Parallel Streaming between Heterogeneous HPC Resources for Real-time Analysis

Thomas Marrinan[a,1,*], Greg Eisenhauer[b], Matthew Wolf[c,2], Joseph A. Insley[d,3], Silvio Rizzi[d], Michael E. Papka[d,3]

[a] *University of St. Thomas, St. Paul, Minnesota, USA 55105*
[b] *Georgia Institute of Technology, Atlanta, Georgia, USA 30332*
[c] *Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA 37830*
[d] *Argonne National Laboratory, Lemont, Illinois USA 60439*

Abstract

Performing analysis or generating visualizations concurrently with high performance simulations can yield great benefits compared to post-processing data. Writing and reading large volumes of data can be reduced or eliminated, thereby producing an I/O cost savings. One such method for concurrent simulation and analysis is *in transit* — streaming data from the resource running the simulation to a separate resource running the analysis. In transit analysis can be beneficial since computational resources may not have certain resources needed for visualization and analysis (e.g. GPUs) and to reduce the impact of performing analysis tasks to the run time of the simulation. When sending and receiving data in transit, data redistribution mechanisms are needed in order to support heterogeneous data layouts that may be required by the simulation and analysis applications. The work described in this paper compares two mechanisms for on-the-fly data redistribution when streaming data in parallel between two distributed memory applications. Our results show that it is often more advantageous to stream data in the same layout as the sender and redistribute data amongst processes on the receiving end than to stream data in the final layout needed by the receiver.

Keywords: In transit, analysis, visualization, high performance computing, in situ

*Corresponding author

Email addresses: tmarrinan@stthomas.edu (Thomas Marrinan), eisen@@cc.gatech.edu (Greg Eisenhauer), wolfmd@ornl.gov (Matthew Wolf), insley@anl.gov (Joseph A. Insley), srizzi@alcf.anl.gov (Silvio Rizzi), papka@anl.gov (Michael E. Papka)

[1] Also affiliated with Argonne National Laboratory.
[2] Also affiliated with Georgia Institute of Technology.
[3] Also affiliated with Northern Illinois University.

1. Introduction

Simulations and analysis run on high-performance computing (HPC) resources are driving large-scale science and engineering. Applications running on these massively parallel, distributed memory systems must divide data and computation amongst individual compute processes. Key findings from the 2014 DOE High Performance Computing Operational Review [1] state that in situ and in transit visualization and analysis will become more common and should be treated as first-class citizens along with HPC simulations. It also reports that in situ and in transit analysis will become necessary as computing reaches exascale due to I/O limitations of saving data to disk.

While in situ visualization and analysis can have performance benefits from not needing to move or copy data, it can also be detrimental to the running simulation. According to the 2013 DOE Data Intensive Science and Exascale Computing Report [2], in situ analysis can incur significant runtime costs due to the analytics consuming CPU time and memory, which are precious to the simulation. In contrast, in transit analysis enables simulation applications to fully utilize CPU and memory on a computing resource, while streaming data over high-bandwidth networks to a separate resource for visualization and analysis.

By shipping data to a separate resource, in transit analysis provides unique affordances that researchers can leverage. One major advantage of in transit analysis is that it can be used to take advantage of systems with specialized hardware. Kress et al. [3] compared in situ analysis to in transit analysis and determined in transit to be more scalable, support data translations better, have greater fault tolerance, and be easier to use. Also, using in transit analysis only loosely couples the analysis with the simulation, making it possible to plug either component into other routines [4].

In this paper, we present our research on improving parallel streaming performance for in transit analysis between two distributed memory applications with heterogeneous data layouts. Our research was motivated by the desire to stream simulation data from remote high-performance computer centers to a researcher's local system for on-site analysis (such as ultra-high resolution visualizations being rendered and viewed on a tiled display wall).

After highlighting related works that motivated our research, we discuss two techniques for streaming and reorganizing data between two remote clusters. The first technique is the current state-of-the-art and involves the simulation knowing the desired data layout for the receiving analysis application and sending the corresponding chunks of data directly to the proper analysis nodes. The second technique is the culmination of our research and involves each node in the simulation sending all of its data to an analysis node in a load balanced fashion, then having the analysis application redistribute data locally so that it ends up in the desired layout.

Since local redistribution of data is a challenge in its own right, we then discuss our development of the Dynamic Data Redistribution (DDR) library [5]. The DDR library calculates what data must be exchanged with each processes and abstracts MPI routines to enable HPC applications to redistribute data

between processes. Application developers simply must specify what data each process currently owns and the data each process desires with respect to the overall data domain.

We have evaluated the two techniques for in transit streaming in order to study their performance and scalability under various conditions. Performance of parallel streaming between remote resources likely depends on the exact configuration of machines at each end. We therefore tested two use cases that we believe to be representative of common ways in transit streaming would be used for real-time analysis. Our results show that our technique (streaming data as is, then redistributing it within the visualization / analysis application) both outperforms and shows better scalability in nearly all cases than the current state-of-the-art (streaming data in the final layout needed by the receiver).

2. Related Work

In order to research in transit streaming between HPC resources, we looked at existing tools and techniques used in the field. First, we investigated frameworks for supporting parallel streaming in the context of distributed memory applications. Then, we looked at methods for handling data layout differences between producers and consumers.

2.1. In Transit Frameworks

We discovered three modern frameworks that support in transit streaming between distributed memory applications. The first framework we looked at was GLEAN [6]. GLEAN is a generic framework for accelerating data analysis and I/O for supercomputer simulations. GLEAN can facilitate in transit analysis via a server/client architecture that forwards simulation data to an analysis application. GLEAN can also perform on-the-fly data transformations between the simulation and analysis applications to make data constructs formatted properly for each application. However, these transformations are limited to the format within each process (such as structure of arrays vs. array of structures).

The next in transit framework we looked at was Henson [7]. Henson supports cooperative multitasking by splitting a distributed memory application into multiple "puppets" and acting as the "master" to coordinate the actions of multiple subroutines. Henson provides a flexible mechanism for routines to execute concurrently and is not limited to simply sending data from simulations to analysis tasks. However, since the master controls the puppets within the scope of a single application, it cannot be used to stream data between completely separate resources.

Finally, we looked at the Adaptable IO System (ADIOS) [8] for enabling in transit analysis. ADIOS affords simulations the option to be run offline and save data to disk using POSIX I/O or MPI-IO. When in transit visualization / analysis is desired, the FlexPath [9] transport method can instead be utilized to stream data over high-bandwidth networks and includes a method for reorganizing data layout on-the-fly. Due to the provided functionality and wide adoption of ADIOS, our work leverages this framework.

2.2. Data Transformation and Redistribution

We also investigated existing techniques for reorganizing data amongst processes in a distributed memory application. Our research found two classifications of tools to help with this procedure – data transformation tools and data redistribution tools. The former alters data layout within a given process, whereas the latter modifies the layout of data between processes.

Perry and Swany [10] developed a method called data type fission that segregates transmitted fields from non-transmitted fields for sending and receiving data between processes. MPI applications can therefore transmit certain fields of a native object while omitting others. The use of data type fission eliminates extra data copies and leads to a significant reduction in communication time. While this type of data transformation can lead to more efficient communication, it does not abstract the redistribution of data. This, in turn, leaves a heavy burden on application developers when data redistribution is necessary.

Kjolstad et al. [11] have developed an algorithm to automate the creation of custom MPI data types. Their work abstracts the transformation of non-continuous data for efficient retrieval. Performing these transformations manually in real-world applications is complex, time consuming, and error-prone. Therefore, their algorithm aims to improve programmer productivity and reliability. Our work on data redistribution has similar goals, but for inter-process data transmission in addition to staging the data for efficient retrieval.

When it comes to actually redistributing data between processes, DIY2 [12] provides some functionality to developers. DIY2 is a data and computation abstraction for parallel workflows. The main use for DIY2 is to enable the same program to execute on various platforms, from HPC distributed memory environments to a single multi-core workstation. However, one feature of DIY2 is to abstract the communication patterns for exchanging data between processes when running on a distributed memory system. This abstraction is intended for iterative processes requiring information from local neighbors or global reductions. In contrast, our data redistribution work is designed to facilitate the staging of data onto the proper processes for computation / analysis.

Esnard et al. [13] present a steering framework for parallel simulations. They couple a visualization system with a running HPC simulation, which allows users to view and modify the simulation in real-time. Since data in the simulation can have a different layout than is needed in the parallel visualization, data must be reorganized. In their work, this redistribution of data occurs at the socket level when transmitting data over the network between simulation processes and visualization processes. Our data redistribution work was designed to accomplish a similar task, but without relying on external network communication.

3. Parallel Streaming with Heterogeneous Data Layout

Since simulation and visualization / analysis applications may require different data layouts, simply transferring the data is not sufficient. Data must also be reorganized from the layout on the simulation into the desired layout on

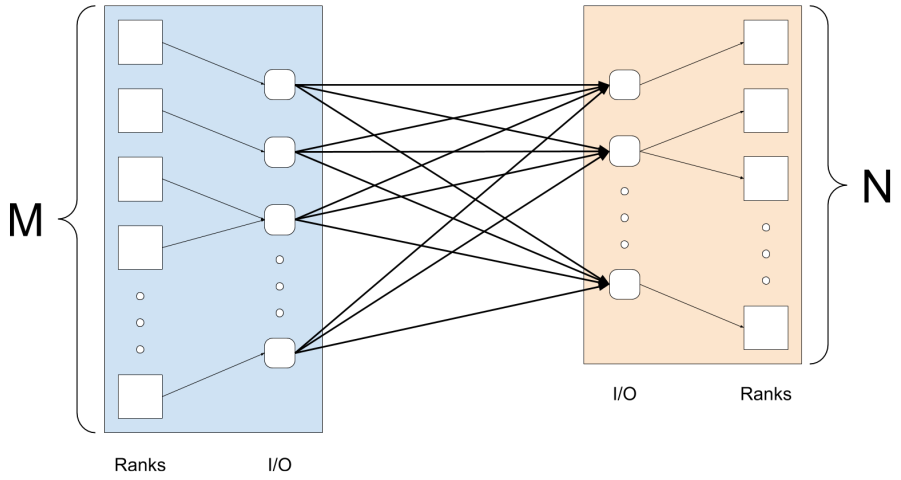

Figure 1: ADIOS bounding-box technique. N visualization / analysis processes receive data from M simulation processes with heterogeneous data layouts. Data is reorganized as it is streamed between resources.

the visualization / analysis resource. The rest of this section will describe two different techniques for data transfer and reorganization.

For both techniques, a subset of dedicated nodes may be needed to perform I/O depending on the topology of the computational resources. If only a subset of nodes on the simulation cluster have external network connection, data must be gathered on I/O nodes prior to streaming data out. Similarly, if only a subset of nodes on the visualization / analysis cluster have external network connection, data must be received on the I/O nodes then scattered to compute nodes. Additionally, the number of processes in the visualization / analysis application reading data (N) are likely to be different from the number of processes in the simulation application writing data (M). In practice, visualization / analysis routines typically use no more compute processes than the simulation ($M \geq N$).

3.1. Bounding-Box Streaming

The first technique for streaming and reorganizing data on-the-fly is the state-of-the-art technique provided by the ADIOS and FlexPath frameworks. Using this technique, each reader specifies a bounding-box for its desired data. The readers then compute the set of writers that own any chunk of needed data and send requests for that data. Once the data is received, each reader stitches the chunks together into contiguous memory at which point it becomes usable by the visualization / analysis application. This handles data reorganization as chunks are being shipped over the network between resources. Figure 1 illustrates the streaming and data reorganization between resources with heterogeneous data layouts while using bounding box streaming.

Streaming data directly from the owning process in the simulation to the requesting process in the visualization / analysis application has a couple of potentially costly side effects. First, each process in the visualization / analysis application must connect directly to each process in the simulation that it needs data from. Depending on how different the data layouts are between the two applications, this could lead to each visualization / analysis processes connecting to each simulation process (total of $M*N$ connections). After connections are established, it also leads to more messages with smaller sizes being sent over the network. Second, the reorganization of data is happening at the socket level as data is transported between computational resources. Typically, this connection is significantly lower bandwidth than the internal network of either cluster.

3.2. Write Block Streaming with Local Redistribution

Due to the potential draw-backs of the bounding-box technique, we developed an alternative technique for streaming and reorganizing data on-the-fly. Our technique is a two step process that decouples the streaming from the data reorganization. Using this technique, each reader uses ADIOS with FlexPath to receive full blocks of data from a mutually exclusive subset of writers. The number of writers each reader receives data from is divided as equally as possible and does not necessarily correspond to the data needed. Once the write blocks are received, the data must be redistributed amongst the visualization / analysis processes so that they each have the data needed by the application.

We leverage the Dynamic Data Redistribution (DDR) library [5], which is designed to perform on-the-fly reorganization of data using `MPI_Alltoall` calls, within the visualization / analysis cluster. The details of the DDR library are described in more detail in the next section, but it essentially provides a mechanism to locally redistribute data received directly from the simulation processes into the final layout needed by the visualization / analysis processes. If a subset of nodes in the visualization / analysis application are performing I/O, then this redistribution of data also replaces the scatter operation needed to move data to the compute nodes. Figure 2 illustrates the streaming and data reorganization between resources with heterogeneous data layouts while using write block streaming and local redistribution.

This technique was designed to mitigate the issues present in bounding-box streaming. Regardless of how different the data layouts are between the two applications, each visualization / analysis process will connect to M/N simulation processes. If M is not a multiple of N, each of the first R visualization / analysis process will connect to an extra simulation process (where R is the remainder of M/N). Since the entire block of data from each writer is streamed at once, this technique also results in fewer messages with larger sizes being sent over the network. Finally, redistributing the data after it is streamed allows the reorganization to occur over the high-bandwidth internal networks.

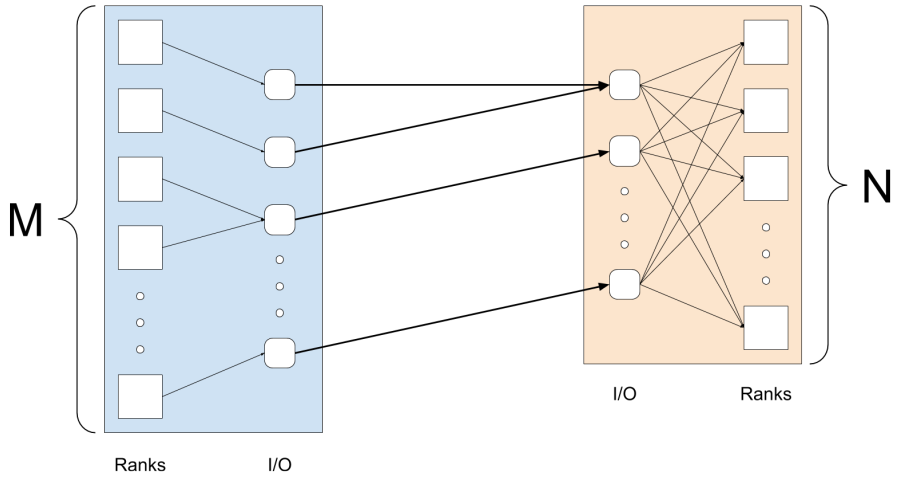

Figure 2: ADIOS write block technique with local redistribution. Each simulation process sends its entire data block to a single corresponding visualization / analysis process in a load balanced fashion. Data is locally redistributed on the visualization / analysis resource in order to match the final data layout needs of the application.

4. Automated Dynamic Data Redistribution

This section covers the library we developed for automated dynamic data redistribution in distributed memory applications. We have broken the process down into three major components: initialization and description of the data, setting up the mapping of data between processes, and the actual transmission of data between processes. Each of these three components have been wrapped into a single public function in our library. `DDR_NewDataDescriptor` is the public function that creates an object to describe the type of data being reorganized. `DDR_SetupDataMapping` is the public function that informs our library what data each process in the application owns and what data each process needs. `DDR_ReorganizeData` is the public function that performs MPI calls to exchange data between processes. By limiting the outward facing code, we have reduced the burden on application developers to integrate our library into existing projects.

Throughout this section we will use a simple example (henceforth referred to as *E1*) to help illustrate the concepts of dynamic inter-process data movement enabled by DDR. *E1* is a distributed memory application with four processes operating on a two-dimensional grid with an overall domain that is 8 cells wide and 8 cells tall. Each process currently owns two non-contiguous and mutually exclusive 8x1 rows from the overall domain, but needs the data located in a single 4x4 quadrant of the overall domain. Therefore, each process must send part of its currently owned data to each other processes, while also receiving a portion of the data it needs from each other process. Figure 3 illustrates the setup and data movement needs for *E1*.

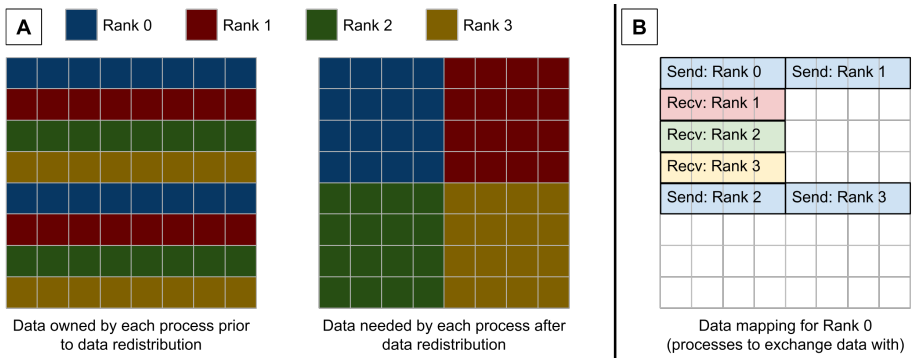

Figure 3: 2D data redistribution in a distributed memory application with four processes. Panel A – data layout before and after redistribution. The left grid shows each process in the application owning two separate 8x1 chunks of data in the overall domain prior to data redistribution. The right grid shows each process needing one continuous 4x4 chunk after redistribution in order to properly process the data. Panel B – data mapping for rank 0. This grid shows the chunks of data owned by rank 0 that need to be sent to other ranks as well as the chunks of data that rank 0 needs to receive from other ranks.

In addition to providing a visual overview of DDR, we outline how applications developers can easily integrate the library into their code. Algorithm 1 provides pseudocode for *E1* using the three public function calls in the DDR library. In the algorithm, data_{own} represents the two 8x1 chunks of data owned by each process prior to data redistribution, and $\text{data}_{\text{need}}$ represents the 4x4 quadrant of data needed by each process after data redistribution.

Algorithm 1 Sample use of the DDR library

Input: data_{own}
Output: $\text{data}_{\text{need}}$

```
1: desc = DDR_NewDataDescriptor(nProcesses, DATA_TYPE_2D,
       MPI_FLOAT, sizeof(float))
2: chunks_own = 2
3: dims_own = {[8, 1], [8, 1]}
4: offsets_own = {[0, rank], [0, rank+4]}
5: right = rank % 2
6: bottom = rank / 2
7: dims_need = [4, 4]
8: offsets_need = [4*right, 4*bottom]
9: DDR_SetupDataMapping(rank, nProcesses, chunks_own, dims_own,
       offsets_own, dims_need, offsets_need, desc)
10: DDR_ReorganizeData(nProcesses, data_own, data_need, desc)
```

4.1. Data Description

The first step to enable DDR is to create a description of the type of data that needs to be redistributed. DDR currently supports 1D, 2D, or 3D arrays stored

in continuous memory with each element having a fixed size. An application creates a DDR descriptor with the `DDR_NewDataDescriptor` function. This function has four parameters: the number of processes in the MPI application; whether the data is organized in a 1D, 2D, or 3D array; the data type of the elements in the array; and the byte size of the elements in the array. The function returns a pointer to an object that stores this information.

4.2. Data Mapping

The second step for DDR is to set up the mapping between processes for sending and receiving data. Each process in the distributed memory application may own several chunks of data from the overall domain. In order to properly process the data, we assume that each process will require a single continuous subsection of data after data redistribution. Therefore, DDR enables each process to send data to other processes from many chunks, while receiving data from other processes into one chunk.

An application sets up the data mapping using the `DDR_SetupDataMapping` function. This function has eight parameters: the rank number of the process calling the function, the number of processes in the MPI application, the number of chunks the process calling the function owns, an array specifying the dimensions of each owned chunk, an array specifying the offsets of each owned chunk into the overall domain, the dimensions of the chunk the process calling the function needs after data redistribution, the offset of the needed chunk into the overall domain, and the DDR descriptor object. Table 1 enumerates *E1*s parameter values for each process using the pseudocode from Algorithm 1.

	P1	P2	P3	P4	P5	P6	P7	P8
Rank 0	0	4	2	{[8,1],[8,1]}	{[0,0],[0,4]}	[4,4]	[0,0]	desc
Rank 1	1	4	2	{[8,1],[8,1]}	{[0,1],[0,5]}	[4,4]	[4,0]	desc
Rank 2	2	4	2	{[8,1],[8,1]}	{[0,2],[0,6]}	[4,4]	[0,4]	desc
Rank 3	3	4	2	{[8,1],[8,1]}	{[0,3],[0,7]}	[4,4]	[4,4]	desc

Table 1: `DDR_SetupDataMapping` parameter values for *E1*. Parameters are abbreviated – P1: rank number, P2: number of processes, P3: number of chunks to send, P4: array of send chunk dimensions, P5: array of send chunk offsets, P6: receive chunk dimensions, P7: receive chunk offsets, P8: DDR descriptor.

Dimensions and offsets for sending and receiving data chunks have a number of elements corresponding to the problem type - [i] for 1D, [i,j] for 2D, and [i,j,k] for 3D. Therefore the number of total elements in the sending dimensions and offsets parameters must be equal to the number of chunks owned prior to redistribution multiplied by the number of dimensions in the problem type. The number of elements in the receiving dimensions and offsets parameters must be equal to the number of dimensions in the problem type.

The chunks of data sent from each process should be mutually exclusive and complete. This means that no two processes should own the same data prior to redistribution and that collectively the entire data domain should be owned by some process. On the receiving end, however, data does not need to be

mutually exclusive or complete. This means that multiple processes can receive overlapping data and that there can be areas of the overall domain not received by any process.

Internally, the `DDR_SetupDataMapping` function creates a series of send and receive objects to be used with MPI commands for data redistribution. Based on the send and receive dimensions and offsets provided by each process, a geometric overlap is computed to detect which subsections of the data chunks should be sent to and received from other processes. Even in applications where the data is dynamic, this set up process is only required once as long as the layout of data remains consistent

4.3. Data Redistribution

The third and final step for redistributing data using DDR is to actually exchange the data between processes in the distributed memory application. A developer can trigger this with a call to `DDR_ReorganizeData`, which takes four parameters: the number of processes in the MPI application, a buffer that has the data owned by the process calling the function, a buffer where the needed data will be stored into, and the DDR descriptor object.

Internally, the `DDR_ReorganizeData` function will make calls to `MPI_Alltoallw` in order to exchange data between processes. The number of `MPI_Alltoallw` calls is equivalent to the maximum number of chunks that any one process owns. `MPI_Alltoallw` is used (rather than `MPI_Alltoallv`) since custom subarray types are needed to describe multidimensional subsets of data. When dealing with dynamic data, `DDR_ReorganizeData` can be called each time processes own new data without needing to initialize the library or set up the data mapping again. The DDR library is available with permission from Argonne's development team at https://xgitlab.cels.anl.gov/fl/ddr/.

5. Use Cases: In Transit Streaming with Data Redistribution

In this section we describe two authentic use cases surrounding a simple Lattice Boltzmann Method (LBM) simulation for computing fluid flows in a two-dimensional space [14]. The LBM simulation breaks fluid properties, such as density and velocity, into a regular grid of floating point values. In each iteration of the simulation, every cell updates its value by simulating particles streaming and colliding. Certain cells, including the edges, are kept at fixed values. Barriers can be placed inside the domain to force the fluid to flow around them, thereby creating more turbulent flow patterns.

The simulation application splits the data into horizontal slices distributed between the processes. This was done so as to minimize the number of processes each rank needed to exchange data with during each iteration of the simulation. By using slices that cover the entire width of the domain, each rank only needs to communicate with two other processes at most, the neighbors with data directly above and below.

5.1. Ultra High-Resolution Visualization

Our first authentic analysis case consists of a simple visualization application, which would take a 2D array of floating point values and apply a colormap in order to create an image. A variable of interest, such as vorticity, would be chosen as the array to stream from the simulation to the visualization application. By streaming data as its being computed, users can investigate the visualization in real-time to quickly determine flow patterns throughout the domain.

Since the simulation can be run with a very fine grid, the visualization requires an ultra high-resolution display in order to view each cell. Today, ultra high-resolution video walls are created by tiling numerous HD or 4K displays, which are often driven by a single PC with high-end graphics cards [15]. Even with one PC, however, a distributed memory application may be used to split tasks (e.g. one process per screen, one process per GPU, etc.).

5.2. Pattern Recognition

Our second authentic analysis case consists of an automated routine for finding areas of interest within the overall domain. Again vorticity could be used as a property of interest, and users may only care about regions with high turbulence. By streaming data as its being computed, analysis can be run in real-time to enable saving only the regions of interest instead of the entire data set (thus saving I/O). Performing such an analysis routine would likely need to be run on a distributed memory cluster in order to handle the large amounts of data coming from the simulation.

6. Evaluation

In order to evaluate the bounding-box and write block with local redistribution techniques, we used sample distributed memory applications as stand-ins for the simulation and visualization / analysis routines. The sample simulation used a 2D regular grid of double precision floating-point numbers and ran for 100 time steps, outputting data 1 out of every 10. For each time step, the simulation would sleep for 200 ms to imitate computational work. The 2D grid was distributed amongst the processes of the simulation in horizontal slices.

The sample visualization / analysis application would continually request the next output time step from the simulation until there were no more time steps remaining. This application measured the time between when the next time step became available from the simulation and when the data was fully received and laid out appropriately amongst its processes. Two different layouts were tested for the visualization / analysis application - distributing the 2D grid in vertical slices and in a rectangular grid with chunks as close to square as possible. An example of these data layouts are illustrated in Figure 4.

These sample applications were used so that we could isolate the effective rate at which the two techniques allowed data to be streamed and reorganized without interference from computational imbalances that may be present in

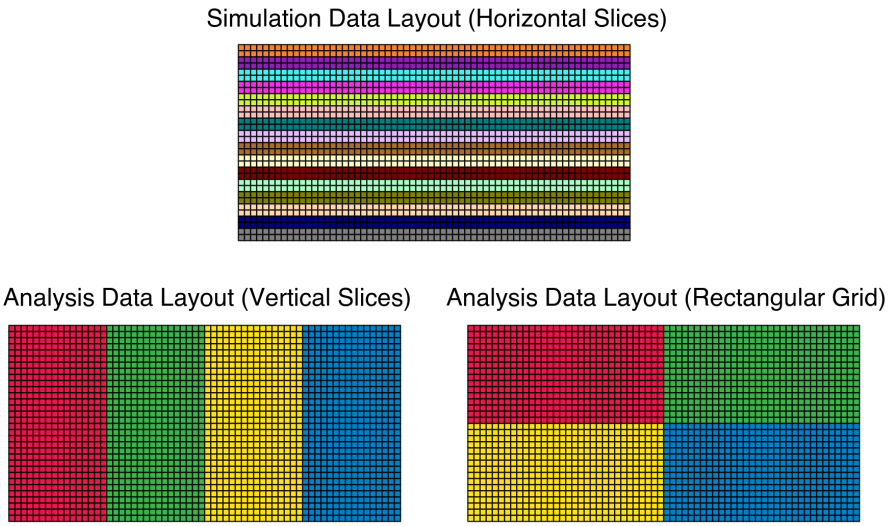

Figure 4: Data layout illustration for distributing a 64x32 overall data domain amongst the processes of simulation and analysis applications. For this illustration, the simulation uses 16 processes and splits data into horizontal rows. The analysis application uses 4 processes and can either split data into vertical columns or a rectangular grid. Colors represent the data needed by each process in its respective application.

production simulation and visualization / analysis codes. The results below all report on "effective bandwidth", which we use as a term to mean the speed at which it takes to transfer and reorganize data between the two applications.

For our experiments, we used between 8 and 128 processes for the simulation and between 2 and 32 processes for the visualization / analysis application. While these scales may seem a bit on the small end at first glance, these tests were designed to isolate the transfer of data between distributed memory applications. In many HPC settings, only a subset of nodes have I/O capabilities. For example, Argonne National Laboratory's Blue Gene/Q system, Mira, has 1 I/O node for every 128 compute nodes.

The overall data size for a single time step used in our experiments ranged from 15.625 MB to 4000 MB. Again, these sizes may appear on the smaller end at first. However, the data transferred for analysis is only a fraction of what's needed for computation in the simulation. Additionally, simulations typically run for numerous time steps. The LBM simulation that our experiments were modeled after requires an order of magnitude more data for computation than is streamed out for analysis and often requires for more than 100,000 time steps, thus reaching terabyte or even petabyte scales.

6.1. Hardware Configuration

To perform tests, we used Argonne National Laboratory's Cooley cluster (126 nodes, each with 12 cores, and a FDR Infiniband interconnect) to run the

simulation application. For one set of tests, we used a dual-CPU Linux workstation that is connected to a tiled display wall (capable of running 40 tasks in parallel) to run the analysis application. The two resources were networked over 40 Gbps Ethernet. These resources represent authentic devices that leverage in transit streaming and have been used to stream and render ultra high resolution visualizations. On a second set of tests, we used Oak Ridge National Laboratory's former Keeneland Initial Delivery System cluster (120 nodes, each with 12 cores, and a QDR Infiniband interconnect) which is now housed at Georgia Tech to run the analysis application. The two resources were networked with 1 Gbps Ethernet to each analysis node. These resources represent authentic devices that could be used for in transit pattern recognition routines.

6.2. Effective Bandwidth Tests

Our first test investigated scaling the number of processes in both the simulation and visualization / analysis applications while maintaining an overall data size of 4000 MB. This resulted in smaller and smaller chunks of data streamed to and from each process as the number of processes is scaled up. We kept the ratio of processes writing data to processes reading data constant at 4:1. The A panels of Figure 5 both show that using the write block with local redistribution technique performs similarly to the bounding-box technique at small scales, but performs much better at larger scales. It is also worth noting that effective bandwidth increases with scale using the write block with local redistribution technique, but decreases with scale using the bounding-box technique.

Second we investigated only scaling the number of processes in the simulation while keeping the number of processes in the visualization / analysis application constant. Again, overall data size was kept at 4000 MB. This tested the effects of varying the ratio of the number of writers to number of readers. The B panels of Figure 5 both show that effective bandwidth remains fairly constant regardless of the writer to reader ratio, but again shows that using the write block with local redistribution technique performs better than the bounding-box technique.

Next we investigated scaling the data size while keeping the number of processes for both the simulation and visualization / analysis constant at 128 and 32 respectively. The C panels of Figure 5 both show that effective bandwidth improves with increased data size for both techniques. However, the write block with local redistribution technique not only performs better at all scales tested, it also demonstrates better scalability than the bounding-box technique.

Last we investigated weak scaling by keeping the amount of data per process constant at 31.25 MB and scaling the overall data size with the number of processes in both the simulation and visualization / analysis applications. We used a ratio of 4:1 for the number of processes writing data to the number of processes reading data. The D panels of Figure 5 both show that using the write block with local redistribution technique scales effective bandwidth quite well as data per process remains constant, whereas using the bounding-box technique causes the effective bandwidth to remain constant or even decrease.

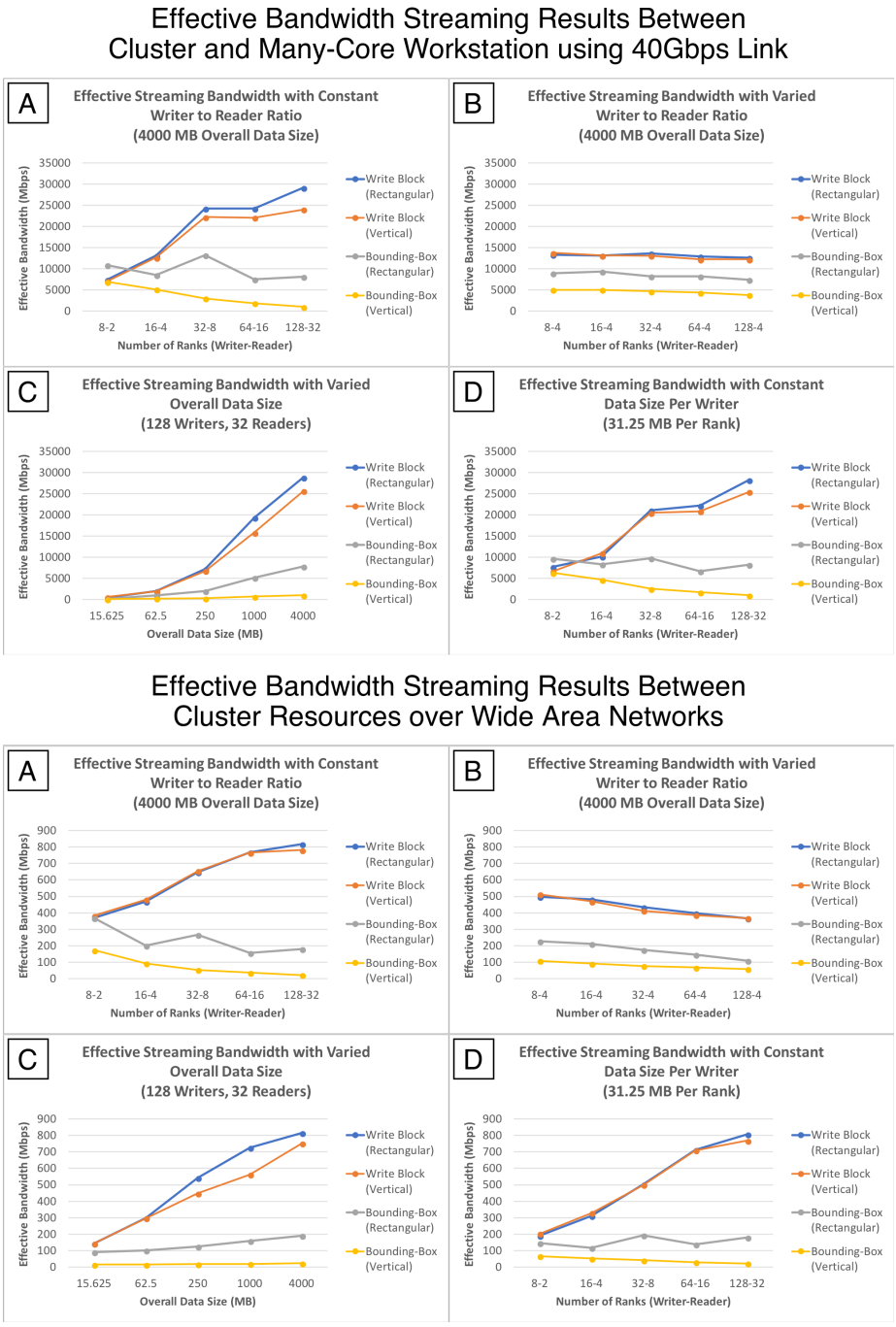

Figure 5: Effective bandwidth results for in transit streaming and reorganization of data from a cluster resource to a single many-core workstation and to a separate cluster resource.

6.3. Data Layout Dependence

One other interesting difference observed in all four test cases is the difference (or lack-there-of) between the visualization / analysis application using vertical and rectangular grid layouts. When using the bounding-box technique, the rectangular grid layout performs noticeably better than the vertical layout. This is not surprising, since transferring data from a sequence of horizontal slices to a sequence of vertical slices requires each process in the visualization / analysis application to request a narrow chunk of data from every single process in the simulation. In contrast, when transferring data from a sequence of horizontal slices to a rectangular grid, each process in the visualization / analysis application can request a larger chunk of data from only a subset of processes in the simulation. Interestingly, this difference is not usually as noticeable when using the write block with local redistribution technique. Since the streaming portion of the data transfer is the same regardless of final data layout, the only differences occur during the redistribution. Since the redistribution phase takes place internally within the visualization / analysis resource, it only contributes to a small amount of the overall data movement time.

7. Conclusion

We have presented research on optimizing data transfer between distributed memory applications for in transit analysis. We have also presented research on automating the redistribution of dynamic data in distributed memory applications. Our main contributions are the design and implementation of a new technique to handle in transit streaming between applications with different data layouts, the development of a library that abstracts the necessary complexities for redistributing data inside a distributed memory application, and the evaluation of our technique against the prior state-of-the-art. Results showed that our technique of streaming data as is, then redistributing it within the visualization / analysis application, both outperforms and shows better scalability than to stream data in the final layout needed by the receiver.

Increasing the effective bandwidth of streaming data between distributed memory applications will decrease the runtime cost associated with performing concurrent analysis while a simulation executes. While it may be possible in some simulations to overlap computation and data streaming, this would require partitioning the cores within a node. One core per node would need to be dedicated for streaming the current time step while the remaining cores compute the next time step. Thus fewer cores would be available for computation by the simulation. Depending on the properties of the simulation (data size vs. computation time for a single time step), this approach may be favored over alternating between computation and data streaming. Even when dedicating one core for streaming, using our in transit data transfer technique with higher effective bandwidth would improve how much of the data transfer could be overlapped with computing the next time step.

For future work, it is our goal to package local redistribution of data after being received in transit as an optional plug-in for the next major release of

ADIOS. We are also investigating introducing feedback mechanisms for analysis or visualization applications to send data back to the simulation in order to steer high performance applications.

Acknowledgment

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] L. B. N. Laboratory, Doe high performance computing operational review (hpcor): Enabling data-driven scientific discovery at doe hpc facilities, Tech. rep. (2014).

[2] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C. Johnson, R. Mount, V. Sarkar, V. White, D. Williams, Synergistic challenges in data-intensive science and exascale computing: Doe ascac data subcommittee report, Tech. rep. (Mar 2013).

[3] J. Kress, S. Klasky, N. Podhorszki, J. Choi, H. Childs, D. Pugmire, Loosely coupled in situ visualization: A perspective on why it's here to stay, in: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV2015, ACM, New York, NY, USA, 2015, pp. 1–6. doi:10.1145/2828612.2828623.
URL http://doi.acm.org/10.1145/2828612.2828623

[4] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld, M. E. Papka, S. Klasky, Examples of in transit visualization, in: Proceedings of the 2Nd International Workshop on Petascal Data Analytics: Challenges and Opportunities, PDAC '11, ACM, New York, NY, USA, 2011, pp. 1–6. doi:10.1145/2110205.2110207.
URL http://doi.acm.org/10.1145/2110205.2110207

[5] T. Marrinan, J. A. Insley, S. Rizzi, F. Tessier, M. E. Papka, Automated dynamic data redistribution, in: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 1208–1215. doi:10.1109/IPDPSW.2017.17.

[6] V. Vishwanath, M. Hereld, V. Morozov, M. E. Papka, Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems, in: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2011, pp. 1–11. doi:10.1145/2063384.2063409.

[7] D. Morozov, Z. Lukic, Master of puppets: Cooperative multitasking for in situ processing, in: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC '16, ACM, New York, NY, USA, 2016, pp. 285–288. doi:10.1145/2907294.2907301.
URL http://doi.acm.org/10.1145/2907294.2907301

[8] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, 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, CLADE '08, ACM, New York, NY, USA, 2008, pp. 15–24. doi:10.1145/1383529.1383533.
URL http://doi.acm.org/10.1145/1383529.1383533

[9] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, N. Podhorszki, Flexpath: Type-based publish/subscribe system for large-scale science analytics, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014, pp. 246–255. doi:10.1109/CCGrid.2014.104.

[10] B. Perry, M. Swany, Improving mpi communication via data type fission, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, ACM, New York, NY, USA, 2010, pp. 352–355. doi:10.1145/1851476.1851528.
URL http://doi.acm.org/10.1145/1851476.1851528

[11] F. Kjolstad, T. Hoefler, M. Snir, Automatic datatype generation and optimization, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, ACM, New York, NY, USA, 2012, pp. 327–328. doi:10.1145/2145816.2145878.
URL http://doi.acm.org/10.1145/2145816.2145878

[12] D. Morozov, T. Peterka, Block-parallel data analysis with diy2, in: 2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV), 2016, pp. 29–36. doi:10.1109/LDAV.2016.7874307.

[13] A. Esnard, N. Richart, O. Coulaud, A steering environment for online parallel visualization of legacy parallel simulations, in: 2006 Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications, 2006, pp. 7–14. doi:10.1109/DS-RT.2006.7.

[14] D. Schroeder, Fluid dynamics simulation.
URL https://physics.weber.edu/schroeder/fluids/

[15] NVIDIA Corporation, Quadro sync ii: User guide (2016).
URL http://images.nvidia.com/content/quadro/product-literature/user-guides/Quadro-Sync-II-User-Guide.pdf