SANDIA REPORT
SAND2021-10271
Unclassified Unlimited Release
Printed August 2021

Sandia National Laboratories

Design and Performance of Kokkos Staging Space toward Scalable Resilient Application Couplings

Bo Zhang, Philip Davis, Pradeep Subedi, Manish Parashar, Francesco Rizzi, Nicolas Morales and Keita Teranishi

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

Design and Performance of Kokkos Staging Space toward Scalable Resilient Application Couplings

Bo Zhang, Philip Davis, Pradeep Subedi and Manish Parashar
Department of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854-8019
{bo.zhang,philip.e.davis,parashar}@rutgers.edu

Francesco Rizzi
Nexgen Analytics
30 N Gould St
Sheridan, WY 82801
francesco.rizzi@ng-analytics.com

Nicolas Morales and Keita Teranishi
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551
{nmmoral,knteran}@sandia.gov

SAND2021-10271

ABSTRACT

With the growing number of applications designed for heterogeneous HPC devices, application programmers and users are finding it challenging to compose scalable workflows as ensembles of these applications, that are portable, performant and resilient. The Kokkos C++ library has been designed to simplify this cumbersome procedure by providing an intra-application uniform programming model and portable performance. However, assembling multiple Kokkos-enabled applications into a complex workflow is still a challenge. Although Kokkos enables a uniform programming model, the inter-application data exchange still remains a challenge from both performance and software development cost perspectives. In order to address this issue, we propose Kokkos data staging memory space, an extension of Kokkos' data abstraction (memory space) for heterogeneous computing systems. This new abstraction allows to express data on a virtual shared-space for multiple Kokkos applications, thus extending Kokkos to support inter-application data exchange to build an efficient application workflow. Additionally, we study the effectiveness of asynchronous data layout conversions for applications requiring different memory access patterns for the shared data. Our preliminary evaluation with a synthetic benchmark indicate the effectiveness of this conversion adapted to three different scenarios representing access frequency and use patterns of the shared data.

ACKNOWLEDGMENT

We thank Robert Clay, Joseph Kenny, Jeremiah Wilke and Christian Trott for the support of this work. We also acknowledge Jeffery Miles for the advice on our initial design and implementation.

1. MOTIVATION

With the trend that various computing devices are integrated in next generation exascale clusters, a growing number of heterogeneous applications are designed for fully utilizing the computing power. However, application programmers and users are finding it challenging to compose scalable, performant and portable in-situ workflow with a multiple of concurrent application executions to enable multi-scale and multi-component simulations and analyses. Such workflows require the users to understand the performance charactersitics of the target heterogeneous computing platforms and application programs in detail. Heterogeneous programming frameworks, such as Kokkos[1] and RAJA[2], are well-designed and widely adopted as solutions to write performance portable applications targeted at all major HPC platforms. But to our best knowledge, none of these programming frameworks have the capabilities to link multiple heterogeneous applications into a complex workflow using similar performance portable abstractions. The efficient implementation of these interaction capabilities is paramount to the coming heterogeneous HPC era. DataSpaces[3] provides capabilities for coupling applications through a shared-space abstraction. As a coupling framework, DataSpaces enables flexible data redistribution between applications using simple put/get and concurrency control semantics. These low-level semantics provides a good foundation for instrumenting scalable application interactions, but does not inherently provide any usable heterogeneous computing abstractions as seen in Kokkos.

As such, the overarching goal of this research is to build a performance-portable data staging service at extreme scale that can couple multiple applications with a single heterogeneous programming model. Specifically, our approach integrates existing data staging solution with the Kokkos ecosystem by supporting inter-application data exchanges between various memory layouts. The resulting portable data staging service allows us to simply transfer data between applications in the Kokkos semantics at runtime regardless of the underlying data layout of these applications. We also develop an in-transit mechanism to manage data reorganization and replication for heterogeneous memory layouts performed at our data staging service. More specifically, we propose three designs with different in-transit data reorganization placement adapted to system resources constraints and workflow characteristics.

2. DESIGN

2.1. Architecture

A schematic overview of the portable data staging service based on Kokkos is presented in Figure 2-1. It is built upon the DataSpaces framework and directly leverages existing components by reusing its data transport, indexing and querying capabilities. The DataSpace client APIs are seamlessly integrated with the Kokkos core library to support Kokkos::Staging APIs. The key components of the portable data staging service include the Data Reorganization module and the Kokkos::Staging Interface. These modules cooperate to facilitate data movement and sharing across heterogeneous HPC workflow applications.

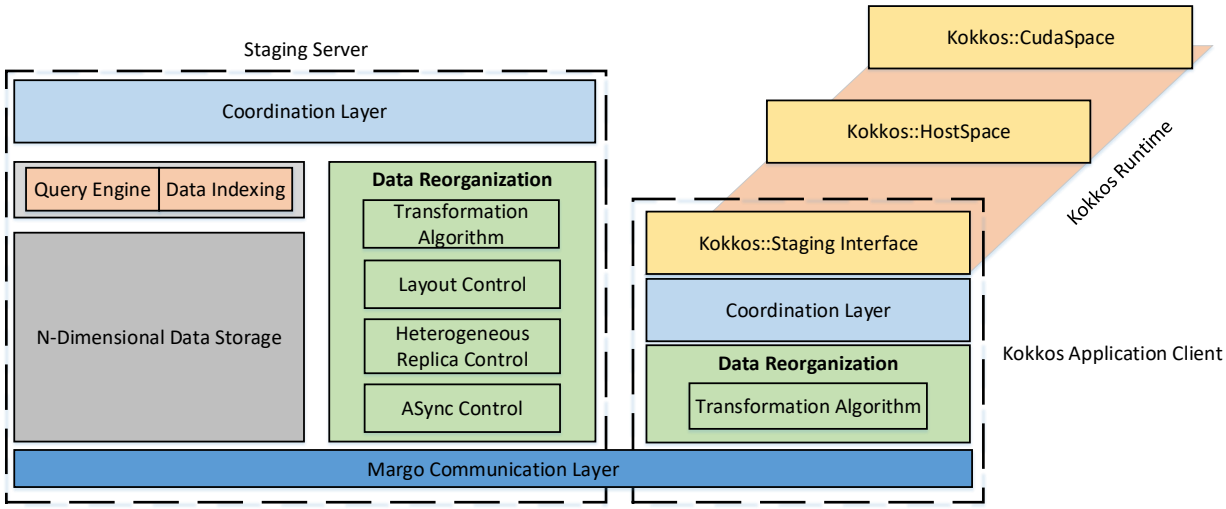

Figure 2-1 Architecture of portable data staging service based on Kokkos. The data reorganization module and Kokkos::Staging APIs were implemented on top of DataSpaces and Kokkos framework respectively.

2.2. Data Reorganization module

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 layouts and scheduling the data reorganization operations. In a complex workflow, the required data layouts for the 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 there are thousands or even millions of asynchronous heterogeneous data requests floods in, concurrency also becomes a major concern if the data reorganization operations are proceeded at the staging server. To overcome this problem, concurrency control and heterogeneous replica control are implemented in the Data Reorganization module. When many requests comes to the staging server for the same data object but in heterogeneous layouts, if the requested layout is replicated at the server, then the server send the data object to the client. However, if the data object with the requested layout does not exist, the server will only launch a data reorganization for the first request and let others wait until it is ready, saving both computational overhead and memory usage at the clients.

2.3. Kokkos::Staging Interface

In order to make our new data staging capability compliant to other memory spaces in Kokkos, we wrap DataSpaces client operations with a new name space `Kokkos::Staging`. In order 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.

```
Kokkos::Staging::initialize();
Kokkos::Staging::finalize();
```

After the initialization, users can declare a `Kokkos::Staging` view similar to what they are supposed to do with `Kokkos::CudaSpace`. The layout of `Kokkos::Staging` view should be explicitly declared for heterogeneity, otherwise it would use the default layout as the host space.

```
using ViewStaging_t = Kokkos::View<Data_t**, Kokkos::StagingSpace>;
ViewStaging_t v_S("StagingView", i1, i2);
using ViewStaging_lr_t = Kokkos::View<double**, Kokkos::LayoutRight,
                                      Kokkos::StagingSpace>;
ViewStaging_lr_t v_S_lr("StagingView_LayoutRight", i1, i2);
```

The `deep_copy` function enables the put/get operations for Kokkos applications to transfer data to/from the staging server. In the `deep_copy` function, zero-copy non-blocking data transfer to the staging server is performed. But before the actual data transfer, setting the version and bounding box of the variable is optional but strongly suggested.

```
Kokkos::Staging::set_version(v_S_lr, version);
Kokkos::Staging::set_lower_bound(v_S_lr, lb0, lb1);
Kokkos::Staging::set_upper_bound(v_S_lr, ub0, ub1);
// from host to staging
Kokkos::deep_copy(v_S_lr, v_P);
// from staging to host
Kokkos::deep_copy(v_G, v_S);
```

When reader applications request the data with a different layout, an extra line is needed to declare the heterogeneity. This will also allow completely different view label for data in other layouts.

```
Kokkos::Staging::view_bind_layout(v_S_ll, v_S_lr);
```

A simple usage example of `Kokkos::Staging` is illustrated here.

```
using ViewHost_lr_t    = Kokkos::View<double**, Kokkos::LayoutRight,
                                      Kokkos::HostSpace>;
using ViewHost_ll_t    = Kokkos::View<double**, Kokkos::LayoutLeft,
                                      Kokkos::HostSpace>;
using ViewStaging_lr_t = Kokkos::View<double**, Kokkos::LayoutRight,
                                      Kokkos::StagingSpace>;
using ViewStaging_ll_t = Kokkos::View<double**, Kokkos::LayoutLeft,
                                      Kokkos::StagingSpace>;
ViewHost_lr_t v_P("PutView", i1, i2);
ViewStaging_lr_t v_S_lr("StagingView_LayoutRight", i1, i2);

Kokkos::Staging::set_version(v_S_lr, version);
Kokkos::Staging::set_lower_bound(v_S_lr, lb0, lb1);
Kokkos::Staging::set_upper_bound(v_S_lr, ub0, ub1);

// from host to staging
Kokkos::deep_copy(v_S_lr, v_P);

ViewStaging_ll_t v_S_ll("StagingView_LayoutLeft", i1, i2);
ViewHost_ll_t v_G("GetView", i1, i2);

Kokkos::Staging::set_version(v_S_ll, version);
Kokkos::Staging::set_lower_bound(v_S_ll, lb0, lb1);
Kokkos::Staging::set_upper_bound(v_S_ll, ub0, ub1);
// bind two staging views in different layout
Kokkos::Staging::view_bind_layout(v_S_ll, v_S_lr);

// from staging to host
Kokkos::deep_copy(v_G, v_S_ll);
```

With these fundamental APIs, we 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.

3. HETEROGENEOUS DATA REORGANIZATION MECHANISM

In extreme scale in-situ workflows using data-staging middleware such as DataSpaces, the data producer applications are typically computationally intensive. Therefore, it is desirable for these producer applications to offload the operations for managing different data layouts across the

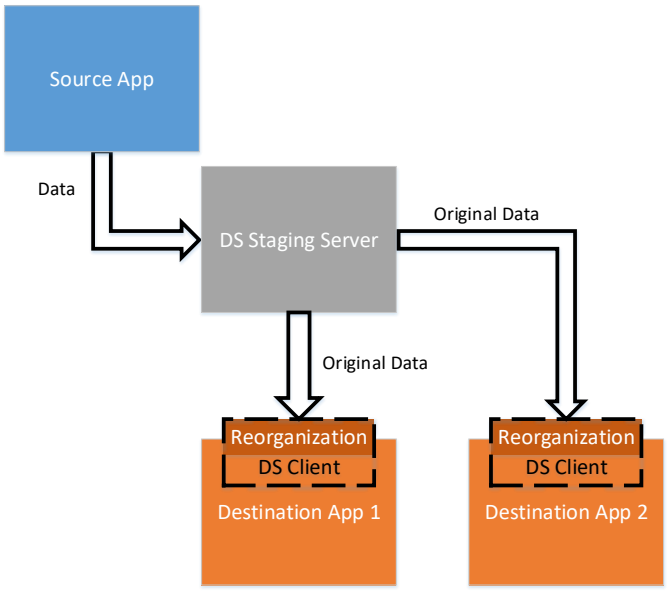

Figure 3-1 A schematic illustration of reorganization at Destination.

coupled applications in their workflow. To achieve this goal, we proposed three data reorganization approaches, i.e., reorganization at destination, reorganization at staging as requested, and reorganization at staging in advance. In this section, we present the design of these three approaches.

3.1. Reorganization at Destination

To prepare the data for a specific layout different from the origin, the most straightforward approach is fetching the original data and reorganizing it at the destination application. We implemented this approach by adding a generic reorganization wrapper after fetching the original data from the server at the DataSpaces client, which is co-located with the destination application. The major advantage of this approach is simplicity in the implementation, and it can scale out with the destination application. The disadvantage of this approach is lack of reuse for multiple application instances with the same data reorganization. As shown in Figure 3-1, if multiple applications request the data in the same layout but different with the original one, every application has to reorganize the data on it own, which is a waste of both computation resources and time to solution.

3.2. Reorganization at Staging as Requested

For the purpose of reorganized data reuse, it is possible to offload the data reorganization operations to the staging server. Figure 3-2 illustrates the data reorganization as requested at the staging server. The first fetch request for the data in a layout different from the origin will invoke the reorganization process at server. To avoid a waste of computation as well as duplicated heterogeneous replica storage, other concurrent requests for the same data object in the same

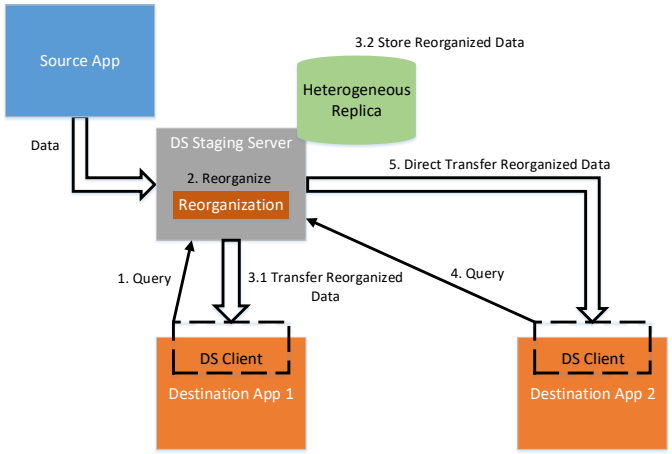

Figure 3-2 A schematic illustration of Reorganization at Staging as Requested.

reorganized layout are halted until the ongoing reorganization finishes. Then, the staging server sends the reorganized data to its multiple destinations, saving the reorganized data into its storage at the same time for future requests. Since the staging server keeps the replicated data in heterogeneous layouts, the subsequent fetch requests for the data in the available layouts are processed efficiently. This design utilizes the idle computing resources at the staging server while staging is always an I/O-bound task. Since the first request of each data object with new data layout still leads to longer I/O time, reorganization at the server might lead to a significant slowdown with an extremely limited scale of the staging server.

3.3. Reorganization at Staging in Advance

In order to make the destination applications unaware of data reorganization from the I/O performance perspective, overlapping the reorganization time in the staging server with the processing time in the destination applications is essential. We achieved this goal by reorganizing the data to all types of the available layouts at the staging server in advance. As shown in Figure 3-3, the staging server will start to reorganize the entire data object immediately after receiving it from the source applications. In this approach, all the subsequent fetch requests are not aware of data reorganization at all, since the staging server keeps the heterogeneous data replica for all possible layouts. However, the major weakness of this approach is the large memory capacity requirement to meet the increasing number of the supported layouts.

4. EVALUATION

In this section, we present the empirical evaluations of our heterogeneous data reorganization mechanism with our synthetic benchmarks, which simulate a variety of data access patterns.

Our experiments have been performed on the Frontera System at the Texas Advanced Computing Center (TACC). Frontera hosts 8368 compute nodes, each containing Dual Intel Xeon Platinum

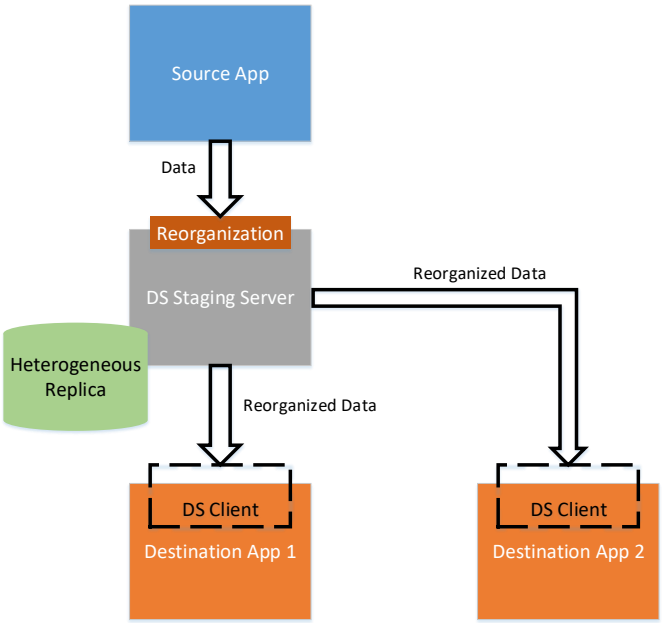

Figure 3-3 A schematic illustration of Reorganization at Staging in Advance.

8280 ("Cascade Lake"), 28-core processor with 192GB of DDR4 RAM and 240GB SSD. All of the tests in subsequent sections are performed 3 times and the average result is reported.

Data read access rate and layout matching between source and destination impacts the performance of simulation/analysis workflows. To better understand the impact of typical data read patterns upon our approach, we select two scenarios similar to [4]: reading entire data domain for all time steps, and reading subset data domain for all time steps. In both scenarios, coupled scientific applications are assumed to write/read data to/from a three-dimensional data domain. The data is assumed to be written over multiple iterations or time-steps in a fixed layout, and read in the similar temporal fashion but reorganized into different layouts.

In our synthetic tests, two application codes, namely readers and writers, are used to emulate a generic end-to-end data movement behavior in real coupled simulation workflows. As their names suggest, the writers produce simulation data and write it to the staging servers and the readers read the data from the staging servers then perform some analysis. The data is organized in a 3-dimensional Cartesian grid format with $X \times Y \times Z$ scale.

In all of the synthetic test cases, one writer application writes the data for the entire domain in a fixed layout over all the (simulation) time steps into the staging servers, while one reader application reads the data in either same or different layout of the writers' data format. To demonstrate the reuse of the reorganized data, we have evaluated the performance of the second reader, which shares the same read pattern of the first one.

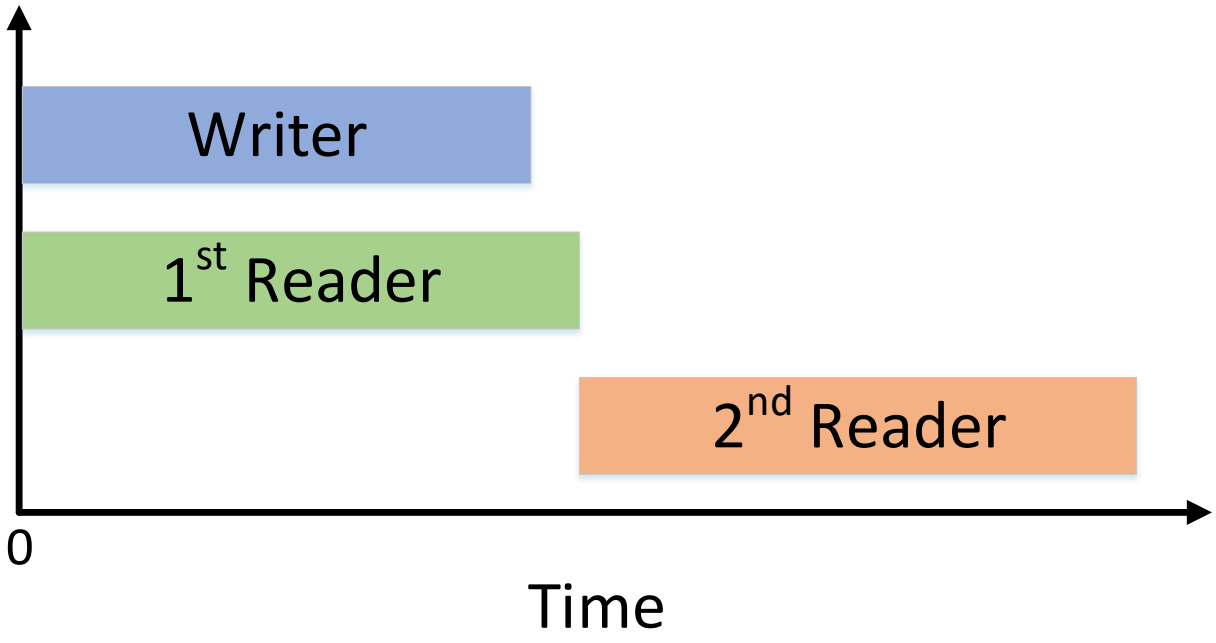

Figure 4-1 Sequence diagram of the workflow used in exploring the task placement of data reorganization

Data Domain	$1024 \times 1024 \times 1024$
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 (20 Time-steps)	160 GB

Table 4-1 experimental setup configurations of core-allocations for writer and staging server, data domain and size of the staged data for data reorganization task placement tests

4.1. Exploring the task placement of data reorganization

This experiment evaluates the impact on the I/O performance for scalable in-transit workflow of three data reorganization task placement scenarios discussed in the previous section. To better understand the tradeoffs between these approaches, three critical metrics in in-transit workflow are selected based on [5]. Table 4-1 details the base setup for all the tests cases in this experiment. This setup might be changed due to different metrics to be evaluated. As shown in figure 4-1, the second reader is designated to start after the first one finishes in this experiment to eliminate the interference between asynchronous reader applications.

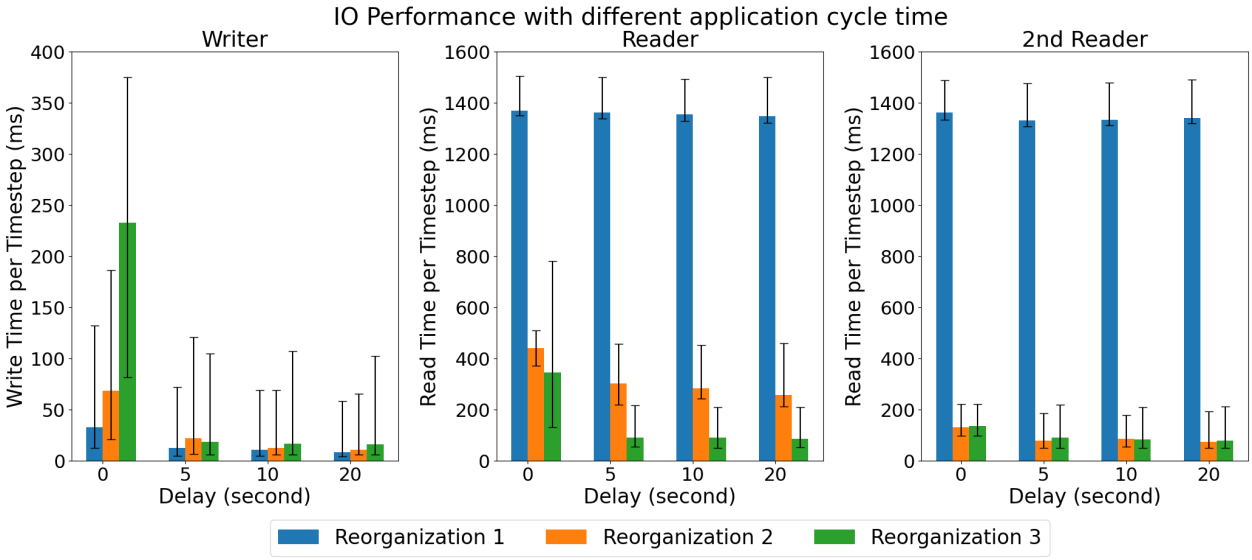

Figure 4-2 Comparison of I/O time per time step among three different data reorganization placement with varying cycle time of applications

4.1.1. Metric 1 - Cycle time of writer and reader

Because our synthetic writers and readers use a simplified data generator, both of the applications has a relatively fast cycle time. Some applications could be represented by such a fast cycle time, but the implications with applications who have longer cycle time should be studied as well. To simulate these longer cycle times, 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 cases used were:

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

Figure 4-2 shows the I/O time per time step for each data reorganization design with varying cycle time of applications. Apparently, longer cycle times benefit data reorganization at staging server. Reorganization 1, where data reorganization resides at the reader side, cannot take advantage of latency hiding and keeps a steady I/O time regardless of delay time. As for Reorganization 2, it benefits from longer cycle times, because infrequent data fetch leaves enough time for data reorganization at staging server, avoiding the cascading slow down for the subsequent data fetches. Reorganization 3 exploits the longer cycles effectively to achieve the speedup of 25× and 5× compared to Reorganization 1 and 2 respectively in *Delay(20)* because the staging server can perform the data reorganization during the idle time of the writer. However, in *Delay(0)*, the writer dramatically degrades its performance because the data staging server has to concurrently perform the data reorganization for the outstanding reader's request and the data transfer from the writer. For the second reader, both reorganization 2 and 3 have nearly identical

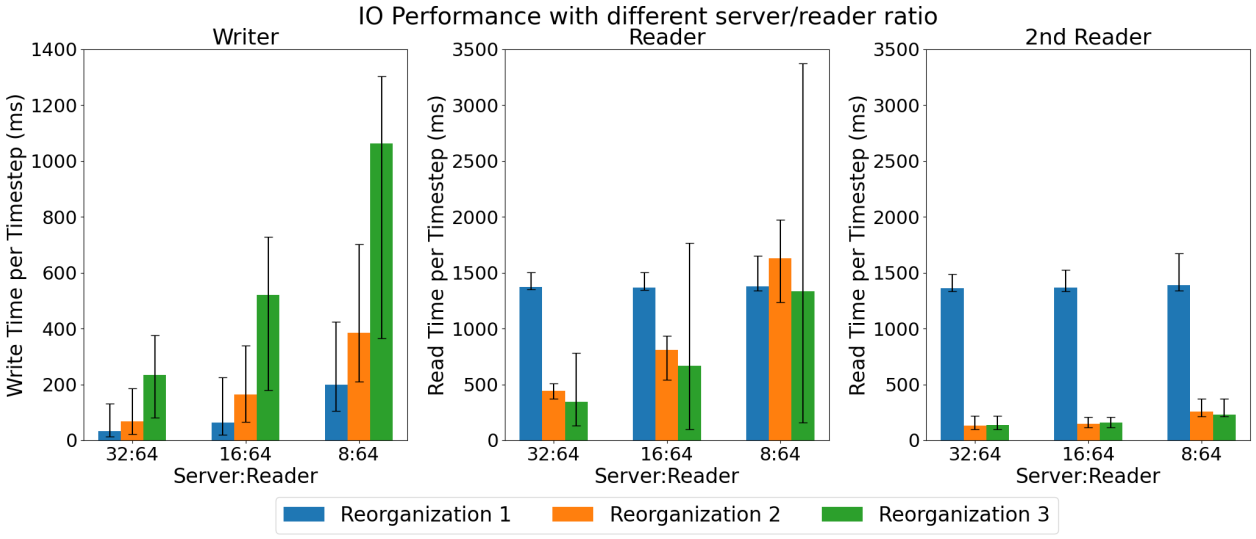

Figure 4-3 Comparison of I/O time per time step among three different data reorganization placement with different staging server scale

read time by directly hitting the reorganized replica at staging server, while reorganization 1 still has to transform every data object. This "cache hit" saves up to 94% read time.

4.1.2. Metric 2 - Staging server scale

Placing the data reorganization task at staging server has a great potential to make the staging server computational bound. Consequently, the respond rate to incoming I/O requests would slow down, which might further stalls the entire workflow disastrously. For the in-transit paradigm, the scale of simulation and analysis are typically predetermined. Thus, we change the number of staging server cores to explore how the computational capability of the staging server impacts the performance. We have investigated the three server configurations with the scale of 512(16) writer cores(nodes) and 64(4) reader cores(nodes) as follows:

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

Figure 4-3 demonstrates the I/O time per time step among three different data reorganization placement with different staging server scale. There are several insights that can be drawn from Figure 4-3. First, all three data reorganization design are subject to slightly I/O time increase as the server:reader ratio decreases. Second, the server scale has a large impact on reorganization 2 and 3. For the first reader, the I/O performance of both Reorganization 2 and 3 degrades as the server scale decreases. In the *server:reader ratio of 8:64*, Reorganization 2 performs the worst while Reorganization 3 has a relatively same performance with Reorganization 1. However, due to the asynchronous workflow we were running, the I/O behavior varies as the time step increases. Figure 4-4 demonstrate the read time of all the data reorganization designs in each time step with

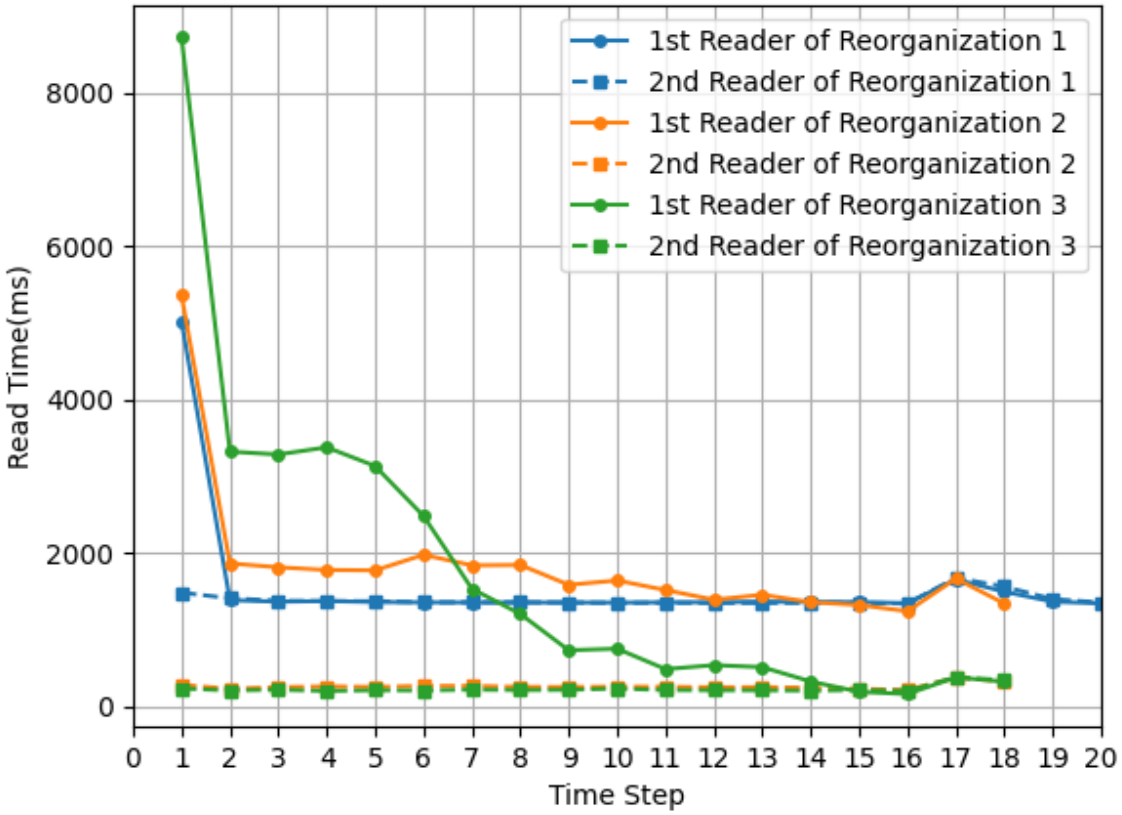

Figure 4-4 Comparison of read time in each time step among three data reorganization with the server:reader ratio of 1:8

the server:reader ratio of 1:8. 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 has to wait for the data generated and transferred to the staging server. From the second time step, the read time of Reorganization 1 is stable as expected, while Reorganization 2 has longer reader time than Reorganization 1 in almost every time step due to the limited parallelism at staging server. The read time of Reorganization 3 is significantly long at the first several time steps, because the reader has to wait for the finish of reorganization, which happens immediately after the staging server receiving the data from the writer. As the writer stops transferring the data to the staging server and the server finishes the reorganization in advance, the first reader starts to take advantages of reorganized replicas at server, thus the read time becomes shorter than the other two from time step 8 and converge to the read time of the second readers who completely reuse the reorganized replicas at server. This also explains the large span of error bar of Reorganization 3 in Figure 4-3.

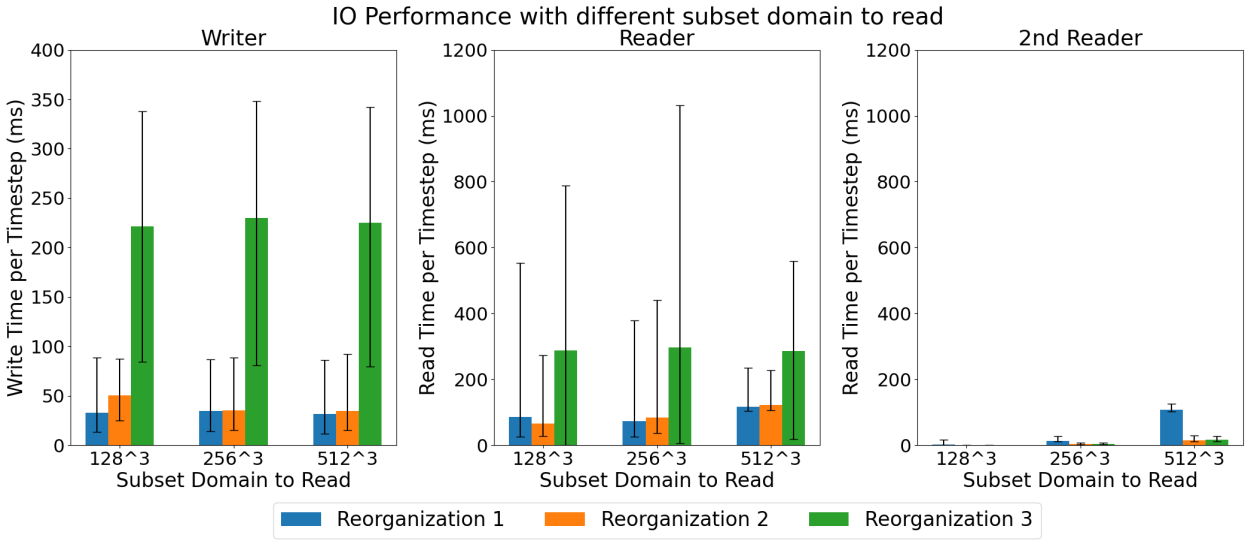

Figure 4-5 Comparison of I/O time per time step among three different data reorganization placement with different size of subset domain to read (any % for the saving from the full-domain?

4.1.3. Metric 3 - Data size of reading subset domain

Besides the scenario where the entire data domain is read, accessing a subset of the data domain is representative for applications such as interactive visualizations and descriptive statistic analysis[6]. Processing the data as requested saves both the computation and storage overheads. It is also important to explore the saving with respect to the size of the subset. Thus, we only reads the data assigned to a single core of the simulation (writer) code, whose coordinates is $\{\frac{1024-d}{2}, \frac{1024+d}{2}\}$ in each dimension of the entire domain. The three distance parameters (d) are given as:

- **d=128:** readers only read a $128\times128\times128$ cube from the core of the entire data domain.
- **d=256:** readers only read a $256\times256\times256$ cube from the core of the entire data domain.
- **d=512:** readers only read a $512\times512\times512$ cube from the core of the entire data domain.

In Figure 4-5, we show the I/O time per time step among the three different data reorganization placement with the different sizes of the subset domain to read. It was observed that for both the writer and the first reader, Reorganization 1 and 2 share almost the same I/O time due to the small size of the subset data to transform. On the other hand, Reorganization 3 has the worst performance and large variation at the writer and the reader due to the concurrent execution of the data reorganization of the entire domain and the data transfer from the writer. As for the second reader, the difference between Reorganization 1 and the others becomes relatively smaller than the writer and the first reader.

The memory usage for the three data reorganization with the different subset domain sizes are presented in Figure 4-6. Reorganization 1 just needs the memory for the original data. Reorganization 2 exhibits the great saving from just maintaining the replica of the subset as it is

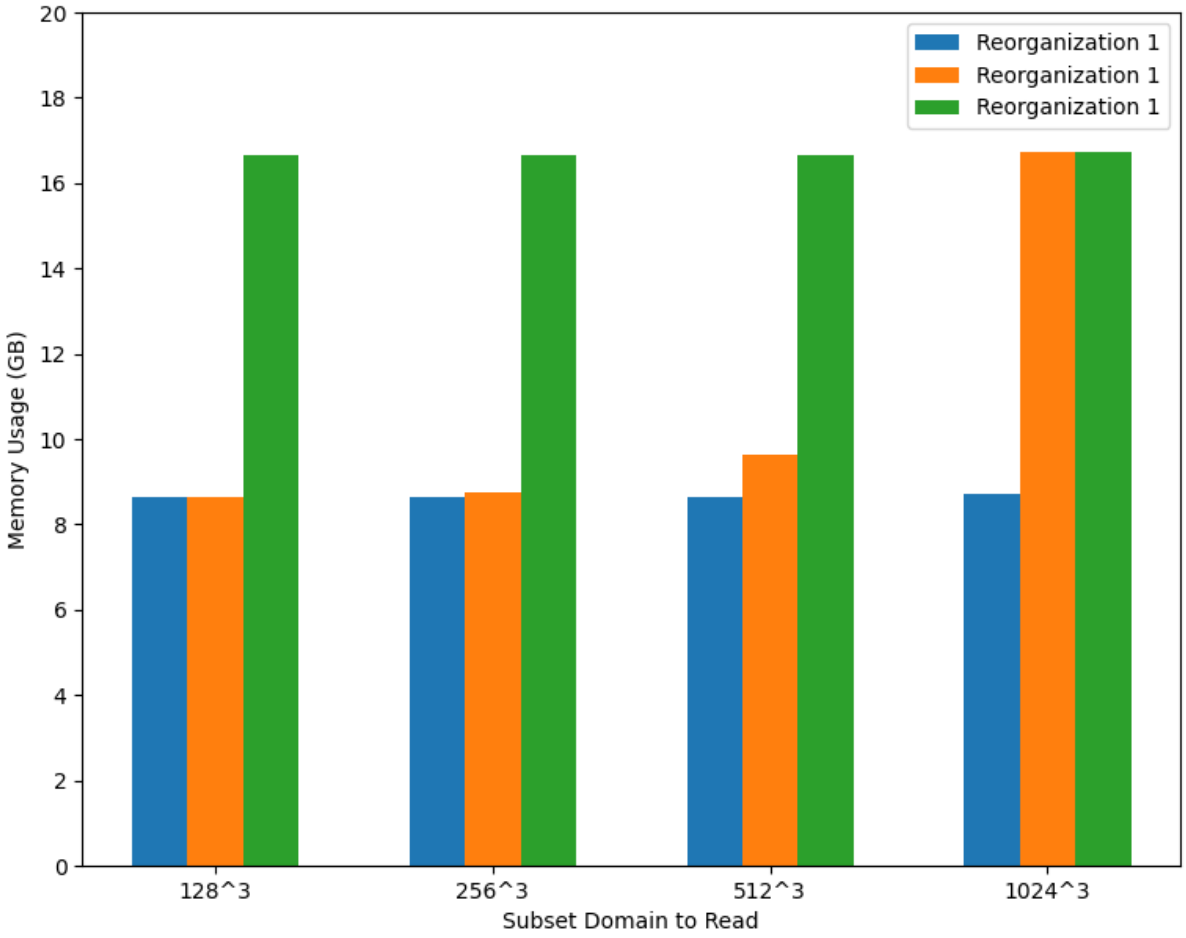

Figure 4-6 Comparison of memory usage among three data reorganization with different size of subset domain to read (The captions needs a fix!!)

designed to create a reorganized subset as needed. Reorganization 3 always keeps the entire data domain in another layout, doubling the memory requirement irrespective to the subset size.

From the aforementioned test cases, a tradeoff is drawn with respect to the available resources and the features of workflow planned to run. Apart from the main applications, if the additional resources for staging server is very limited, Reorganization 1 turns to perform the best in both time to solution and memory usage, since the other two are likely to be slow due to heavy workload at server and even breaks down due to exhausted memory. Applications with long cycle time will benefit from reorganization in advance as Reorganization 3 works. For the situation where applications only need a subset of data, Reorganization 2 outperforms others by computing as needed.

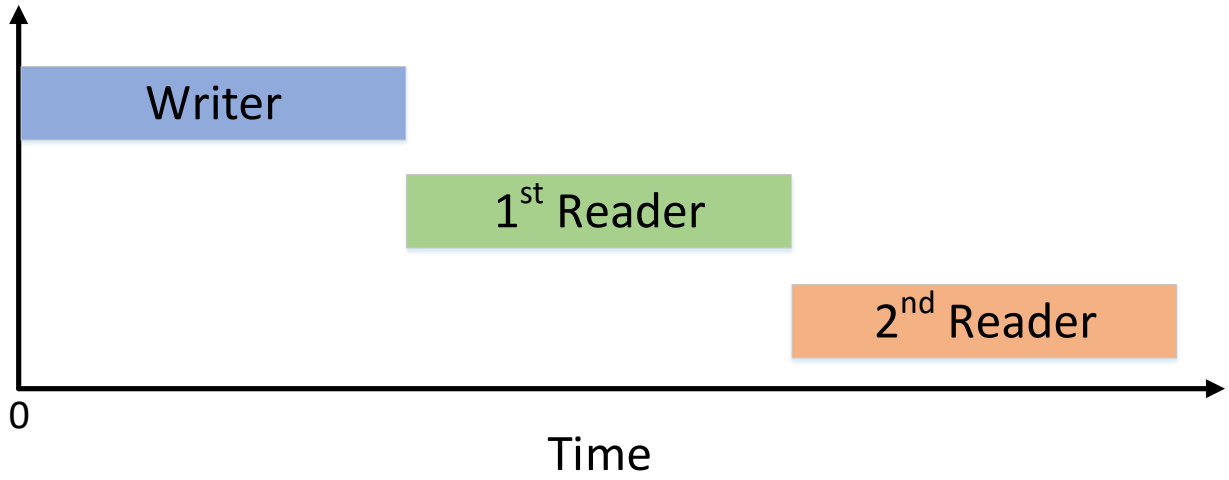

Figure 4-7 Sequence diagram of the workflow used in strong scaling comparison to existing methods

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 (20 Time-steps)	160 GB				
Cycle Time	20 second				

Table 4-2 experimental setup configurations of data domain, core-allocations and size of the staged data for strong scaling tests

4.2. Strong scaling comparison to existing methods

Besides the experiment to explore the tradeoffs between three data reorganization placements, we compared our heterogeneous data staging system with two other existing methods of inter-application data exchange based on file in the Kokkos framework: standard I/O (data is stored as binary files in disk using C++ standard I/O implementation and involves ad-hoc data reorganization if the layouts between simulations and analyses are mismatched.), HDF5[7] (data is stored as HDF5 files in disk using HDF5 implementation and involves ad-hoc data reorganization if needed.). Because the data coupling through file systems does not support asynchronous I/O between applications, writer, reader and the second reader are running in sequence, as shown in Figure 4-7, in all the cases of this study. To simulate a typical extreme scale in-situ workflow, such as XGC1[8], FLASH[9], according to[10], a strong scaling test has been performed with the detailed configuration described in Table 4-2.

In Figure 4-8, we show the result of strong scaling comparison among C++ standard I/O, HDF5 and DataSpaces in the fixed data domain for both homogenous and heterogenous data exchange between the writer and the reader. It was observed clearly that almost all cases using DataSpaces outperforms the other two existing baseline approaches by the I/O time reduction of 20%-87%

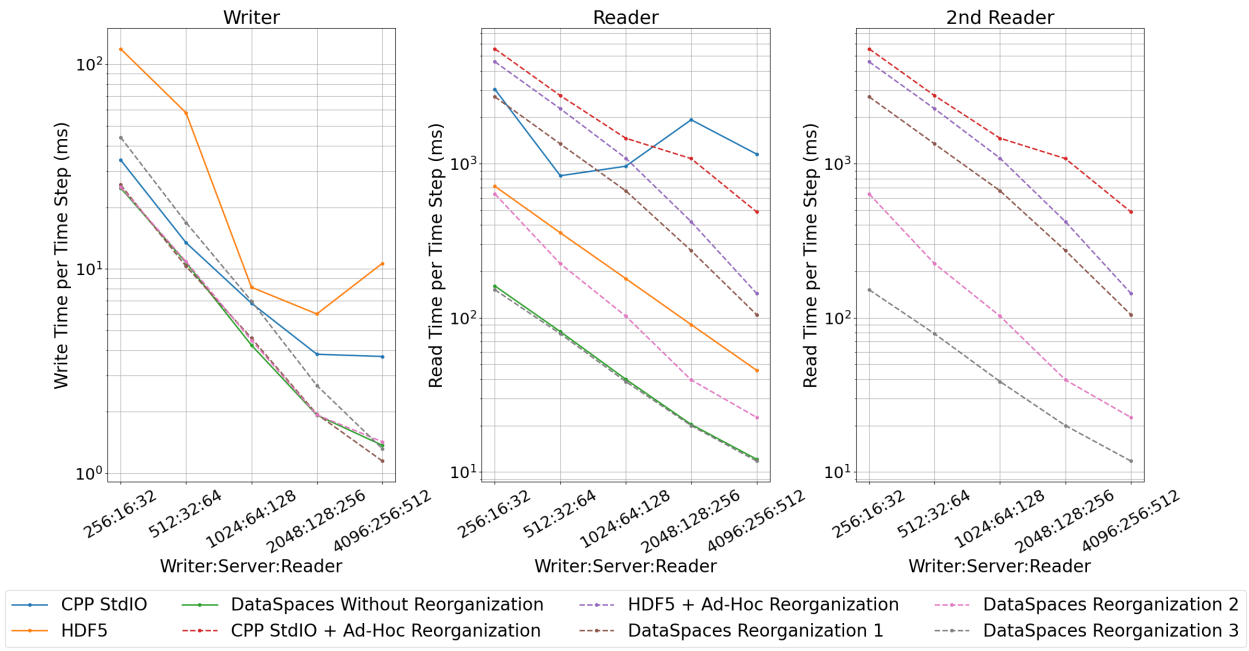

Figure 4-8 Strong scaling comparison of I/O time per time step among C++ standard IO, HDF5 and DataSpaces

and 27%-97% for the writer and the reader respectively, except for DataSpaces Reorganization 3, which sacrifices a little at the writer's side but gains the overhead hidden at the reader's side with the performance identical to the homogenous data fetch. This is expected because the data storage and management in DataSpaces avoids the involvement of secondary storage. In contrast to the result of the fixed scale experiments, DataSpaces exhibits a great overall scalability and an acceptable overhead with the increasing number of application processing elements from 300 to 5k in total, compared to the existing baseline approach.

From our strong scaling workflow simulations, we can infer that the file-based in-situ data exchange between applications performs poorly at extreme scale due to the unnecessary involvement of file systems. In addition, heterogeneous workflow makes the file-based in-situ data exchange difficult to provide a portability form both performance and development perspectives. On contrary, our heterogeneous staging service is able to tackle these cases easily and provide an I/O time reduction up to 97% in comparison to C++ standard I/O and HDF5. From our data reorganization placement exploration, we also observe that putting data reorganization tasks into staging server would achieve the best performance for a large scale workflow as long as the resources for staging is not extremely limited. Also, our staging service provides identical APIs for heterogeneous applications so that developers could easily couple them by adding extra few lines. In summary, our staging service can effectively provide both performance and development portable data staging service for heterogeneous workflows at extreme scale with efficient data reorgnization on the fly.

5. FUTURE WORK

While 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. In this paper, we designed data reorganization mechanisms, which simplifies data exchange between heterogeneous applications that require different memory access pattern for performance. We implemented the data reorganization mechanisms within Kokkos Staging Space, an extension of Kokkos data abstraction, based on DataSpaces data staging framework. Kokkos Staging Space is deployed on the Frontera system at TACC, and experimentally evaluated using a synthetic benchmark. Our experiment results gave insight into the effectiveness and tradeoffs between the three data reorganization mechanisms: reorganization at destination, reorganization at staging as requested, and reorganization at staging in advance, under different scenarios representing access frequency and use patterns of the shared data. The result also demonstrated that Kokkos Staging Space performs better than existing file-based Kokkos data abstraction in terms of time-to-solution and scalability for inter-application data exchange.

As future work, we plan to build an adaptive data reorganization mechanism by combining the three design we implemented in this paper, which could learn the data access pattern and dynamically choose the most appropriate reorganization method according to both time to solution and memory usage. Besides, we also plan to support more data reorganization type, such as the transformation between Array of Struct(AoS) and Struct of Array(SoA). In addition, seeking a real in-situ workflow consist by heterogeneous applications and evaluating it is an important task in our following work as well.

REFERENCES

[1] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[2] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. Raja: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*, pages 71–81. IEEE, 2019.

[3] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[4] Tong Jin, Fan Zhang, Qian Sun, Hoang Bui, Melissa Romanus, Norbert Podhorszki, Scott Klasky, Hemanth Kolla, Jacqueline Chen, Robert Hager, Choong-Seock Chang, and Manish Parashar. Exploring data staging across deep memory hierarchies for coupled data intensive

simulation workflows. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1033–1042, 2015.

[5] James Kress, Matthew Larsen, Jong Choi, Mark Kim, Matthew Wolf, Norbert Podhorszki, Scott Klasky, Hank Childs, and David Pugmire. Comparing the efficiency of in situ visualization paradigms at scale. In Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan, editors, *High Performance Computing*, pages 99–117, Cham, 2019. Springer International Publishing.

[6] Philippe Pebay, David Thompson, and Janine Bennett. Computing contingency statistics in parallel: Design trade-offs and limiting cases. In *2010 IEEE International Conference on Cluster Computing*, pages 156–165, 2010.

[7] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana 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, 2011.

[8] Tuomas Koskela and Jack Deslippe. Optimizing fusion pic code performance at scale on cori phase two. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, pages 430–440, Cham, 2017. Springer International Publishing.

[9] Bruce Fryxell, Kevin Olson, Paul Ricker, FX Timmes, Michael Zingale, DQ Lamb, Peter MacNeice, Robert Rosner, JW Truran, and H Tufo. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, 2000.

[10] Andrew C Bauer, Hasan Abbasi, James Ahrens, Hank Childs, Berk Geveci, Scott Klasky, Kenneth Moreland, Patrick O'Leary, Venkatram Vishwanath, Brad Whitlock, et al. In situ methods, infrastructures, and applications on high performance computing platforms. In *Computer Graphics Forum*, volume 35, pages 577–597. Wiley Online Library, 2016.

Sandia National Laboratories

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 under contract DE-NA0003525.