Eurographics Symposium on Parallel Graphics and Visualization (2020)
S. Frey, J. Huang, F. Sadlo (Editors)

Improving Performance of M-to-N Processing and Data Redistribution in In Transit Analysis and Visualization

B. Loring [1], M. Wolf [2], J. Kress [2], S. Shudler [3], J. Gu [1], S. Rizzi [3], J. Logan [2], N. Ferrier [3], and E. W. Bethel [1]

[1] Lawrence Berkeley National Laboratory – https://www.lbl.gov
[2] Oak Ridge National Laboratory – https://www.ornl.gov
[3] Argonne National Laboratory – https://www.anl.gov/

Abstract

In an in transit setting, a parallel data producer, such as a numerical simulation, runs on one set of ranks M, while a data consumer, such as a parallel visualization application, runs on a different set of ranks N. One of the central challenges in this in transit setting is to determine the mapping of data from the set of M producer ranks to the set of N consumer ranks. This is a challenging problem for several reasons, such as the producer and consumer codes potentially having different scaling characteristics and different data models. The resulting mapping from M to N ranks can have a significant impact on aggregate application performance. In this work, we present an approach for performing this M-to-N mapping in a way that has broad applicability across a diversity of data producer and consumer applications. We evaluate its design and performance with a study that runs at high concurrency on a modern HPC platform. By leveraging design characteristics, which facilitate an "intelligent" mapping from M-to-N, we observe significant performance gains are possible in terms of several different metrics, including time-to-solution and amount of data moved.

CCS Concepts
• ***Software and its engineering*** $\rightarrow$ *Software performance;* • ***Human-centered computing*** $\rightarrow$ *Visualization systems and tools;* • ***Computing methodologies*** $\rightarrow$ *Parallel algorithms;*

1. Introduction

In situ processing refers to the scenario where analysis and visualization is performed on data as it is being generated, rather than first being saved to persistent storage for *post hoc* use (c.f., [B*16]). In the regime of *in situ* processing, one of many particular configurations entails a scenario where data is moved across a network as it is produced to a separate application running on a separate set of hardware resources for analysis and visualization. In this scenario data is produced on M simulation ranks, then consumed on N consumer ranks, where typically $M >> N$. This configuration is often referred to as "in transit" processing, see Fig. 1.

In the in transit configuration, a central challenge is the problem of M-to-N data redistribution from producer ranks to consumer ranks. One way the challenge arises is when producer and consumer run at markedly different levels of concurrency. Another is when the data distribution scheme used by each is also different. Yet another way is when producer and consumer have vastly different scaling characteristics, which in turn lead to different levels of concurrency M and N for a given producer-consumer pairing on a given problem configuration. Finally, it is often the case that the consumer ranks do not require the complete problem domain from the producer, a situation that can occur during data reduction or subsetting operations, such as slicing or isocontouring.

The focus of our work here is on an approach for solving the M-to-N data redistribution problem in a way that is broadly applicable to a diversity of data producer and consumer methods and that scales to high concurrency on modern HPC platforms. We demonstrate an implementation and evaluate its performance at scale on a large HPC platform. As part of the study, we show generality and broad applicability through the use of different data producers, different data consumers, varying levels of concurrency, and multiple options for data transport. The study includes evaluation of performance gains that result when the consumer is able to request only the data it needs to complete its operation, and contrast the cost savings with a configuration where the producer sends the entire problem domain to the consumer.

The contributions of our work are as follows:

- A design pattern for solving the M-to-N data redistribution problem that arises in all in transit processing.
- A description of an implementation, and demonstration of its generality and scalability where we run at scale on HPC platforms using different data producers and data consumers.

DOI: 10.2312/pgv.20201073

- An in-depth performance evaluation that examines multiple dimensions of performance, including runtime, amount of data moved, and total cost of solution. This study helps to provide insight into questions like “what are the right ratios of M and N to use for in transit processing?”
- Demonstrable performance gains from being able to leverage the design to tune the M-to-N data redistribution for a particular use scenario, gains that can result in substantial cost savings in terms of lowering the amount of data moved between producer and consumer ranks.

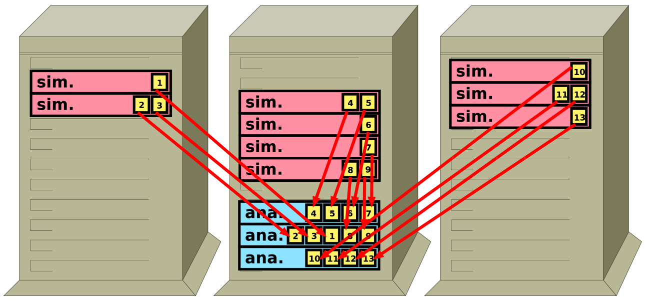

Figure 1: *M-to-N in transit. In the M-to-N in transit scenario, a simulation job running on M ranks (red boxes) with B data blocks (yellow boxes) cooperates with an analysis job running on N ranks (blue boxes), necessitating data movement. Determining the mapping of blocks from the M simulation ranks to the N analysis ranks is a critical challenge that can impact performance in a number of ways.*

2. Design

One key difference between *in situ* and in transit scenarios is the data decomposition used by producer and consumer. In an *in situ* setting, a simulation code running at N-way parallel invokes *in situ* methods that are also run, by definition, at N-way concurrency. Furthermore, the simulation’s data decomposition dictates what data is processed by each of the *in situ* ranks: this decomposition is dictated by the simulation. In contrast, in an in transit scenario, while an M-way parallel simulation code uses one data decomposition, the N-way parallel analysis code has the opportunity to use a different data decomposition.

In this section, we discuss several design considerations that are centered around the ability to solve this very problem: defining a mapping of data from M-to-N ranks, and to do so in a way that can accommodate a variety of different producer/consumer code pairs, run at varying concurrency, and using a number of different potential mechanisms for moving data.

To simplify and accelerate implementation, we chose to extend the SENSEI generic *in situ* interface [A*16] with new capabilities and then we evaluate their performance characteristics (in §3). SENSEI is designed around a common data model that enables the exchange of data in between the data producers, such as a numerical simulation, and the data consumers, such as a parallel visualization or analysis application. Therefore, our presentation of M-to-N design principles is grounded in SENSEI’s existing design, and the terminology we use is drawn from the terms that describe SENSEI’s design elements.

To begin, we present some background material, namely the notion of SENSEI’s endpoint (§2.1) and adaptor design (§2.2), which are foundational to the ability to swap in and out different *in situ* methods without having to recompile the simulation, or data producer, code. Then, we proceed to describe the metadata needed to describe the producer’s data model to the in transit consumer ranks (§2.3). The metadata is input to the partitioner (§2.4), which is responsible for computing a mapping from M producer to N consumer ranks.

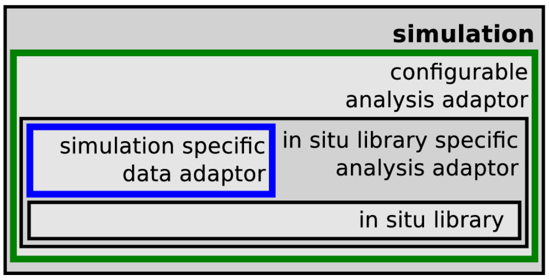
simulation
configurable analysis adaptor
simulation specific data adaptor
in situ library specific analysis adaptor
in situ library

Figure 2: *In situ architecture schematic. A simulation accesses the system via the configurable analysis adaptor(green box). When in situ processing is invoked a simulation specific data adaptor(blue box) provides the means for the configured in situ library specific analysis adaptor to fetch and transform the data needed and feed it to the in situ library for processing.*

2.1. End point

In in transit configurations our system makes use of an MPI parallel application, compiled and installed with our libraries, we call the endpoint. Fig. 3 shows a fictitious example. During in transit processing data is move from a simulation with 5 blocks of data distributed on 5 MPI ranks to the endpoint which is running on 2 MPI ranks. The endpoint is universal in the sense that it may be configured at run time to receive and process data from any instrumented simulation without modifications to either the endpoint or the simulation. This is achieved via XML files provided on the command line, and any supported I/O or *in situ* data processing library may be used. Simulations wishing to run in an in transit configuration only need supply these two XML configurations. The details of how our design accomplishes this are provided below.

2.2. Adaptor pattern

Adaptors define APIs that enable the invocation of a specific action without the need by the invoking code to know who is making it happen or how it is being done. The adaptor pattern allows for the interchange of the performers of an action without modifying the invoking code. In our system there are only two actions. The first action is the *invoke* action. *In situ* processing including analysis, visualization, and I/O is periodically *invoked* by a simulation.

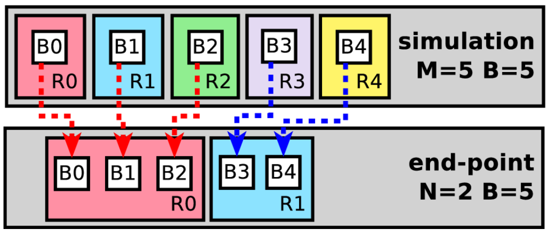

Figure 3: *M to N in transit example. 5 blocks of data (B0 - B4) residing on 5 MPI ranks in the simulation (R0 - R4) are moved to and processed on the endpoint running on 2 MPI ranks (R0 - R1).*

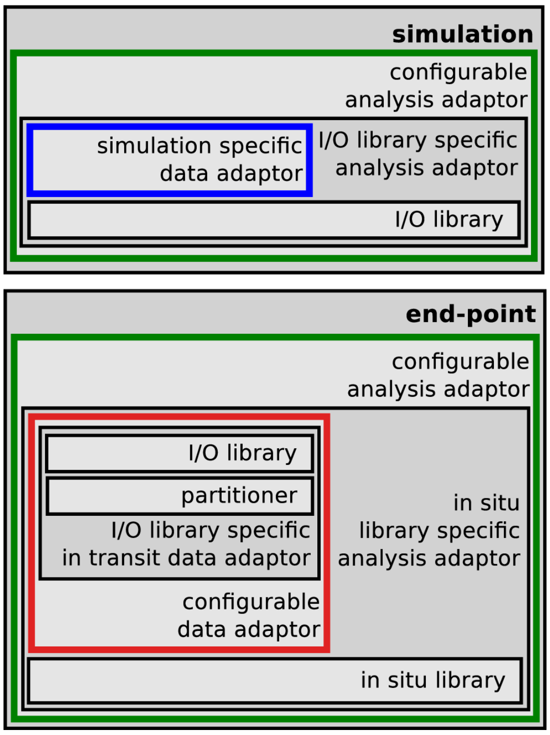
simulation
configurable analysis adaptor
simulation specific data adaptor
I/O library specific analysis adaptor
I/O library
end-point
configurable analysis adaptor
I/O library
partitioner
I/O library specific in transit data adaptor
configurable data adaptor
in situ library specific analysis adaptor
in situ library

Figure 4: *M to N in transit architecture schematic. A simulation accesses the system via the configurable analysis adaptor(green box). When in situ processing is invoked a simulation specific data adaptor(blue box) provides the means for the configured I/O library specific analysis adaptor to fetch and transform the data needed for staging by the I/O library. The endpoint interfaces to the I/O library through a configurable in transit data adaptor(red box) and the in situ data processing library via the configurable analysis adaptor(green box). When the simulation invokes in situ processing, the endpoint is signaled, whence it in turn invokes processing, and the I/O specific in transit data adaptor is provided to fetch the needed data.*

The second action is the *fetch* action. In response to the *invoke* action data processing code *fetches* data to process. Our system is comprised of two fundamental adaptor types, the *analysis adaptor* which is used to *invoke* processing, and the *data adaptor* which is used to *fetch* the needed data.

The *data adaptor* is used to *fetch* data. Any consumer of data be it for I/O, visualization, or analysis makes use of the data adaptor to fetch data. The data adaptor API is used to fetch data, in both *in situ* and in transit configurations. Every simulation provides a *simulation specific data adaptor* that is passed as a part of the *invocation* of *in situ* processing. Similarly the fetch operations of every I/O library is exposed to the system via an *I/O library specific data adaptor* that is passed as part of the invocation of processing. Through the use of data adaptors, data processing codes that consume data need not be modified when run either in an *in situ* or an in transit configuration. In Figs. 2 and 4, the simulation specific data adaptor is shown by the blue box.

Compared to the *in situ* configuration, the in transit configuration has additional complexities of connecting to the simulation running in another job, listening for invocation of processing from the simulation, and load balancing data as it is moved to the endpoint running at a different level of concurrency. In our design all data processing can be run in either an *in situ* or an in in transit configuration without code modification.

To deal with the additional complexities of the in transit configuration, I/O library specific data adaptors are derived from the *in transit data adaptor* that is itself derived from the data adaptor type. The in transit data adaptor adds to the adaptor API, APIs for use by the endpoint for management and control of connection and data movement from the simulation, as well as APIs for interfacing with our partitioning mechanisms discussed below. In this way the I/O library specific data adaptor can be used in place of any simulation specific data adaptor with out modification to the calling code. Note also that RTTI enables the discovery of an in transit data adaptor hence when desirable data processing codes may take control of the partitioning step. This is the basis for the metadata enabled optimizations presented later in Results, §3.

The *analysis adaptor* is used to *invoke* processing, both in *in situ* and in transit configurations. Simulators need access to a diverse set of *in situ* and I/O libraries, ranging from high performance visualization to machine learning to high performance I/O and data movement. The analysis ecosystem is rapidly evolving and no one library does it all. New analysis and I/O capabilities are exposed to simulators through the introduction of *library specific analysis adaptors*. The role of library specific analysis adaptors is to initialize and configure an *in situ* or I/O library for some run time user specified processing or movement, and in response to invocation by the simulation, fetch and and transform data and feed it into the library for processing or movement to the endpoint.

In both *in situ* and in transit configurations the simulation initiates the invocation, but in the in transit configuration this invocation initiates a data movement phase, where data is transferred to the endpoint for processing. In the endpoint an I/O library specific in transit data adaptor listens for the invocation, and forwards it into a library specific analysis adaptor running there. This I/O library specific data adaptor is passed with the invocation and is subsequently used to fetch the necessary data across the network or from disk as it the case be.

The system implements run time configurability through a delegation pattern. Configurable adaptor implementations create and initialize a library specific adaptor instance from a run time user provided XML configuration. Calls made to a configurable adaptor are forwarded directly to the library specific instance. Hence a simulation instrumented to use the *configurable analysis adaptor* gains access to all available I/O and *in situ* libraries. In the endpoint both configurable analysis adaptor and configurable in transit data adaptor are used to gain access to all available I/O and *in situ* libraries through a single interface. In Figs. 2 and 4, the configurable analysis adaptor is shown by the green box. In Fig. 4, the configurable in transit data adaptor is shown by the red box.

2.3. Metadata

Metadata play a key role in our design with respect to in transit processing. We introduced a metadata object with 34 fields that describes spatial domain decomposition and its mapping to MPI ranks of mesh and array based data in a detailed yet compact way. The metadata object encodes the information necessary for load balancing such as array and mesh geometry sizes. The complete list appears in Table 2 in Appendix A.

The simulation fetches the metadata object through the data adaptor API and used by data processing codes to discover what data is available. Partitioning makes use of metadata for load balancing purposes, the output of the partitioning step is a second metadata object encoding the recipe for how data is to be moved. The I/O codes that move data make use of the recipe as the analysis codes make requests to fetch specific data. Analysis codes fetching metadata are given the recipe as well, because this describes how the data will be once it is moved to the endpoint.

2.4. Partitioner

Running in an M-to-N in transit configuration necessitates a partitioning step where data distributed on the M simulation ranks is mapped onto the endpoint's N ranks. A number of partitioning algorithms exist with each having advantages in particular situations. Certain data processing algorithm have specific partitioning requirements. For instance some global parallel FFT implementations require pencil or slab domain decomposition [MDK19]. For these reasons a flexible partitioning mechanism is a requirement. In our approach data processing and analysis codes can be used either *in situ* or in transit without modification. Therefor analysis code should not need to be involved in the partitioning step unless it is beneficial. Additionally our approach supports the run time selection of one of a number of I/O libraries for data movement and as a result partitioning needs to be implemented externally to the I/O library. For those reasons we introduce a *partitioner* object with an API that takes a description of simulation data and its mapping onto simulation ranks, and produces a new description of the data mapped onto endpoint ranks which is used as a recipe for moving data as it is fetched by the analysis code.

2.5. Slice extract

In situ extracts are a commonly used technique that can be used to reduce I/O costs while capturing relevant features of the simulated data. To explore the impacts of partitioning on in transit processing we developed a new analysis adaptor, called the slice extract, which

Extract	Partitioner	M	N	Dataset Size
isosurface	optimized	8192	128	4096^3 (576 GB)
			256	
			512	
			1024	
			2048	
			4096	
			8192	
	default	8192	128	4096^3 (576 GB)
			256	
			512	
			1024	
			2048	
			4096	
			8192	
slice	optimized	8192	128	4096^3 (576 GB)
			256	
			512	
			1024	
			2048	
			4096	
			8192	
	default	8192	128	4096^3 (576 GB)
			256	
			512	
			1024	
			2048	
			4096	
			8192	

Table 1: *Matrix of runs used to analyze the default vs. optimized partitioners at varying concurrency. The setup is described in §3.1.1 and the results presented in §3.1.2.*

computes geometric extracts consisting of either simulation data sampled onto a run time specified slice plane or a run time specified set of iso-surfaces. The extracted geometry is written to disk in VTK format for potential post processing and/or visualization.

Fig. 5 shows the data blocks that are required to compute the extracts defined in the experiments presented in section 3.1. The left column shows a visualization of the extracted geometry. These visualizations were made after the experiments using data written to disk during the experiments. The middle column shows the set of blocks that were used in the calculation. The right columns shows all the blocks. As noted, a default partitioner would need to distribute the data using a standard scheme like striping or block decomposition. However, the light grey blocks are not useful for the intended end product, and their movement represents a pure performance overhead. Therefore, a partitioner that moved only the required blocks from the simulation to the end-point would be more efficient in space utilization and in timeliness.

We exploit this observation to present two different alternatives in the tests that follow. First, we have a *default partitioner* that simply invokes a default block-based equipartitioning algorithm over the entire simulation domain. We also developed *optimized partitioners* for each of the scenarios that would be able to use the metadata provided by the simulation to determine the relevant blocks to transfer. In the case of the slice extract, per-block bounding box

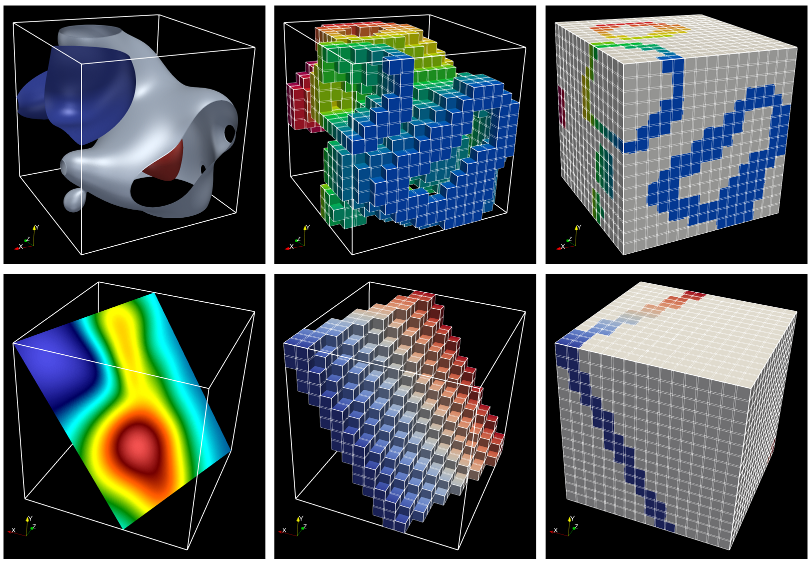
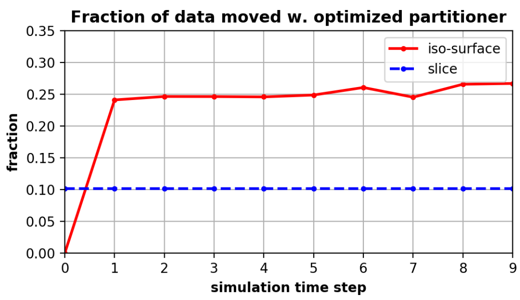

Figure 5: *Optimized partitioners determine which blocks are needed by the consumer and assign them to endpoint ranks while excluding all other blocks from the dataset. Top row, isosurface extract. Bottom row, planar slice extract. Left column, extracted geometry. Center column, blocks needed to compute the extract. Right all blocks. Line plot: the fraction of simulation cells moved as a function of simulation time.*

metadata is tested for intersection with the plane. Only blocks intersecting the plane are needed to compute the extract. In the case of the iso-surface extract, per block array range metadata are tested for intersection with the set of iso-values. Only blocks where the array range brackets an iso-surface value are needed to compute the extract. Blocks not needed in the calculation are not assigned to any rank and as a result are not moved to nor processed by the slice extract. Once the needed set of blocks are identified the block-based equipartitioning algorithm assigns them to available endpoint ranks.

3. Results

The key focus of this section is to evaluate the efficacy of our design. To that end, the primary research questions we focus on include:

1. What are the performance characteristics of M-to-N in transit configurations for a variety of producers, consumers, and at varying levels of concurrency?
2. What are the performance gains that result in an M-to-N in transit setting when making use of data model metadata to move subsets of a problem domain, as compared to moving the entire problem domain from producer to consumer?
3. Are these results consistent across multiple data producers and consumers?

We begin with a study that uses a miniapplication (§3.1). That study reveals some of these key performance insights where we see the advantages that result from our design in terms of reduced runtime, reduced amount of data moved, and other measures. We extend this testing paradigm to a full-scale numerical simulation, IAMR [ABC*98], and see that these same performance gains from the miniapplication study also persist when using a full scale numerical code (§3.2).

3.1. Default vs. Optimized Partitioner at Varying Concurrency with a Miniapplication

3.1.1. Methodology

We focus on measuring the performance gains that result when we are able to leverage the data model metadata to reduce the amount of data being moved from the M producer ranks to the N endpoint ranks. Further, we seek to gain some insight into the selection of a reasonable value for the ratio of simulation ranks, M, to data processing ranks, N, for the operations at hand. To study these questions, we use one of the SENSEI miniapplications, `oscillators`, on NERSC's Cori Cray XC40 supercomputer. For two common extract operations, slice and isosurface, we explore four different performance measures under varying levels of endpoint concurrency when using default and optimized partitioners. In this battery of tests, we wish to better understand a broad set of performance metrics and at varying levels of concurrency.

Here, we configure the `oscillators` to run for 10 time steps with with a 4096^3 mesh decomposed into 8192 blocks onto 8192 MPI ranks. The plane used to compute the slice extract is defined by the point, $\vec{x} = .5, .5, .5$, and the normal, $\vec{n} = .8, -.5, .3$. Three iso-values, $-.25, 1.25, 3.25$, were used when computing iso-surface extracts. These configurations were selected to produce non-trivial extracts and are shown at one time step in the left most panel of figure 5. Both `oscillators` and endpoint were configured to use the ADIOS 1.13.1 I/O library with the FLEXPATH staging method for data movement.

In a series of runs we vary endpoint concurrency from 128 to 8192 ranks in power-of-2 steps. The total concurrency accounting for both miniapp and endpoint therefor varies from 8320 up to 16384 MPI ranks. A group of runs were made for each geometric extract mode, slice and iso-surface. Within each group of runs, there are two sub-groups of runs, the first makes use of optimized partitioners, the second does not. The set of runs we made are summarized in table 1.

To assess the impact that the partitioning method and the ratio of

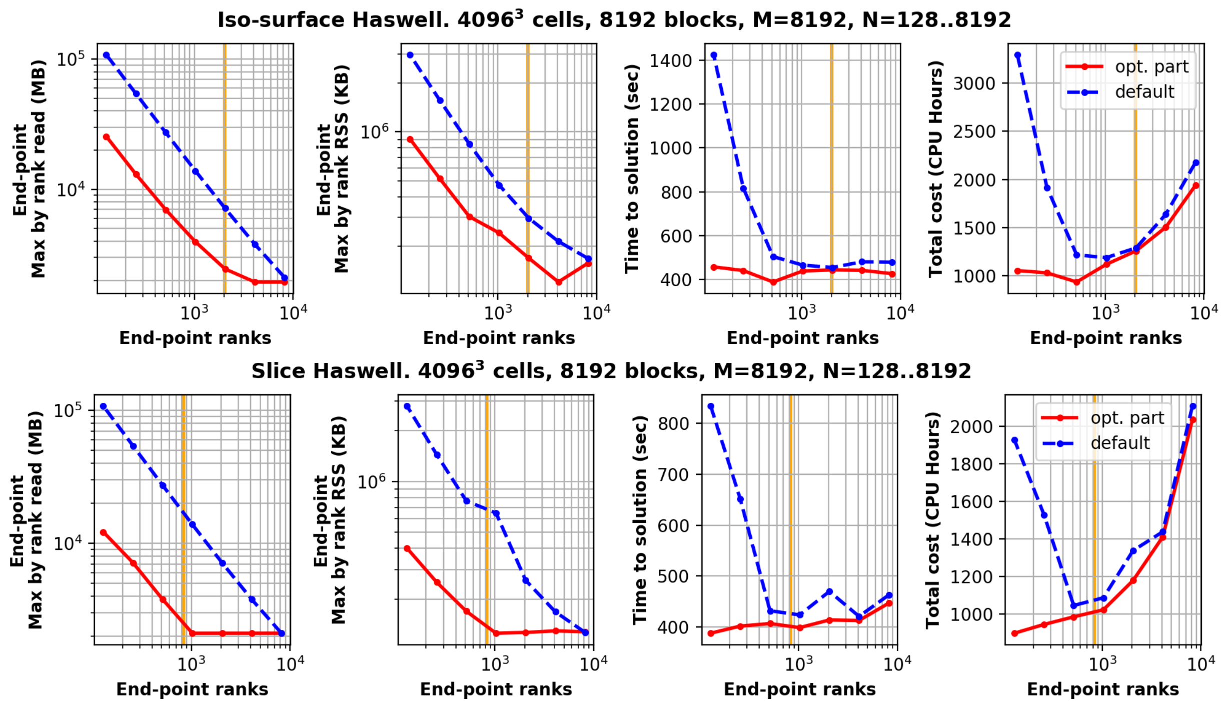

Figure 6: *Scaling study results for M=8192 simulation ranks, with 8192 blocks, while N end-point ranks vary from 128 up to 8192 in power of 2 steps. Top row: iso-surface extract. Bottom row: slice-extract. Column 1: cumulative data moved to the end-point on the rank w/ largest data movement. Column 2: RSS high mark on rank w/ largest memory use. Column 3: time to solution. Column 4: total cost of run.All: red line represents runs w/ optimized partitioner, blue represents runs w/ default partitioner, and orange shows the average number of active blocks.*

simulation ranks, M, to data processing ranks, N, has on the given extract operation, for each run we measure data moved per-rank, memory used per-rank, and total runtime.

3.1.2. Results

The results shown in Fig. 5 illustrate the visual output as well as a measure of the amount of data moved in each of the default and optimized partitioner configuration. The left column shows the extracted geometry, the middle column shows the set of blocks that were used in the calculation, the right columns shows all the blocks. When the optimized partitioner is in use, the white colored blocks were not moved from the simulation to the endpoint nor processed by the endpoint when the optimized partitioner was in use. The line plot shows, at varying simulation time steps, the fraction of the full dataset moved from producer to endpoint for both isosurface and slice computation. For the three isosurfaces computed, only about 25% of the data contributes to a solution, and needs to be moved. An exception is at the first timestep, where the simulation has not evolved the computation to the point where the output has any cells that contain the isovalue. In the case of the slice, only about 10% of the mesh cells intersect the slice plane, and contribute to the final solution.

Measurements from our experiments are presented in Fig. 6. The top row shows isosurface extract run results, while the bottom row shows slice extract run results. We present 4 metrics, from left to right: cumulative data moved to the end-point on the rank with the largest value, RSS (resident set size) memory utilization high water mark on the endpoint rank with the largest value, time to solution in seconds, and total cost of solution in terms of CPU hours.

Time to solution, t_s, is defined as the period from the start of the simulation to the end of either the simulation or end-point which ever ends last. The cost of solution, c_s, metric is defined as $c_s = t_s \cdot (N+M)/3600$ where M is the number of simulation ranks and N is the number of end-point ranks. In all plots a red solid line corresponds to runs made with optimized partitioners, while a blue dashed line to those made with the default partitioner. A vertical orange line shows the average number of active blocks as identified by the optimized partitioners. For the optimized partitioners this line falls at or approximately at the number of end-point ranks where each rank has 1 block of data to process. The middle column in the left of Fig. 5 shows a visualization of the active blocks at one time step.

For both extract geometries, runs made with the default partitioner show the expected doubling trend in the rank wise data movement and RSS memory use metrics as endpoint ranks are halved in each step. In the case of the optimized partitioners, which exclude blocks not needed in the calculations, rank wise data movement and RSS are flat at end-point concurrencies above the number of active blocks. Below that level of concurrency the expected doubling trend is present. Note that RSS metrics follow the general trend but have some jitter, which we think could be explained by the combination of data dependencies such as varying block sizes and dynamic memory allocation algorithms.

With the default partitioner, the time to solution curves show a distinct knee at 512 ranks. Decreasing endpoint resources below this level results in drastic increases in runtime as each rank is given more blocks to process. On the other hand, with the optimized partitioner for the isosurface extract runtime only slightly increases when decreasing endpoint ranks all the way down to 128 ranks. With the optimized partitioner for the slice extract, runtime is lowest at 128 ranks, the smallest endpoint concurrency in our test matrix.

With the default partitioner for both extract types there is a distinct minimum cost of solution. The parabolic shape of this curve makes sense given that at endpoint concurrencies above the active number of blocks not all of the blocks moved and processed contribute to the solution, while at concurrencies below this level multiple blocks are moved to and processed by each endpoint rank. In the case of the former, adding more ranks results in no speed up. In the case of the latter increasing the number of blocks moved to and processed by each rank slows things down. This result shows that when trying to minimize the cost of solution it is important to select the appropriate M to N ratio.

With the optimized partitioner for the isosurface extract cost is lowest at 512 ranks, but not drastically higher below this, while for the slice extract cost is lowest at the smallest run we made, 128 ranks. The optimized partitioners provide a clear benefit when running with fewer endpoint resources resulting in drastically lower time to and cost of solution.

3.2. Results with an AMR-based Numerical Simulation

3.2.1. Methodology

Here we explore the performance differences that result when using a default and optimized partitioner for an in transit use scenario that includes a complex, AMR-based numerical simulation. We wish to better understand the potential gains realizable in actual simulation code as compared to a miniapplication.

We instrumented the AMReX framework [ea19] for use with SENSEI. This gives us access to a wide variety of block structured adaptive mesh refinement(AMR) simulations for testing. In these experiments we made use of the AMReX based IAMR compressible Navier-Stokes code [ABC*98] configured for the simulation of a Rayleigh-Taylor instability modeling the mixing of two fluids of different densities under the influence of gravity. The Rayleigh-Taylor instability produces a set complex iso-surfaces that evolve in time.

The experiments presented in Sec. §3.1 hint that lower time to and cost of solution may be obtained when $M >> N$. In light of this, we selected $M = 8192$ simulation ranks and $N = 128$ data processing ranks for the following experiments. In preparation for a detailed performance analysis we ran IAMR for 440 time steps to evolve the simulation to a point where the fluid interface is complex. During this initial phase, cursory measurements were made. Later, once the simulation had evolved, detailed performance experiments were made over ten time steps while an isosurface extract was produced at each step. As in Sec. §3.1 these experiments

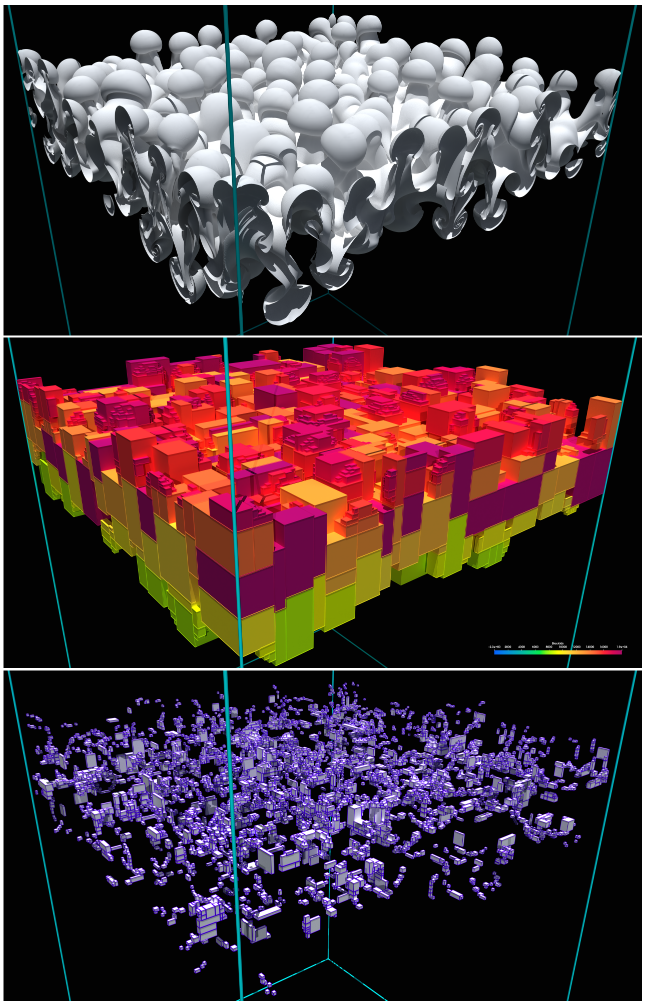
Figure 7: *Top: Isosurface. Middle: The blocks that were used to compute the isosurface colored by block id. Bottom: Level 1 blocks that were not needed to compute the isosurface. Level 0 blocks were not rendered in the bottom most panel since they occlude the level 1 blocks. When the optimized partitioner is used only the blocks shown in the middle panel are moved and processed.*

center around running with and without the optimized partitioner and the same metrics are analyzed.

In these runs IAMR was configured for a base level of 1024^2 x 2048 cells, 1 level of refinement, and was run with M=8192 ranks, with 4 OpenMP cores per rank, on 1025 KNL nodes of NERSC's Cori system. The total number of cores used by IAMR was 32768. The endpoint was run on 9 nodes with N=128 MPI ranks. The slice extract was configured to calculate an iso-surface in fluid density at the value 9536669.198.

One challenge in processing AMR data is that data blocks from refined levels duplicate and cover, either partially or fully, data blocks from coarse levels. Care must be taken when computing

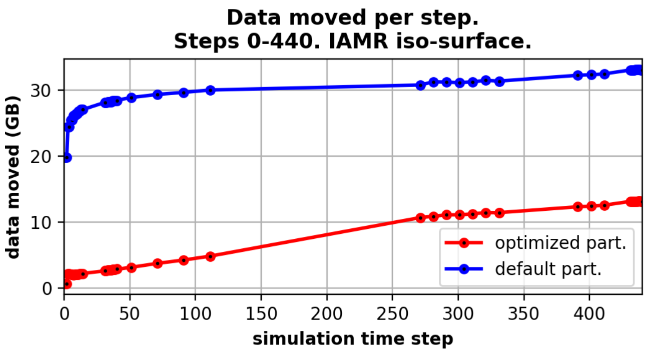

Figure 8: *Data moved during initial runs of the IAMR Rayleigh-Taylor problem. The data moved when the optimized partitioner is used (red line) is substantially less than the data that would be moved if a default partitioning algorithm is used (blue line).*

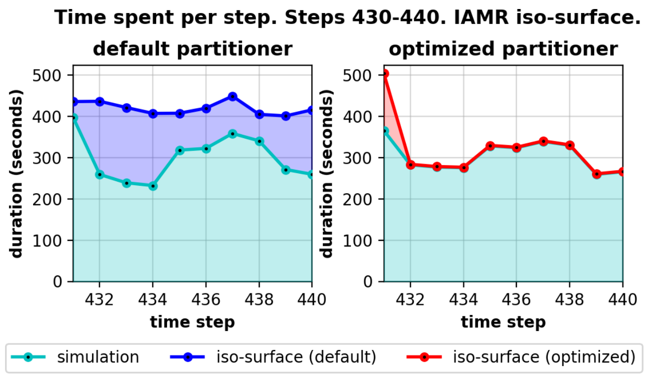

Figure 9: *Time spent during simulation and iso-surface extraction during the 10 time steps 430 to 440. Height of the shaded areas gives the time for that operation. The areas are stacked showing the total time for both operations.*

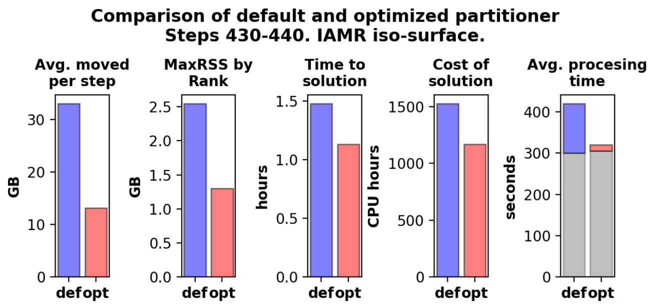

Figure 10: *A comparison between default and optimized partitioner performance over the 10 time steps 430 - 440. From left to right, time averaged per-step total data moved from simulation to endpoint, Max RSS of all endpoint ranks across all time steps, Time to solution in seconds, Cost of solution, time averaged per-step processing time. In all plots the blue bar indicates the default partitioner while the red bar indicates the optimized partitioner. In the right most plot gray bars indicate the solver time.*

metadata and applying partitioning algorithms. For instance in the calculation of per-block array min and max we use to determine if an iso-surface intersects a block, one should not make use of data from the cells of that block which are covered by cells from a block in a more refined level, since covered cells are duplicated in the refined level and hence the iso-surface will be as well. Our AMReX specific data adaptor handles this aspect of the metadata calculation and as a result the optimized iso-surface partitioner can run without modification. The ability to handle complex dataset types such as AMR meshes, illustrates the flexibility of our approach.

3.2.2. Results

During the initial runs, where the simulation was advanced to an interesting state, the isosurface extraction with the optimized partitioner was periodically invoked and data movement and block partitioning metrics were captured. The isosurface extracted at time step 420 is shown in Fig 7 in the top panel, along with the 4771 level 1 blocks that intersect this isosurface in the middle panel, and the 5691 level 1 blocks that do not intersect it in the bottom panel. In time step 420 there were a total of 18654 blocks, only 4795 of which intersect the isosurface. The middle and bottom plots in Fig 7 leave out all 8192 level 0 blocks, of which only 24 intersected the isosurface, since if rendered they would occlude the more interesting level 1 blocks. Fig 8 shows the amount of the data moved at each time step for isosurface extraction with the optimized partitioner (red line) compared to the total data size (blue line). In the worst case, the optimized partitioner moved less than 40% of the data.

The results of detailed experiments conducted over time steps 430 to 440 are presented in Figs. 9 and 10. Fig. 9 shows the time spent, per time step, updating the simulation and computing isosurface extractions. Results for the run made with the default partitioner are on the left and the optimized partitioner on the right. In these plots the height of the shaded regions give the time of the operation, here either simulation or isosurface extraction. Shaded regions are stacked such that the total height gives the total time for both operations. In the plots, time spent updating the simulation is on the bottom in cyan with time spent isosurfacing stacked above. The panel on the left shows the results obtained with the default partitioner and the panel on the right those obtained with the optimized partitioner. With the optimized paritioner, for all but the first step, the time spent in the isosurface extraction (red) is less than 2 seconds per step, a significant speed up compared to when the default partitioner is used.

Fig. 10 shows a comparison between the default (red) and optimized (blue) partitioner, using the metrics: GB of data moved per step; GB memory used (max RSS high water mark) on the endpoint rank with the largest value; time to solution; cost of solution; and processing time per-step. Improved performance across all metrics is observed when the optimized partitioner is used. These results illustrate that the technique can be beneficial when applied to a highly complex realistic simulation.

The software and hardware configurations used in this experiment differ from those presented in §3. Due to a limitation in the ADIOS 1.13.1 FLEXPATH implementation in order to handle AMR data we had to upgrade to ADIOS 2.5.0. This also enabled us to make use of a new RDMA engine for data movement that leverages Cray GNI infrastructure deployed with 2.5.0. RDMA over GNI should be a substantial improvement over the socket based implementation available in 1.13.1. Lastly, due to long queue wait times on Cori we made use of KNL partition for these experiments. Cori's KNL nodes have one socket with 68 cores and 96 GB of RAM per node. Generally speaking, in our benchmarks KNL architecture proved slower compared to the Haswell.

4. Related Work

The idea of processing data as it is generated has been around for decades, with some of the earliest work consisting of a direct-to-film recording process from the 1960s. That work, along with a thorough survey of work in the *in situ* and in transit space is in a 2006 Eurographics STAR report [B*16].

Early work on in transit infrastructure in the HPC space includes the CUMULVS project, which is middleware for coupling codes running at different levels of concurrency and for moving data between them in a M-to-N fashion [GKP97]. A more recent CUMULVS report from 2006 [KWB06] includes a survey of related projects focusing on M-to-N data partitioning and distribution on HPC platforms, some of which go back to the mid 1990s.

More recently, ADIOS [L*14], a parallel I/O library with a POSIX like API, provides in transit processing through it memory based staging methods, including FLEXPATH [D*14], Dataspaces [DPK12], and DIMES [Z*17]. HDF5 is also an I/O library, which has been shown to be useful for in transit processing where data between producer and consumer is staged on high speed NVRAM [G*19]. libIS [U*18] is a lightweight library for in transit data transport, which uses a client-server model where clients (consumers) can request data from servers (producers) on an as-needed basis.

Over the years, several efforts have studied whether an *in situ* or in transit configuration will produce lowest cost, typically time-to-solution, for a given problem configuration. Oldfield et al., 2014 [O*14] evaluate *post hoc*, *in situ*, and in transit in the context of analysis and tracking of features in simulation output. They identify situations in which in transit or *in situ* approaches are more or less advantageous, such as in transit being advantageous when analysis computations are more complex and time consuming. Morozov and Lukic, 2016 [ML16] examine *in situ* and in transit configurations of a cosmological simulation coupled with a two-stage analysis pipeline, and find that when the analysis and simulation codes have different scaling properties, that it is advantageous to use in transit configurations. Kress et al., 2019 [K*19] study scalable rendering and aim to find the best balance of M to N producer and consumer ranks across different levels of concurrency by considering cost models for both *in situ* and in transit configurations.

The focus of our work here is on the design, implementation, and performance evaluation with focus on the performance gains that can result from leveraging metadata for partitioning and moving data from producer to consumer ranks in an in transit configuration. A similar idea appears in Childs et al., 2005 [C*05], which describes the "contract" system in the VisIt application that results in optimizations of data movement through the visualization pipeline: downstream processing stages inform upstream stages of the data subsets needed to perform a specific computation. Our performance study uses some of the same use scenarios from that work, namely planar slicing and iso-contouring, to illustrate the gains that result from optimizing data partitioning and movement. Whereas that earlier work measured and reported runtime improvement in the setting of an interactive GUI based post processing visualization application where the stages of their pipelines ran in the same process address space and data was provided by disk based I/O, our work here investigates similar approach applied in a setting where the data produced by a simulation is immediately moved to and processed in a separate application running at a different level of concurrency. In our work fast interconnects are used to move data and as such it never hits the disk, and we look beyond runtime into deeper levels of performance analysis. For each of the default and optimized configurations, we measure and report, in addition to runtime, the amount of data moved between producer and consumer ranks, and the memory footprint of producer and consumer ranks. These additional measurements beyond runtime provide significantly deeper insight into the benefit of the optimizations for in transit data partitioning and placement.

5. Discussion and Future Work

The two examples, slicing and isosurfacing, both represent relatively common visual analytics tasks for scientific applications as well as good ways to illustrate the power of a uniform metadata model in enabling more intelligent access and query support at runtime. Our metadata supports multiple partitioning choices that we demonstrated: a quick-to-adopt default scenario with striping, and an efficient way through careful pre-selection of data in the "optimized" partitioner. The role of the partitioning strategy had a significant runtime performance impact as the ratio of simulation ranks (M) to end-point ranks (N) increased, and the simple infrastructure allowed us to rapidly explore how best to tune the relative scaling of the application and the visualization analytics.

As future work, one key aspect to continue to explore is how visualization applications can move from the desktop to extreme scale computing platforms. Because analysis and simulation scale in different ways, overall performance improvement for computation experiments may lie in composing workflows to utilize in transit and additional resources rather than doing analysis *in situ* at scale. How to best utilize that capability leverages thinking about the metadata extensions that we've presented to create partitioners that minimize the trade-offs of network communication across a variety of workflows.

Aligned with that future work direction is a need to better categorize and understand the performance classes of different analytics components as they scale. We anticipate rich future work involving classification of such analysis and quantifying how best to co-design cooperative components so that they leverage M to N to L to P to .. compositions. There is also a substantial need to address how performance and provenance information is captured so that results generated in transit or *in situ* can be made reusable and verifiable. Additionally, the data flow necessary to enable better partitioners is a form of computational steering, based on a bidirectional control flow. Extending the metadata model to understand a general framework for advanced controls of execution management is another exciting opportunity.

Challenges do exist for this approach. For example, composition of reduction with visualization/analysis changes the nature of scaling and the types of metadata required. Some applications, like ray-tracing, do not have easy, index-based techniques for making partitioning choices, since rays can end up anywhere. This represents an interesting opportunity to explore other sorts of runtime compositions, but its use of partitioners would likely focus more on duplication of data, rather than priority selection. Lessons learned from these applications might lead towards a more general query interface between data and analysis adaptors, which would require careful performance tuning.

6. Conclusion

In in transit processing, one of the central challenges is moving data from the *M* producer ranks to the *N* consumer ranks. We present a design pattern for a flexible, general purpose solution to this challenging problem. Our implementation adds new in transit capabilities to the SENSEI generic *in situ* interface, and we demonstrate its scalablility by running at up to 16K-way concurrency on a large HPC platform. We demonstrate its generality through two different examples, one of which is a sophisticated adaptive mesh refinement (AMR) code. Our performance evaluation measures runtime, amount of data moved, and time to solution to help reveal the nature of performance gains possible when using an optimized partitioner that moves only those portions of the data needed by the consumer. Future work will include broadening scalability studies to include additional transport options, and to explore more broadly across a spectrum of heterogeneous resources that are emerging on large HPC platforms.

References

[A*16] AYACHIT U., ET AL.: The SENSEI Generic In Situ Interface. In *Proceedings of In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV 2016)* (Salt Lake City, UT, USA, Nov. 2016). 2

[ABC*98] ALMGREN A. S., BELL J. B., COLELLA P., HOWELL L. H., WELCOME M. L.: A conservative adaptive projection method for the variable density incompressible navier–stokes equations. *Journal of Computational Physics 142*, 1 (1998), 1 – 46. URL: http://www.sciencedirect.com/science/article/pii/S0021999198958909, doi:https://doi.org/10.1006/jcph.1998.5890. 5, 7

[B*16] BAUER A. C., ET AL.: *In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report. *Computer Graphics Forum, Proceedings of Eurovis 2016 35*, 3 (June 2016). 1, 9

[C*05] CHILDS H., ET AL.: A Contract-Based System for Large Data Visualization. In *Proceedings of IEEE Visualization (Vis05)* (Minneapolis, MN, Oct. 2005), pp. 190–198. 9

[D*14] DAYAL J., ET AL.: 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), IEEE, pp. 246–255. 9

[DPK12] DOCAN C., PARASHAR M., KLASKY S.: Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing 15*, 2 (Jun 2012), 163–181. 9

[ea19] ET AL. Z.: AMReX: A Framework for Block-Structured Adaptive Mesh Refinement. *Journal of Open Source Software 4*, 37 (2019), 1370. doi:10.21105/joss.01370. 7

[G*19] GU J., ET AL.: HDF5 as a vehicle for in transit data movement. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2019), ACM. 9

[GKP97] GEIST G. A., KOHL J. A., PAPADOPOULOS P. M.: CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. *International Journal of High Performance Computing Applications 11*, 3 (Aug. 1997), 224–236. 9

[K*19] KRESS J., ET AL.: Comparing the efficiency of in situ visualization paradigms at scale. In *International Conference on High Performance Computing* (2019), Springer, pp. 99–117. 9

[KWB06] KOHL J. A., WILDE T., BERNHOLDT D. E.: Cumulvs: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance. *The International Journal of High Performance Computing Applications 20*, 2 (2006), 255–285. 9

[L*14] LIU Q., ET AL.: Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience 26*, 7 (2014), 1453–1473. 9

[MDK19] MORTENSEN M., DALCIN L., KEYES D.: mpi4py-fft: Parallel fast fourier transforms with mpi for python. *Journal of Open Source Software 4* (04 2019), 1340. 4

[ML16] MOROZOV D., LUKIĆ Z.: Master of puppets: Cooperative multitasking for in situ processing. In *Proceedings of the Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2016), pp. 285–288. 9

[O*14] OLDFIELD R. A., ET AL.: Evaluation of methods to integrate analysis into a large-scale shock shock physics code. In *Proceedings of the 28th ACM International Conference on Supercomputing* (2014), ICS '14, pp. 83–92. 9

[U*18] USHER W., ET AL.: libis: a lightweight library for flexible in transit visualization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2018), ACM, pp. 33–38. 9

[Z*17] ZHANG F., ET AL.: In-memory staging and data-centric task placement for coupled scientific simulation workflows. *Concurrency and Computation: Practice and Experience 29*, 12 (2017), e4147. 9

Appendix A: Metadata Table

Name	**Purpose**
	dataset
GlobalView	tells if the information describes data on this rank or all ranks
MeshName	name of mesh
MeshType	container mesh type
BlockType	block mesh type
NumBlocks	global number of blocks
NumBlocksLocal	number of blocks on each rank
Extent	global index space extent†,§,*
Bounds	global bounding box*
CoordinateType	type enum of point data‡
NumPoints	total number of points in all blocks*
NumCells	total number of cells in all blocks*
CellArraySize	total cell array size in all blocks*
NumArrays	number of arrays
NumGhostCells	number of ghost cell layers
NumGhostNodes	number of ghost node layers
NumLevels	number of AMR levels (AMR)
PeriodicBoundary	indicates presence of a periodic boundary
StaticMesh	non zero if the mesh does not change in time
	array
ArrayName	name of each data array
ArrayCentering	centering of each data array
ArrayComponents	number of components of each array
ArrayType	type enum of each data array
ArrayRange	global min,max of each array*
	block
BlockOwner	rank where each block resides*
BlockIds	global id of each block*
BlockNumPoints	number of points for each block*
BlockNumCells	number of cells for each block*
BlockCellArraySize	cell array size for each block‡,*
BlockExtents	index space extent of each block†,§,*
BlockBounds	bounds of each block*
BlockLevel	AMR level of each block§
BlockArrayRange	min max of each array on each block*
	AMR level
RefRatio	refinement ratio in i,j, and k direction§
BlocksPerLevel	number of blocks in each level§

$*$ - present only if requested, † - with Cartesian meshes, ‡ - with unstructured meshes, § - with AMR meshes

Table 2: *Metadata fields available in the enhanced data model describe data and its mapping onto a set of hardware resources.*