can be applied to the extension of any portable programming framework for heterogeneous workflow support. We develop an in-transit mechanism to collect data access patterns and manage data reorganization as well as replication for heterogeneous memory layouts based on the DataSpaces staging service. Specifically, we explore three approaches with different placements for in-transit data reorganization and finally propose a self-adaptive hybrid reorganization(SAHR)

method to automatically adapt with system resource constraints and workflow characteristics. In addition, we integrate our heterogeneous data staging solution with Kokkos ecosystem by supporting inter-application data exchanges between various memory layouts. The resulting Kokkos Staging Space allows us to simply transfer the data between applications in the Kokkos semantics at runtime irrespective to the data layout and the underlying memory subsystems of the individual application instances.

A schematic overview of the Kokkos Staging Space is presented in Figure 5. It is built upon the DataSpaces framework and directly leverages the existing components by reusing its data transport, indexing, and querying capabilities. The DataSpaces client APIs are seamlessly integrated with the Kokkos core library to support `Kokkos::Staging` APIs. The key components of the Kokkos Staging Space include the Access Pattern Collection module and the Data Reorganization module inside Heterogeneous DataSpaces as well as the `Kokkos::Staging` Interface on the top. These modules cooperate to facilitate data movement and sharing across heterogeneous HPC workflow applications.

A. Heterogeneous DataSpaces

Heterogeneous DataSpaces is extended from existing solutions by adding an Access Pattern Collection module and a Data Reorganization module. The Access Pattern Collection module is responsible for new data access pattern recognition and record as detailed in Section III-D. The Data Reorganization module accommodates a unified data layout management abstraction for the data staging movement. Specifically, it implements general-purpose transposition algorithms for arbitrary data structure but provides a plugin interface for third-party algorithms as well. It is also responsible for managing the supported data layout and scheduling the data reorganization operations by cooperating with the Access Pattern Collection module. In a complex workflow, the required data layouts of multiple applications can be varied. If such a workflow scales out, the complexity of data access requests would be a Cartesian product of the number of layout types and the number of data objects. When thousands or even millions of asynchronous heterogeneous data requests flood in, concurrency also becomes a major concern if the data reorganization operations are performed at the staging server. To overcome this problem, we have integrated concurrency and heterogeneous replica controls in the Data Reorganization module. When the staging server receives data access requests for heterogeneity, the server only launches a data reorganization for the first request and keeps a replica with the reorganized layout for the subsequent requests so that the particular layout is served through the replica.

B. Kokkos::Staging Interface

For the purpose of making our new data staging capability compliant to other memory spaces in Kokkos, we wrap DataSpaces client operations with a new namespace `Kokkos::Staging`. Figure 6 presents a sample code that

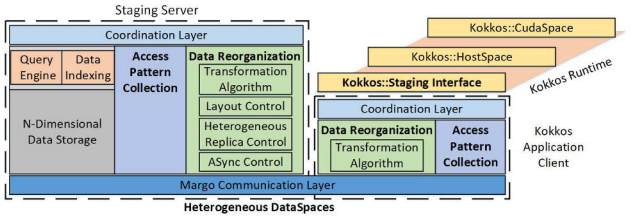


Fig. 5. Architecture of Kokkos Staging Space. The data reorganization module and Kokkos::Staging API were implemented on top of the DataSpaces and Kokkos framework respectively.

exchanges data between two Kokkos applications whose views are in different layouts. To use `Kokkos::Staging` functionalities, an initialization call is required. This call is responsible for initializing an internal DataSpaces client, assuming that Kokkos initialization call has been made. At the end of each program but before Kokkos finalizes, `Kokkos::Staging` needs to be called in order to release the resource binding to the DataSpaces server. After the initialization, users can declare a `Kokkos::Staging` view similar to what they are supposed to do with other Kokkos memory spaces [6]. The layout of `Kokkos::Staging` view should be explicitly declared for heterogeneity, otherwise, it would use the default layout of the host space. In accordance with the data copy semantics in Kokkos memory spaces, a deep copy between `Kokkos::Staging` view and other memory space views is served by zero-copy non-blocking put/get operations for Kokkos applications to transfer data to/from the staging server. However, setting the version and bounding box of the variable before the actual data transfer is optional but strongly suggested. When reader applications request the data with a different layout, an extra line is needed to declare the heterogeneity, as shown in line 26 in Figure 6. An implicit data reorganization will be performed in `deep_copy` function after this explicit call for user awareness.

With these fundamental APIs, users can exchange data between heterogeneous applications. Coupled applications are expected to be aware of the variable name and local bounding box of the data. They can then simply call `deep_copy()` to enable data exchange between the coupled applications. Users are free to implement complex functions by encapsulating these basic operations.

V. Evaluation

We test our heterogeneous data reorganization methods using a synthetic workflow emulator, which simulates various data read patterns. We performed these experiments on the Frontera system [26] at the Texas Advanced Computing Center (TACC). Frontera hosts 8368 compute nodes, each containing a Dual Intel Xeon Platinum 8280 (“Cascade Lake”) 28-core processor with 192GB of DDR4 RAM and 240GB SSD. All of the test runs in subsequent sections have been executed 3 times and the average result is reported.

To better understand the impact of the read access rate and layout matching between a source and a destination on workflow performance, we select two scenarios similar to

```
Kokkos::Staging::initialize();
{
using ViewHost_lr_t    = Kokkos::View<Data_t**,
      Kokkos::LayoutRight, Kokkos::HostSpace>;
using ViewHost_ll_t    = Kokkos::View<Data_t**,
      Kokkos::LayoutLeft, Kokkos::HostSpace>;
using ViewStaging_lr_t = Kokkos::View<Data_t**,
      Kokkos::LayoutRight, Kokkos::StagingSpace>;
using ViewStaging_ll_t = Kokkos::View<Data_t**,
      Kokkos::LayoutLeft, Kokkos::StagingSpace>;
ViewHost_lr_t v_P("PutView", i0, i1);
ViewStaging_lr_t v_S_lr("StagingView_LayoutRight",
                        i0, i1);
ViewStaging_ll_t v_S_ll("StagingView_LayoutLeft",
                        i0, i1);
ViewHost_ll_t v_G("GetView", i0, i1);
// global domain geometric descriptor
Kokkos::Staging::set_lower_bound(v_S_lr, lb0, lb1);
Kokkos::Staging::set_upper_bound(v_S_lr, lb0, lb1);
Kokkos::Staging::set_lower_bound(v_S_ll, lb0, lb1);
Kokkos::Staging::set_upper_bound(v_S_ll, lb0, lb1);
// global iteration
Kokkos::Staging::set_version(v_S_lr, version);
Kokkos::Staging::set_version(v_S_ll, version);
// bind two staging views in different layout
Kokkos::Staging::view_bind_layout(v_S_ll, v_S_lr);
// from host to staging
Kokkos::deep_copy(v_S_lr, v_P);
// from staging to host
Kokkos::deep_copy(v_G, v_S_ll);
}
Kokkos::Staging::finalize();
```

Fig. 6. Code example of data exchange between Kokkos views in different layouts

[27]: reading the entire data domain and reading a subset data domain for all time steps. In both scenarios, the coupled scientific applications are modeled to read/write data to/from a 3-D data domain. The data is modeled as writes over multiple iterations or time steps in a fixed layout, and reads in a similar temporal manner but with heterogeneous layouts.

In our synthetic tests, two application codes, namely readers and writers, are used to emulate generic end-to-end data movement behaviors in real coupled simulation workflows. As their names suggest, writers produce simulation data and write it to the staging servers and readers read the data from staging servers and then perform some analysis. In all of the test cases, one writer application writes the data for the entire domain in a fixed layout over all of the simulation time steps into the staging servers. One reader application also reads the data, using either the same layout or a different layout as needed, but varies the data access pattern. To demonstrate reorganized data reuse, a second reader, which shares the same read pattern as the first one, is added when the reader requests the data in a layout different from that of writer.

A. Exploring the task placement of data reorganization

This experiment evaluates the impact on I/O performance in the scalable in-transit workflow of the four data reorganization methods introduced in Section III. To better understand the trade-offs between these approaches, three critical workflow metrics are selected according to [28]. Table I details the base setup for all tests cases in this experiment. The second reader starts after the first reader finishes data reads in this experiment to eliminate interference between asynchronous reader applications.

TABLE I
EXPERIMENTAL SETUP CONFIGURATIONS FOR SYNTHETIC EXPERIMENTS

Data Domain	1024 × 1024 × 1024
No. of Parallel Writer Cores (Nodes)	512(16)
No. of Parallel Reader Cores (Nodes)	64(4)
No. of 2nd Parallel Reader Cores (Nodes)	64(4)
No. of Staging Cores (Nodes)	32(8)
Total Staged Data Size (15 Time-steps)	120 GB

1) Metric 1 - Cycle time of writer and reader: Because our synthetic writers and readers use a simplified data generator, both of the applications have a relatively fast cycle time. While this fast cycle time may accurately represent some applications, applications with longer cycle times should be studied as well. To simulate a longer cycle time, a pause is added to our synthetic writers and readers after the completion of computation but before the data movement in each cycle. The four simulated cases were:

- **Delay(0):** writers and readers ran with no sleep command.
- **Delay(5):** writers and readers ran with 5 seconds of sleep after each computation time step.
- **Delay(10):** writers and readers ran with 10 seconds of sleep after each computation time step.
- **Delay(20):** writers and readers ran with 20 seconds of sleep after each computation time step.

Figure 7 shows the I/O time per time step for each data reorganization method with varying application cycle time. We see that longer cycle times benefit data reorganization at the staging server. RAD, where data reorganization takes place at the reader side, cannot take advantage of any latency hiding and keeps a steady I/O time regardless of the delay time. Although the data reorganization in RASAR happens at the staging server at the start of receiving get requests, it benefits from the longer cycle time, because less frequent data get requests leave enough time for the data reorganization at the staging server, avoiding a cascading slowdown in the following requests. RASIA and SAHR obtain the most advantage from the longer cycles, with a speedup of up to 96% and 78% compared to RAD and RASAR respectively in *Delay(20)*, because the data reorganization overhead can be hidden by the staging server and the readers can proceed in parallel. However, in *Delay(0)*, the reorganization in the server means the staging servers are computationally bound, which causes both the read and write times of RASAR, RASIA, and SAHR to increase. For the second reader, RASAR, RASIA, and SAHR have nearly identical read times due to directly hitting the reorganized replica at the staging server, while RAD still has to transform every data object. This "cache hit" saves up to 95% read time.

2) Metric 2 - Staging server scale: Placing the data reorganization task at the staging server has a great potential to make the staging operations computationally bounded. Consequently, the response rate to incoming I/O requests would slow down, which might further affect the entire workflow

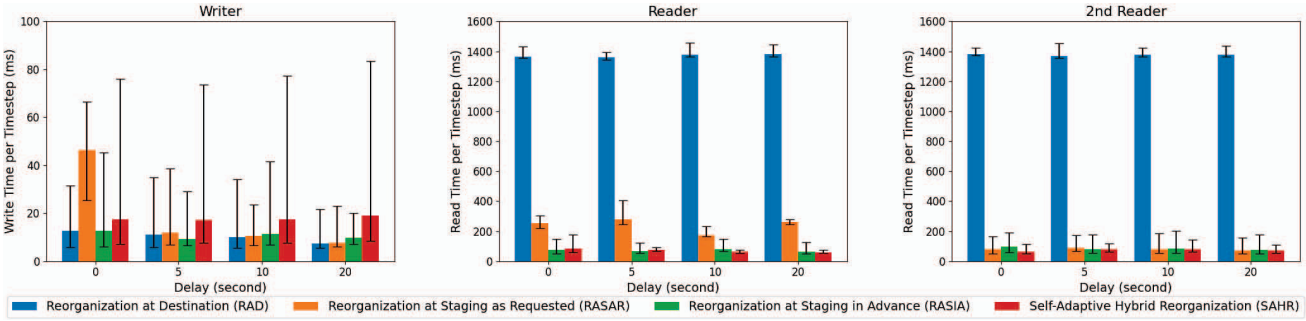


Fig. 7. Comparison of I/O time per time step among four data reorganization methods with varying cycle time of applications

adversely. For an in-transit paradigm, the scale of simulation and analysis is predetermined. Thus, we change the number of the staging server cores to explore how the staging server scale impacts the performance. The three server configurations with the scale of 512(16) writer cores(nodes) and 64(4) reader cores(nodes) used were:

- **8(2):** staging server ran with 8(2) cores(nodes).
- **16(4):** staging server ran with 16(4) cores(nodes).
- **32(8):** staging server ran with 32(8) cores(nodes).

Figure 8 demonstrates the I/O time per time step among the four data reorganization methods with different staging server sizes. Since the first reader starts with the writer at the same time, the read time of the first time step is extremely long because the reader must wait for the data generated and transferred to the staging server. The statistic in the first time step is always excluded due to its uncertainty. Several insights can be drawn from Figure 8. RASAR, RASIA, and SAHR are subject to a dramatic I/O time increase as the server:reader ratio decreases while RAD keeps constant performance because the limited server resources cannot afford the computational overhead incurred by reorganization. Such a heavy workload can even lead to performance degradation for the writers since the write call is blocking and the response time to the write requests increases. For the first reader, RASAR is sensitive to the server scale changes because of no time overlapping. It performs the worst in the *server:reader ratio of 8:64* case while RASIA and SAHR have relatively the same performance as RAD on average. The read time of RASIA and SAHR witness an extraordinary slowdown in the beginning, but finally converge to the read time of the second readers who completely reuse the reorganized replicas at the server, which is due to the I/O behavior variation in the asynchronous workflow we were running as time step increases. From the second time step, RASIA and SAHR must wait for the reorganization, which happens immediately after the staging server receives the data from the writer, to be completed. As the writer stops putting data to the staging server and the server finishes the reorganization in advance, they start to benefit from reorganized replicas at the server. The read time of them becomes shorter than the other two from time step 7 and converges from time step 9. This also explains the large span of the error bar of RASIA and SAHR in Figure 8. As for the second reader, because the first reader did not cache the heterogeneous replica at the server in the initial time step, SAHR always uses the RAD method and thus spends more time in the first time step than other methods that use cache.

3) Metric 3 - Data size of reading subset domain: Besides the scenario where the entire data domain is read, the mere need for a subset of data domain is representative for applications such as interactive visualizations and descriptive statistical analysis [29]. Reorganizing the data as requested saves both computation and storage overhead. How the subset data size results in terms of time and memory also draw our interest. Thus, we only read a geometric core, whose coordinates are $\{\frac{1024-d}{2}, \frac{1024+d}{2}\}$ in each dimension, of the entire domain as the subset data. The three distances(d) used were:

- **d=128:** readers only read a 128×128×128 cube from the geometric core of the entire data domain.
- **d=256:** readers only read a 256×256×256 cube from the geometric core of the entire data domain.
- **d=512:** readers only read a 512×512×512 cube from the geometric core of the entire data domain.

In Figure 9, we show the I/O time per time and memory consumption step among proposed reorganization methods with different sizes of subset domain to read. It is observed that all the methods perform identically on the writer side. For the first reader, while RAD and RASAR require longer time as subset size increases, RASIA and SAHR stay constant because of the computation in advance. Although SAHR typically requires slightly more read time than RASIA, it uses the memory more efficiently by collecting the data access pattern. RAD only uses the memory for the original data despite the always longer read time. RASAR saves only the part of reorganized data as reader queried, which consumes up to the double size of the original data. However, RASIA always saves the entire data domain in another layout, which remains stable memory usage to the double size of the original data.

From the above-mentioned test cases, a trade-off is drawn with respect to the available resources and the features of the workflow planned to run. Apart from the main applications, if the additional resource for the staging server is very limited, RAD turns to perform the best in both time to the solution and the memory usage, since others are likely to be slow due to the heavy workload at the server and even breaks

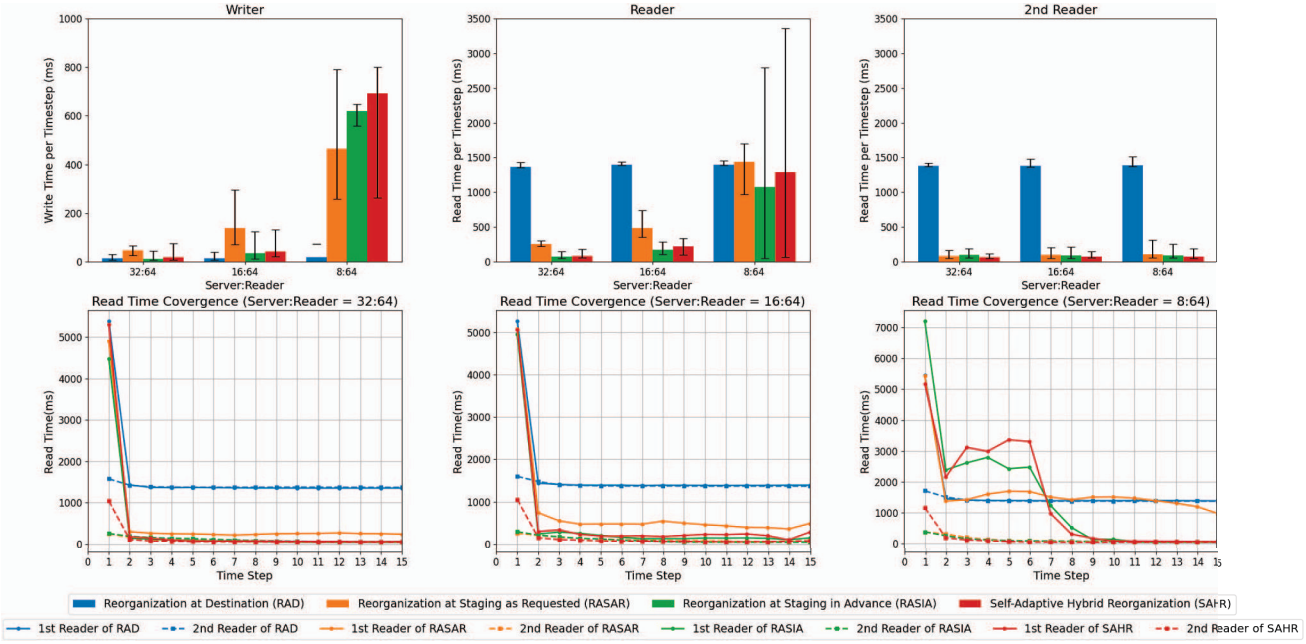


Fig. 8. Comparison of I/O time per time step among four data reorganization methods with different staging server scale

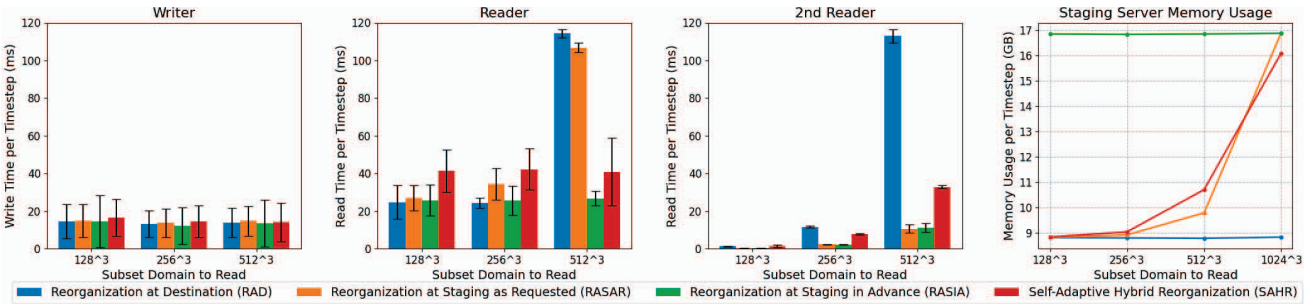


Fig. 9. Comparison of I/O time & staging server memory usage per time step among four data reorganization methods with different size of subset domain to read

down due to the insufficient memory size. Applications with a longer cycle time benefit from the reorganization in advance as RASIA works. For the situation where applications only need a subset of data, RASAR outperforms others by computing as needed. SAHR presents its advantage in adaption to a particular workflow by balancing memory consumption and I/O time.

B. Strong scaling comparison to existing Kokkos backends

Besides the experiment to explore the trade-offs between different data reorganization placements, we compare the SAHR method implemented in our heterogeneous data staging system with two other existing backends of inter-application data exchange based on the file in Kokkos framework: StdFile (C++ standard I/O binary files who need ad-hoc data reorganization if layouts between applications are mismatched.), Parallel HDF5 [30] (HDF5 files who need ad-hoc data reorganization if layouts between applications are mismatched.). Because the data coupling through the file system does not support asynchronous I/O between applications, writer, reader, and the second reader are running in sequence in all the cases of this test. To simulate a typical extreme scale in-situ workflow, such as XGC1 [31], FLASH [32], according to [33], a strong scaling test is performed with the detailed configuration described in Table II.

In Figure 10, we show the result of strong scaling comparison among C++ standard I/O, HDF5, and DataSpaces in the fixed data domain for both homogeneous and heterogeneous data exchange between writer and reader. It is clearly observed that DataSpaces outperforms the other two existing baseline approaches by the I/O time reduction of 27%-79% and 77%-98% for the writer and reader respectively, except for DataSpaces SAHR, which sacrifices a little at writer side but gains the overhead hide at reader side with the performance identical to homogeneous data get. This is expected because data storage and management in DataSpaces avoids the involvement of secondary storage. In contrast to the result of fixed scale experiments, DataSpaces shows great overall scalability and acceptable overhead with the increasing number of application processors from 300 to 5k in total,

TABLE II
EXPERIMENTAL SETUP CONFIGURATIONS OF DATA DOMAIN, CORE-ALLOCATIONS AND SIZE OF THE STAGED DATA FOR STRONG SCALING TESTS

Data Domain	1024 × 1024 × 1024				
No. of Parallel Writer Cores (Nodes)	256(8)	512(16)	1024(32)	2048(64)	4096(128)
No. of Parallel Reader Cores (Nodes)	32(2)	64(4)	128(8)	256(16)	512(32)
No. of 2nd Parallel Reader Cores (Nodes)	32(2)	64(4)	128(8)	256(16)	512(32)
No. of Staging Cores (Nodes)	16(4)	32(8)	64(16)	128(32)	256(64)
Total Staged Data Size (15 Time-steps)	120 GB				
Cycle Time	20 second				

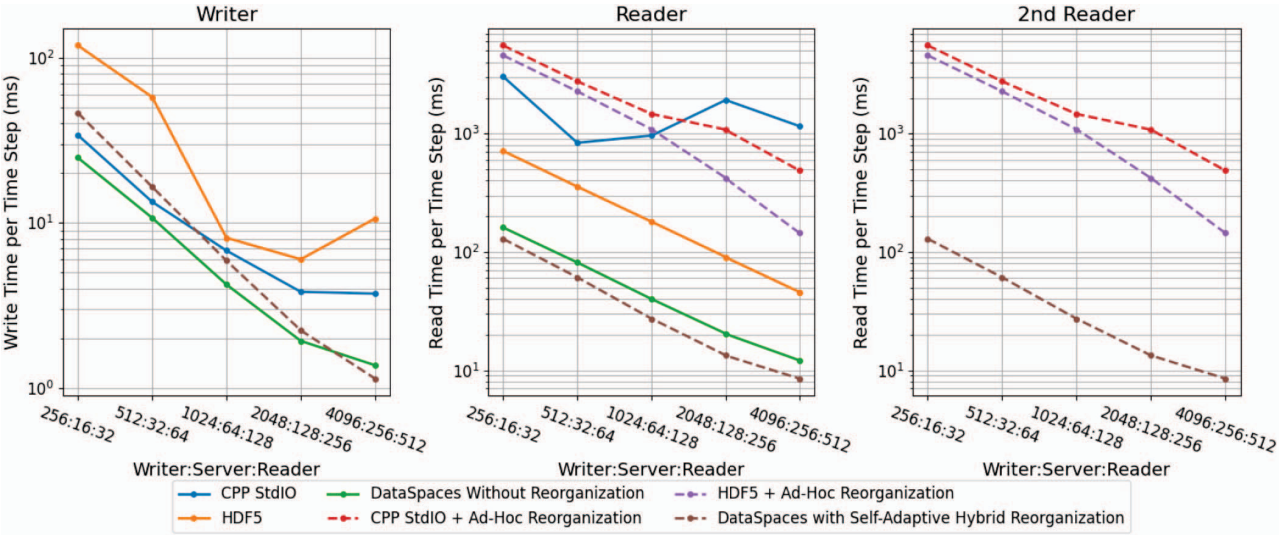


Fig. 10. Strong scaling comparison of I/O time per time step among C++ standard I/O, HDF5 and DataSpaces

compared to the existing baseline approach.

From our strong scaling workflow simulations, we can infer that file-based in-situ data exchange between applications performs poorly at the extreme scale due to the unnecessary involvement of the file system. In addition, heterogeneous workflow makes file-based in-situ data exchange difficult to provide portability from both performance and development perspectives. On the contrary, our heterogeneous staging service is able to tackle these cases easily and provide an I/O time reduction of up to 97% in comparison to C++ standard I/O and HDF5. Also, our staging service provides the common APIs for heterogeneous applications, so developers could easily couple them by adding extra a few extra lines. In summary, our staging service can effectively support heterogeneous multi-component scientific workflow thus guaranteeing the portability of the individual applications at the extreme scale, by efficiently reorganizing the data on the fly.

VI. Conclusion and Future Work

Although heterogeneous programming frameworks have emerged as effective solutions for porting applications to various platforms, they are not capable of assembling these applications into a heterogeneous in-situ workflow while maintaining individual portability. We propose four data reorganization methods, RAD, RASAR, RASIA, and SAHR, which simplify the exchange of data between heterogeneous applications requiring different memory access layouts and then implement these methods within the Kokkos Staging Space, an extension of Kokkos' original data abstraction based on the DataSpaces data staging framework. We evaluate the Kokkos Staging Space on the TACC Frontera system using a synthetic benchmark; our experimental results give insight into the effectiveness and trade-offs between the four methods under different access frequencies and use patterns in the shared data. We demonstrate that the Kokkos Staging Space outperforms the existing file-based Kokkos data abstraction both in time-to-solution and scalability for inter-application data exchange. The source code for our prototype implementation of the Kokkos Staging Space is publicly available at https://github.com/Zhang690683220/kokkos-staging-space. As future work, we plan to support more data reorganization types, such as the transformation between Array of Structs(AoS) and Struct of Arrays(SoA), and to evaluate these methods using a real-world scientific workflow of heterogeneous components.

Acknowledgment

We thank Elisabeth Giem, Patricia Gharagozloo, and Joseph Kenny for the consistent support of our research. This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under con-

tract DE- NA0003525. This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. The authors acknowledge Zhao Zhang and the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC, visualization, database, or grid resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu

REFERENCES

[1] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. (2021) Top500 list, june 2021. [Online]. Available: https://www.top500.org/lists/top500/2021/06/

[2] A. Goswami, Y. Tian, K. Schwan, F. Zheng, J. Young, M. Wolf, G. Eisenhauer, and S. Klasky, "Landrush: Rethinking in-situ analysis for gpgpu workflows," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 32–41.

[3] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[4] W. M. Brown, "Gpu acceleration in lammps," in *LAMMPS User's Workshop and Symposium*, 2011.

[5] B. Wang, S. Ethier, W. Tang, T. Williams, K. Z. Ibrahim, K. Madduri, S. Williams, and L. Oliker, "Kinetic turbulence simulations at extreme scale on leadership-class systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[6] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[7] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[8] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 2019, pp. 71–81.

[9] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[10] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[11] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[12] B. Zhang, P. Davis, M. Parashar, N. M. Morales, and K. Teranishi, "Toward resilient heterogeneous computing workflow through kokkos-dataspaces integration." Sandia National Lab.(SNL-CA), Livermore, CA (United States), Tech. Rep., 2020.

[13] F. Rizzi, P. J. Blonigan, and K. T. Carlberg, "Pressio: Enabling projection-based model reduction for large-scale nonlinear dynamical systems," *arXiv preprint arXiv:2003.07798*, 2020.

[14] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang *et al.*, "Exploring traditional and emerging parallel programming models using a proxy application," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 919–932.

[15] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Citeseer, Tech. Rep., 2008.

[16] (2021) Cuda c++ programming guide - v11.5.1. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[17] N. A. Awar, S. Zhu, G. Biros, and M. Gligoric, "A performance portability framework for python," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 467–478.

[18] C. Trott, L. Berger-Vergiat, D. Z. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, M. Gligoric, N. Al Awar, G. Shipman, and G. Womeldorff, "The kokkos ecosystem: Comprehensive performance portability for high performance computing," *Computing in Science & Engineering*, 2021.

[19] R. Reyes, G. Brown, R. Burns, and M. Wong, "Sycl 2020: More than meets the eye," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–1.

[20] S. Khuvis, K. Tomko, J. Hashmi, and D. K. Panda, "Exploring hybrid mpi+ kokkos tasks programming model," in *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+ X (PAW-ATM)*. IEEE, 2020, pp. 66–73.

[21] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of hpc-style sycl applications," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–11.

[22] L. Wan, A. Huebl, J. Gu, F. Poeschel, A. Gainaru, R. Wang, J. Chen, X. Liang, D. Ganyushin, T. Munson *et al.*, "Improving i/o performance for exascale applications through online data layout reorganization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 878–890, 2021.

[23] S. He, Y. Yin, X.-H. Sun, X. Zhang, and Z. Li, "Optimizing parallel i/o accesses through pattern-directed and layout-aware replication," *IEEE Transactions on Computers*, vol. 69, no. 2, pp. 212–225, 2019.

[24] H. Tang, S. Byna, S. Harenberg, X. Zou, W. Zhang, K. Wu, B. Dong, O. Rubel, K. Bouchard, S. Klasky *et al.*, "Usage pattern-driven dynamic data layout reorganization," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 356–365.

[25] (2021) Apache arrow: A cross-language development platform for in-memory data. [Online]. Available: https://arrow.apache.org/

[26] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The evolution of leadership computing at the national science foundation," in *Practice and Experience in Advanced Research Computing*, 2020, pp. 106–111.

[27] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, C.-S. Chang, and M. Parashar, "Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 1033–1042.

[28] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire, "Comparing the efficiency of in situ visualization paradigms at scale," in *International Conference on High Performance Computing*. Springer, 2019, pp. 99–117.

[29] P. Pebay, D. Thompson, and J. Bennett, "Computing contingency statistics in parallel: Design trade-offs and limiting cases," in *2010 IEEE International Conference on Cluster Computing*, 2010, pp. 156–165.

[30] 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*, 2011, pp. 36–47.

[31] T. Koskela and J. Deslippe, "Optimizing fusion pic code performance at scale on cori phase two," in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Cham: Springer International Publishing, 2017, pp. 430–440.

[32] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, and H. Tufo, "Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *The Astrophysical Journal Supplement Series*, vol. 131, no. 1, p. 273, 2000.

[33] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," in *Computer Graphics Forum*, vol. 35, no. 3. Wiley Online Library, 2016, pp. 577–597.