Collective I/O Optimizations for Adaptive Mesh Refinement Data Writes on Lustre File System

Dharshi Devendran*, Suren Byna*, Bin Dong*,
Brian Van Straalen*, Hans Johansen*, Noel Keen*, and Nagiza F. Samatova†

*Lawrence Berkeley National Laboratory, CA 94720, USA Email: pdevendran@lbl.gov
†North Carolina State University, NC 27695, USA

***Abstract*—Adaptive mesh refinement (AMR) applications refine small regions of a physical space. As a result, when AMR data has to be stored in a file, writing data involves storing a large number of small blocks of data. Chombo is an AMR software library for solving partial differential equations over block-structured grids, and is used in large-scale climate and fluid dynamics simulations. Chombo's current implementation for writing data on an AMR hierarchy uses several independent write operations, causing low I/O performance. In this paper, we investigate collective I/O optimizations for Chombo's write function. We introduce Aggregated Collective Buffering (ACB) to reduce the number of small writes. We demonstrate that our approach outperforms the current implementation by 2× to 9.1× and the MPI-IO collective buffering by 1.5× to 3.4× on the Edison and Cori platforms at NERSC using the Chombo-IO benchmark. Using the Darshan I/O characterization tool, we show that ACB makes larger contiguous writes than collective buffering at the POSIX level, and this difference gives ACB a significant performance benefit over collective buffering.**

***Index Terms*—Collective Buffering, Chombo, Parallel I/O, Lustre File System**

I. Introduction

Adaptive Mesh Refinement (AMR) is a significant advance in large-scale scientific simulations on high-performance computers (HPC) toward improving efficiency of computational and memory requirements. AMR methods allow applications to dynamically refine simulation resolution across space and time. Several scientific simulations from cosmology, combustion, climate, etc. are moving toward using AMR. Over the last decade, several block-structured AMR libraries, such as Chombo [1], BoxLib [3], FLASH [5], have become popular. As a result of refinement based on information, AMR applications produce hierarchical, multi-level, and multi-resolution meshes. Writing this complex data structure to storage devices is often tricky as there are several small and disjoint pieces of data to be stitched together, often causing poor parallel I/O performance. In this paper, we study the parallel I/O module of the Chombo library and optimize its performance using aggregation techniques.

Chombo is a software library for solving partial differential equations over AMR grids, and is currently being used for large-scale climate and fluid dynamics simulations. Chombo specializes in block-structured AMR grids, which divides the grids into rectangular regions of uniform mesh-width, called "boxes". In general, a Chombo application spends most of its time in solving the partial differential equations (PDEs) that model a scientific problem. As a result, there has been a significant research and development effort on improving the performance of Chombo's solvers and the algorithms involved in computations [2], [13], [11], [9]. As these algorithms become faster, Chombo's I/O routines become a more prominent bottleneck in performance.

Chombo uses the HDF5 library [12] to write out its data to storage. In the current Chombo implementation, each MPI process calls a separate HDF5 write function for each box on the process, resulting in several independent small write operations, and ultimately low performance. Moreover, the layout of boxes in the file and in memory affect the type of optimizations we can use to improve I/O performance. In the HDF5 file, the boxes are organized in lexicographic order to facilitate fast reads of grid sub-regions. In the memory, the boxes are distributed across MPI processes in a way that balances the loads (usually measured by the number of points in a box) across the processes as much as possible. The distribution of boxes on the processes appear randomly distributed, and are typically independent of lexicographic order. As a result, each process writes to several non-contiguous regions in the file.

Collective I/O is a popular optimization strategy for applications that make several small non-contiguous accesses to a file. There are a few studies that investigate the performance of collective I/O in AMR-based codes [14], [8], [15], [7]. We have found only one other paper, however, that studies I/O performance for Chombo [6]. Howison et al. [6] introduce optimizations for collective buffering on a Lustre file system, and show that this new mode of collective I/O improves Chombo's I/O performance when applied to a Chombo I/O benchmark.

In this paper, we build on the work in [6] by introducing **Aggregated Collective Buffering (ACB)**, a new strategy for optimizing Chombo's I/O. In Chombo's current implementation, each MPI process only sends a single box for writing, limiting the optimizations that MPI-IO's collective buffering can achieve. In ACB, we aggregate the boxes on each process into one buffer in Chombo and use a union of HDF5 hyperslabs to specify the locations of the boxes in the file, so the boxes are still organized in lexicographic order in the file.

We test the proposed ACB method on the Chombo I/O

benchmark on two Cray systems at the National Energy Research Scientific Computing Center (NERSC), *Edison and Cori*, and compare it to the performance of MPI-IO with independent and collective buffering I/O modes. We show that enabling collective buffering is 2.3× to 6.1× faster than the current Chombo I/O implementation that is based on independent I/O mode on Cori. Our new ACB mode is 3.9× to 9.1× faster than the independent I/O version on Cori. On Edison, Chombo with collective I/O mode is 1.1× to 1.6× faster than that with independent I/O and ACB is 2× to 3.4× faster. We have also investigated how Lustre file system striping affects performance, and used the I/O characterization tool Darshan [4] to analyze the behavior of the three methods.

In Section 2, we provide background on Chombo I/O and MPI-IO collective buffering. In Section 3, we introduce our Aggregated Collective Buffering optimization strategy for Chombo. In Section 4, we present our experiments and evaluation of I/O performance and conclude our discussion in Section 5.

II. Background

A. Chombo

Chombo is a software package for solving partial differential equations over a block-structured AMR mesh. This mesh is a hierarchy of nested, uniform resolution grids. In general, only the coarsest grid covers the problem domain. The refinement ratio, which is a vector of integers, relates the resolutions of two consecutive grid levels. In block-structured AMR, each grid can be decomposed into rectangular regions, called boxes.

Application variables, such as temperature, velocity, pressure, etc., are evaluated at centers of the cells making up a box, at cell corners, or at the centroids of cell faces. Different variables may be evaluated at different locations. Ghost cells, or extra layers of cells surrounding a box, are used to communicate information between boxes as well as between different grid levels. In Chombo, the data on a box and its ghost cells is contained in a multidimensional array. We show an example AMR mesh with three grid levels in Figure 1.

To store this data, Chombo uses HDF5, a library and portable file format for storing complex data. HDF5 abstracts out the details of writing data to storage, so application developers can maintain an object-oriented view of their data. It uses parallel MPI-IO to handle the low-level details of writing data to a parallel file system.

Chombo writes out the data on the entire AMR mesh to a single HDF5 dataset. In particular, the multidimensional array containing the data on a box is flattened, and is represented as a 1D array in the HDF5 file. In the HDF5 file, the AMR boxes are organized in lexicographic order to facilitate fast reads of grid sub-regions. Across MPI processes, however, the boxes and their data are not distributed in lexicographic order. Instead, the boxes and their data are divided up among MPI processes to balance the load across the processes as much as possible. The default load assigned to a box is the number of cells in the box, but the application developer may

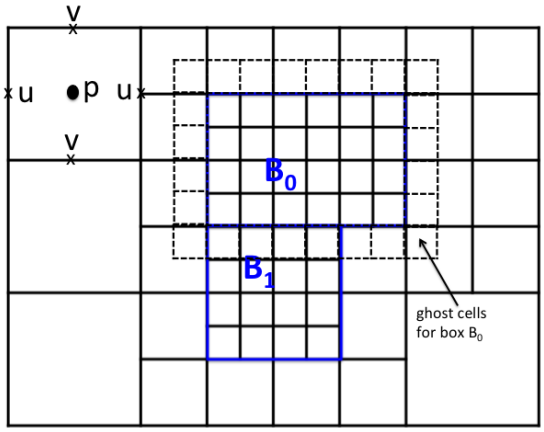

Fig. 1. An AMR mesh with three grid levels. The refinement ratio is (2, 2). Only the coarsest grid covers the problem domain, and each grid is nested within the next coarsest grid. Each grid can be decomposed into boxes or rectangular regions. In this example, grid level 2 is decomposed into 2 boxes (marked by thick blue lines): B_0 and B_1. Also in this example, the AMR data consists of 3 variables: u, v, and p. u is evaluated at the vertical face centers, v at the horizontal face centers, and p at the cell centers. Ghost cells are used to exchange this data between boxes and across levels.

choose a different set of load assignments. Once loads are assigned to the boxes, many applications sort the list of boxes using Morton ordering, which favors putting spatially adjacent boxes close together in the list. Finally, Chombo applies the Kernighan-Lin algorithm to assign boxes to processes and balance loads [1]. As a result of this procedure, the boxes may appear randomly distributed across processes, and often each process writes to non-contiguous regions in the file. In Figure 2, we show an example on how boxes distributed across MPI processes are written to disjoint locations in a HDF5 file. When data is written to a disk-based file system, due to poor performance of disks with non-contiguous accesses that require large number of costly disk seeks, I/O performance is typically poor.

In addition, in the current Chombo implementation, each process performs an independent write for each box on the process. In particular, it first copies the data on a box into a 1D buffer, and then it calls an HDF5 write function to store the data. As a result, Chombo issues a large number of independent write function calls that output small chunks of data. The large number of small I/O requests is another inefficiency on disk-based file systems, as the start-up time of I/O requests is costly.

B. MPI-IO Collective Buffering

The parallel HDF5 library is dependent on MPI-IO for performing data read and write operations to parallel file systems, such as Lustre and GPFS. ROMIO, the most popularly used MPI-IO implementation, provides various features for MPI processes to write the data independently or collectively. In the independent mode, each MPI process issues their own I/O calls. As mentioned above, AMR applications often result in small non-contiguous independent writes. However, disk-

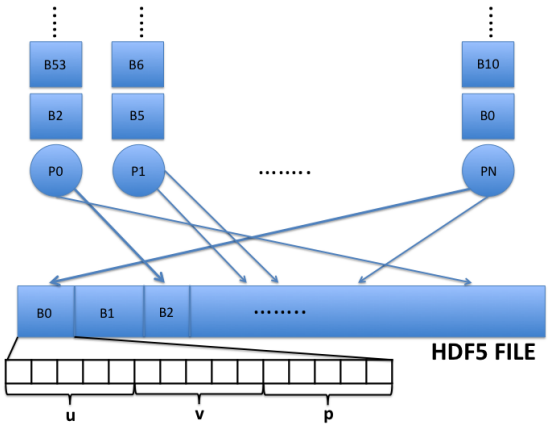

Fig. 2. Boxes are distributed across MPI processes to balances loads across the processes. In the HDF5 file, however, the boxes are organized in lexicographic order. Because load balancing does not distribute the boxes according to lexicographic order, each process writes to non-contiguous regions of the file. In the HDF5 file, the data on a box (in this example, the data are the variables u, v, and p evaluated on the grid) are represented as a 1D array.

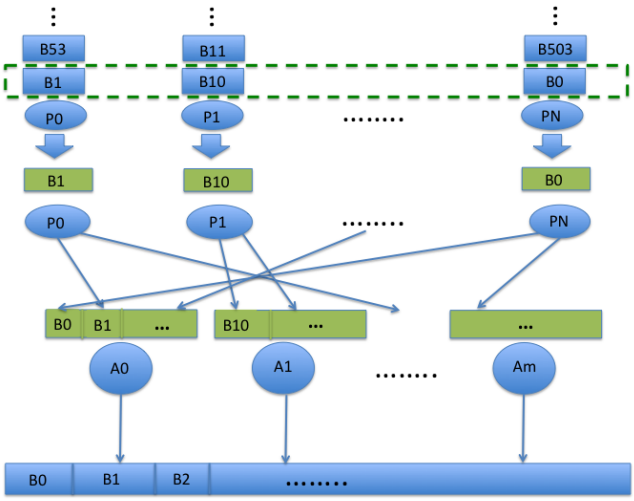

Fig. 3. Illustration of collective buffering. Chombo first copies the data on a box (denoted by B0, B1, ...) into a 1D buffer (green), and then sends these buffers to HDF5. In collective buffering, aggregators (A0, A1, ...) collect the data sent by each of the processes (P0, P1, ...) into buffers (in green) to write out as many contiguous chunks as possible. Note that aggregators only have access to a subset of all the boxes at a time in this implementation of Chombo.

based parallel file systems perform very poorly when disk has to be accessed many times and to write small data to non-contiguous locations as the overheads of accessing and seeking different locations on disk is costly. To reduce the number of independent accesses to the file system by each process, MPI-IO provides the *collective buffering* feature, also known as two-phase I/O [10]. With collective buffering, MPI-IO assigns a subset of the processes to be aggregators, which gather data that will be written to contiguous sections of the file into buffers (phase 1). As a result, these aggregators can write out large chunks of contiguous data (phase 2) resulting in fewer accesses to the file system. In Figure 3, we show how MPI-Processes of Chombo work with collective buffering. In HDF5, collective buffering of MPI-IO can be turned on by specifying a dataset transfer property in an HDF5 write call.

There are several modes of collective buffering implementations in MPI-IO. The CB2 mode of the collective I/O routine performs optimizations for Lustre file systems. Lustre uses several Object Storage Targets (OSTs) to provide parallel access to the file system. A file can be split into blocks across these OSTs to read/write the file in a parallel fashion. The size of each block is called a *stripe*, and the size of the block is called the *stripe size*. The number of OSTs used for storing a file is called the *stripe count*. Several studies have shown that the write performance of Lustre significantly deteriorates when the amount of data written to an OST is not a multiple of the stripe size [6]. Also, Howison, et al. demonstrated that choosing the number of MPI-IO collective buffering aggregators to be equal to the number of OSTs gives the best performance on Lustre. As a result, in the CB2 mode, the aggregator buffer size is set to a multiple of the stripe size, and the number of aggregators is set equal to the number of OSTs. On Cray machines, CB2 mode is the default, and we maintain this default in our experiments.

1) Using collective I/O in Chombo: One of the requirements of HDF5 to work using MPI-IO collective buffering is that all the processes have to participate in collective I/O. Because collective I/O synchronizes writes across processes, all processes must issue the same number of write calls, or the application will hang. In Chombo, because every process calls a write operation for each of its boxes, and the number of boxes are not equal across all MPI processes, some processes may not call write functions the same number of times. To overcome this hurdle, we have a process call a write function with empty data, if the process has finished writing out all of its data, and other processes still have more data to write.

III. Aggregated Collective Buffering (ACB)

Even in the collective buffering mode, each MPI process of Chombo only sends a single box to the aggregators, which restricts the amount of performance tuning that collective I/O can achieve. For example, if the boxes are organized in lexicographic order on the processes, which is the case in some applications, each collective I/O call contains non-contiguous chunks that cannot be aggregated further. Consequently, if we can send multiple boxes on a process in a single write call, collective I/O can perform much better. This is the motivation for our *aggregated collective buffering (ACB)* approach.

In ACB, we copy several boxes on the process into a buffer, instead of only copying one box into the buffer. In Figure 4, we show an example aggregation of boxes on processes and how they are sent to MPI-IO aggregators that write out the data to file system. We generate a union of hyperslabs to specify the locations of the boxes in the file. In HDF5, a hyperslab specifies a contiguous section in a file. Using a union of hyperslabs, a HDF5 user can specify a set of file regions

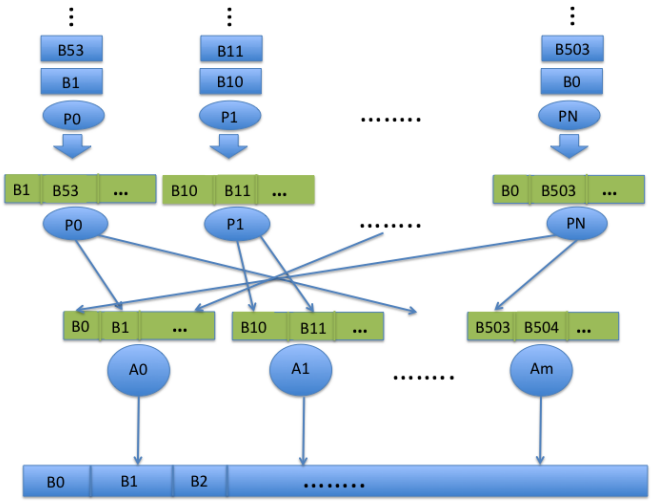

Fig. 4. In Aggregated Collective Buffering (ACB), the boxes (B0, B1,...) on the MPI processes (P0, P1,,PN) are first aggregated into a buffer for each process. These buffers along with the locations of the boxes in the file are sent in an HDF5 write function call. MPI-IO aggregators (A0, A1,,Am) collect the boxes from all the processes and reorganize them into new buffers to minimize the number of write calls.

in a single HDF5 write call. The hyperslabs in the union can have different shapes and can be non-contiguous. With this utility, we can maintain lexicographic order of boxes in the file, and write multiple non-contiguous boxes in a single HDF5 write. A union of hyperslabs is generated using the function `H5S_SELECT_OR` [12]. Once we have the union, we pass the buffer and hyperslab union into a HDF5 write.

Note that ACB works at the Chombo level. In particular, the aggregation of boxes occurs in the Chombo code before any data is sent to HDF5 for writing, and is separate from the aggregation performed by collective buffering.

Because ACB only aggregates boxes on a single process, it does not require any extra communication. In particular, we do not collect or shuffle boxes across multiple MPI processes. The trade off is that ACB requires extra memory to aggregate all the boxes. In the future, we will explore how MPI datatypes can be used to reduce the amount of extra memory used [7].

We have implemented ACB within the Chombo library source code. In particular, we added a *for* loop to aggregate the boxes into a single buffer, and another *for* loop to form a union of the hyperslabs. Because the method does not change the file layout, none of Chombo's read functions (or the applications that depend on these functions) have to be modified.

We expect ACB will perform best when MPI-IO collective buffering is turned on. As a result, we only consider the performance of ACB in the collective buffering mode in this paper. However, ACB can be used in independent I/O mode, and we plan to explore this case in the future. Also, the number of boxes that are aggregated into the buffer is a configurable parameter in ACB. In this paper, we consider the effect of aggregating all the boxes on an MPI process into the buffer. Exploration of how different numbers of boxes affect the performance of ACB is another future direction.

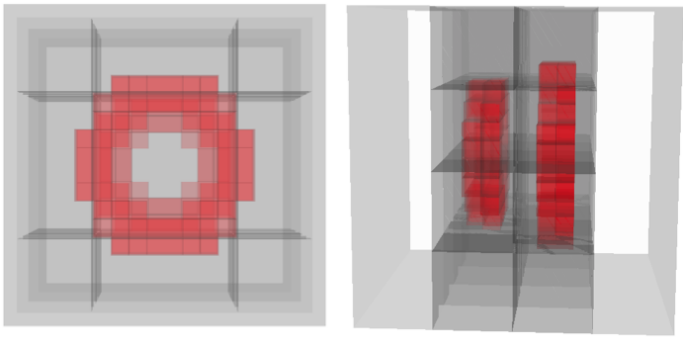

Fig. 5. Two views (front and side) of the AMR grid used in experiments on Chombo I/O benchmark.

IV. Results

A. Experimental Setup

We evaluate the performance of the three methods (independent I/O, collective buffering, and ACB) using the Chombo I/O benchmark. The benchmark was designed to study Chombo's write performance without the extra overhead of floating point computations. The benchmark provides control over the size of the data file through a vector parameter called the *replicating factor*. In constructing the AMR grid for the test, the benchmark takes a unit grid and duplicates the grid in each of the coordinate directions. The replicating factor specifies the number of times the grid should be replicated in each of the coordinate directions. Figure 5 shows one unit of the grid (or equivalently the resulting grid when the replicating factor is set to (1, 1, 1)). This grid contains 3 levels, and is $64\times64\times64$ on the coarsest domain. The refinement ratio is (4, 4, 4) for all levels. In our experiments, we vary the replicating factor to create different size files. The Chombo I/O benchmark also has options for the load balancing scheme. We use the algorithm described in Section II with Morton ordering.

Our experiments were performed on the supercomputers Edison and Cori (Phase 1) at the National Energy Research Scientific Computing Center (NERSC). Edison is a Cray XC30 supercomputer with a Lustre parallel file system peak aggregate I/O bandwidth of 72 GB/s. Cori, which is designed for data-intensive simulations, is a Cray XC40 supercomputer system with a Lustre file system that has a peak aggregate I/O bandwidth of 744 GB/s. On Edison, 96 Object Storage Targets (OSTs) are available for storing files (*scratch2* file system) and on Cori, 248 OSTs are available. On Cori, the ratio of the number of Object Storage Servers (OSS) to OSTs is one (1), i.e., each OSS manages one OST. On Edison, the number of OSSs is 24, i.e., each OSS manages four OSTs.

B. Scalability Tests

First, we explore the scalability of the three methods (independent I/O, collective buffering, and ACB) for the Chombo I/O benchmark. Figures 6 and 7 show the write times of the three methods at three different scales on Cori and Edison, respectively. For the 61 GB file, we used 576 processes (24

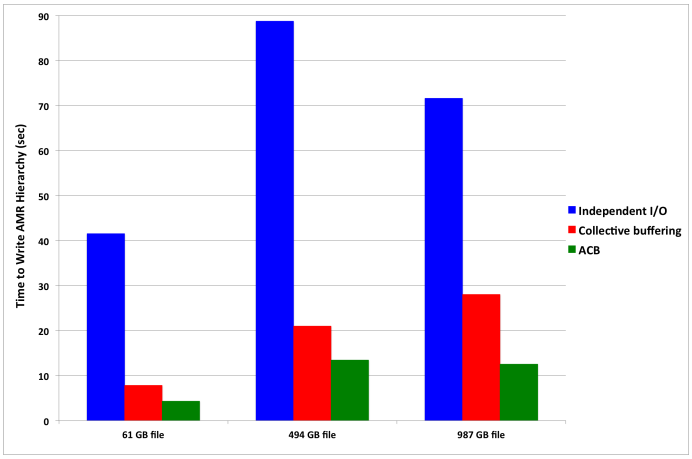

Fig. 6. Scalability runs on Cori.

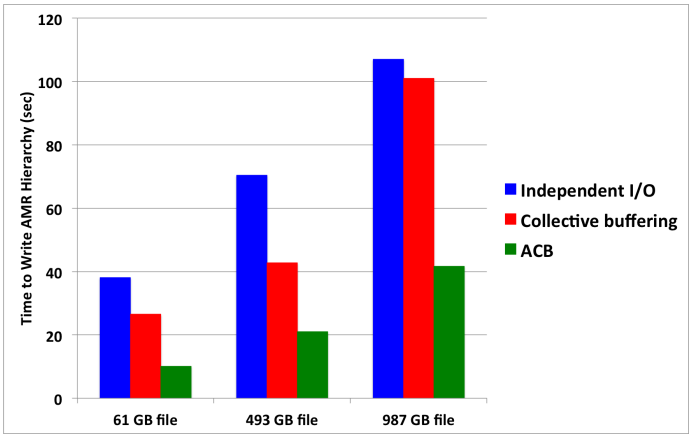

Fig. 7. Scalability runs on Edison.

compute nodes). The resultant HDF5 file was striped across 24 OSTs with a stripe size of 4 MB on both Cori and Edison. For the 494 GB file, we used 3072 processes (96 compute nodes). The file was striped across 96 OSTs with a stripe size of 8 MB on Cori and Edison. Finally, for the 987 GB file, 5856 processes (244 compute nodes) were used. The file was striped across 96 OSTs on Edison and on 244 OSTs on Cori, with a stripe size of 16 MB.

On both systems, collective buffering and ACB outperform the current implementation of Chombo with independent I/O. The write times scale differently on the two systems because of striping differences. On Cori, the 494 GB file is striped across 96 OSTs, and the 987 GB file is striped across 244 OSTs. On Edison, however, both files are striped across 96 OSTs, the maximum number of OSTs available on Edison.

On Cori, collective buffering achieves a speedup of 2.6× to 5.3× over independent I/O, while ACB is 5.7× to 9.6× faster than independent I/O, resulting in an extra factor of 1.6× to 1.8× performance improvement over collective buffering. On Edison, collective buffering is only 1.1× to 1.6× faster than independent I/O, whereas ACB is 2.6× to 3.8× faster than independent I/O. This is a factor of 2× to 2.6× additional

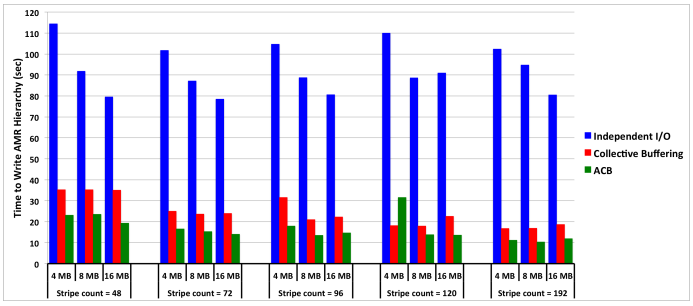

Fig. 8. Striping test on Cori. The test was run with 3072 MPI processes across 96 nodes, and it output a 494 GB data file. For each stripe count, we tested against stripe sizes of 4 MB, 8 MB, and 16 MB.

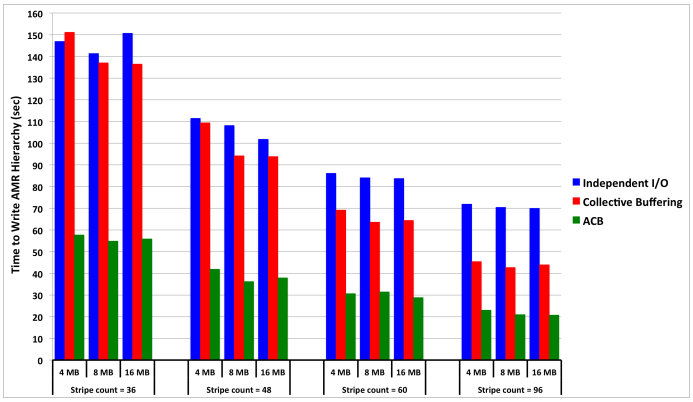

Fig. 9. Striping test on Edison. The test was run with 2304 MPI processes across 96 nodes, and it output a 494 GB data file. For each stripe count, we tested against stripe sizes of 4 MB, 8 MB, and 16 MB.

speedup over collective buffering.

C. Striping Tests

In our scaling studies, we chose a particular set of striping parameters. Next, we explore how much striping effects the performance of the three methods. In these tests, we vary the stripe count and stripe size for the 494 GB file test case, and record the write times. On both Cori and Edison, we look at stripe sizes of 4 MB, 8 MB, and 16 MB. For the experiments on Cori, we choose the stripe count to be 48 OSTs, 72 OSTs, 96 OSTs, 120 OSTs, and 196 OSTs. On Edison, however, we are restricted to a maximum of 96 OSTs, so we vary the stripe count among 36 OSTs, 48 OSTs, 60 OSTs, and 96 OSTs. Figures 8 and 9 show the results of this study.

The striping tests demonstrate that all three methods perform better when the file is striped across more OSTs, especially on Edison. The stripe size does not significantly affect the write times for collective buffering and ACB, but increasing stripe size slightly improves performance for independent I/O mode.

On Cori, collective buffering is 2.3× to 6.1× faster than independent I/O, while ACB is 3.9× to 9.1× faster than independent I/O. This is a 1.5× to 1.8× speedup over collective buffering. On Edison, ACB provides even more speedup over collective buffering than Cori. In particular, collective

	Ind. I/O, 48 OSTs	Ind. I/O, 96 OSTs	Coll. I/O, 48 OSTs	Coll. I/O, 96 OSTs	ACB, 48 OSTs	ACB, 96 OSTs
MPI-IO writes	115268	115268	119808	119808	6912	6912
POSIX writes	115628	115268	164270	164270	63241	63241
Top access size	4M	4M	4M	4M	8M	8M
Count of top access size	115201	115201	42640	42689	63174	63177
Access size 2	272	272	8M	8M	272	272
Count of access size 2	15	15	4779	4830	16	16

TABLE I
TABLE OF STATISTICS FROM DARSHAN. IND. I/O = INDEPENDENT I/O. COLL. I/O = COLLECTIVE I/O (USING COLLECTIVE BUFFERING). TOP ACCESS SIZE REFERS TO THE WRITE SIZE ACCESSED MOST OFTEN. ACCESS SIZE 2 IS THE SECOND MOST ACCESSED WRITE SIZE. 4M = 4599936 AND 8M = 8388608

buffering is only 1.1 × to 1.6× faster than independent I/O, whereas ACB is 2× to 3.4× faster than independent I/O.

D. Analysis of Write Calls Using Darshan Characterization

To understand the performance behavior of the three methods, we used Darshan, an I/O characterization tool, to collect statistics on the underlying write calls. We analyzed the statistics for two test cases on Edison: the 494 GB file striped across 48 OSTs with a stripe size of 8 MB and the 494 GB file striped across 96 OSTs with a stripe size of 8 MB. Table I compares the total number of POSIX write calls and MPI-IO write calls for the three methods as well as the top two access sizes. Figures 11 and 10 display the distributions of MPI access sizes and POSIX access sizes, respectively.

In the tests, the total number of boxes is 115628, and the graphs confirm that independent I/O uses a separate write operation for each box. For collective I/O, the 2304 processes collectively call write 52 times (the number of 'layers" of boxes) for a total of 119808 write calls. In ACB, on the other hand, the 2304 processes make one collective write call on each of the 3 levels, for a total of 6912 write calls.

As predicted, independent I/O makes several small write operations. In particular, most of the independent I/O operations are only 4 MB in size, with a few smaller writes. ACB, on the other hand, has several large write calls between 100 MB and 1 GB. At the POSIX level, this translates into 8 MB-sized writes, based on the Lustre stripe size. Because of the small number of relatively large and contiguous data writes, ACB achieves a significant performance benefit.

In contrast, the distribution of POSIX writes for collective I/O is much more spread out. At the MPI level, most of the collective calls are between 4 MB and 10 MB. However,

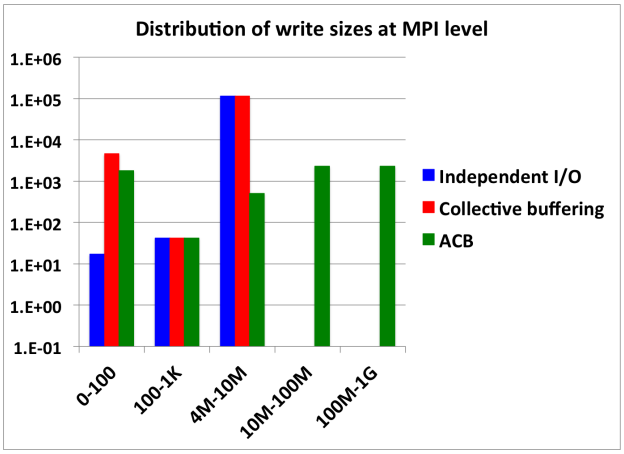

Fig. 10. Distribution of write sizes at the MPI level for a 494 GB file test case on Edison. The file was striped across 96 OSTs with a stripe size of 8 MB. The test was run with 2304 MPI processes across 96 nodes. Note that the y-axis is on a log scale. Because all three methods had very few write accesses in the 1K-4M range, we don't include those accesses in this distribution.

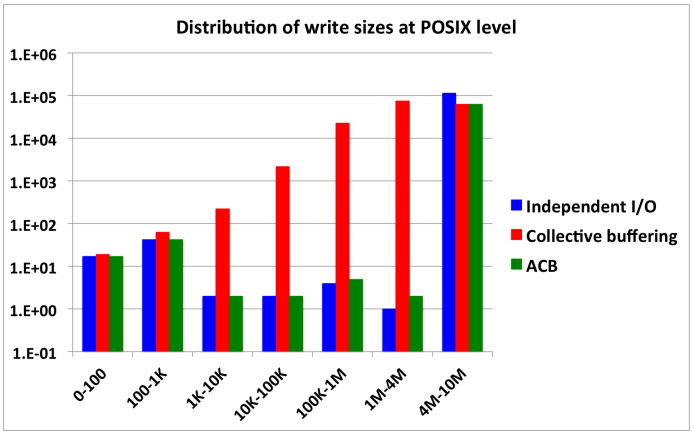

Fig. 11. Distribution of write sizes at the POSIX level for a 494 GB file test case on Edison. The file was striped across 96 OSTs with a stripe size of 8 MB. The test was run with 2304 MPI processes across 96 nodes. Note that the y-axis is on a log scale.

these calls are divided into several POSIX write operations of varying sizes. From Table I, most of the write operations are 4 MB in size, and some of them are 8 MB in size. Collective buffering also requests several small writes between 0 and 100 bytes. We posit that the smaller write operations may be aggregated on the server side, which reduces the write time for collective I/O over independent I/O.

V. CONCLUSION

In this paper, we introduced aggregated collective buffering (ACB), a new strategy for improving Chombo's I/O performance, and tested this method using the Chombo I/O benchmark on Edison and on Cori. On both systems, ACB outperforms independent I/O and collective I/O modes of MPI-IO. Using the Darshan I/O tool, we analyzed the number of write calls made by each of the three methods. ACB's

performance can be attributed to the relatively small number of large-sized write operations it makes.

For simplicity, we considered the case where ACB aggregates all boxes on a process. In general, the number of aggregated boxes is a parameter that can be set to balance the tradeoff between extra memory usage and performance. In the future, we will study how this parameter affects ACB's performance.

Although we implemented ACB within Chombo, there is potential for applying this method to other applications. Other AMR-based codes have data structures similar to boxes, and ACB may be modified to work with other applications. Finally, Cori provides burst buffers, a non-volatile storage that is between processes' memory and the disk-based parallel file system. Burst buffers offer a different way to optimize I/O performance, and we will explore in the future how ACB performs on burst buffers.

ACKNOWLEDGMENT

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC).

REFERENCES

[1] M. Adams, P. Colella, D. T. Graves, J.N. Johnson, N.D. Keen, T. J. Ligocki, D. F. Martin, P.W. McCorquodale, D. Modiano, P.O. Schwartz, T.D. Sternberg, and B. Van Straalen. Chombo software package for AMR applications-design document. *Lawrence Berkeley National Laboratory Technical Report LBNL-6616E*, 2000.

[2] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Mary Hall. Converting stencils to accumulations for communication-avoiding optimization in geometric multigrid. *Workshop on Stencil Computations (WOSC)*, 2014.

[3] J. Bell, A. Almgren, V. Beckner, M. Day, M. Lijewski, A. Nonaka, and W. Zhang. Boxlib users guide. *Lawrence Berkeley National Laboratory Technical Report*, 2013.

[4] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage*, (7:8:1-8:26), October 2011.

[5] Anshu Dubey, Katie Antypas, Murali K. Ganapathy, Lynn B. Reid, Katherine Riley, Dan Sheeler, Andrew Siegel, and Klaus Weide. Extensible component based architecture for flash, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[6] Mark Howison, Quincey Koziol, David Knaak, John Mainzer, and John Shalf. Tuning hdf5 for lustre file systems. *Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, September 2010.

[7] Rob Latham, Chris Daley, Wei keng Liao, Kui Gao, Rob Ross, Anshu Dubey, and Alok Choudhary. A case study for scientific i/o: improving the flash astrophysics code. *Computational Science and Discovery*, 2012.

[8] Jianwei Li, Wei keng Liao, A. Choudhary, and V. Taylor. I/o analysis and optimization for an amr cosmology application. *Proceedings 2002 IEEE International Conference on Cluster Computing*, 2002.

[9] P. McCorquodale, P. Colella, G. Balls, and S.B Baden. A scalable parallel poisson solver with infinite-domain boundary conditions. *Proceedings of the 7th Workshop on High Performance Scientific and Engineering Computing*, 2005.

[10] B. Nitzberg and V. Lo. Collective buffering: Improving parallel i/o performance. *HPDC 97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, 1997.

[11] Chaopeng Shen, David Trebotich, Sergi Molins, Daniel T Graves, BV Straalen, DT Graves, T Ligocki, and CI Steefel. High performance computations of subsurface reactive transport processes at the pore scale. *Proceedings of SciDAC*, 2011.

[12] The HDF Group. Hierarchical data format version 5, 1997-2016.

[13] D. Trebotich, B.V. Straalen, D. Graves, and P. Colella. Performance of embedded boundary methods for cfd with complex geometry. *J. Phys.: Conf. Ser.*, 2008.

[14] P. Wauteleta and P. Kestenera. Parallel io performance and scalability study on the prace curie supercomputer. *White paper, Prace*, 2011.

[15] Yongen Yu, Douglas H. Rudd, Zhiling Lan, Nickolay Y. Gnedin, Andrey Kravtsov, and Jingjin Wu. Improving parallel io performance of cell-based amr cosmology applications. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.