HPC I/O Innovations in the Exascale Era

International Journal on High Performance Computing Applications
XX(X):1–13

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Greg Eisenhauer[1], Norbert Podhorszki[2], Ana Gainaru[2], Scott Klasky[2], Junmin Gu[3], Vicente Bolea[4], Liz Dulac[2], Dmitry Ganyushin[2], William F. Godoy[2], Qing Liu[5], Caitlin Ross[4], Lipeng Wan[6], Scott Wittenburg[4], Kesheng Wu[3]

Abstract

As high performance computing architectures have evolved to deliver ever increasing performance and applications have changed to take advantage of new innovations, middleware that supports those applications has had to adapt as well. The Adaptable Input Output System (ADIOS), which provides scalable IO performance for exascale HPC applications is one such middleware. During the Exascale Computing Project (ECP), key portions of the ADIOS environment were adapted to respond to ongoing developments in exascale computing and the stresses and opportunities inherent in those changes. This paper examines those changes and where appropriate compares them to pre-exascale implementations.

Keywords

High Performance Computing, ADIOS, I/O

Introduction

ADIOS 2, the latest version of the Adaptable Input Output System Godoy et al. (2020) was developed as part of the Exascale Computing Project (ECP). As part of that development, key aspects of the ADIOS implementation were re-imagined and optimized to maximize scalability and performance. One of those innovations is the new BP5 engine and data format.

The key changes in BP5 compared to older approaches are a) two level metadata representation, b) new memory management to reduce memory consumption during the I/O phase, and c) GPU-aware IO. BP5 also enables filtering output steps for reading so that an analysis code can avoid reading all steps of data. The new metadata organization allowed us to use ADIOS in the E3SM application effectively and to provide fast writing performance when running on the full scale of the Frontier supercomputer. The new memory management allowed for saving up to 50% of memory used for I/O purposes for particle-in-cell codes, like PIConGPU and WarpX, that wanted to output particles that consume the majority of the GPU memory. New GPU-integrating allows computing block statistics and applying GPU-based data operators, increasing the throughput of write operations.

In order to put these developments into context and provide a basis for later discussion, we first briefly describe the development history of ADIOS and its I/O strategies, and how they have evolved as HPC architectures have changed.

A brief history of BP format

The original ADIOS data format, called Binary Pack (BP), addressed one of the main writing performance issue of parallel IO that contemporary parallelization of logically contiguous file formats like NetCDF-3 and HDF5 suffered from, namely the interleaved access to file regions by multiple processes for the sake of organizing an N-dimensional array output from multiple processes into a single, contiguous, dump of the array, so that any reader could know a-priori, what offsets in the file corresponds to each element of the array. BP simply dumped each process' output into a separate region of the file and avoided much of the locking contention the other IO libraries suffered from. This format gave superior write performance and good scalability for applications running before the appearance of petascale computers. Obviously, reading the entire variable on a single reader was slower, since the ADIOS library had to read and merge multiple blocks of data into a global, contiguous, array, while the other libraries only had to issue a single read operation for the entire data. BP was, however, better at scale for parallel reading, as long as the decomposition of the readers were not intentionally orthogonal to the decomposition of the writers Polte et al. (2009), and it was also competitive and mostly superior when reading slices of large datasets (e.g. 2D slices of a 3D array) Tian et al. (2011). Moreover, applications could experience maximum performance when writing and reading

[1]Georgia Institute of Technology
[2]Oak Ridge National Lab
[3]Lawrence Berkeley National Lab
[4]Kitware
[5]New Jersey Institute of Technology
[6]Georgia State University

Corresponding author:
Greg Eisenhauer, College of Computing, Georgia Institute of Technology, Atlanta, GA 30339
Email: eisen@cc.gatech.edu

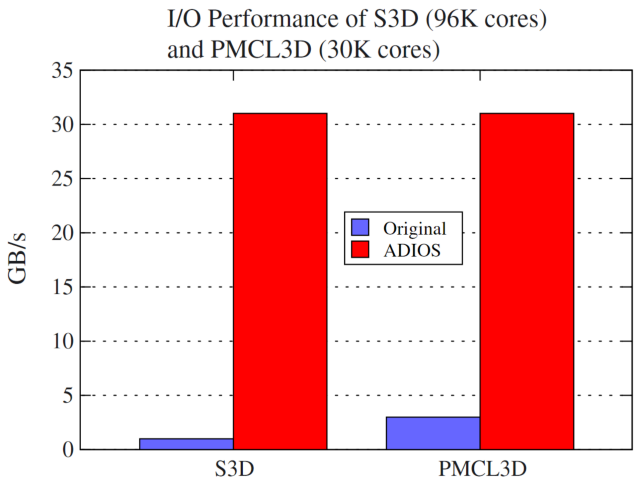

Figure 1. IO performance comparison of original IO solution vs ADIOS with BP3 file format for two HPC applications on Jaguar at Oak Ridge Leadership Facility, a petascale computer.

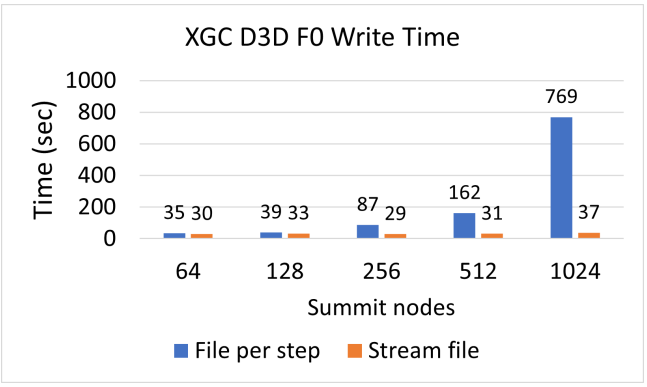

Figure 2. Impact of file creation on a moderately small output data that is written every time. XGC application on Summit, fixed 40GB output at all scales, per step, 4TB data total in 100 steps. Total runtime of the application was 1240 seconds on 1024 nodes. The strong-scaling performance of ADIOS writing 40GB of data from increasingly more compute nodes (Stream file case, BP4 format) is completely shadowed by the cost of creating a new output file (folder) at every output step (File per step case).

checkpoint data. Each process could dump and retrieve all the checkpoint data from the BP file without interfering with other processes.

When a process was writing many variables, there were still too many IO requests overall from many processes. ADIOS buffered all variable outputs in memory and dumped them all at once into one location in the output file, therefore issuing one IO request per process in an output step. Thus, ADIOS traded memory consumption for maintaining scalability and maximum performance.

With the increase of the number of compute cores to reach petascale, HPC applications started running on hundreds of thousands of MPI ranks, which quickly proved to be a bottleneck when writing to a single file. ADIOS implemented a two-phase IO by aggregating data across processes and thus reducing the number of writers to the file. Maintaining a limited number of processes that request write operations at the same time was crucial to avoid crushing the file system, and to provide good IO performance at large scale.

One issue remained however, namely, that a single file could not exhaust the total bandwidth available from the file system. Therefore, the BP output format became a folder with multiple subfiles in it, directing aggregators to different file targets. When selecting enough but not too many aggregators, the application could use the entire file system bandwidth for a single large output. We explained this format in more detail and explored the aforementioned issues and our solutions in Liu et al. (2014) and showed great application write and read performance on then-current HPC systems. The first figure from this paper is shown here in Figure 1 to showcase the IO performance improvement, that applications experienced on early petascale machines when switching to ADIOS and its BP file format.

Years later, we studied the WarpX application IO performance on the pre-exascale computer Summit at the Oak Ridge Leadership Facility, in Wan et al. (2022) and showed that performing IO with subfiles and chunking of datasets was far superior to anything else, both for writing and for overall read performance when retrieving various sub-selections of multi-dimensional arrays.

In the beginning of ECP, we implemented ADIOS-2 from scratch and first we re-implemented the BP format as it was in the old code base, that we named BP3. As the file systems kept growing (e.g. a GPFS installation for Summit at Oak Ridge Leadership Facility, a Lustre installation for Theta at Argonne Leadership Facility), the BP3 performance was growing as well, and applications could produce large amount data with low overhead. On Summit, several application maintained over a terabyte per second write performance with ADIOS. Increasingly, another issue become a bottleneck in IO, namely, the cost of creating a directory with subfiles in it, that took several seconds when a large parallel application was doing it which became vastly more than writing datasets that were once considered large, see Figure 2. It was increasingly more beneficial to append new output steps into the existing dataset. However, BP3 was exponentially slowing down due to its metadata management, which happened to sort each variables metadata across all output steps, and therefore had to be completely updated every time a new output was appended. The BP4 file format organized the metadata per step, so that appending only increased the metadata but did not modify the existing one, and thus, it cost nothing to append new output steps. Moreover, in case of an unexpected failure, the existing output steps of a BP4 output would not become corrupted, even if the error occurred during writing, when the last output step became incomplete or corrupt. The latter was essential to convince any user to dump all simulation data into a single large dataset with many output steps, but the performance gain was substantial, when they did. Another added benefit of BP4 was that a consumer could treat it as an ”ADIOS stream” and start consuming the existing steps before the producer closed the output, then wait for incoming new steps and continue processing all steps until the producer stops.

Applications could easily produce several petabytes of output data in a day on pre-exascale machines using ADIOS BP format with only a few percent of overhead cost to their runtime. This success, however, opened up new use

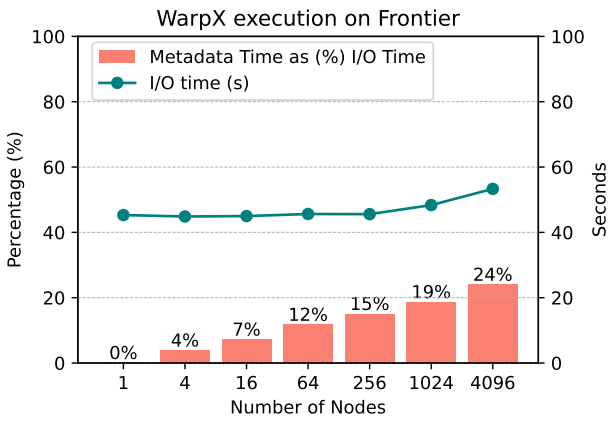

Figure 3. The Escalating Relative Cost of ADIOS Metadata Time in Comparison to Overall I/O Time

cases that brought new problems with scalability. First, the metadata size to maintain all information about the individual data blocks on disk is linearly increasing with the number of processes, number of variables and number of output steps. At some point, a reader may run out of memory just trying to process all that metadata when opening the file. Moreover, collecting and processing all this metadata is a good chunk of the overall write time and read time, decreasing the overall IO performance observed by the application. Second, considering the overall speed of ADIOS, new GPU applications had the luxury of attempting to output all data from all GPUs, but there was sometimes not enough memory on the host to manage all that data with ADIOS' aggressive buffering scheme. Therefore, we set out to design a better metadata representation, a faster implementation for metadata processing, better memory management, and support for GPU data in the latest BP5 format. In this paper, we describe the changes from BP4 to BP5, and then show ECP application performance on Frontier, the first open science exascale computer.

Metadata Concerns

Traditionally, the focus in HPC I/O has been getting application *data* to storage as quickly as possible and metadata costs have been neglected, sometimes for good reason. For example, if data in files is proactively organized so that data that was decomposed across multiple writer ranks is written into physically contiguous file areas, and/or readers and writers have *a priori* agreements on those locations, metadata may not be needed at all. However *a priori* agreement is untenable in many cases, and as noted above, writing from multiple ranks into physically contiguous file areas can have significant performance penalties. Therefore ADIOS, like many HPC I/O solutions Folk et al. (1999); Latham et al. (2003), creates metadata in the course of moving the data to storage in order to keep track of the particulars regarding where individual chunks of data are actually stored.

Additionally, because ADIOS doesn't require readers to have full *a priori* knowledge of what data is written and instead supports data discovery, metadata also includes the names, types and dimensionality of the written ADIOS variables. In the case of structures like ADIOS global arrays (in which a single logical array can be decomposed arbitrarily across many writer ranks), metadata for each written block also includes the N-dimensional geometry of the array as well as the N-D position and size of each written block, as well as either the value (for atomic data types) or information about where the data block has landed in storage (for arrays).

Prior to exascale systems, metadata was not as significant a performance concern as data storage, but as shown in Figure 3 which illustrates characteristics of WarpX I/O at various scales (measurements performed on Frontier), this is beginning to change. In particular, the figure shows the overall I/O time (green line, scale at right) and relative contribution of the metadata handling overhead within that I/O time (percentage represented by orange bars). With increasing scale, the per-rank I/O time remains relatively constant with ADIOS writing data from all CPU ranks. However, the ADIOS metadata gather and write time increases to as much as 24% of the total I/O time. These trends are not unique to WarpX, but have also been captured for other metadata heavy applications, including E3SM. Therefore a re-examination of metadata was a significant focus of our BP5 efforts.

Metadata basics

Abstractly, each ADIOS output step produces both data and metadata on each writer rank. Metadata and data have always been handled separately, in part because of the assumption that data is large and must be handled carefully, while metadata could be handled with less caution. But it is also the case that because data and metadata are consumed differently by the reader, which may pick and choose what data it wants, but generally requires significant information from metadata to support that functionality. Specifically, ADIOS reader-side semantics allow a reader rank to ask if a variable was written in a given step (`InquireVariable()`) and to ask for its value (for variables of atomic data types) or details about blocks written by the writer (for arrays). For arrays, the reader may `Get()` a specific hyperslab, which might be derived from blocks contributed by multiple writers. In order to correctly implement this, each reader rank must essentially have access to *all* the writer side metadata (that is, the metadata block from each writer rank for that step).

In order to support this reader-side requirement for complete access, the internal ADIOS writer engine generally:

1. gathers all the metadata blocks to rank 0 using MPI operations
2. processes the gathered data to produce a single aggregate block
3. writes the aggregate metadata to a single location.

In the reader implementation of `BeginStep()`, reader rank 0 reads that aggregate metadata and distributes it to all reader ranks where it is placed into internal data structures for later queries. Those internal data structures are then traversed during ADIOS read operations both to answer application queries about variables and find the location and geometry information for data blocks (which is necessary to acquire the actual data to satisfy `Get()` operations).

Metadata handling is characterized by three basic costs: writer-side metadata consolidation, processing and storage, reader-side metadata reading/distribution and installation, and reader-side data structure traversal costs. Usually the overhead of metadata handling are inconsequential, but when an application is producing thousands of variables on tens of thousands of processes and repeatedly many times, this isn't always the case.

BP5 metadata design

Upon examination we identified a number of areas in BP4 metadata handling where there seemed to be room for improvement. In particular BP4 metadata:

- is relatively large, always including full variable name and type information with every appearance of information about a data block
- is built-up piecewise as ADIOS Put()s occur, which requires full "parsing" when later read
- doesn't take advantage of the often-repetitive nature of HPC communication. I.E.:
  - Often each rank is writing the same variables as every other rank
  - Step `N+1` tends to repeat the pattern of step `N`

Additionally we observed that because the BP4 metadata for an array entry contained the exact byte-offset-in-the-file location of the data, BP4 metadata construction was deeply entangled with the BP4 routines responsible for writing data to disk. While not a performance problem *per se*, this property of BP4 metadata made it more difficult both to change the way data was stored and handled and made it more difficult to implement the more innovative data handling techniques described later in this paper.

In order to tackle these problems, BP5 represents metadata using FFS (Fast Flexible Serialization) Eisenhauer et al. (2011a), a package developed for high-performance messaging. FFS has a lot of features, including heterogeneity management, support for message morphing Agarwala et al. (2005) and dynamic code generation for filtering and processing, but for the purposes of ADIOS the feature that mattered was FFS' separation of "message format" from "message data". Specifically, if one thought of a message as a C-style structure (extended with strings, dynamic arrays, etc.), then the "message format" is the logical equivalent to the textual struct definition, while the "message data" is the actual representation of that struct in memory. In FFS messaging, each message format has an associated "format ID", a cryptographic hash over the struct definition. Message data is tagged with the format ID of the struct definition that describes it, and upon receipt, these format IDs are used to quickly identify the handler or specific processing associated with that message type.

In FFS-based messaging systems, such as EVPath Eisenhauer et al. (2009), FFS is generally used to specify the details of a compile-time structure to be transmitted as a message (this is described in detail in Eisenhauer et al. (2011b)). In this usage, FFS encodes structures directly from sender memory and upon receipt, transforms incoming messages into local in-memory structures resolving any differences in memory representation between sender and receiver (such as byte-order, pointer sizes, field alignment). However in

```
adios2::Dims shape { 500, 150 };
adios2::Dims start {  0, 0 };
adios2::Dims start2 {  200, 75 };
adios2::Dims count { 100, 75};
auto VarDouble = io.DefineVariable<double>("scalar_r64");
auto VarArray =
  io.DefineVariable<int8_t>("array_i8", shape, start, count);
auto VarLong = io.DefineVariable<long>("scalar_long");
engine.Put(VarDouble, dval);
engine.Put(VarArray, array_data);
engine.Put(VarLong, lval);
VarArray.SetSelection({start2, count});
engine.Put(VarArray, array_data);
```

Figure 4. Example ADIOS code segment.

BP5 we use FFS in a more dynamic way. Specifically we represent our step metadata as an aggregate "structure" that never exists in any compile-time format, but the structures description and its memory representation are built up on the fly on each writer rank as data is presented to ADIOS for output.

With each Put() of an ADIOS variable on a writer rank, we build up a virtual "structure" (the message data), along with a corresponding struct definition (the message format). The struct definition includes relatively fixed information about the variable being output, such as its name, type and dimensionality, while the structure itself contains only the few bytes of value (for atomic data types) or the geometry and block location information (for arrays).

For example, consider the simple ADIOS code sequence in Figure 4. When this sequence of Put()s is executed in an ADIOS writer rank, we build up a structure description and memory representation such as is shown in Figure 5.* Note the color correspondence between the Put() calls in Figure 4 and the structure fields in Figure 5. For the first step on each rank, each `Put()` operation (which is not collective in ADIOS) builds up the structure description and corresnponding entries in the metadata output buffer piece by

*N.B. This representation is conceptually correct but has been simplified for presentation. For efficiency, the actual implementation uses nested structures and encodes more information in the field names (variable name, as well as data types, applied operators and other internal ADIOS information).

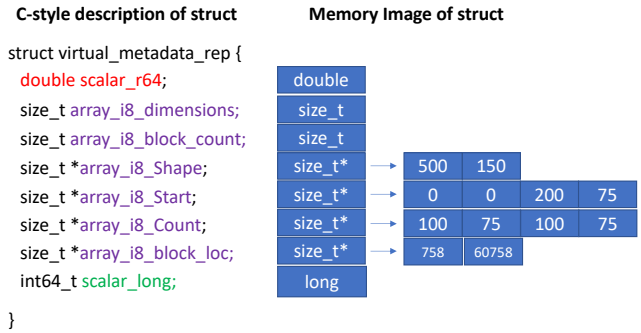

Figure 5. Virtual metadata structure description (left) and memory representation (right) after the execution of the code blocks of Figure 4.

piece. Subsequent steps use already created structures unless the Put sequence changes. For example, the code sequence of Figure 4 is happening on the first step, when the `Put()` of `VarDouble` happens, BP5 adds the field "scalar_r64" (red in Fig 5) to the structure description and copies the data value into the metadata output buffer (ADIOS scalar variables are always stored directly in metadata.) When the `Put(VarArray)` happens, a number of fields (in purple) are added to our virtual structure declaration and to the output buffer. These fields track the number of dimensions the output array has, how many output blocks (initially 1), the Shape, Start and Count values specified in the put, and finally, the location of the actual data (relative to the start of this rank's output data block). Note that several of these fields are pointers to dynamically sized arrays (a feature supported by FFS structure marshalling). Similar to what occurred with the `Put(VarDouble)`, the `Put(VarLong)` in green adds a single field to the virtual structure declaration and to the output buffer. ADIOS allows multiple chunks of a global variable to be output from any rank on a single step, so here we depict a 2nd `Put(VarArray)`. In this case, because `VarArray` already appears in the metadata structure, rather than adding more fields, BP5 increments the count in the existing `array_i8_block_count` field and appends the 2nd set of Start and Count information after those of the first `Put()`. More `Put()`s would follow a similar pattern, extending our virtual structure or adding to existing fields, gradually building up what is essentially a memory image of the metadata for this step on this rank.

Upon ADIOS `EndStep()`, the struct definition (the left side of Figure 5) is registered with FFS and a format ID calculated. The struct data (the in-memory representation of the ADIOS metadata on the right side of Figure 5) is then encoded using FFS and tagged with the struct definition's format ID. *These two items together, the struct definition and the struct data, completely describe the metadata for a single rank in a self-describing way creating BP5's "two-level metadata"*. Generally the struct description will be the larger of the two with the relative sizes impacted by factors such as the relative frequency of scalar variables vs. arrays and the length of the variable names employed by the application. In the simple example of Figure 5, the struct definition is 484 bytes and the encode data structure is 136 bytes. If a single rank and a single output step were all that were involved, this separation of struct definition (called "metametadata" in BP5) and struct data (simply called "metadata") would merely add additional overhead and the total metadata size (metadata and metametadata together) would likely be larger than the BP4 approach. However, with multiple ranks and multiple steps there are advantages. Because of the nature of HPC applications, the same set of variables tend to be written by many different ranks. Because the metametadata essentially describes the *structure* of the metadata that is produced and changes only when the set of written ADIOS variables changes, different ranks (which perform `Put()`s without coordination) independently generate the identical metametadata block with identical format IDs and because those IDs are a cryptographic hash of the metametadata block's contents those blocks will have the same IDs.

Each unique metametadata block must be available to the reader in order to understand the metadata, so the metametadata blocks must be aggregated like the metadata blocks, but the nature of metametadata offers opportunities for optimization. First, on a single rank, the metametablock is always offered for aggregation on the first step, but thereafter it is only offered for aggregation if the set of variables being written changes (typically rarely in our experience of HPC applications). Second, when the metametadata and metadata from multiple ranks are aggregated only the unique metametadata entries need to be kept. Because each block's ID is a cryptograph hash of its contents simply comparing hashes is an easy way to determine uniqueness. However, because uniqueness across ranks can only be determined after metametadata is aggregated and every rank must offer its metametadata for aggregation on the step, the amount of information each rank contributes to the aggregation can be large. In fact, early implementations of BP5 quickly hit MPI's 31-bit size limitation for gathering data as the total size of repeated metametadata spread across over all processes would become larger than 2GB. This was resolved by introducing a 2-level metadata aggregation into the BP5 engine and eliminating duplicate metametadata entries at each level of aggregation, thus keeping its size in check on each rank at all times.

	Number of writer ranks				
Engine	896	1792	3548	7168	14336
BP4	.145	.277	.572	1.173	2.385
BP5	.066	.141	.265	.512	1.099

Table 1. Per Step writer side metadata cost in seconds averaged 1000 steps with moderately-intensive metadata (525 arrays, 1000 variables).

Comparison of metadata overheads

Table 1 shows a comparison of BP4 and BP5 writer-side metadata overheads, including per-rank serialization, aggregation (MPI_Gather) and computation overheads of merging prior to writing to storage. The authors have chosen this as a good relative measure of metadata performance because it is independent of storage speed and also representative of the use of the BP5 format in other engines, such as in ADIOS' Sustainable Staging Transport where metadata is aggregated similarly but is sent over network connections to simultaneously running ADIOS consumers. BP5's reduced metadata size produces improvements in file storage and network transmission times, but it also shows gains in computational overhead as compared to previous implementations. In BP4, after the metadata is gathered to rank 0, each metadata block is deserialized into local data structures, then the block information is merged to consolidate information about each variable, then the metadata is serialized again into an aggregate block. In contrast, BP5 does no rank 0 processing of metadata, but simply concatenates the individual blocks. As is evident in Table 1 this produces a significant savings over BP4. This measure does include the `MPI_Gather()` time, so it is influenced by metadata size, but the most of the gains are due to reduced writer-side processing.

However, as BP4 is essentially doing up-front metadata organization (assembling into a coherent whole metadata that

	Number of writer ranks				
Engine	896	1792	3548	7168	14336
BP4	.074	.147	.302	.596	1.24
BP5	.113	.235	.467	.934	1.92

Table 2. Reader side metadata installation cost in seconds.

was produced over the disparate writer ranks) and rather than eliminating that cost, BP5 could just be shifting it to the reader. In order to address this concern, we also have to measure the costs of making the information in the metadata available for query in the reader. Both BP4 and BP5 read raw metadata on rank 0 and use MPI collectives to broadcast it to the other reader ranks. However, due to the different designs of BP4 and BP5, BP4 generally does most of its metadata processing during file `Open()`, while BP5 does it on a per-step basis during `BeginStep()`. Because of these differences we focus on the computational overhead of preparing the metadata for use rather than its read or broadcast cost. This focus also serves our purpose of determining if BP5's approach has simply shifted organizational costs from the writer to the reader.

Table 2 gives reader-side metadata preparation (or "installation") costs. These costs are incurred on each reader rank, but since each writer produces metadata they vary with the number of writer ranks. While Table 1 showed that BP4's writer-side metadata organization adds an additional cost, we do not see those significant costs shifted to the reader in BP5. While the less-organized metadata does make BP5 metadata slightly more expensive, the additional cost is more than offset by the savings in writer-side aggregation costs. This confirms that BP5's approach to metadata represents an overall improvement over BP4 and is not just shifting costs from the writer to the reader. Additionally, as shown in Figure 3, BP5 metadata is roughly `4x` faster than BP4 for *traversal*, that is, for actually navigating the installed metadata to locate the information necessary to satisfy a query. (Here, the measurement represents the total time to process an ADIOS `Get()` for each of the written variables while neglecting the actual data access time).

Given this, and in order to put the metadata overhead savings in the context of a full application, we compare the total write time and the produced metadata size with an E3SM application use case which happens to be the most difficult case for ADIOS as far as we know. In Figure 6, we can see that BP5's metadata size is less than a third of the BP4 metadata. The overall IO performance is improved by decreasing the cost of gathering and writing the metadata. Note that even though the almost 10 GB metadata produced by 21504 processes sounds a lot, the application data was 29 TB for that run, and therefore it is not the size of the metadata but the processing time of it that matters. In the last

	Number of writer ranks				
Engine	896	1792	3548	7168	14336
BP4	.224	.437	.879	1.71	3.46
BP5	.052	.112	.220	.448	.934

Table 3. Reader side metadata traversal cost in seconds.

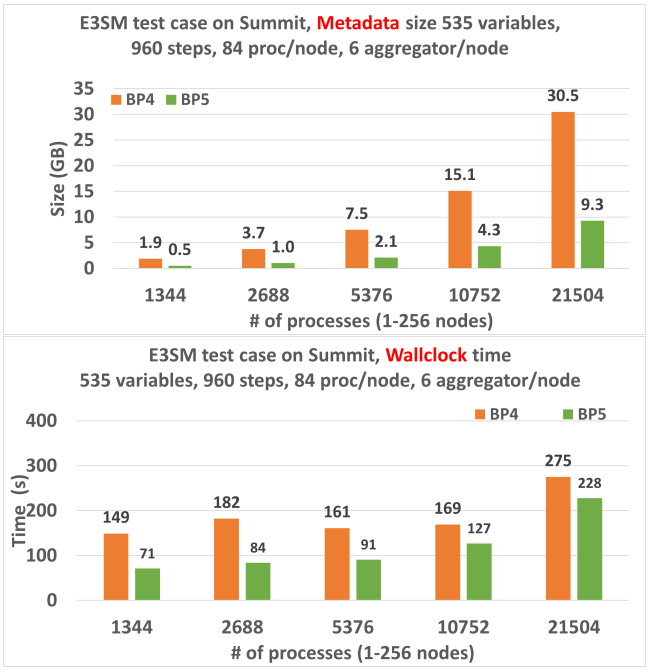

Figure 6. Metadata size and total runtime performance comparison of BP4 and BP5 with our E3SM test case. Data size on 21504 processes is 31 GB per output step.

section of the paper, we will present E3SM results comparing with the default parallel NetCDF IO driver, as evaluated and reported by their application team.

Memory management

While Figure 3 showed that metadata can be a significant performance concern, that was only the case because data storage was being handled efficiently. However, new architectures and applications are forcing changes in data handling as well. For a long time, ADIOS has successfully traded memory for performance. The BP3/4 engines always buffer the process' output in a large buffer, and create a single I/O request for the entire buffer in each process. Buffering avoids the most common issue with user level I/O, namely the *small writes*, where latency becomes a major player in downgrading overall performance by orders of magnitude. Second, since there are many of these large buffers, one per process, an aggregation scheme must be used to control the number of active writes at a given time to avoid the second major pitfall, *too many I/O requests*. BP3/BP4 uses a very expensive aggregation scheme in terms of network and memory usage in order to continuously write with a limited number of requests while eliminating any gaps between finishing a request and starting additional write requests. Contrary to expectations, the large memory consumption was never an issue for HPC applications as they have been severely limited by computation power compared to memory size. Similarly, networks in supercomputers have way much higher bandwidth for moving data than the underlying file systems, therefore, moving data from many processes to the select few aggregator processes paid off for decreasing the number of active writers to the file system. Even the largest producers, like global adjoint tomography Bozdağ et al. (2016), that produced over a petabyte of output, or particle-in-cell codes (like XGC, GTC fusion codes, and the

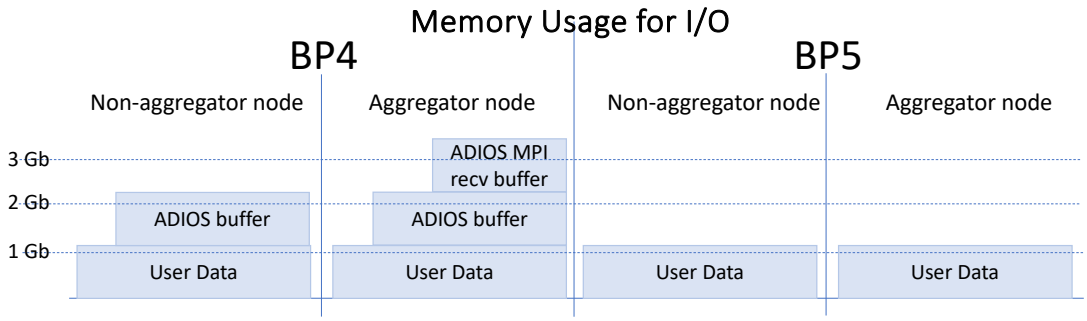

Figure 7. Memory consumption during I/O involving 2 processes and about 1.1 GB user data per process, BP4 vs BP5. Best case scenario where all user variables are larger than 4 MB.

PIConGPU particle acceleration code) that output all their particles for post-processing could run at the largest scales possible without any issues with memory or networking.

GPU computing in the time of exascale computers, however, changed the situation with memory. First, computation is performed on the GPUs, much faster than on CPUs, and second, the host main memory is about the size of the aggregate size of the GPUs in the host. IO still needs to be performed from the host and its memory therefore, the data has to be downloaded from the GPU to the host memory by the application and then pass that data to the IO library, like ADIOS. For applications like XGC or PIConGPU, there is simply not enough memory available on the host anymore for ADIOS buffering all data once again. Therefore, we had to design a more conservative memory management as well as aggregation mechanism, that did not consume so much memory. We devised two separate approaches for this problem, one is chunked memory buffer, described here, while the other is GPU-aware IO, where the user can simply pass the GPU data pointers to ADIOS, which will then copy the data into the ADIOS buffer on the host memory. The latter is described in the next section.

The single large buffer was used to eliminate small individual IO requests, but it is not so important for large user arrays. We can afford to issue a handful of large IO requests on one process in quick succession instead of a single one without performance penalty as the total number of IO requests across the application at any given time does not increase. The new `chunked memory` buffer is composed of chunks allocated and managed by ADIOS to buffer all small user data in one or more large buffer chunks. Large user arrays are not copied anymore, rather, the user data pointer is added to the list of chunks. During the writing phase, the process issues the write requests in a loop, and all the data from the chunks go into a single file into a contiguous space. Note, that what is small and what is large is dependent on the system and the application, therefore, the user can set the limit for buffering. The default size is 4 MB. Any user array smaller than this limit is still copied into a large contiguous buffer.

Figure 7 shows the improvement of the memory consumption from BP4's buffering to BP5's chunked buffering where all user variables are larger than the buffering limit. Two processes are shown (left vs right), where process 0 is the aggregator. BP4 engine on the top half, and BP5 on the bottom half. In BP4, the ADIOS buffer holds a copy of the user data on each process, and during aggregation, process 1 sends its data asynchronously to process 0, hence the additional receiving buffer on process 0. The aggregator process is alternating between two buffers for writing one to disk while at the same time receiving data from a another process. In case of BP5, the pointers to user variables are kept as the entries of the "buffer chunks", with no ADIOS-owned buffer chunk in this ideal scenario. Hence, there is no copy of the user data. Moreover, the aggregation strategy is different here, where every process writes its own data to disk, and only the order of processes, i.e., who is writing at a given time, is controlled by the aggregation. In this scenario, ADIOS uses no extra memory for IO.

GPU Aware IO

The BP4 engines is operating exclusively on data allocated on the Host. Application using the GPU are required to transfer the data to the Host before handling it over to ADIOS. During the final years of the ECP project, the GPU-backend in ADIOS has been developed allowing applications to pass directly the GPU pointer for both reading and writing data from/to storage. Figure 8 shows the API and logic changes inside the BP5 engine to support the GPU-backend. An ADIOS variable holds two new concepts: the memory space and layout. These can be either set by the user directly or set by the `Put/Get` function by ADIOS automatically detecting the memory space used to allocate the passed buffer (`Host` or `Device`) and setting the layout corresponding to the memory space (`Right` or `Left`). Based on the default layout used by the programming language used by an application (e.g. row major for C++, column major for Fortran) ADIOS will know if there is a mismatch between the variable memory space and what BP5 expects and will adjust the ordering of the dimensions to keep the correct local offsets for global arrays (more details in the following subsections).

Once the user or ADIOS sets the memory space and layout for a variable, the BP5 engine can apply operators directly on the GPU pointer provided by the user. If the operator is executing on the GPU, this would remove the need for the operator to allocate memory on the device and copy the data from Host to Device. The GPU-backend also allows the operator to return data stored on either the GPU or CPU. Depending on what the user provides and optionally what the operator returns (if any operator is applied) BP5 will compute the min and max for each block using the

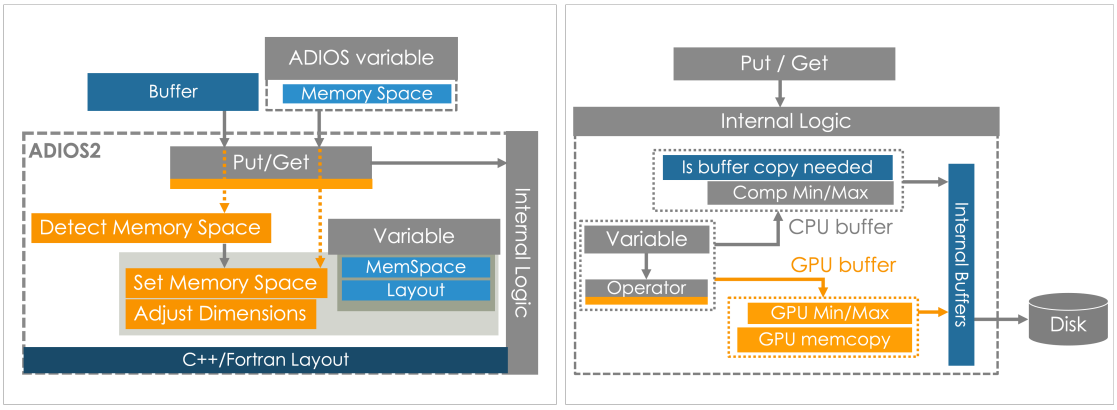

Figure 8. GPU backend framework inside the AIODS2 library: 1) changes to the API on the left; and 2) changes to the internal logic for moving the data from the user space to storage

corresponding device for each memory space. For GPU buffers, the memory has to be copied to ADIOS internal buffers before it can be stored to the disk. For CPU pointers this step can be skipped depending on the strategy used by the BP5 engine (more details in the following subsections).

GPU-backend implementation

The API changes to ADIOS are minimal and agnostic to the implementation solution used to enable the GPU-backend. Calling the `Put` and `Get` functions with GPU pointers is exactly the same as for CPU pointers. The only changes are represented by two new functions: i) one that allows users to set the memory space of a variable to bypass the internal automatic detection; and ii) one that allows to get the Shape of a variable on the read side according to a given layout. In this section we present the implementation considerations of introducing these two concepts into a variable object.

Automatic detection of the memory space of a given buffer uses functions provided in Cuda, HIP and SYCL (e.g. `cudaPointerGetAttributes` is used for Cuda) to check if the data array provided by the user has been allocated on the Host or Device. The detection time is in order of μs regardless of the backend and system. This is negligible compared to the write time for most applications since the detection is triggered only once for the first `Put/Get` call on each variable. Once the memory space is set to Host or Device, it is pinned to this value for the entire existence of the ADIOS variable. Users can also manually pin the memory space of a variable if they want to avoid detecting the space automatically. Even though the overhead of detecting the memory space is low, this could be desirable for example when a buffer is allocated on the unified memory space and the performance implications are not completely transparent (more on this in the performance section).

An array layout describes how multidimensional arrays are stored in linear storage. A layout right, or row major ordering, provides a layout mapping where the rightmost extent has stride 1, and strides increase right-to-left as the product of extents. A layout left, or column major has a reversed mapping with strides increasing left-to-right. ADIOS was built on the assumption that Host multi dimensional (MD) arrays are typically using row major when the code is written in C++ and column major for Fortran. The ADIOS library itself handles the interoperability between

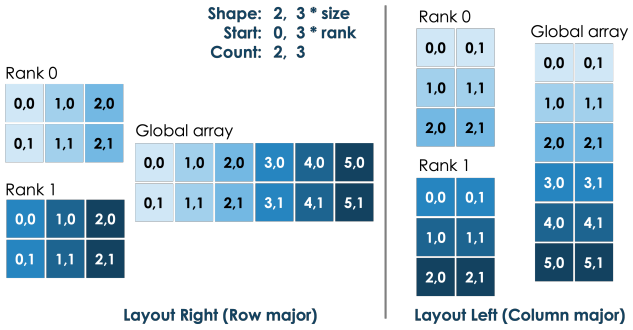

Figure 9. Example of constructing the global array from local data distributed over two ranks using different layouts

these two ordering by setting the programming model when initializing the ADIOS library. This is done at the level of the entire code. On GPUs, the typical mapping uses column major regardless of the ordering used by the programming language for the reason of loop computing performance. In addition, programming models that are just starting to support multi-dimensional arrays natively, like mdspan Hollman et al. (2019) in C++23, boost Koranne (2011), and Kokkos Trott et al. (2022), now can support both layouts. Adding the concept of layout inside each variable allows ADIOS to handle the dimensions of each multi-dimensional variable regardless of where the array for output/input was allocated, what layout is being used at write and read, and in what programming language the producer and consumer applications were written.

Since ADIOS implements the concept of MD arrays by default, users need to provide the shape of the global array as well as the local shape of the array that each rank is responsible for. For example, in Figure 9, ranks are writing a 2x6 global array with each rank populating half of the global array. The GPU-aware backend allows different layouts for the global array (the figure is illustrating the same array using Layout Right and Left) without requiring the user to update the code for each case. The user defines the shape of the global array and ADIOS adjusts the dimensions for each rank according to the buffer layout and memory space.

To enable the computation of the block statistics (min/max) on the GPU and the copy of GPU allocated data into the ADIOS internal buffers, ADIOS is using two GPU

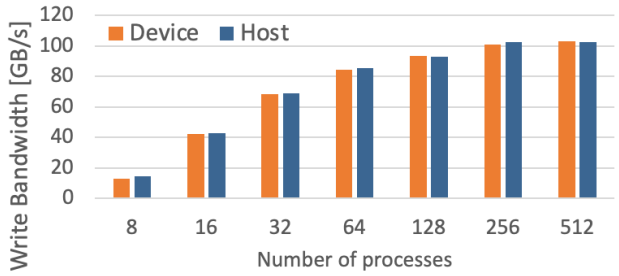

Figure 10. Write bandwidth at 10GB per process on Summit (allocated on the host and without computing min/max) when using BP5 with and without the GPU backend

backends using a Cuda and a Kokkos based implementations. The Cuda backend has the advantage of providing the desired functionality with the native NVidia GPU library but it only supports NVidia devices, while the Kokkos backend is portable on any system with a GPU but requires the Kokkos library. The performance between the two backends is extremely similar, thus for the rest of the paper, we only show the results using the Kokkos backend. Moreover, for buffers allocated on the Host, the GPU-aware ADIOS uses the initial BP5 logic so there is only an overhead for detecting the Host buffer (which is negligible). Thus, the restuls when using Host buffers when the GPU-backend is enabled are identical with the original code. For the rest of the paper, Host buffers will only use BP5 without the GPU-backend enabled.

Figure 10 shows the write bandwidth when writing 1GB of data per rank (without computing the min/max) using the GPU-backend and the original BP5 code to highlight the overheads of using GPU allocated buffers. The bandwidth is computed as the total amount of data written divided by the total time it takes ADIOS to write the data (the time between when the Put is called to when the data is writen to the storage). This means the bandwidth for Host pointers does not include the time it takes the user to transfer GPU data to Host. In our experiments, we measured an approximate bandwidth of 20 GB/s for transferring data between the device and host on an AMD Mi250 for 1 GB of data, which represents around 50 ms. Overall this overhead represents less than 4% of the total write time. Other GPU architecture might have higher impact, but if the simulation/user data is on the Device, this cost will have to be paid anyway either on the user side on within ADIOS. The BP5 engine does not always copy the host data to internal buffers (if the host array is larger than 4MB, BP5 uses by default the user buffer directly for writing to storage). The sync parameter can be used in a Put call to force ADIOS to copy the data to internal buffers. This is useful, for example, when the user wants to reuse their buffer after a Put call. For Device buffers, the GPU-backend always has to copy the user buffer leading to the performance difference between the GPU-backend and the default strategy for BP5 in Figure 10. Note that for the GPU-backend the user can immediately reuse the buffer after a Put call and also that the BP5 default times do not include allocating data on the host and copying the data between device and host. If these times are included the overhead for the GPU-backend compared to BP5 will be almost 0.

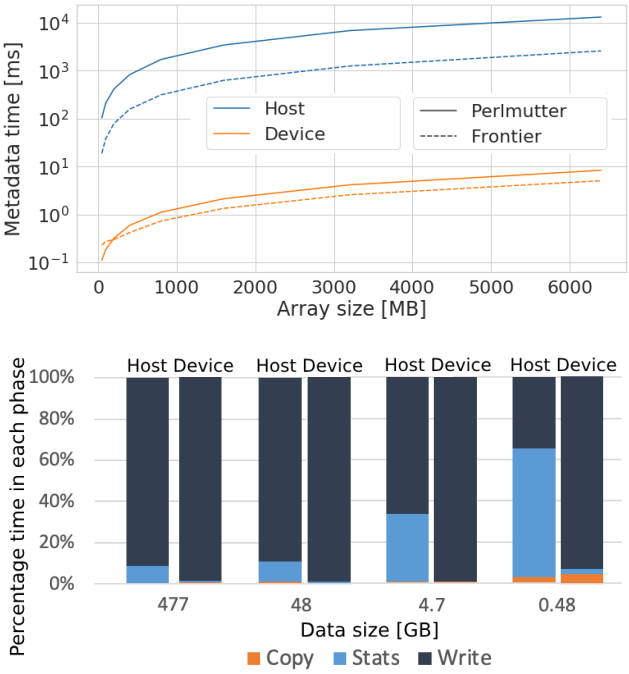

Figure 11. On node performance of the GPU-backend compared to BP5 using Host pointers: 1) time spend in BP5 related to computing the stats; 2) percentage of time spent in different parts of BP5: write time, stats computation, copy

As long as the simulation is using Device pointers, the memory footprint of the GPU-backend is the same as the default BP5 with the difference that the simulation will not be the one handling the Host pointers. The GPU-backend will use less memory compared to the BP5 in Sync mode since the Host array will not have to be kept in two copies (the host user buffer and the ADIOS internal buffer).

GPU-backend performance

In this section we look at the performance of each component in BP5 influenced by the GPU-backend, mainly: i) we investigate the performance of computing the stats (min/max values) for each block of data; ii) we show the impact of allocating buffers on the unified memory space; iii) we show the performance of operators that can be applied to data before writing and have the option of using the Device.

Stats computation. ADIOS computes by default the min and max value of each block for each variable (where a block is represented by data on one rank). Figure 11 top shows the performance of computing stats for different data sizes using Host and Device pointers on different architectures for one node runs. For very small array sizes (less than a few MB of data) the overhead of launching kernels to compute the stats exceeds the speed-up of using the GPU. However, using the GPU is one or two orders of magnitude faster than computing the stats on the CPU even for small array size. The process is embarrassingly parallel since the stats are independently computed on each rank so these results hold at scale as well. Figure 11 bottom show the percentage of time spent in each stage of the writing process inside ADIOS (POSIX write, stats computation, copy the data to the internal buffers). For the Host results, the user buffer is used to write data to storage so the cost of copy only accounts for updating the

Figure 12. I/O bandwidth for one step of S3D data (1.5TB total size) accessed by 900 ranks using Host or Device pointers

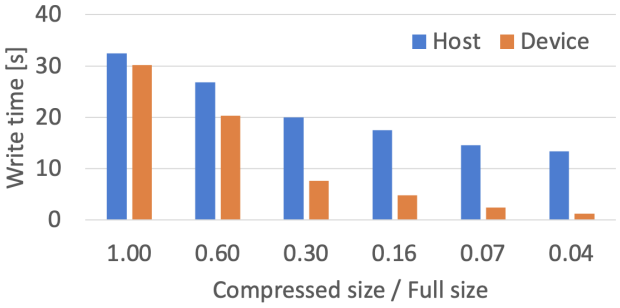

Figure 13. Write time with BP5 for 9 GB of data per rank allocated on Host or Device, when applying the ZFP compressing operator (for different compression rates)

metadata and it is very low. As in our previous results, since the copy between device and host is not happening inside ADIOS, this time is not measured in the figure for Host pointers but should be taking into consideration is the user code is dealing with Device data. The results in figure 11 are single node, at scale the write time might increase slightly, making the percentage spent in computing the stats slightly smaller. Overall, the results clearly show that for the GPU-backend, computing the stats becomes almost negligible.

To better understand the GPU-backend performance at scale, we ran the S3D application Im et al. (2012), a code that solves a direct numerical simulation of turbulent combustion and we did separate runs using Host and Device pointers to read and write the data. The S3D application is outputting 1.5 TB of data, at every step, containing five 3D and one 4D global arrays. Figure 12 shows the write and read bandwidth for BP5 when using 900 ranks spread out on 17 nodes and using 68 GPUs. The amount of data per rank is 1.3 GB for the 4-D array and 400 MB divided between the rest of the 3-D arrays. BP5 using Host buffers does not copy the data to internal buffers. When using device buffers, the overhead of transferring the data from device to host is completely hidden by the speed-up in computing the stats for the 4-D array, achieving a 20% speed-up compared to using Host buffers. The read time is slightly higher when using Device pointers due to the cost of moving the data to the device after reading (but the same cost would have to be paid anyway on the application for GPU codes).

Unified Virtual Memory. UVM is a single memory address space accessible from any processor in a system. Depending on where the memory is physically initially allocated, accessing data could trigger page migrations and will have different order of magnitude performance. We made experiments on NVIDIA Quadro RTX with Cuda 12 and noticed that the detection CUDA detection system inside ADIOS identifies UVM allocated buffers as Host buffers regardless of where they were initially allocated. While the data is reachable from the CPU, computing the stats and moving the data to storage encounters a penalty in this case. On a NVIDIA Quadro RTX, the page migration bandwidth was around 1 GB/s compared to around 50 GB/s for explicitly using Kokkos to copy the memory to the host. Additionally the stats would not be computed on the GPU when Host is detected as the buffer memory space. For 1GB of data per rank this slowdown would account for spending 1s compared to less than 0.05s in stats computation for a write operation of around 2-3s. Other processors/drivers will have a different trade-off, but UVM performance needs to be better understood when used together with the GPU-backend in ADIOS. Our recommendation is for users to set the ADIOS backend to the appropriate memory space for each given architecture and not rely on the automatic detection when using UVM. We plan to investigate this matter further in the future.

GPU-aware operators. ADIOS is capable of attaching operators to data and automatically applying them before writing data to storage. One type of operators frequently used is a compressor like ZFP Diffenderfer et al. (2019) or Mgard Gong et al. (2023). If Device pointers are being used by the simulation, the GPU-backend in ADIOS will forward the Device pointer directly to the operators (if they are capable of running on the GPU). Figure 13 shows the total write time when applying the ZFP compressor operator to 9 GB of data per rank allocated on either Host on Device. Depending on the memory space used, ZFP is compressing the data on the device or host before returning control to ADIOS that will afterwards write the compressed data to the storage. The horizontal ax in the figure is represented by the compression rate (the ratio between the compressed data over the total amount of data). For the first bar (compression rate of 1), the data is not compressed so the 10% performance difference is given by the speed-up of computing the stats on the Device. Overall, BP5 is able to compress and write data much faster when using the GPU-backend. For a compression rate of 0.16, for example, the data per rank is compressed to 1.4 GB (regardless of the backend) and the write time is decreased by a factor of 3.6x due to both compressing the data and computing the stats on the GPU.

Application performance

WarpX particle accelerator simulations

WarpX is an AMR-based particle-in-cell (PIC) accelerator physics modeling code that uses the AMReX framework. The winners of the 2022 Gordon Bell Prize Fedeli et al. (2022) used WarpX to successfully simulate a Laser Wake-field electron Accelerator targeting a hybrid solid-gas target, a large computational problem requiring an exascale computer.

OpenPMD Huebl et al. (2015) is an open meta-data schema that provides meaning and self-description for data sets in science and engineering. WarpX uses openPMD for application level I/O while in turn openPMD uses ADIOS2

or HDF5 as drivers for permanent storage. The ADIOS2 driver also enables staging data between applications using the openPMD API.

The BP4 file format was developed during the ECP project to improve the WarpX application's IO performance, by eliminating the cost of appending new output steps to an existing dataset and doing that robustly so that in case of an application or IO failure the entire dataset would be still readable up to the one before the last output step. We also studied both write and read performance of ADIOS in WarpX in Wan et al. (2022) and showed that performing IO with subfiles and chunking of datasets was far superior to anything else, both for writing and for overall read performance when retrieving various sub-selections of multi-dimensional arrays.

WarpX-related BP5 development was not for improving the I/O performance of WarpX further, but for decreasing the memory usage of ADIOS. This application writes all particle and field data from the GPUs to disk, writing about 80% of the GPU memory content. The new chunked memory management avoids making a copy of the large variable during IO in each local process. Additionally, BP5's new data aggregation implementation avoids sending one process output at once to the aggregator, further decreasing the overall memory consumption during the writing phase.

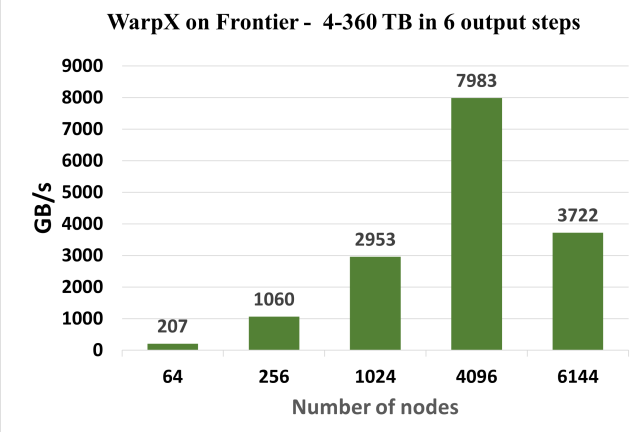

Figure 14. WarpX I/O performance on Frontier during early access using ADIOS. Best runs at each node size.

We evaluated WarpX I/O during the early access to Frontier, running the standard Laser Wake-field electron Accelerator test case that can be scaled up to arbitrary sizes. We only increased the output frequency beyond what is practical to stress test the IO without spending too much machine time for the evaluation. Figure 14 shows that the IO using ADIOS BP5 scaled very well up to the half of Frontier, reaching a peak of almost 8 TB/s throughput in the best run, and still held up well on 6K nodes. In general, large WarpX simulation runs can expect close to 5 TB/s writing throughput on Frontier and in practice, i.e., with less frequent outputs, the IO overhead compared to the runtime should be negligible.

XGC fusion simulation

XGC (X-point Gyrokinetic Code) is a global gyrokinetic particle-in-cell code, which specializes in the simulation of the edge region of magnetically confined thermonuclear fusion plasma. This code has been using ADIOS since before the first release of the IO library and has been driving the requirements for large scale IO development from before petascale computing. In the ECP Whole Device Modeling Application project, XGC was used to simulate the edge of the plasma, the turbulent region of the tokamak, while it was strongly coupled to another code (either GENE Merlo et al. (2021) or GEM Cheng et al. (2020)) that simulated the core of the plasma. ADIOS staging was used to perform the frequent data transfer between the simulation codes, as well as to the many in situ visualization and performance tracing tools running at the same time.

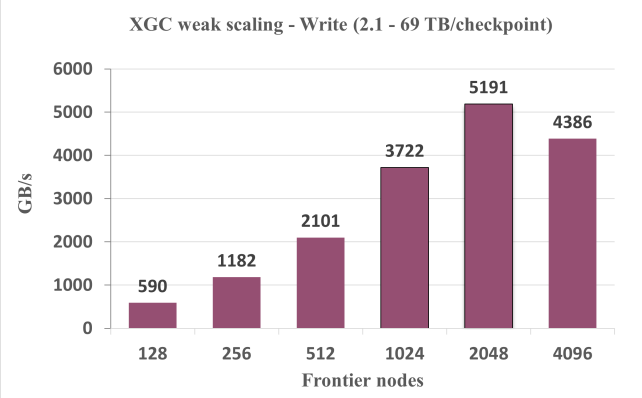

Figure 15. XGC checkpoint I/O performance on Frontier during early access using ADIOS. Best runs at each node size.

We measured the checkpoint writing of XGC when running the `XGC_ITER_15MA` case, one of the largest ITER fusion device simulations that the XGC team was using at that time. This simulation produced a 69 TB checkpoint when running on 4096 node. We measured XGC after the WarpX studies, still in the early access phase of Frontier.

Simple Cloud Resolving E3SM Atmosphere Model (SCREAM)

ADIOS was used as one of the I/O back-end for the ultra high-resolution atmospheric simulations Taylor et al. (2023) that won the 2023 Gordon Bell Prize. The Simple Cloud Resolving E3SM Atmosphere Model (SCREAM) was run on almost the entire Frontier, with a 3.25 km horizontal resolution and 128 vertical layers with a model top at 40 km for global cloud resolving. SCREAM uses the Software for Caching Output and Reads for Parallel I/O (SCORPIO) library, which provides an interface layer to various I/O libraries, including Parallel NetCDF Latham et al. (2003) and ADIOS for HPC, for reading input and writing model output as well as restart files.

We have worked closely with the SCORPIO development team during the ECP project to develop the ADIOS driver in SCORPIO. We have used various test cases (atmospheric (F case), ocean-ice (G case) and land simulation (I case)) with different IO requirements, and finally a SCREAM test case. The performance of the BP5-based ADIOS driver was 2-16x faster than the second best PNetCDF driver as seen in Figure 16. It is worth to point out that the F-case and its higher resolution twin, the SCREAM case, would run out of memory when writing to the BP4 format. The new metadata design and the new memory management of BP5

was necessary in this project to be able to output all the data from this applications.

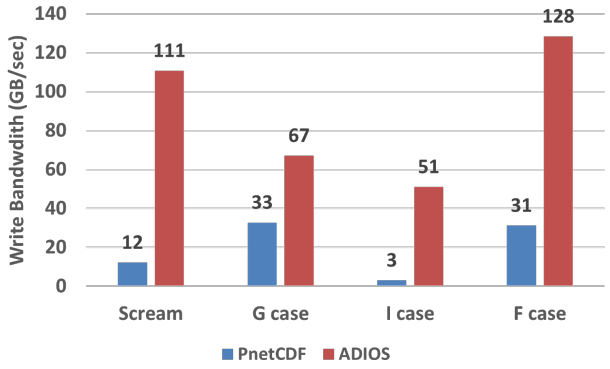

Figure 16. E3SM I/O performance on Frontier using ADIOS vs PNetCDF. SCREAM, ocean/ice, land, and atmospheric cases. Result provided by the SCORPIO team.

The runs for the Gordon Bell winning paper Taylor et al. (2023) achieved even better performance with ADIOS, achieving 135 GB/s overall write throughput on 8K nodes (and 234 GB/s on 2K nodes). It is worthwhile to mention that the largest data file, the actual simulation data they were interested in, was being written at 1576 GB/s, showing that ADIOS works best with large output arrays. ADIOS still struggles when lots of small arrays are written from across many processors due to the metadata cost that cannot be amortized with the data writing phase.

Notes on the application tests

We ran the WarpX and XGC tests in the early access phase of Frontier, when the machine network was not stable when large jobs were performing communication, and therefore we got too few successful runs at 4K and above sizes to establish a reliable throughput value. We present the best runs in this paper that shows the potential write speed of this applications using ADIOS. The E3SM tests results were reported by the SCORPIO developers and the SCREAM developers themselves at a later stage when they were evaluating E3SM for the Gordon Bell Prize. Nevertheless, we can see both from Figure 14 and 15 and from the numbers reported by SCREAM, that the write performance peaks around 2K-4K nodes for these applications, and drops beyond that. In previous smaller supercomputers, ADIOS was keeping performance close to the limits, not dropping at the top. In the future, we will investigate where ADIOS may over stress the file system at large scale and tune its runtime parameters or tune an algorithm inside ADIOS to keep the performance at the maximum.

Conclusion

In this paper, we have presented the latest design choices in the ADIOS system that provide scalable IO performance for exascale HPC applications. We discussed the implications of the file format design on its metadata sizes and designed a new one that is smaller and faster. We changed the memory management in ADIOS so that even the largest data producers can use it on the largest systems. We made ADIOS GPU-aware, so that applications can pass device pointers to ADIOS to write and read data from GPU to disk and back to GPU memory. Finally, we showed the IO performance of three applications from the Exascale computing program running on Frontier and show how our new designs can achieve performance close to the peak of the leadership computing systems.

Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

Agarwala S, Eisenhauer G and Schwan K (2005) Lightweight morphing support for evolving middleware data exchanges in distributed applications. *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)* : 697–706URL https://api.semanticscholar.org/CorpusID:2190657.

Bozdağ E, Peter D, Lefebvre M, Komatitsch D, Tromp J, Hill J, Podhorszki N and Pugmire D (2016) Global adjoint tomography: first-generation model. *Geophysical Journal International* 207(3): 1739–1766. DOI:10.1093/gji/ggw356. URL https://doi.org/10.1093/gji/ggw356.

Cheng J, Dominski J, Chen Y, Chen H, Merlo G, Ku SH, Hager R, Chang CS, Suchyta E, D'Azevedo E, Ethier S, Sreepathi S, Klasky S, Jenko F, Bhattacharjee A and Parker S (2020) Spatial core-edge coupling of the particle-in-cell gyrokinetic codes GEM and XGC. *Physics of Plasmas* 27(12): 122510. DOI: 10.1063/5.0026043. URL https://doi.org/10.1063/5.0026043.

Diffenderfer J, Fox AL, Hittinger JA, Sanders G and Lindstrom PG (2019) Error analysis of zfp compression for floating-point data. *SIAM Journal on Scientific Computing* 41(3): A1867–A1898. DOI:10.1137/18M1168832. URL https://doi.org/10.1137/18M1168832.

Eisenhauer G, Wolf M, Abbasi H, Klasky S and Schwan K (2011a) A type system for high performance communication and computation. *2011 IEEE Seventh International Conference on e-Science Workshops* : 183–190URL https://api.semanticscholar.org/CorpusID:7893343.

Eisenhauer G, Wolf M, Abbasi H, Klasky S and Schwan K (2011b) A type system for high performance communication and computation. In: *2011 IEEE Seventh International Conference on e-Science Workshops*. pp. 183–190. DOI:10.1109/eScienceW.2011.16.

Eisenhauer G, Wolf M, Abbasi H and Schwan K (2009) Event-based systems: opportunities and challenges at exascale. In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09. New York, NY, USA: Association for Computing Machinery. ISBN 9781605586656, pp. 1–10. DOI:10.1145/

1619258.1619261. URL https://doi.org/10.1145/1619258.1619261.

Fedeli L, Huebl A, Boillod-Cerneux F, Clark T, Gott K, Hillairet C, Jaure S, Leblanc A, Lehe R, Myers A, Piechurski C, Sato M, Zaïm N, Zhang W, Vay JL and Vincenti H (2022) Pushing the frontier in the design of laser-based electron accelerators with groundbreaking mesh-refined particle-in-cell simulations on exascale-class supercomputers. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press. ISBN 9784665454445, pp. 1–12.

Folk M, Cheng A and Yates K (1999) Hdf5: A file format and i/o library for high performance computing applications. In: *Proceedings of Supercomputing*, volume 99. pp. 5–33.

Godoy WF, Podhorszki N, Wang R, Atkins C, Eisenhauer G, Gu J, Davis P, Choi J, Germaschewski K, Huck K, Huebl A, Kim M, Kress J, Kurc T, Liu Q, Logan J, Mehta K, Ostrouchov G, Parashar M, Poeschel F, Pugmire D, Suchyta E, Takahashi K, Thompson N, Tsutsumi S, Wan L, Wolf M, Wu K and Klasky S (2020) ADIOS 2: The adaptable input output system. A framework for high-performance data management. *SoftwareX* 12: 100561. DOI:https://doi.org/10.1016/j.softx.2020.100561.

Gong Q, Chen J, Whitney B, Liang X, Reshniak V, Banerjee T, Lee J, Rangarajan A, Wan L, Vidal N, Liu Q, Gainaru A, Podhorszki N, Archibald R, Ranka S and Klasky S (2023) Mgard: A multigrid framework for high-performance, error-controlled data compression and refactoring. *SoftwareX* 24: 101590. DOI:https://doi.org/10.1016/j.softx.2023.101590. URL https://www.sciencedirect.com/science/article/pii/S2352711023002868.

Hollman DS, Lelbach BA, Edwards HC, Hoemmen M, Sunderland D and Trott CR (2019) mdspan in c++: A case study in the integration of performance portable features into international language standards. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. pp. 60–70. DOI:10.1109/P3HPC49587.2019.00011.

Huebl A, Lehe R, Vay JL, Grote DP, Sbalzarini I, Kuschel S, Sagan D, Mayes C, Pérez F, Koller F and Bussmann M (2015) openPMD: A meta data standard for particle and mesh based data. https://github.com/openPMD. DOI:10.5281/zenodo.591699. URL https://www.openPMD.org.

Im HG, Trouve A, Rutland CJ and Chen JH (2012) Terascale high-fidelity simulations of turbulent combustion with detailed chemistry. DOI:10.2172/1048137. URL https://www.osti.gov/biblio/1048137.

Koranne S (2011) *Boost C++ Libraries*. Boston, MA: Springer US. ISBN 978-1-4419-7719-9. DOI:10.1007/978-1-4419-7719-9_6. URL https://doi.org/10.1007/978-1-4419-7719-9_6.

Latham R, Zingale M, Thakur R, Gropp W, Gallagher B, Liao W, Siegel A, Ross R, Choudhary A and Li J (2003) Parallel netcdf: A high-performance scientific i/o interface. In: *SC Conference*. Los Alamitos, CA, USA: IEEE Computer Society, p. 39. DOI:10.1109/SC.2003.10053. URL https://doi.ieeecomputersociety.org/10.1109/SC.2003.10053.

Liu Q, Logan J, Tian Y, Abbasi H, Podhorszki N, Choi JY, Klasky S, Tchoua R, Lofstead J, Oldfield R, Parashar M, Samatova N, Schwan K, Shoshani A, Wolf M, Wu K and Yu W (2014) Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience* 26(7): 1453–1473. DOI:https://doi.org/10.1002/cpe.3125. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3125.

Merlo G, Janhunen S, Jenko F, Bhattacharjee A, Chang CS, Cheng J, Davis P, Dominski J, Germaschewski K, Hager R, Klasky S, Parker S and Suchyta E (2021) First coupled GENE–XGC microturbulence simulations. *Physics of Plasmas* 28(1): 012303. DOI:10.1063/5.0026661. URL https://doi.org/10.1063/5.0026661.

Polte M, Lofstead J, Bent J, Gibson G, Klasky SA, Liu Q, Parashar M, Podhorszki N, Schwan K, Wingate M and Wolf M (2009) ...and eat it too: high read performance in write-optimized hpc i/o middleware file formats. In: *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09. New York, NY, USA: Association for Computing Machinery. ISBN 9781605588834, p. 21–25. DOI:10.1145/1713072.1713079. URL https://doi.org/10.1145/1713072.1713079.

Taylor M, Caldwell PM, Bertagna L, Clevenger C, Donahue A, Foucar J, Guba O, Hillman B, Keen N, Krishna J, Norman M, Sreepathi S, Terai C, White JB, Salinger AG, McCoy RB, Leung LyR, Bader DC and Wu D (2023) The simple cloud-resolving e3sm atmosphere model running on the frontier exascale system. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23. New York, NY, USA: Association for Computing Machinery. ISBN 9798400701092, pp. 1–11. DOI:10.1145/3581784.3627044. URL https://doi.org/10.1145/3581784.3627044.

Tian Y, Klasky S, Abbasi H, Lofstead JF, Grout RW, Podhorszki N, Liu Q, Wang Y and Yu W (2011) Edo: Improving read performance for scientific applications through elastic data organization. *2011 IEEE International Conference on Cluster Computing* : 93–102URL https://api.semanticscholar.org/CorpusID:18030868.

Trott CR, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, Gayatri R, Harvey E, Hollman DS, Ibanez D, Liber N, Madsen J, Miles J, Poliakoff D, Powell A, Rajamanickam S, Simberg M, Sunderland D, Turcksin B and Wilke J (2022) Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33(4): 805–817. DOI:10.1109/TPDS.2021.3097283.

Wan L, Huebl A, Gu J, Poeschel F, Gainaru A, Wang R, Chen J, Liang X, Ganyushin D, Munson T, Foster I, Vay JL, Podhorszki N, Wu K and Klasky S (2022) Improving i/o performance for exascale applications through online data layout reorganization. *IEEE Transactions on Parallel and Distributed Systems* 33(4): 878–890. DOI:10.1109/TPDS.2021.3100784.