2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Automated Dynamic Data Redistribution

Thomas Marrinan[1], Joseph A. Insley[1], Silvio Rizzi[1], François Tessier[1], and Michael E. Papka[1,2]
[1]Argonne National Laboratory
Lemont, IL, USA
{tmarrinan, insley, srizzi, ftessier, papka}@anl.gov
[2]Northern Illinois University
DeKalb, IL, USA
papka@niu.edu

***Abstract*—High-performance distributed memory applications often load or receive data in a format that differs from what the application uses. One such difference arises from how the application distributes data for parallel processing. Data must be redistributed from how it was laid out by the producer to how the application needs the data to be laid out amongst its processes. In this paper, we present a large-scale distributed memory library, provided to developers in an easily integrated API, for automating data redistribution in MPI enabled applications. We then present the results of two scientific computing use cases to evaluate our library. The first use case highlights how dynamic data redistribution can greatly reduce load time when reading three-dimensional medical imaging data from disk. The second use case highlights how dynamic data redistribution can facilitate in-transit analysis of computational fluid dynamics, which results in smaller data output size and faster time-to-discovery.**

Keywords-data redistribution; distributed memory; MPI; scalable algorithms; scientific applications

I. 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. Current trends, elucidated in the 2014 DOE High Performance Computing Operational Review (HPCOR), show that the cost of data relative to computation is growing [5]. This means that HPC applications must dedicate more time and resources to loading and saving data, which results in less utilization for scientific computation. Findings from HPCOR also state that science use cases produce and use a wide variety of data with different access patterns. Efficient data transformations can help address these issues by load-balancing data accesses and supporting a variety of access patterns.

Existing research on data transformation to enable compatibility between applications primarily surrounds how data is organized within a process (e.g. storing multivariate data in a planar configuration or interleaved configuration) [15]. When it comes to handling the distribution of data between processes, most solutions are application specific and not easily integrated into other software packages [1,19]. We identified the flexible redistribution of data between processes as an open research area. Therefore, we aimed to automate the redistribution of dynamic data with minimal instructions in order to reduce the burden placed on application developers.

In this paper, we address handling layout differences between data producers and data consumers, as well as provide methods for load-balanced parallel data management routines. We have developed the Dynamic Data Redistribution (DDR) library, which can be seamlessly integrated into existing scientific and engineering codebases with three simple function calls. 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 in respect to the overall data domain.

DDR was designed to achieve two majors goals: 1) reduce overall application disk read and write time by facilitating load-balanced I/O, and 2) transform inter-process data layout on-the-fly to enable various data access patterns. By accomplishing these goals, DDR serves as an efficient technique for accessing data that is stored in a fashion that is not directly compatible with the application wishing to use it.

We have evaluated DDR with two authentic scientific use cases. First, we highlight how DDR can facilitate load-balanced parallel I/O when reading a stack of TIFF images to perform parallel visualization of three-dimensional medical imaging data. In this case, DDR enables the visualization application to read the image files from disk in a load-balanced fashion. After the images are read in, individual pixels are redistributed so that each process can properly access the data it needs to create the final visualization. Results from this use case show that DDR can lead to nearly a 25X I/O speed-up compared to the traditional parallel I/O technique previously used.

For the second use case, we highlight how DDR can enable real-time analysis of a computational fluid dynamics (CFD) simulation. In this case, a visualization application can use DDR to enable on-the-fly transformation of in-transit data received from the CFD simulation. This transformation allows data to be redistributed from how it was laid out in the simulation to how it needs to be laid out in the visualization application. Rendering images while the simulation is running allows the scientists to monitor its progress and gain situational awareness, which can lead to a faster time-to-discovery. Additionally, saving rendered images to disk, rather than the raw simulation data, results in significantly smaller data output size as well as faster I/O [12].

978-1-5386-3408-0 2017

DOI 10.1109/IPDPSW.2017.17

II. Related Work

Parallel distributed memory applications are traditionally written using MPI. However, many common design patterns are not sufficiently abstracted, causing unnecessary complexities for application developers. Kaushik et al. [8] talk extensively about the issue of accessing distributed data. They write: "Different phases of a program vary in their access patterns to a shared array and a different data distribution of the array is often best suited for each phase… Scientific libraries are tuned to provide peak performance for a fixed set of distributions for the input arrays. These distributions may not conform with the distributions of the actual parameters, leading to performance degradation." Our DDR library complements the current field of work that attempts to address this problem.

A. Data Transformations

Perry and Swany [13] developed a method called *data type fission* that segregates transmitted fields from non-transmitted fields for sending and receiving data between processes. This process enables MPI applications to efficiently 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 improvement in performance in communication heavy applications. 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. [9] 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 has similar goals, but for inter-process data transmission in addition to staging the data for efficient retrieval.

Sharma et al. [14] investigated *array interleaving*, a data transformation technique that combines elements from multiple arrays in continuous memory. This transformation can reduce the number of memory accesses and lead to greater computational efficiency due to spatial locality of data. Experimental results also show a significant decrease in memory energy when using array interleaving. While our research looks at a different aspect of data transformations, this work could be utilized along side of DDR to efficiently stage data for redistribution.

B. Data Redistribution

DIY2 [9] provides 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. One feature of DIY2 is to abstract the communication patterns for exchanging data between processes when running on a distributed memory system. However, this abstraction is intended for iterative processes requiring information from local neighbors or global reductions. In contrast, our work is designed to facilitate the staging of data onto the proper processes for computation.

Esnard et al. [5] present a steering framework for parallel simulations. They enable the coupling of a visualization system with a running HPC simulation, allowing users to view and modify the simulation as it runs. Since data on the HPC simulation likely has 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 work was designed to accomplish a similar redistribution of data, but without relying on external network communication.

C. Scientific Applications

Our research on automating dynamic data redistribution can lead to efficiencies in a variety of common large-scale scientific and engineering workflows. One common technique is visualizing 3D data using volume rendering [7,18]. In order to handle ultra high-resolution data sets, research has been conducted on performing distributed volume rendering using many compute nodes with many GPUs [4,14,16]. However, loading large 3D data sets into common distributed rendering packages can be time consuming. One such software is ParaView [2], which requires preprocessing data into a custom format in order to leverage parallel data distribution. Our research could be integrated into such packages to enable on-the-fly conversion from data formats that are laid out in an otherwise incompatible fashion.

Another scientific workflow that has increased demand is in-situ analysis of running simulation. One way to perform in-situ analysis is to have two separate resources, one dedicated to the simulation and the other dedicated to the analysis. In this scenario data must be sent in-transit from one distributed memory application to another. ADIOS [10] and GLEAN [17] are two such frameworks that enable this type of data movement. However, there is still potential for data needing to be redistributed once it arrives on the analysis resource due to differences in the number of processes in each application or how the applications expect data to be laid out. Therefore integration of DDR into analysis applications receiving simulation data in-transit can facilitate the efficient processing of real-time data.

III. Methods

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

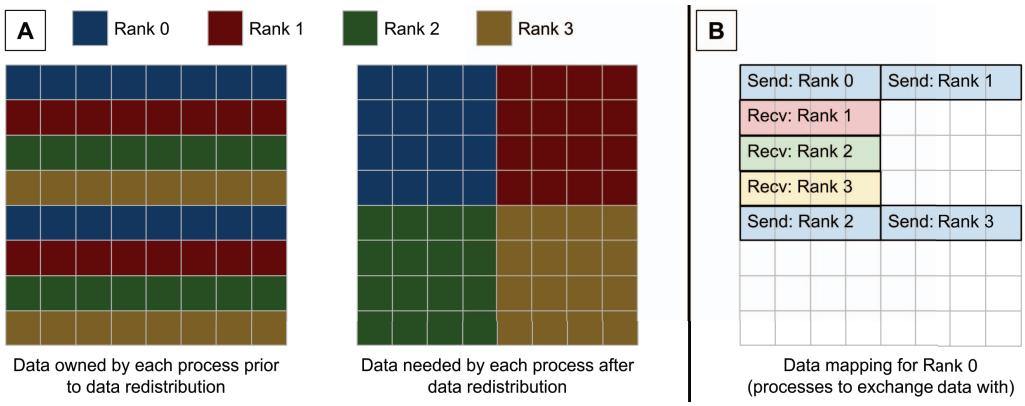

Figure 1. 2D data redistribution in a distributed memory application with four processes. Panel A – data layout before and after redistribution. The left grid shows each process in the application owning two separate 8x1 chunks of data in the overall domain prior to data redistribution. The right grid 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 rank 0 needs to receive from other ranks

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 reference the following example, E1, to help illustrate the concepts of dynamic data movement enabled by DDR. E1 is a distributed memory application with four processes operating on a two-dimensional grid with an overall domain of 8x8. Each process owns two separate rows, but needs the data located in one of the quadrants of the overall domain. Figure 1 illustrates the setup and data movement needs for E1. In addition to providing a visual overview of DDR, we outline how the library can be easily integrated into an application. 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 $data_{need}$ represents the quadrant of data needed by each process after data redistribution.

Algorithm 1 Sample use of the DDR library.

Input: $data_{own}$
Output: $data_{need}$

```
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)
```

A. 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.

B. 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 I enumerates E1's parameter values for each process using the pseudocode from Algorithm 1.

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.

TABLE I. 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.

	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

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.

C. 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 at https://xgitlab.cels.anl.gov/fl/ddr/.

IV. USE CASES

Automating the redistribution of dynamic data in distributed memory applications can have significant impact on a number of authentic large-scale scientific and engineering applications. This section highlights two use cases that utilize DDR to improve efficiency and enable real-time analysis with reduced storage needs. The first use case surrounds efficiently loading ultra high-resolution 3D medical images in parallel. The second use case surrounds enabling real-time parallel visualization of HPC simulations via in-transit streaming.

A. Parallel Visualization of 3D Medical Images

The medical images generated from Magnetic Resonance Imaging (MRI), Computed Tomography (CT), and Positron Emission Tomography (PET) are able to generate ultra high-resolution three-dimensional data sets. These data sets are typically generated by capturing a series of slices through the medium being imaged. The slices are then saved in a standard image format, such as TIFF. In order to visualize the three-dimensional volume, the series of 2D images can be stacked on top of each other and rendered in a process known as direct volume rendering (DVR) [7,18]. Figure 2 depicts a 3D image of a primate tooth, showing both individual 2D images stacked next to each other, and visualized using DVR.

Unfortunately, GPUs have limited resources and

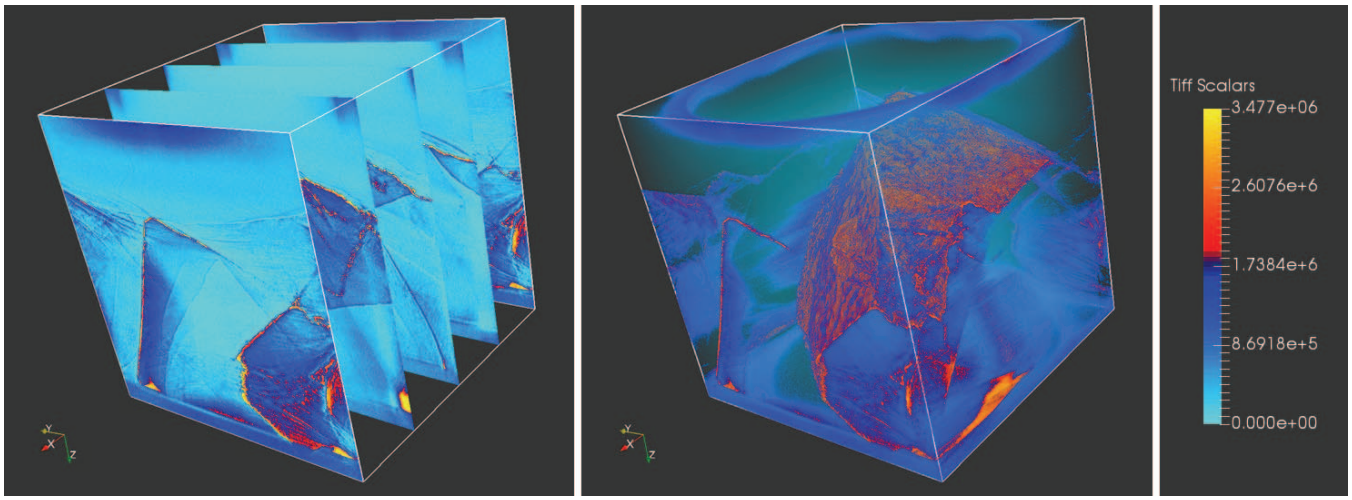

Figure 2. Visualization of a 3D TIFF stack representing a primate tooth. Left – individual 2D images stacked next to each other (only a small subset shown for clarity). Middle – volume visualization using DVR of the 3D data set. Right – colormap used to render the primate tooth image data.

therefore cannot render data sets that are too large to fit in their graphics memory (e.g. a GPU with 4GB of usable graphics memory could handle a maximum volume of 2048x2048x1024 1-byte ints or 1024x1024x1024 4-byte floats). When the data set is too large for a single GPU to render, parallel visualization techniques must be employed to use multiple GPUs on multiple machines.

In order to perform efficient distributed memory DVR, the entire volume is broken into equally sized boxes that are as close to cubes as possible. This leads to each process only needing data from a subset of images in the entire series. It also leads to each process only needing a subset of pixels from each image it needs data from. Unfortunately, common 2D image formats such as TIFF require a program to decode and extract the entire image from file, even if the application only needs the values of a few pixels. Reading and decoding entire images on each process leads to many processes loading the same image. It also leads to each process throwing away much of the data it spends effort on to extract.

To address the time-consuming and unnecessary overheads, we have integrated our DDR library into the loading of a series of TIFF images. The total number of images in the TIFF series can be equally divided amongst the processes regardless of what chunks of data each process eventually needs. Our DDR library is then used to automatically redistribute pixel data from the processes that read and decoded each image to the processes that need partial images to properly perform distributed memory DVR. Using DDR results in each TIFF image only being read by one process and avoids reading and decoding pixel data that would just be thrown away.

Our collaborators have gathered ultra high-resolution three-dimensional CT images on Argonne National Laboratory's Advanced Photon Source (APS). Two such data sets are a 2028x2048x2048 volume of a primate tooth stored as a series of 32-bit grayscale TIFF images, and a 4096x2048x4096 volume of a mouse brain stored as a series of 8-bit grayscale TIFF images.

In order to perform benchmark tests to evaluate the performance of DDR, we generated an artificial TIFF data that had the largest resolution and bit-depth of our authentic data sets. Therefore our artificial data set consisted of 4096 TIFF images, each with a resolution of 4096x2048 and 32-bit grayscale color. This resulted in a volume with a total data size of 128GB. We have utilized Argonne National Laboratory's visualization cluster, Cooley, to load the artificial TIFF image series for parallel DVR. Cooley has 126 nodes, each node has two GPUs, and each GPU has 12GB of graphics memory. Therefore, a minimum of 11 GPUs (6 nodes) would be required to load the entire data set. Cooley nodes are interconnected with a FDR Infiniband CLOS network. Each node has a single 56 Gbps link available for MPI communications.

To evaluate load time, we tested three different cases - without DDR, using DDR where each process reads and decodes images assigned from the series round-robin, and using DDR where each process reads and decodes images assigned from the series in consecutive chunks. The difference between the latter two cases is that the round-robin assignment requires each image to be a separate chunk to redistribute with DDR, whereas consecutive images can be grouped together into a single chunk to redistribute with DDR. We ran these tests at four different scales - always splitting the volume into an equal number of chunks in each dimension – 3^3 (27) processes, 4^3 (64) processes, 5^3 (125) processes, and 6^3 (216) processes. Each test was repeated 10 times.

TABLE II. TIFF LOAD TIME RESULTS.

Number of Processes	No DDR	DDR (Round-Robin)	DDR (Consecutive)
3^3 (27)	283.0 ± 1.7 sec	39.3 ± 0.2 sec	49.2 ± 0.2 sec
4^3 (64)	204.6 ± 1.2 sec	18.9 ± 0.2 sec	18.9 ± 0.1 sec
5^3 (125)	188.2 ± 1.2 sec	11.1 ± 0.1 sec	10.4 ± 0.1 sec
6^3 (216)	165.3 ± 5.9 sec	9.7 ± 0.4 sec	6.6 ± 0.0 sec

Results, enumerated in Table II, show that using DDR can significantly reduce the load time of a stack of TIFF images. This is due to the reduction in overall image reads needed by the application. These results mean that the overhead associated with transmitting data between processes is more than offset by the file reading efficiencies gained. Using DDR with one consecutive chunk on 6^3 (216) processes led to the maximum improvement in performance – an average of 6.6 seconds compared to an average of 165.3 seconds when not using DDR (24.9X speed up).

Since we ran tests at various scales, we were able to show that DDR achieves strong scaling. Figure 3 shows this strong scaling of loading TIFF images in parallel. The two

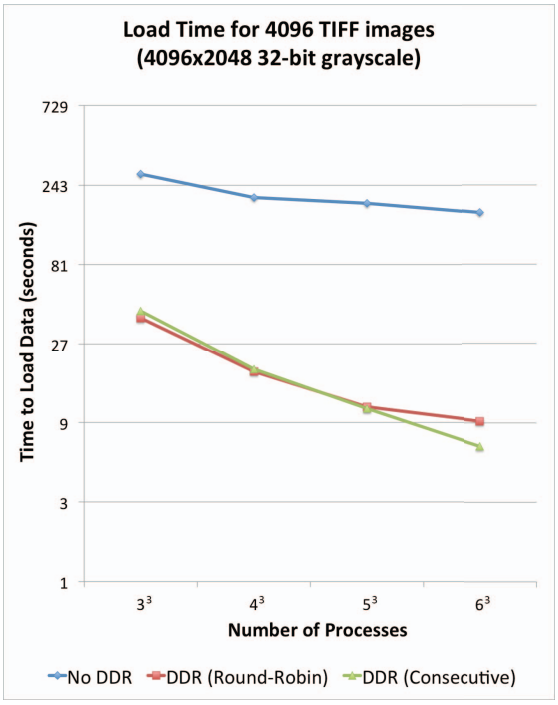

Figure 3. Strong scaling results for parallel TIFF loading. Using DDR with many small chunks resulting from a round-robin distribution of file reading and using DDR with one large chunk resulting from reading consecutive files both exhibit strong scaling. However, DDR with one large chunk results in maximum performance at larger scales.

TABLE III. COMMUNICATION SCHEDULING OF MPI_ALLTOALLW ACCORDING TO THE DATA REDISTRIBUTION TECHNIQUE.

Number of Processes	DDR (Consecutive)		DDR (Round-Robin)	
	Rounds	*Data Size (MB)*	*Rounds*	*Data Size (MB)*
3^3 (27)	1	4315.12	152	30.81
4^3 (64)	1	1920.00	64	31.50
5^3 (125)	1	1006.63	33	31.74
6^3 (216)	1	589.95	19	31.85

data redistribution techniques we implemented (consecutive and round-robin) are plotted along with a baseline case that does not use DDR. Since we increased the number of processes in a cubic fashion, the time is depicted with a $\log_3$ scale.

Beyond the comparison with the baseline case, we can notice performance differences between the two techniques for data redistribution. At small scale, the round-robin method outperforms the consecutive method by 20%, while this trend reverses at larger scales with the consecutive approach 32% faster on 216 processes. This behavior can be explained by the trade-off between network contention and communication scheduling of each technique, as shown in Table III. This table presents, for each number of processes tested, the number of rounds (calls to `MPI_Alltoallw`) to perform and the data size sent and received per process per round. Using the consecutive approach, the data is contiguous in memory requiring only one round with a large amount of data per process (up to 4.3 GB). This creates network contention on the single 56 Gbps link available per node. When distributing images in a round-robin way, the number of calls to `MPI_Alltoallw` is equal to the total number of images divided by the number of processes. However, despite the overhead of these consecutive collective operations, the data size per process per round remains constant and allow for full utilization of the network bandwidth capacity. A strong scaling of our experiments highly reduces the data size per process in the consecutive method, optimizing at the same time the network bandwidth usage. On the other hand, the round-robin method still pays the price of the overhead of multiple calls of `MPI_Alltoallw` with a larger number of processes.

B. *In-Transit Streaming*

Performing data analysis on intermediate results of a running high-performance computing application has several advantages. It produces an I/O cost savings by performing analysis without the need to write to or read from disk. This in turn enables a higher sampling rate for analysis, which can elucidate complex behaviors occurring at fine temporal resolution. Additionally, specialized hardware such as high-bandwidth networks and GPUs can be leveraged for analysis at the same time as the CPUs are computing simulation results. Our second DDR use case focuses on in-transit analysis, where data is streamed from a distributed memory computational resource performing a simulation to a separate distributed memory resource responsible for performing analysis. Data is sent from M simulation ranks to N analysis ranks. After receiving intermediate data, the analysis resource leverages our library to redistribute data from how it was laid out in the simulation application to how it needs to be laid out for the application performing analysis.

We have used a simple Lattice Boltzmann method (LBM) for computing fluid flows in a two-dimensional space [3]. The density and velocity of the fluid is broken into a regular grid of floating point values. In each iteration of the simulation, every cell updates its value by simulating particles streaming and collisions. Certain cells, including the edges, are kept at fixed values. For our evaluation tests, we place a barrier inside the domain that forces the fluid to flow around it, creating more turbulent flow patterns.

The simulation application splits the data into slices to distribute between ranks. This was done so as to minimize the number of ranks 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 ranks at most, the neighbors with data directly above and below.

For analysis, we created a simple visualization application, which would take the 2D array of floating point values and apply a colormap in order to create an image. In this use case, rotational velocity was chosen as the variable of interest. The visualization enables users to quickly determine flow patterns throughout the domain. We ran our tests on Argonne National Laboratory's visualization cluster, Cooley, with 128 processes for the LBM simulation and 32 processes for the visualization application. While 32 is a factor of 128, resulting in an equal number of simulation ranks streaming data to each analysis rank, in-transit streaming can be achieved without uniform mapping. Figure 4 depicts the M-to-N parallel data streaming for the LBM fluid dynamics example.

The visual analysis application expects data to be split into a grid that was as close to square as possible (given the total number of analysis ranks). Therefore, the data being received from the simulation was laid out in a different manner than the analysis application required. Using our DDR library ensures that data ends up in the proper location for analysis. Additionally, the redistribution of data must happen each time the analysis application receives new data from the simulation. However, the mapping of what piece of the overall domain each rank receives from the simulation

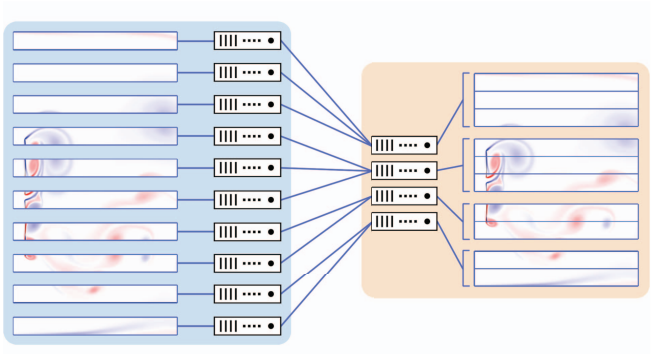

Figure 4. Parallel data streaming of 2D LBM data. This illustration shows 10 simulations ranks streaming data to 4 analysis ranks. The first two analysis ranks receive data from 3 simulation ranks, whereas the last two analysis ranks receive data from 2 simulation ranks.

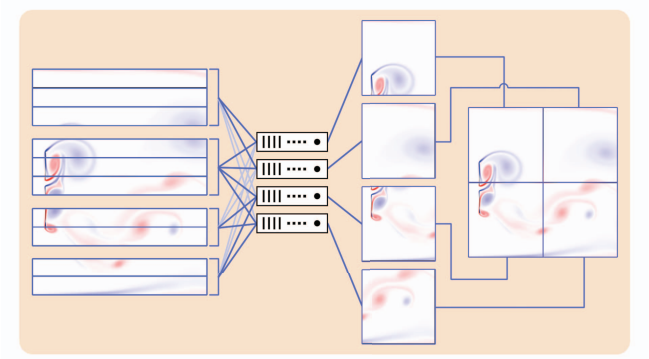

Figure 5. Data redistribution by the analysis application. The DDR library automatically reorganizes the data to fit the layout specified by the analysis application. For the 2D LBM fluid flow example, incoming slices of data were redistributed into nearly square rectangles.

and what piece each rank needs to have in order to render the proper visualization remains constant. Figure 5 shows how data is automatically redistributed inside the analysis application by our library.

Once data had been redistributed, the image could be rendered using a blue-white-red colormap. Rendered images were saved to disk rather than the raw data, which compressed the data resulting in smaller data output. Table IV shows results from running the 2D LBM fluid flow simulation at different grid sizes. The simulation ran for 20000 iterations, outputting data every 100th iteration. Raw data was saved to disk directly from a 4-byte float array. Processed data was generated from the analysis application that visualizes the streamed raw data and saves to disk as a compressed JPEG image. The values in Table IV represent one variable of interest (vorticity) for both raw and processed data. However, many other variables (e.g. velocity, density, etc.) are required for computation and could also be streamed and rendered, achieving similar data compression.

While using visualization to render an image results in a loss of data, it can yield a much higher output frequency without needing more storage capacity. This tradeoff could be beneficial in many cases, and may eventually become necessary as we approach exascale computing. Additionally, it is possible to do both raw data output and in-transit analysis at different frequencies. For example, in our LBM fluid flow use case, we could still output raw data every 100 iterations, but additionally stream data every 10 iterations for visual analysis. This would increase temporal resolution 10-fold, but only marginally increase data storage size. Scientists could use the additional temporal information to detect complex phenomena that were otherwise missed and direct future simulations.

TABLE IV. DATA SIZE ON DISK WITH AND WITHOUT IN-TRANSIT STREAMING. DATA WAS SAVED 200 TIME STEPS DURING THE SIMULATION.

Grid Dimensions	Raw Data Size	Processed Data Size	Data Reduction
3238 x 1295	3.2 GB	19.9 MB	99.38%
6476 x 2590	12.8 GB	61.0 MB	99.52%
12952 x 5180	51.2 GB	217.8 MB	99.57%
25904 x 10360	204.7 GB	830.9 MB	99.59%

V. CONCLUSION

We have presented research on automating the redistribution of dynamic data in distributed memory applications. Our main contributions are the development of a library that abstracts the necessary complexities for redistributing data inside a distributed memory application and highlighting the benefits of such a library through two authentic scientific use cases. Experimental results show that automated dynamic data redistribution can significantly reduce file load time when dealing with three-dimensional medical images and enable analysis applications to properly handle and process real-time data from a running simulation.

To reduce the burden of integrating our library into existing simulations or analysis applications, we have consolidated our code into three public function calls. Application developers simply need to initialize the library, declare what data each process owns and what data each process wants, then ask the library to exchange data between processes. Since there is little coding required to integrate automated data redistribution into existing simulations and analysis tools, we foresee a wide range of large-scale scientific and engineering applications benefiting from integration of our DDR library. In addition to large-scale visualization, such as the use cases highlighted in this paper, DDR can be utilized in any distributed memory application when loading data that was produced with a different layout than the distribution needed for parallel processing.

While highly useful, DDR does have certain limitations. First, it only supports 1D, 2D, or 3D arrays stored in continuous memory with each element having a fixed size. Second, the library requires additional memory usage to store both the data each process owns prior to redistribution and the data each process receives after redistribution. Therefore, certain applications that are already memory bound may need to utilize a greater number of nodes in an HPC system.

For future work, we would like to extend the capabilities of the DDR library. First, we would like to optimize the MPI communication. Currently the redistribution is handled via `MPI_Alltoallw` calls, which may not be most efficient when data only needs to be sent and received from a subset of other processes. By looking at how an application sets up the data mapping, we could determine if data only needs to be redistributed to a few neighboring processes and use direct send and receive calls to improve efficiency. Finally, we would like to add support for more data patterns, so application developers could redistribute more complex structures organized with arbitrary data layout.

ACKNOWLEDGMENT

We would like to thank our collaborators, Narayanan Kasthuri (Argonne National Laboratory / University of Chicago), Callum Ross (University of Chicago), and Carmen Soriano (Argonne National Laboratory), who provided data collected at Argonne National Laboratory's APS. 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] F. Affinito and C. Cavazzoni, “FFT data distribution in plane-waves DFT codes. A case study from Quantum ESPRESSO,” Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016), pp. 212, 2016.

[2] J. Ahrens, B. Geveci, and C. Law, “ParaView: An End-User Tool for Large Data Visualization, Visualization Handbook,” Elsevier, 2005.

[3] S. Blair, C. Albing, A. Grund, and A. Jocksch, “Accelerating an MPI Lattice Boltzmann code using OpenACC,” Proceedings of the Second Workshop on Accelerator Programming using Directives (WACCPD '15), article 3, 2015.

[4] B. Corrie and P. Mackerras, “Parallel volume rendering and data coherence,” Proceedings of the 1993 symposium on Parallel rendering (PRS '93), pp. 23-26, 1993.

[5] Department of Eneergy, “DOE High Performance Computing Operational Review (HPCOR): Enabling data-driven scientific discovery at DOE HPC facilities,” 2014.

[6] A. Esnard, N. Richart and O. Coulaud, “A Steering Environment for Online Parallel Visualization of Legacy Parallel Simulations,” Proceedings of the Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '06), 2006.

[7] K. A. Frenkel, “Volume rendering,” Communications of the ACM, vol. 32, no. 4, pp. 426-435, April 1989.

[8] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan, “An approach to communication-efficient data redistribution,” Proceedings of the 8th international conference on Supercomputing (ICS '94), pp. 364-373, 1994.

[9] F. Kjolstad, T. Hoefler, and M. Snir, “Automatic datatype generation and optimization,” Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12), pp. 327-328, 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),” Proceedings of the 6th international workshop on Challenges of large applications in distributed environments (CLADE '08), pp. 15-24, 2008.

[11] D. Morozov and T. Peterka, “Block-Parallel data analysis with DIY2,” Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV), 2016.

[12] P. O'Leary, J. Ahrens, S. Jourdain, S. Wittenburg, D. H. Rogers, and M. Petersen, “Cinema image-based in situ analysis and visualization of MPAS-ocean simulations,” Parallel Computing, vol. 55, pp. 43-48, July 2016.

[13] B. Perry and M. Swany, “Improving MPI communication via data type fission,” Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10), pp. 352-355, 2010.

[14] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, and V. Vishwanath, “Performance modeling of vl3 volume rendering on GPU-based clusters,” Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization (PGV '14), pp. 65-72, 2014.

[15] N. Sharma, P. R. Panda, F. Catthoor, P. Raghavan, and T. Vander Aa, “Array interleaving—An energy-efficient data layout transformation,” ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 20, no. 3, article 44, June 2015.

[16] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens, “Multi-GPU volume rendering using MapReduce,” Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10), pp. 841-848, 2010.

[17] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, “Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems,” Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11), article 19, 2011.

[18] L. Westover, “Interactive volume rendering,” Proceedings of the 1989 Chapel Hill workshop on Volume visualization (VVS '89), pp. 9-16, 1989.

[19] G. Zhang, J. Shu, W. Xue, and W. Zheng, “SLAS: An efficient approach to scaling round-robin striped volumes,” ACM Transactions on Storage (TOS), vol. 3, no. 1, article 3, March 2007.