Performance Characterization of Irregular I/O at the Extreme Scale

S. Herbein^a, S. McDaniel^a, N. Podhorszki^b, J. Logan^b, S. Klasky^b, M. Taufer^{a,*}

^aUniversity of Delaware, Newark, DE 19716

^bOak Ridge National Laboratory, Oak Ridge, TN 37831

Abstract

This paper reports our experience with irregular I/O and describes lessons learned when running applications with such I/O on supercomputers at the extreme scale. Specifically, we study how irregularities in I/O patterns (i.e., irregular amount of data written per process at each I/O step) in scientific simulations can cause increasing I/O times and substantial loss in scalability. To this end, we quantify the impact of irregular I/O patterns on the I/O performance of scientific applications at the extreme scale by statistically modeling the irregular I/O behavior of two scientific applications: the Monte Carlo application QMCPack and the adaptive mesh refinement application ENZO. For our testing, we feed our model into I/O kernels of two well-known I/O data models (i.e., ADIOS and HDF) to measure the performance of the two applications' I/O under different I/O settings. Empirically, we show how the growing data sizes and the irregular I/O patterns in these applications are both relevant factors impacting performance.

Keywords: exascale, irregular I/O, QMCPack, ENZO, aggregation rate, ADIOS, HDF5

1. Introduction

The I/O times of applications at the extreme scale play key roles in the applications' overall performance. As the number of processes grows, the I/O fraction of the whole execution time grows, causing substantial loss in overall scalability. In our past work with the QMCPack application and the integration of the Adaptable IO System (ADIOS) I/O library in its code, we experienced this problem first-hand [1, 2, 3]. Figure 1 shows results from this work; specifically the figure shows the average percentage of time spent by QMCPack simulations of a large graphite system in execution (i.e., for computation and communication) versus the time spent for I/O. The values are obtained over a set of 6 QMCPack runs, each performing 100 Monte Carlo steps and saving to disk the traces every 10 steps. We considered two popular libraries, HDF5 [4] and ADIOS [5]; for both we considered different aggregation ratios for the writing processes (i.e., one aggregator for two processes, one aggregator for four processes, and one aggregator for eight processes). Independently from the I/O library and aggregation ratios, the I/O grows to become 30% of the total execution time when half the nodes of the Titan supercomputer at the Oak Ridge National Laboratory (ORNL) are used.

Identifying the reasons for increasing I/O times is not always intuitive or trivial. At first we thought that the irregularity of the I/O pattern (i.e., irregular amount of data written per process at each I/O step) in QMCPack was among the main causes of the increasing I/O times. QMCPack uses quantum Monte Carlo methods to find solutions to the many-body Schrödinger's equation by stochastic samplings [6] and calculates total energies of condensed

¹E-mail of corresponding author: taufer@acm.org; Fax number: +1-302-831-8458

Figure 1. Percentage of QMCPack time spent in execution vs. I/O for different I/O libraries and aggregation ratios using different numbers of nodes on Titan.

systems. The core of the QMCPack algorithm works on many trial solutions called walkers. A process runs multiple walkers per step; at the end of each step, walkers with low energy are duplicated, while walkers with high energy are terminated. Thus, within each simulation step, processes deal with different numbers of walkers and write different amounts of data to disk. To study whether our hypothesis about the impact of I/O irregularities on performance is true, we consider different I/O patterns (both irregular and regular) and study their performance under different I/O settings. We investigate the problem when using two I/O data models with their libraries and file formats, namely, ADIOS and HDF5.

The spectrum of applications with irregular I/O is broad. To cover such a spectrum and at the same time consider meaningful scientific applicators at the extreme scale, we considered the Monte Carlo application QMCPack [6] and an adaptive mesh refinement (AMR) application ENZO [7]. For each application, we ran several simulations and built a statistical model of their I/O patterns, which we present in this paper. For completeness, we also modeled the regular I/O of two scientific applications, S3D [8] and GTC [9], whose processes write the same amount of data at each I/O step. In our study, we fed the models into two I/O kernels supported by the two I/O models (i.e., ADIOS and HDF) and we studied the impact of different I/O settings. In this experience paper, we present lessons learned when measuring the impacts of irregular I/O on performance on Titan at ORNL. Techniques to tune the applications' I/O performance are beyond the scope of this paper and can be found in [2].

The contributions of this paper are as follows:

- We statistically model the I/O behavior of two irregular and a regular scientific applications.
- We feed the models into the ADIOS and HDF I/O kernels to measure the performance of the two applications' I/O under different I/O parameter values and settings.
- We critically compare and contrast the results for a large set of I/O scenarios and show how the growing I/O size (i.e., from a small I/O size of few megabytes per writing process to the larger I/O size with an increase of one order of magnitude) and the irregular I/O patterns in applications with irregular I/O are relevant factors when tuning performance.

The rest of this paper is organized as follows. Section 2 provides an overview of the applications and their I/O patterns; Section 3 describes the I/O methods studied in this paper and the I/O kernels used for the testing; Section 4 defines the key time components profiled in this work for both ADIOS and HDF5; Section 5 presents the several tests at the small and large scales; Section 6 discusses relevant related work; and Section 7 concludes this paper with a brief summary and ideas for future work.

2. Profiling I/Os of Real Applications

We model the irregular I/O behavior of the Monte Carlo application QMCPack and the AMR application ENZO, as well as the regular I/O behavior of S3D and GTC. QMCPack and ENZO represent two classes of applications with two quite different irregular I/O patterns. QMCPack's I/O pattern features all the processes writing to disk; the amount of data written by each process resembles the shape of a normal distribution. On the other hand, ENZO's pattern features very few processes writing very large amounts of data, while the other processes write only a few KBytes. As the name also suggests, applications with regular patterns exhibit processes which all write the same amount of data at each I/O step.

2.1. QMCPack

QMCPack's many trial solutions are called walkers and evolve over many steps during which they refine a system of particles using the diffusion Monte Carlo (DMC) method. At the end of each step, walkers with low energy are duplicated, and walkers with high energy are terminated. Consequently, at each step, each process has a different number of walkers whose particle positions and energies it must write out to disk. Figure 2.a shows the box plot graphic for the number of walkers being written to output for real QMCPack simulations when performing 15 DMC steps of a large graphite system with 128 carbons and 512 electrons and using 256 processes (two per node) on Titan. The box plot consists of seven pieces of information. The whisker on the bottom extends from the 10th percentile (bottom decile) to the top 90th percentile (top decile). Outliers are placed at the end of the bottom and top decile whiskers (outliers caps). The top, bottom, and lines through the middle of the box correspond to the 75th percentile (top box line), 25th percentile (bottom box line), and 50th percentile (middle box line), respectively. A square is used to indicate the arithmetic mean. The height of the box indicates a large variability in the number of walkers across the 256 processes at each step. For this QMCPack simulation, for each step, the mean (square) is equal to the median (middle box line), suggesting that a normal distribution of the number of walkers across processes per step can statically model the number of writing walkers per process at a given simulation step. Note that each walker contains the same amount of data per write (i.e., 65 KB); thus the total size of the I/O data per process can be obtained by multiplying the number of walkers by 65 KB. Figure 2.b shows an example of normal distribution used to generate the I/O profile of QMCPack for one single step. While the number of walkers per process changes across steps, the overall distribution remains normal. Figure 2.c shows the box plot for the number of walkers being written to output for the synthetically generated I/O of QMCPack simulations used in our tests when performing 15 DMC steps of the large graphite system.

Figure 2. Profile of the number of walkers per process for 15 steps of a QMCPack simulation studying a 4x4x2 graphite on 256 nodes of Titan (a); statistical model of the I/O pattern mimicking a normal distribution for one simulation step (b); and example of synthetic I/O patterns generated with the statistical model (c).

2.2. ENZO

ENZO is a parallel AMR application for computational astrophysics and cosmology. The ENZO algorithm uses high spatial and temporal resolution for modeling astrophysical fluid flows. Data is organized in Cartesian meshes in one, two, and three dimensions, and different meshes with different sizes are written by different processes at runtime. An ENZO simulation changes the accuracy of a solution in certain regions as the simulation evolves. In ENZO, the processes write a variable amount of data per step as the simulation expands and contracts the several meshes. Figure 3.a shows the plot box of the ENZO simulation for 15 steps of the NFW Cool-Core Cluster test on 256 nodes of Titan. The simulation of the cooling flow is performed in an idealized cool-core cluster that resembles the Perseus cluster. The test considers a root mesh of 64^3 , a maximum refinement level of 12, and a minimum mass for refinement level exponent of -0.2. In this simulation, as the cooling catastrophe happens, the temperature drops to the bottom of the cooling function. We observe that very few outliers (processes) deal with very large meshes while many processes deal with smaller meshes. As for QMCPack, we observe a large I/O variability (i.e., wide box plot). Contrary to QMCPack, the variability is not distributed normally but more closely resembles an exponential distribution. This can be deduced from the mean that is biased by the high size of the data records written by few outliers and the median that is overlapping with the bottom box line. Figure 3.b shows an example of exponential distribution used to generate the I/O profile of ENZO for one single step. Across steps, as the number of blocks per process changes, the overall distribution remains exponential. Figure 3.c shows the box plot graphic for the number of blocks being written to output for the synthetically generated I/O of ENZO simulations that we use for our testing when performing 15 steps.

Figure 3. Profile of the data written per process for 15 steps of an ENZO simulation studying the NFW Cool-Core Cluster test on 256 nodes of Titan (a); statistical model of the I/O pattern mimicking an exponential distribution for one simulation step (b); and example of synthetic I/O patterns generated with the statistical model (c).

2.3. Applications with Regular I/O Patterns

To prove our hypothesis, we compare and contrast the I/O performance of the two applications described above with the I/O performance of a hypothetical application with regular I/O pattern. Several applications like S3D and GTC exhibit such a regular I/O pattern since in their simulations each process writes the same amount of data at each I/O step. These applications are easy to mimic synthetically. In an ideal case, the box plot graphic for this type of applications presents the overlapping of the 10th, 25th, 50th, 75th, and 90th percentiles in a single horizontal line, while no outliers are detected. The statistical model used here to mimic the I/O of regular applications assigns constant data to each writing process at each I/O step.

3. I/O Libraries

We consider two data models with their library and file format, ADIOS and HDF5, that are commonly used in supercomputers. In this section we provide a general overview of the two models.

3.1. ADIOS Background

ADIOS consists of a suite of I/O methods, an easy-to-use read-and-write API, a set of utilities, and metadata stored in an external XML file [5]. Different I/O methods, such as POSIX and MPI_AGGREGATE, can be specified in the XML file, which can be changed at runtime rather than compile time. ADIOS_POSIX (or POSIX) is the simplest of the ADIOS methods; it uses the standard POSIX file API. Each process writes to one file, and an extra global metadata file is created to reference the data output. POSIX obtains high performance when using few processes since it has low overhead; for large process counts, however, the metadata server of a distributed file system such as Lustre can become a bottleneck. ADIOS MPI_AGGREGATE is a hybrid method that first aggregates data to a small subset of processes and then uses MPI-I/O to write to disk. By accumulating data, ADIOS MPI_AGGREGATE keeps the load on the Lustre metadata server low and thus continues to perform even with very large numbers of processes. The simple API allows for minimal changes to the existing code, while the XML file enables I/O method switching without recompiling the code. The XML file is parsed on ADIOS initialization; its metadata contains a description of the data generated and information on how the data should be written out to disk.

The ADIOS I/O library comes with Skel, a tool that allows us to decouple the I/O from the simulation. Skel builds the skeleton of an application's I/O by decoupling the I/O component of a complex code from its computation and communication components [10]. Skel takes an ADIOS XML metadata descriptor and a set of parameters and generates C or FORTRAN source code with the appropriate ADIOS I/O calls, data generation, and timers. The execution of the benchmarks on real platforms is faster because Skel does not include computation and communication but just I/O; users can easily and efficiently collect a large set of I/O performance information. The original implementation of Skel had a major limitation that reduced its applicability: it provided a method to specify only a fixed I/O size per "time step." We addressed this limitation and extended Skel to enable variable I/O size per process and per I/O step using three different techniques: an inline definition of functions modeling the I/O of each process the generation of processes' I/Os based on a statistical distribution, and the use of I/O traces from a real simulation. Each approach is selected by adding additional tags to Skel's XML description of the I/O. When these tags are encountered, Skel uses the XML-specified approach to generate the variable I/O across processes at runtime.

3.2. HDF5 Background

Hierarchical Data Format (HDF) [4] has become a standard for scientific data archiving and data sharing because it enables scientists to create and record the structure and metadata about the dataset being stored. HDF includes a data model, library, and file format for storing and managing complex heterogeneous data. HDF5's flexible and simple data model provides the ability to define a higher-level data model on top of HDF5 to describe domain-specific data relationships. HDF5's data model consists of two components: groups and datasets. Groups are structures that resemble directories and contain links to datasets, subgroups, and other objects. A dataset is contained within a group and consists of a multidimensional array of elements, together with supporting metadata. A dataset allows for a wide variety of data types. Datasets have a single data type that applies to every data element. HDF5's library defines a set of predefined data types that are consistent with commonly used storage formats, which include two's complement integers, IEEE floating-point numbers, and big-endian and little endian byte orders. A dataset can be stored as a linear array of contiguous bytes or broken into separate chunks; it may be compressed or stored in a separate file. Combinations of these storage options are also possible. HDF uses a portable file format, with no limit on the number or size of data objects in the collection. HDF5 offers a number of options for organizing data within the file, to take advantage of the access patterns employed in reading and writing data. The open source library of HDF runs on a wide variety of computing platforms, from personal PCs to petascale-class machines such as Titan and Blue Gene. HDF runs on a wide variety of high-level APIs with C, C++, FORTRAN 90, and Java interfaces.

The key features HDF offers for optimizing storage space and access time are compression, metadata caching, chunking, and an extensible set of I/O drivers. Moreover, the HDF group provides users with tools for analyzing and managing the data, including HDFView, a visual tool for browsing and editing HDF5; h5check, a tool to check the validity of an HDF file; and h5edit, a tool for editing an HDF5 file. Moreover, HDF can be used by other middle-ware packages (e.g., FLASH, MATLAB, Mathematica, ParaView, and R) to supports data managing, manipulating, viewing, and analyzing in applications.

4. Taxonomy of I/O Times

While performing the same type of writing, the ADIOS and HDF5 total times can be decomposed into different I/O time components.

4.1. Time Components in ADIOS Kernel

In previous work, we showed how POSIX and MPI_AGGREGATE are the most effective ADIOS methods for small and large numbers of writers, respectively [1, 2, 3]. Here we focus our analysis on these two methods. For both methods, the I/O time can be divided into several phases. In an initial phase, ADIOS reads in the specified XML file and parses it for the I/O parameters to be used for the application run. In this phase, the ADIOS buffer is also allocated. If MPI_AGGREGATE is selected as the I/O method, then all the writers in the application are split into process groups (PGs). Each PG is a collection of writers; one of the writers in each PG is designated as the PG aggregator. The aggregator is responsible for gathering the data from the other writers in the PG and saving the data to disk. If the number of aggregators, which is specified in the XML file, evenly divides the number of writers, then each PG will be exactly the same size. If not, one PG will have slightly more or slightly fewer writers than the average. Each aggregator creates a group metadata file on the filesystem using MPI_File.Open and writes out the PG header to it. The process with rank zero is responsible for creating the global metadata file. We call the I/O time to create the groups and define the aggregators ADIOS.Group. If POSIX is selected as the I/O method for the run, then each writer creates its own metadata file on the filesystem. This step requires each writer to contact the metadata server; the connection to the metadata server can quickly become a bottleneck as the number of writers expands. Again the process with rank zero is responsible for creating the global metadata file that contains a duplicate copy of all the metadata in the “subfiles” created by the other writers. We call the I/O time to create the files and contact the metadata server ADIOS.Open.

Independently from the I/O method used, each writer copies its data from its application’s buffers into its ADIOS buffer. This is a simple write to memory operation; we call the associated I/O time ADIOS.Write. In the last I/O phase, the data is saved to disk. In the case of POSIX, each writer sends its data to disk. In the case of MPI_AGGREGATE, each writer sends its data to its PG’s aggregator, and then the aggregator writes out the data to disk. Once all of the data is on disk, each writer is then responsible for its metadata. In the case of POSIX, each process is a writer and writes out the data to the metadata file. Each piece of metadata is saved in two places. The first place is in the subfile with the metadata’s associated data. The second place is together with all the other metadata in the global metadata file. The process with rank 0 is responsible for this global file and must gather all the metadata from each writer and write it to the global file. In the case of MPI_AGGREGATE, the writers’ tasks are performed by the aggregators. We call this time component ADIOS.Close.

ADIOS also supports the option of disabling the generation and storage of the global metadata file. This can potentially improve the write performance by eliminating the metadata aggregation performed during the ADIOS.Close stage. This performance increase, however, comes at the cost of increasing the complexity when the data is read back in. To obtain a global view of the data during a read without the global metadata file, each individual subfile must be opened and its metadata parsed, tasks that can be costly with large numbers of subfiles. This tradeoff is explored in Section 5.

4.2. Time Components in HDF5 Kernel

The HDF kernel was designed to mimic the behavior of the MPI_AGGREGATE method in ADIOS and has many of the same phases. In the initial phase, the HDF kernel parses the command line arguments for the I/O parameters to be used for the application run. Also in this phase the application buffers are allocated and the PGs are created based on the aggregation ratio. The aggregator of each PG calls H5Fcreate to create an HDF5 file on the Lustre filesystem. This step involves communication with the Lustre metadata server and can be costly at large scales. Fortunately, this cost is paid only once at the beginning of the kernel; subsequent modifications to the file just require reopening the existing file, which is a much cheaper operation. This differs from ADIOS, where appending to files is not supported in the MPI_AGGREGATE method and a new file must be created for each timestep. We call this the Init phase. For the next phase, each aggregator opens an HDF file created in the Init phase and preps it for the appending of data. This step involves the creation of a new HDF group, dataspace, and dataset. We call this the HDF.Open phase. For the next phase, the data is aggregated from every process in each PG down to the aggregator. While the data is being

aggregated down, the aggregator is simultaneously writing the data out to the HDF5 file using the H5Dwrite function. We call this the HDF_Write phase. The final stage involves the closing of the HDF5 groups, dataspace, datasets, and files. When the HDF5 file is closed using H5Fclose, the remaining data, both raw and metadata, is flushed out to disk, and access to the file is terminated. We call this the HDF_Close phase.

In addition to replicating the functionality found in MPI_AGGREGATE, the HDF kernel supports the ability to choose between two different techniques for aggregating the data: brigade aggregation and asynchronous gather. The brigade aggregation mimics the aggregation done in ADIOS. Each MPI process in a process group with rank i sends its data to the process with rank $i - 1$ while simultaneously receiving data from the process with rank $i + 1$. This creates a “line” of processes that the data flows down on its way to the aggregator. This method is described in further detail in Section 5.2.3. The second type of aggregation, asynchronous gather, achieves the same result as brigade aggregation but with much less data movement. Each nonaggregator in a process group does a blocking send to its aggregator. As the aggregator is writing out a block of data to disk, it does an asynchronous receive from any MPI rank in the process group. Hence, the first nonaggregator to send its data to the aggregator gets served first. This asynchronous gather eliminates all of the redundant data movement present in brigade aggregation. The tradeoff is that each MPI process within a process group must communicate with the aggregator directly. If the network path between the aggregator and any of the MPI processes in the process group is slow because of network congestion or other factors, the penalty will be large. Since the aggregator in brigade aggregation communicates only with rank 1 in the process group, however, the effects of these slowdowns can be diminished.

5. Performance Characterization

Our study reports observed performance tendencies of irregular I/Os when mimicking the patterns of well-known applications on real supercomputers.

5.1. Experimental Setting

We ran our tests on Titan, a Cray XK7 machine with 18,688 compute nodes and 299,008 cores. The performance analysis is structured in two parts: the analysis at the small scale on Titan using the older *Spider* Lustre filesystem with 10.7 PB of space and a bandwidth of 240GB/s, and the analysis at the large scale on the same supercomputer but with the newer *Atlas* Lustre filesystem, with 32 PB of disk space and a bandwidth of 1TB/s. The data analyzed was collected during the transition from the old to the new filesystem on Titan, thus the two sets of measurements. At the small scale, each process can write up to 2.4MB. The data size is tailored to match the size ranges in QMCPACK for which each process had approximately 35 walkers and each walker generates 65KB, summing up to approximately 2.4MB. At the larger scale, each process can write up to 24MB. The size range is a next-generation approximation: we expect that the computational abilities of the next-generation machines will be one order of magnitude larger. For instance, QMCPACK should be able to run with 10X as many walkers per process on next-generation supercomputers. We repeated each test three times on 256, 512, 1024, 2048, 4096 nodes of Titan with two process per node; each test performed 15 consecutive writes. The measured I/O times for each set of 15 writes are presented in box plots, each collecting multiple pieces of information. The height of the boxes indicates the time variability across the multiple samples. The 75th percentile (top box line), 25th percentile (bottom box line), and 50th percentile (middle box line) are captured by top, bottom, and middle lines of the box. The whisker on the bottom extends from the 10th percentile (bottom decile) to the top 90th percentile (top decile). Red outliers caps are at the end of the bottom and top decile whiskers. The arithmetic mean is represented by a square.

5.2. Performance at the Small Scale

At the small scale, we profile the three main ADIOS times (i.e., ADIOS_Total, ADIOS_Write, and ADIOS_Close) with ADIOS supporting metadata and with I/O data sizes approximately 2.4 MB per process. We consider the three different I/O patterns (i.e., exponential, normal, and uniform) with different numbers of aggregators (i.e., 1:4 and 1:16). In our analysis we focus on the following performance comparisons: threading vs. nonthreading in ADIOS; POSIX vs. MPI_AGGREGATE; all-gather aggregations vs. brigade aggregations; and diverse numbers of object storage targets (OSTs) and aggregators.

5.2.1. Threading vs. Nonthreading

POSIX has no threading capabilities as an ADIOS method. For MPI_AGGREGATE, the main difference when turning on or off the threading is how the open operation is performed. When threading is enabled, the MPI_File_Open is performed in a separate thread. The thread is joined back into the main thread during the ADIOS_Close, right before the data is written to disk. Thus, while all the ADIOS_Writes (and possibly other application computation) are being performed, the open is happening asynchronously. This approach helps mitigate the large overhead associated with the Lustre metadata server on systems such as Titan. We note that threading is also performed within the ADIOS_Close when using the MPI_AGGREGATE method. Once all the data has been written to the subfiles, the associated metadata must also be written to the subfiles. Moreover, the metadata must be sent to the process with rank 0 and written out to the global metadata file. These two operations are performed simultaneously when using threading.

To study the impact of threading, we compare the performance in terms of total time for the regular I/O pattern (i.e., “constant” pattern) and irregular I/O patterns (i.e., “normal” for the QMCPack-like I/O pattern and “exponential” for the ENZO-like I/O pattern) without and with threading. In the Uniform pattern, each process writes 2.4 MB of data. In the normal pattern, samples are taken from a normal distribution with a mean of 2.4 MB and a standard deviation of 0.8 MB. In the exponential pattern, samples are taken from an exponential distribution with a lambda of .5. The samples are then scaled so that the average is 2.4 MB. This size is considered a medium I/O size for our tests. In Figure 4 we compare the total time with and without threading for 15 steps of the three I/O patterns using 2,048 writers (two processes per node and 1,024 nodes of Titan). We use a fixed aggregation ratio of two writers per aggregator. We observe that threading clearly provides better performance by reducing the ADIOS_Group time to a negligible fraction of the total time. Once threading is turned on, the main time component is the ADIOS_Close

Figure 4. Total times for regular (i.e., constant) and irregular (i.e., normal and exponential) patterns when using medium I/O size and 2,046 writers on Titan without (a-c) and with threading (d-f). The number of I/O steps on the x-axis ranges from 1 to 15; the total time on the y-axis ranges from 0 to 4 seconds.

time. We observed that the total times and the ADIOS_Close times match for all three I/O patterns, indicating how for medium I/O size, this time component is the driving performance factor. An increase in the number of writers and

associated nodes does not substantially change this observation. Figure 5 shows how ADIOS_Close times match for all three I/O patterns when using 2,048 and 4,096 writers (with two writers per node) on Titan with threading. The

Figure 5. ADIOS_Close times for regular (i.e., constant) and irregular (i.e., normal and exponential) patterns when using medium I/O size with 2,048 writers (a-c) and 4,096 writers (d-f) on Titan. The number of I/O steps on the x-axis ranges from 1 to 15; the ADIOS_Close time on the y-axis ranges from 0 to 4 seconds.

general conclusion that threading should be “on” as the default configuration for ADIOS is not surprising. What it is surprising is that the performance is dictated by the ADIOS_Close times and that the performance values are immune to the type of patterns at this level of I/O size, system sizes, and aggregation ratios.

5.2.2. POSIX vs. MPIAGGREGATE

Intuitively, POSIX should be outperformed by the MPIAGGREGATE as the number of writers grows. To assess this hypothesis, we compare POSIX versus MPIAGGREGATE for the three I/O patterns; the results are shown in Figure 6 when using 2,048 writers. Similar performance was measured for 4,096 writers (not shown in this paper). The results in Figure 6 are again not surprising as turning on threading “removes” the impact of the ADIOS_Group time on the total time. On the other hand, the opening phases for POSIX do not benefit from threading. In results not shown in this paper, we also observe that turning off threading causes the MPIAGGREGATE method to become more costly than POSIX, even for 4,096 writers, confirming the need for threading in I/O operations. The relevance of these tests is that they confirm the insensitivity of the I/O times to the I/O patterns for both POSIX and MPIAGGREGATE for medium I/O size: the total times exhibit the same behavior for the three I/O patterns for POSIX and for MPIAGGREGATE.

5.2.3. Aggregation: All-Gather vs. Brigade

Our tests point out the critical role of ADIOS_Close times on the overall performance. Specifically, we observe how the ADIOS_Close times match the total time in size and behavior across the different I/O patterns. The closing operation used as the default in MPIAGGREGATE is brigade aggregation. ADIOS also supports a simple aggregation

Figure 6. Total I/O times for all three I/O patterns (i.e., constant, normal, and exponential) when using 2,048 writers on Titan with MPIAGGREGATE with threading (a-c) and with POSIX (d-f). The number of I/O steps on the x-axis ranges from 1 to 15; the total time on the y-axis ranges from 0 to 4 seconds.

called all-gather. To better understand the dynamics of the closing operation, we compare the ADIOS_Close times for simple aggregation with those for brigade aggregation.

The simple all-gather aggregation (called here AGG=1) is a naive way of gathering data. Initially each aggregator allocates a block of memory large enough to store all the data from all the writers in the same PG. This operation is followed by an MPI_Gatherv. Once all the data is within the aggregators memory, the data is saved to disk. The obvious disadvantage is that each aggregator must have enough physical memory available to store all the data written by its PG. This is a most unlikely event given todays scientific applications. The brigade aggregation (called here AGG=2) is a more sophisticated way of gathering the data; it avoids the memory limitations present in the simple aggregation. All the writers within a PG do an MPI_Allgather communication with the amount of data they plan to write. Each writer then allocates a block of memory equal to the size of the largest block of data to be written by a process in the PG. A “line” of process is created, ordered from largest rank to smallest rank (which should be the aggregator). A cascade of data begins to flow from the highest-ranked process down to the aggregator. Each process prepares to receive data from its higher-rank neighbor while simultaneously sending data to its lower-rank neighbor. The aggregator begins saving data out to the disk, one writer block at a time. Thus, as each iteration is completed, the data moves down the line of processes, ending at the aggregator. Although more data movement is involved in the brigade aggregation than in the simple aggregation, the memory requirement of the brigade aggregation is only that of the largest block to be written.

To study the performance dynamics of the two closing algorithms, we compare the ADIOS_Close time for the I/O pattern when using 2,048 and 4,096 writers on Titan with simple aggregation (AGG=1) and brigade aggregation (AGG=2). Once again for the medium I/O size we observe no difference in terms of performance behavior across the three I/O patterns, and thus we display in Figure 7 only the results of the normal distribution. We conclude that there is no difference between the two closing approaches in terms of the maximum I/O times of the writers for each

I/O step. On the other hand, within each single step the writers exhibit a different performance profile for the two closing approaches. Note that both sets of tests were run with two writers per node and an aggregation ratio of 1:2. In the case of simple aggregation, a thread is spawned, and both the data and metadata are simultaneously gathered at each aggregator. Because of the parameters of these tests, this aggregation can be done through shared memory rather than over Titan's Gemini network. Thus, the aggregation happens quickly. Any nonaggregator writers (half the writers in these test cases) rapidly finish their portion of ADIOS_Close and exit the function. The remaining writers, the aggregators, must write all the data and metadata out to disk. This operation is much slower than the aggregation and causes the large gaps we see in the figures. With brigade aggregation, each aggregator immediately begins writing its data out to disk while simultaneously receiving data from the other writer in the PG. This approach causes the nonaggregators to block inside the ADIOS_Close while waiting for the aggregators to finish writing out the first block of data. Once the aggregators finish writing the first block, the nonaggregators can quit blocking on their MPI communication and can exit the ADIOS_Close function. The aggregators then write out the second block of data and exit. Hence, half the writers exit halfway through the ADIOS_Close. We note that from the point of view of the memory use on each single node, the brigade aggregation with its lower memory requirements seems to better fit with the growing data size of applications; thus this is the approach used in the rest of this paper.

(a) Normal, AGGR=1, 2048 writers

(b) Normal, AGGR=1, 4096 writers

(c) Normal, AGGR=2, 2048 writers

(d) Normal, AGGR=2, 4096 writers

Figure 7. ADIOS_Close time for the normal I/O pattern when using 2,048 and 4,096 writers on Titan with simple aggregation (i.e., AGG=1 (a-b)) and brigade aggregation (i.e., AGG=2 (c-d)). The number of I/O steps on the x-axis ranges from 1 to 15; the I/O time on the y-axis ranges from 0 to 4 seconds.

5.2.4. Number of OSTs and Aggregators

One critical question that is driving the I/O research is the selection of the optimal number of OSTs and aggregators. While the search for such values has been studied in other work [11] and is not the scope of this paper, we

include here the study of their impact on performance for the sets of values that are normally considered when running applications on Titan. Figure 8 shows the ADIOS_Close times for the normal distribution pattern when using 2,048 writers (two writers per node) on Titan with different aggregation ratios (i.e., 1:8 (a), 1:4 (b), and 1:2 (c)). The figure also shows a case in which the number of aggregators and OSTs do not match (i.e., the ratio of aggregators over OSTs is two (d)). Figure 9 shows the ADIOS_Close time for the normal pattern when using 4,096 writers (two writers per node) on Titan with different parameters values. In both figures we can observe that as the number of writers grows, the optimal ratio of aggregation also changes. For 2,048 writers the optimal ratio is 1:8, whereas for 4,096 the ratio increases to 1:16. On the other hand, in both case studies a different number of OSTs does not seem to impact performance. Our observations confirm the importance of selecting the proper number of aggregators for the sake of

Figure 8. ADIOS_Close time for the normal I/O pattern when using 2048 writers on Titan with different aggregation ratios and OST numbers. The number of I/O steps on the x-axis ranges from 1 to 15; the I/O time on the y-axis ranges from 0 to 4 seconds.

performance. The observations also confirm that the optimal numbers of aggregators may vary based on the number of writers and the data size (results not shown in this section). On the other hand, the type of I/O patterns (regular or irregular) seems again not to impact performance for the type of tests considered in this paper. Here we present results for the normal distribution, but similar results were observed for the other two patterns.

5.3. Performance at the Large Scale

At the large scale, we profile the main ADIOS times (i.e., ADIOS_Total, ADIOS_Write, and ADIOS_Close) with ADIOS metadata support on and off. We also profile the main HDF5 times (i.e., HDFtotal, H5Dwrite, and H5Fclose) using the two aggregation algorithms—namely, the brigade algorithm, which uses an approach to aggregate data similar to that of the ADIOS brigade method, and the asynchronous gathering algorithm (both algorithms were described

Figure 9. ADIOS.Close time for the normal I/O pattern when using 4,096 writers on Titan with different aggregation ratios and OST numbers. The number of I/O steps on the x-axis ranges from 1 to 15; the I/O time on the y-axis ranges from 0 to 4 seconds.

in Section 3.2). Different numbers of processes and nodes in Titan are used, where each process writes data in the order of 24 MB. We study the two irregular I/O patterns (i.e., exponential and normal) with different numbers of aggregators (i.e., 1:2, 1:4, 1:8, 1:16, and 1:32). We perform a comprehensive set of tests with up to 4,096 writers (two processes per node and 2,048 nodes of Titan). Here we present results for the test set using 4,096 processes, but similar results were observed for the test sets with fewer processes. In our study we focus on the following performance aspects: performance impact of metadata support in ADIOS; driving I/O time components in ADIOS; performance of HDF brigade versus asynchronous gather; and the pros and cons of ADIOS versus HDF5.

5.3.1. Impact of Metadata Support in ADIOS

As described in Section 3.1, ADIOS comes with metadata support. While the additional information on the actual data is a plus for the user, the support may come at an additional performance cost, especially when many processes perform large I/O writing. Thus we explore the impact of the ADIOS metadata support on performance at the large scale. Figures 10.a-e show the ADIOS I/O times of the 4,096 processes per steps when metadata is supported for the exponential I/O pattern with aggregation ratios of one aggregator for 2, 4, 6, 16, and 32 writing processes respectively; for the sake of comparison Figures 11.a-e show the same time components for the same exponential I/O pattern but without metadata support.

Figures 12.a-e show the ADIOS total times of the 4,096 processes when the metadata support is on for the normal I/O pattern with aggregation ratios of one aggregator for 2, 4, 6, 16, and 32 writing processes; Figures 13.a-e show the same time components for the same exponential I/O but without metadata support.

Figure 10. Total I/O times of ADIOS with metadata for the *exponential I/O pattern* using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 16 seconds.

Two main observations from the analysis of the test sets are also illustrated in Figures 10, 11, 12, and 13. First, metadata support causes a larger variation of the I/O times per step across the writing processes at the smaller aggregation ratios (i.e., 1:2, 1:4, and 1:8). This can be associated with the fact that the generation and storage of the metadata requires additional work at the aggregator level; these processes that serve as aggregators are also those experiencing the larger I/O times causing the variations. On the other hand, for the larger aggregation ratios (i.e., 1:16 and 1:32), the impact of the metadata support on the processes' times is hidden by other I/O factors. Thus, in both cases (i.e., with and without metadata support), we observe the same large variations of times per I/O step. Second, supporting metadata seems to come at a negligible increase of total I/O time per step that can be tolerated, considering the advantages of the metadata support. For small aggregation ratios (i.e., 1:2, 1:4, and 1:8), the loss in performance with respect to the total I/O time is less than 5%. For larger aggregation ratios (i.e., 1:16 and 1:32), no difference in total time per I/O step is observable. In general, the fact that I/O times seem not to be sensitive to the metadata support when using large aggregation ratios can be related to two factors: (1) the cost of aggregating and storing the global metadata is dwarfed by the cost of aggregating and storing the actual data; and (2) the MPI_Reduce to aggregate the global metadata is latency bound, whereas the MPI_Reduce to aggregate the actual data is bandwidth bound.

5.3.2. Driving Time Components in ADIOS

Just as in the small scale, we study the behavior of the main ADIOS time components at the larger scale (i.e., ADIOS_Write and ADIOS_Close) and their impact on the total I/O times. Figures 14.a-e show the ADIOS_Write times for ADIOS without metadata for the exponential I/O pattern with an aggregation ratio of one aggregator for 2, 4, 6, 16, and 32 writers; Figures 15.a-e show the ADIOS_Close times for same exponential I/O pattern. Here we present results for ADIOS without metadata support, but similar results were observed for ADIOS with metadata support.

Figures 16.a-e show the ADIOS_Write times for ADIOS without metadata for the normal I/O pattern with a level of aggregation of one aggregator for 2, 4, 6, 16, and 32 writers; Figures 17.a-e show the ADIOS_Close times for same normal I/O pattern.

Figure 11. Total I/O times of ADIOS *without metadata* for the *exponential I/O pattern* using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 16 seconds.

In all our tests, we observe that ADIOS_Write times closely match the behaviors of the actual data sizes. Moreover, ADIOS_Write times of the single processes at each I/O step are one order of magnitude smaller than the ADIOS_Close times. We also note that the ADIOS_Write times are in the same range independently from the type of I/O pattern and aggregation ratios. As also observed at the small scale, ADIOS_Close times mostly impact the overall I/O times and are sensitive to the aggregation ratios. The aggregation impact is further discussed below.

5.3.3. Impact of Aggregations in ADIOS

Not surprisingly, different aggregation ratios impact performance in different ways. Figures 10 and 12 indicate how, when metadata support is on, both exponential and normal I/O patterns exhibit similar performance for lower aggregation ratios (up to 1 aggregator for 8 processes) and have a sweet spot in terms of the aggregation ratio equal to 1 aggregator for 4 processes for the normal distribution and 1 aggregator for 8 processes for the exponential distribution. Turning off metadata support slightly improves the performance of 1 aggregator for 4 processes aggregation but comes at the cost of not benefiting from the additional information embedded in the metadata. The figures also show that for both irregular I/O patterns (i.e., for the normal and exponential patterns), high aggregation ratios come not only with a high performance degradation but also with a high variability in terms of the time each process needs to complete its I/O phase. In general, the observations with lower aggregation ratios match and confirm the results in Figure 1 where we showed that the three smaller aggregation ratios in the real application QMCPack resulted in similar I/O overheads. Based on our observations, we conclude that a further increase in the aggregation ratio will result in a further increase in I/O overhead and deterioration of the I/O scalability.

5.3.4. HDF Brigade versus Synchronous Gather

HDF5 is a valuable alternative to ADIOS, and thus we explore the impact of the two different aggregation algorithms described in Section 4.2 on the I/O performance for the two irregular I/O patterns and the same numbers of nodes on Titan. Figures 18.a-e show the HDF5 I/O total times when using brigade for the exponential I/O pattern

Figure 12. Total times of ADIOS with metadata for the *normal I/O pattern* using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 3.5 seconds.

with aggregation ratios of one aggregator for 2, 4, 6, 16, and 32 writing processes; Figures 19.a-e show the same time components for same exponential I/O but with asynchronous gather.

Figures 20.a-e show the HDF total I/O times when using brigade for the normal I/O pattern with aggregation ratios of one aggregator for 2, 4, 6, 16, and 32 writing processes; Figures 21.a-e show the same time components for same exponential I/O but with asynchronous gather.

At very small aggregation ratios (i.e. 1-to-2 and 1-to-4), brigade and asynchronous gather have similar performance profiles. As the aggregation ratio increases, the runtime of brigade increases much faster than that of asynchronous gather. In the worst case of the exponential I/O with an aggregation ratio of 1:32, the difference in performance is greater than a factor of 5. This is most likely due to the extra data movement that occurs in brigade aggregation as actual data is transferred from neighbor to neighbor, and this movement of data increases the probability that both congestion and system noise slow the transfer rate. Even a small performance penalty incurred 32 times, as in the case of a 1:32 aggregation ratio, can have a large impact on performance.

5.3.5. Pros and Cons of ADIOS vs. HDF

Ultimately the comparison between ADIOS and HDF is not a perfect one. Each has its own features. ADIOS supports a global metadata file, which incurs a small cost at write time but greatly decreases the complexity of reading. HDF does not have this global metadata file feature and thus does not have the overhead, but it also does not support the additional information in the metadata. In addition, HDF supports creating a file once and reopening it for appending, while the version of ADIOS that we used (i.e., 1.5) does not support appending to files in the `MPIAGGREGATE` method. Thus ADIOS must incur the cost of creating files on the Lustre filesystem at every time step, whereas HDF has to pay this cost only once, on the first time step.

Despite the limitations of the comparison, we claim that in most cases the performance of HDF and ADIOS is similar. This is consistent with what we saw in our previous work on real applications and summarized in Figure 1. This is not to say that the performance of ADIOS and HDF cannot be improved. The data suggests that at large aggregation ratios, the ADIOS method `MPIAGGREGATE` could benefit from using asynchronous gather rather than

Figure 13. Total times of ADIOS *without metadata* for the *normal I/O pattern* using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 4 seconds.

brigade when aggregating data. The data also suggests that support for appending to files in the MPIAGGREGATE method could be beneficial to ADIOS.

6. Related Work

The study of I/O performance is still an open challenge. To the best of our knowledge, none of the existing work systematically studies the I/O performance of simulations exhibiting irregular I/O patterns. Recent efforts targeting the performance profile and tuning of I/O parameters include [12, 11, 13, 14]. As for the work presented in our paper, work in [12, 11, 13] studies I/O performance at extreme scale for a specific filesystem and specific I/O libraries. Specifically, in [12], the authors use an evolutionary method to explore the large space of I/O parameters for HDF5-based application. In [11], the authors present extensive characterization, tuning, and optimization of parallel I/O on the Jaguar supercomputer. In [13], the authors use a mathematical model to reproduce the filesystem behavior; simulation results are used to validate the model values and an auto-tuning tool searches for optimal parameters, starting from the validated model values. In [14], the authors model disk I/O time for a specific type of technology (i.e., SAD) and a specific platform (i.e., Dash - a prototype for the large, 1024-node Gordon system at SDSC). Other I/O efforts study the overall I/O performance for one or multiple applications [15, 16, 17, 18].

7. Conclusions

In this paper we present our experience in benchmarking and profiling the I/O of two scientific applications with irregular I/O patterns, QMCPack and ENZO. Our initial hypothesis was that when the simulation processes write irregular amount of data per process at each I/O step, the overall simulation exhibits increasing I/O times and substantial loss in scalability. The results in this paper indicate that our hypothesis is true for large data sizes (i.e., when each process write data in the range of tens of megabytes) but not for smaller I/O size (i.e., when each process write data in the range of few megabytes or lower). Since the I/O of applications is growing in size as computing

Figure 14. ADIOS.Write times of the 4,096 processes per step for the *exponential I/O pattern* with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 16 seconds.

approaches the extreme scale, we expect that the I/O patterns will play a larger role in the performance tuning. Future work of the group includes defining strategies that integrate the I/O patterns as a first-class citizen in the I/O tuning process.

Acknowledgments

This work is supported in part by the NSF grant CCF 1318445 and through the Predictive Theory and Modeling for Materials and Chemical Science program by the Office of Basic Energy Science (BES), Department of Energy (DOE). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

- [1] S. Herbein, M. Matheny, M. Wezowicz, J. Kroger, J. Logan, J. Kim, S. Klasky, M. Taufer, Performance Impact of I/O on QMCPack Simulations at the Petascale and Beyond, in: Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering (CSE), 2013.
- [2] M. Matheny, S. Herbein, N. Podhorszki, S. Klasky, M. Taufer, Using Surrogate-based Modeling to Predict Optimal I/O Parameters of Applications at the Extreme Scale, in: Proceedings of the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2014.
- [3] S. Herbein, S. Klasky, M. Taufer, Benchmarking the Performance of Scientific Applications with Irregular I/O at the Extreme Scale, in: Proceedings of the Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), 2014.
- [4] The HDF Group: Hierarchical data format Version 5, <http://www.hdfgroup.org/HDF5> (2000-2010).
- [5] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, C. Jin, Flexible I/O and Integration for Scientific Codes through the Adaptable I/O System (ADIOS), in: Proceeding of the 6th ACM/IEEE International Workshop on Challenges of Large Applications in Distributed Environments (CLADE), 2008.
- [6] J. Kim, K. Esler, J. McMinis, M. Morales, B. Clark, L. Shulenburg, D. Ceperley, Hybrid Algorithms in Quantum Monte Carlo, J. of Physics: Conference Series 402 (1).

Figure 15. ADIOS Close times of the 4,096 processes per step for the *exponential I/O pattern* with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 16 seconds.

- [7] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J. hoon Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, Y. Li, ENZO: An Adaptive Mesh Refinement Code for Astrophysics, *The Astrophysical Journal Supplement Series* 211 (19).
- [8] J. Chen, A. Choudhary, B. deSupinski, M. DeVries, E. Hawkes, S. Klasky, W. Liao, K. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, C. Yoo, Terascale Direct Numerical Simulations of Turbulent Combustion using S3D, *Comput. Sci. Disc.* 2 (015001).
- [9] J. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, J. Chen, Combining In-situ and In-transit Processing to Enable Extreme-scale Scientific Analysis, in: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.
- [10] J. Logan, S. Klasky, H. Abbasi, Q. Liu, G. Ostrouchov, M. Parashar, N. Podhorszki, Y. Tian, M. Wolf, Understanding I/O Performance Using I/O Skeletal Applications, in: *Proceedings of the Euro-Par 2012 Conference*, 2012.
- [11] B. Behzad, S. Byna, Q. Koziol, H. Luu, A. Prabhat, J. Huchette, R. Aydt, M. Snir, Timing Parallel I/O Complexity with Auto-tuning, in: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
- [12] W. Yu, J. S. Vetter, H. Oral, Performance Characterization and Optimization of Parallel I/O on the Cray XT, in: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2008.
- [13] H. You, Q. Liu, Z. Li, S. Moore, The Design of an Auto-tuning I/O Framework on Cray XT5 System, in: *Cray Users Group Conference (CUG'11)*, 2011.
- [14] M. R. Meswani, M. A. Laurenzano, L. Carrington, A. Snively, Modeling and Predicting Disk I/O Time of HPC Applications, in: *Proceedings of the 2010 DoD HPC Modernization Program Users Group Conference*, 2010.
- [15] M. Gamell, I. Roderio, M. Parashar, J. Bennett, H. Kolla, J. Chen, P.-T. Bremer, A. G. Landge, A. Gyulassy, P. McCormick, S. Pakin, V. Pascucci, S. Klasky, Exploring Power Behaviors and Trade-offs of In-situ Data Analytics, in: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
- [16] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, H. Abbasi, Using Cross-layer adaptations for dynamic data management in large scale coupled scientific workflows, in: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
- [17] N. Podhorszki, S. Klasky, Q. Liu, C. Docan, M. Parashar, H. Abbasi, J. F. Lofstead, K. Schwan, M. Wolf, F. Zheng, J. Cummings, Plasma Fusion Code Coupling using Scalable I/O Services and Scientific Workflow, in: *Proceedings of Workshops for the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
- [18] M. Slawiska, M. Clark, M. Wolf, T. Bode, H. Zou, P. Laguna, J. Logan, M. Kinsey, S. Klasky, A Maya Use Case: Adaptable Scientific Workflows with ADIOS for General Relativistic Astrophysics, in: *Proceedings of the XSEDE 2013 Conference*, 2013.

Figure 16. *ADIOS.Write* times of the 4,096 processes per step for the *normal I/O pattern* with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 4 seconds.

Figure 17. *ADIOS.Close* times of the 4,096 processes per step for the *normal I/O pattern* with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 4 seconds.

Figure 18. Total I/O times of HDF5 when using brigade for the exponential I/O pattern using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 10 seconds.

Figure 19. Total I/O times of ADIOS when using *asynchronous gather* for the exponential I/O pattern using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 10 seconds.

Figure 20. Total I/O times of HDF5 when using *brigade* for the normal I/O pattern using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 4 seconds.

Figure 21. Total I/O times of HDF5 when using *asynchronous gather* for the normal I/O pattern using large I/O size with 1:2, 1:4, 1:8, 1:16, and 1:32 aggregation ratios. The number of I/O steps on the x-axis ranges from 1 to 10; the I/O time on the y-axis ranges from 0 to 4 seconds.