A case study for scientific I/O: improving the FLASH astrophysics code

COMPUTATIONAL SCIENCE&DISCOVERY

A case study for scientific I/O: improving the FLASH astrophysics code

Rob Latham[2], Chris Daley[1], Wei-keng Liao[3], Kui Gao[3], Rob Ross[2], Anshu Dubey[1] and Alok Choudhary[3]

[1] DOE NNSA/ASCR Flash Center, Astronomy and Astrophysics, University of Chicago, Chicago, IL, USA
[2] Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, USA
[3] Center for Ultra-scale Computing and Information Security, Northwestern University, Evanston, IL, USA
E-mail: robl@mcs.anl.gov, rross@mcs.anl.gov, cdaley@flash.uchicago.edu, dubey@flash.uchicago.edu, wkliao@ece.northwestern.edu, kgao@ece.northwestern.edu and choudhar@ece.northwestern.edu

Received 23 December 2010, in final form 7 February 2012
Published 20 March 2012
Computational Science & Discovery **5** (2012) 015001 (16pp)
doi:10.1088/1749-4699/5/1/015001

Abstract. The FLASH code is a computational science tool for simulating and studying thermonuclear reactions. The program periodically outputs large checkpoint files (to resume a calculation from a particular point in time) and smaller plot files (for visualization and analysis). Initial experiments on BlueGene/P spent excessive time in input/output (I/O), making it difficult to do actual science. Our investigation of time spent in I/O revealed several locations in the I/O software stack where we could make improvements. Fixing data corruption in the MPI-IO library allowed us to use collective I/O, yielding an order of magnitude improvement. Restructuring the data layout provided a more efficient I/O access pattern and yielded another doubling of performance, but broke format assumptions made by other tools in the application workflow. Using new nonblocking APIs in the Parallel-NetCDF library allowed us to keep high performance and maintain backward compatibility. The I/O research community has studied a host of optimizations and strategies. Sometimes the challenge for applications is knowing how to apply these new techniques to production codes. In this case study, we offer a demonstration of how computational scientists, with a detailed understanding of their application, and the I/O community, with a wide array of approaches from which to choose, can magnify each other's efforts and achieve tremendous application productivity gains.

Contents

1. Introduction

The time spent performing input/output (I/O) on today's leadership-class machines is recognized as a common bottleneck in many existing HPC applications. This is not expected to change soon, as the push to simulate larger scientific problems often means production of larger volumes of data for checkpointing and analysis purposes. In addition, there is also the hardware consideration that the rate at which future storage can accept data is being outpaced by the rate at which results can be calculated. It is critical therefore that application I/O interacts well with storage for the application to scale well at large processor counts.

Computational science applications represent physical phenomena with models and abstractions. Storage systems and file systems, however, operate on bytes and files with minimal structure. In order to bridge that 'interface gap', computer scientists have created an I/O *software stack*, depicted in figure 1. Our work to optimize FLASH I/O behavior required an understanding of all layers of this stack.

FLASH [1, 2] is a publicly available code originally designed to solve problems with compressible, reactive flows. It has evolved into a huge collection of components to solve a wide range of astrophysical, CFD and plasma physics problems. The full FLASH application provides both an adaptive mesh refinement (AMR) grid and a uniform grid (UG) to store Eulerian data (the configurations in this work use the AMR grid). FLASH also contains implementations of parallel I/O using either HDF5 [3] or Parallel netCDF (PNetCDF) [4] high-level I/O libraries.

In the spring of 2009 we faced an application challenge. Applications running on Leadership Class Facilities are allocated a fixed amount of 'CPU hours' to run simulations. The FLASH code spent such a large portion of that allocated time outputting data that insufficient CPU hours remained to compute useful scientific results. In this paper we discuss the three main improvements made to FLASH and the I/O stack to reduce the time spent in I/O.

- We fixed a defect in the MPI-IO library, allowing us to enable collective MPI-IO optimizations.
- We altered the output file format, allowing for an ideal access pattern at the expense of breaking compatibility with existing analysis and visualization tools.
- Finally, we examined a 'best of both worlds' solution via recent extensions to the Parallel-NetCDF high-level I/O library, yielding performance as good as altering the file format while maintaining backwards compatibility.

The FLASH I/O implementations can be tested (independently of science) through an I/O unit test application that has been used as a benchmark in machine acceptance tests and to aid the development of various layers of the I/O software stack. The benchmark has been used consistently for nearly a decade to help evaluate the performance of components of the I/O stack, including HDF5 hyperslab processing [5],

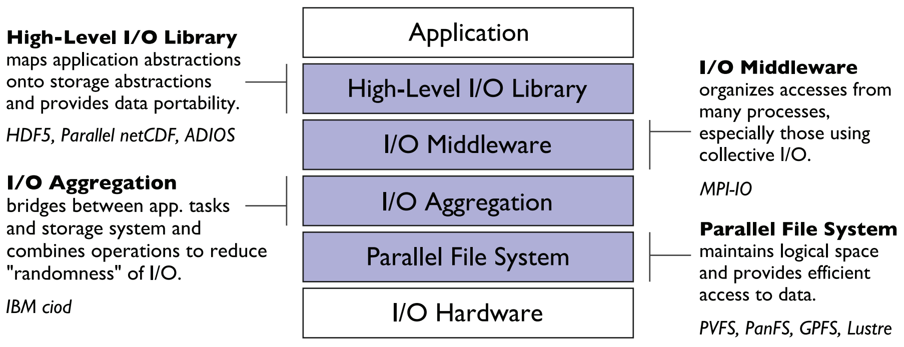

High-Level I/O Library
maps application abstractions onto storage abstractions and provides data portability.
HDF5, Parallel netCDF, ADIOS
I/O Aggregation
bridges between app. tasks and storage system and combines operations to reduce "randomness" of I/O.
IBM ciod
Application
High-Level I/O Library
I/O Middleware
I/O Aggregation
Parallel File System
I/O Hardware
I/O Middleware
organizes accesses from many processes, especially those using collective I/O.
MPI-IO
Parallel File System
maintains logical space and provides efficient access to data.
PVFS, PanFS, GPFS, Lustre

Figure 1. The I/O software stack presents numerous opportunities for optimization.

the Parallel Virtual File System (PVFS) [6, 7], experimental MPI-IO implementations [8] and MPI datatype processing [9, 10]. In fact, download figures from 2007 show that 28% of FLASH downloads are for parallel I/O related studies [11]. With this decade-long attention to parallel I/O, it came as a surprise when early runs on Argonne's BlueGene/P spent exceptionally long times to produce checkpoint files.

This paper describes our analysis and performance tuning approach and provides a model of attack for other applications exhibiting low I/O performance on a new system or as a consequence of scaling. Two broad lessons emerge from this work. Firstly, real scientific codes can achieve high I/O rates with today's I/O stack. Secondly, when I/O experts and application experts collaborate closely, the two groups bring backgrounds which not only complement each other, but in fact amplify the impact of modifications.

One notable aspect of this work is the willingness of the FLASH developers to experiment with altering the file format. Occasionally, computational scientists have an opportunity to design file formats with I/O in mind [12]. More commonly, the file format is fixed and optimizations and improvements must happen at other layers in the software stack [5, 13, 14]. Thus, while the FLASH application has been around for over a decade, we were given nearly a clean slate when it came to evaluating approaches for improving I/O performance. We were able to re-work the file format to provide an ideal I/O workload. Next, we applied recent I/O research to match that upper bound of performance while still maintaining support for the existing workflow. Furthermore, we were able to evaluate several research ideas at scales not yet published in the context of an actual science application.

The paper is organized as follows. We describe the core FLASH mesh data structure, the standard and experimental output file layouts and creation of memory-derived datatypes in section 2. These application-specific details will play an important role when in later sections we discuss some of the more advanced optimizations we applied. We introduce the target BlueGene/P platform and the chosen FLASH test application in section 3. Sections 4–6 cover the performance improvement techniques we applied and their benefits. The optimizations presented have been discussed elsewhere but this work collects these optimizations and evaluates them at higher levels of scalability than studied previously. Finally, we summarize the work and discuss lessons learned during this I/O project in section 8. In short, I/O performance plays too important a role in scientific computing for individual groups to tackle the problem on their own.

2. FLASH memory and file layout

Before discussing I/O experiments and results, we provide some background on the FLASH data model.

FLASH simulations evolve physical quantities such as density, pressure and temperature over time on a Cartesian, structured mesh. The mesh consists of cells that contain the value of physical quantities (also known as mesh variables) at different locations in the computational domain. Each cell is assigned to a block, where a block is a self-contained grid that contains a fixed number of cells and several layers of guard cells. There can be a huge number of blocks in a simulation, and different blocks may be assigned to different processors because the guard cells contain the required neighboring block data or boundary condition data.

2.1. Memory layout

The cell-centered data for cells of blocks assigned to the current processor are stored in a five-dimensional (5D) array of double-precision typed data, named `unk`. It is allocated once at the start of the FLASH run and has the same size in each MPI process. The array contains a dimension for mesh variables, cells in each coordinate direction and blocks. For example, the data for the density mesh variable (DENS VAR) in cell (i,j,k) of block (lb) can be accessed using `unk(DENS VAR,i,j,k,lb)`. The 5D array has the following dimensions:

$$\begin{aligned}&\text{unk(NUNK\_VARS,}\\&\text{NXB} + \text{K1D} * 2 * \text{NGUARD},\\&\text{NYB} + \text{K2D} * 2 * \text{NGUARD},\\&\text{NZB} + \text{K3D} * 2 * \text{NGUARD, MAXBLOCKS).}\end{aligned} \tag{1}$$

Here, `NUNK_VARS` is the number of cell-centered mesh variables, e.g. density, pressure and temperature. `NXB`, `NYB`, `NZB` are the number of x,y,z internal cells, and `NGUARD` is the number of guard cells. The variables (`K1D`, `K2D`, `K3D`) are integer values that take the values (1,0,0), (1,1,0) and (1,1,1) for 1D, 2D and 3D applications, respectively. These integer values allow the same data structure to be used for different dimensionality simulations without wasting guard cell space in unused dimensions. Finally, `MAXBLOCKS` is the maximum number of blocks that can reside in a single MPI process. The array is presented in Fortran column major ordering, i.e. the dimension `NUNK_VARS` varies most rapidly.

There is no reallocation of the mesh data structure during the application run, so new blocks must fit in the `1:MAXBLOCKS` space in the process that they are placed. This presents a challenge for computational scientists as they must select a value for `MAXBLOCKS` that enables `unk` to fit in memory but also provide sufficient free space for new blocks.

2.2. File layout

Only actual blocks are stored in file (`nblocks` of the entire `MAXBLOCKS` dataspace); guard cells are excluded. The standard file layout used by FLASH places each of the `NUNK_VARS` mesh variables into its own 4D variable in the file (one application variable corresponds to one in-file variable). In section 5, we will discuss an experimental file layout that uses a single 5D dataset, or variable, to hold all `NUNK_VARS` mesh variables in file. Henceforth, we refer to file layouts as either *standard* or *experimental*. We will discuss later the performance implications of the standard and experimental file layouts.

The existing I/O strategy for transferring data from memory to file, implemented on top of both HDF5 and PNetCDF, has been used in FLASH over the last several years and is well studied. It involves copying internal cell data for a single mesh variable into a temporary contiguous buffer and then passing a pointer to this temporary buffer to the I/O library write function. This leads to a straightforward data transfer for MPI-IO as there is a contiguous data layout in memory and file. The process is repeated for each of the mesh variables, meaning that the test application we will discuss in section 3 will need to make ten write calls for checkpoint files and three write calls for plot files. This approach works well on most architectures, especially if the application requests collective I/O. Copying into a temporary buffer also maps well to the programming interfaces provided by high-level I/O libraries—one variable per function call. This approach has two drawbacks. The application must make multiple calls to the I/O subsystem, potentially incurring a latency cost with each request. Secondly, using a temporary memory buffer may not be desirable on future architectures where the memory per core is expected to be lower than that on the present systems [15].

In this work, we evaluate a second strategy (new to FLASH) in which we select the data in `unk` directly by selecting relevant areas in memory using MPI-derived datatypes (PNetCDF library) or hyperslabs (HDF5 library). Previous experiments have studied HDF5 hyperslab behavior [5, 16], but in those studies hyperslabs selected the file region each process would write. In that previous work the FLASH application would still copy the variable from the `unk` buffer into a temporary (contiguous) memory region before calling the I/O library routine. Our new strategy makes two changes: firstly, bypassing the temporary buffer by selecting memory regions directly and, secondly, altering the file format so that all application variables reside in a single large

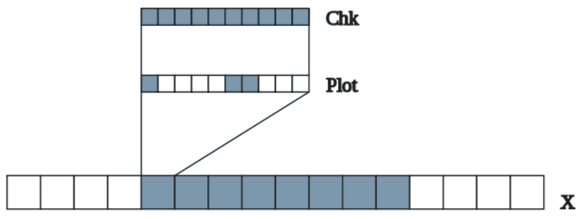


Figure 2. Data extracted from a single block in a 1D simulation with NUNK_VARS = 10 (ten mesh variables), nPlotVar = 3 (three mesh variables for visualization), NXB = 8 (eight internal cells) and NGUARD = 4 (four guard cells).

Table 1. MPI / HDF5-derived datatypes for `unk`.

Type	Selected mesh variables	File layout	File type
A	1	Standard	Checkpoint/plot
B	NUNK_VARS (10)	Experimental	Checkpoint
C	nPlotVar (3)	Experimental	Plot

and contiguous region on disk. By having the application use one large variable or dataset in the high-level I/O library to represent all application variables, we can then make a single library call to write out the variable.

Often, additional memory copies raise performance concerns. In contexts dominated by I/O time, such as checkpoint I/O, applications spend negligible time in memory copies. We wish to point out to the reader that the performance benefits from these approaches result entirely from changes to the I/O access pattern.

Figure 2 shows the exact data that must be extracted from memory into checkpoint and plot files for a simplified 1D simulation. The figure shows selected cells in gray and ignored cells in white. In total, we use three different datatypes to select data for standard and experimental layouts for both checkpoint and plot files. The first datatype, type A, selects all memory locations containing the same single-mesh variable. It is used to produce files that are laid out in the standard FLASH file format and is applicable to checkpoint and plot files. Type A is re-used for each mesh variable in turn by simply adjusting the start memory position. The next datatypes, type B and type C, select all memory locations for all mesh variables for checkpoint and plot files, respectively. A brief summary of the datatype properties is shown in table 1.

All three derived datatypes incorporate the same pattern of guardcell exclusion. Since this is a simple, regular pattern, it is described using an `MPI_Type_create_subarray` (PNetCDF) and `H5Sselect_hyperslab` (HDF5). These API calls are all that is needed to create type A and type B; the only difference between these types is the extent of the subarray in the first dimension. It is more complicated to create type C because the selection is not a simple subarray of a primitive type. The required extra step for PNetCDF is to first create an intermediate MPI datatype that selects the mesh variables at indices 1,6,7 (see figure 2). This involves using `MPI_Type_indexed` and then `MPI_Type_create_resized` to adjust the memory space extent to `NUNK_VARS`. Finally, the intermediate derived datatype is passed to `MPI_Type_create_subarray` to exclude guardcells as before. The required extra step for HDF5 is to take type A and then accumulate the mesh variables at indices 1,6,7 using `H5S_SELECT_OR`. All block data can be selected by repeating the derived datatypes `nblock` times because all blocks have the same size over this grid.

Note that the construction of these memory descriptions provides a good example of collaboration. The I/O experts can educate domain scientists about the optimizations provided by the I/O libraries. The application developers possess the familiarity with the application data model to make full use of the provided optimizations.

3. Input/Output (I/O) experiments

All experiments are performed on the IBM Blue Gene/P, Intrepid, at Argonne National Laboratory (ANL) [17]. This Blue Gene installation has 160K cores, each operating at 850 MHz with four cores per compute node. It is configured with one I/O node per 64 compute nodes, and can deliver approximately 300 MiB per second of I/O bandwidth per I/O node. Earlier hardware studies using IOR, a synthetic benchmark, achieved 40 GiB per second with large block I/O at 65536 cores [18]. The Intrepid hardware configuration has changed since that earlier study: some caches have been disabled, trading some performance for additional durability of data. Furthermore, the workloads in these application experiments are not like the large, block-aligned requests generated by IOR. We only used the production GPFS file system. All experiments are submitted as Virtual Node (VN) jobs, which means that all four cores in a processor run an independent MPI process, and each core has access to a private 512 MiB memory region.

The test application is the standard Sedov simulation that is included in the FLASH distribution. Sedov evolves a blast wave from a delta-function initial pressure perturbation (for further details see [19]). The Sedov problem exercises the infrastructure (AMR and I/O) of FLASH with minimal use of physics solvers. It can, therefore, produce representative I/O behavior of FLASH without spending too much time in computations. Each block consists of ten mesh variables, and the problem size is controlled by adjusting the global number of blocks. Because our interest is focused on I/O behavior, we choose to advance only four timesteps and to produce I/O output every single step so that most application runtime is spent performing I/O rather than computation.

In these experiments we switch off adaptivity, again to focus on I/O behavior, by setting the minimum mesh refinement level equal to the maximum mesh refinement level using parameter values of `lrefine_min = lrefine_max = 5`. As a result, all blocks will recursively bisect the same number of times to the same fully refined level and then remain at that level. A single block at the base level mesh produces $(2^D)^{L-1}$ leaf and $\sum_{n=1}^{L}(2^D)^{n-1}$ total blocks, where D is the dimensionality and L is the finest fully refined level. In our case of $L = 5$ and $D = 3$, a single block at the base level mesh creates an oct-tree mesh with 4096 leaf blocks and 4681 total blocks. We can easily control the total block count by adjusting the number of base level blocks in each coordinate direction using the configuration parameters `nblockx`, `nblocky` and `nblockz`. Adaptivity and refinement costs can be a source of runtime overhead in FLASH science runs, but the FLASH timing infrastructure records that cost separately. The FLASH application stores mesh data in a canonical (array) format before conducting I/O, as discussed in section 2.1. Thus, the underlying I/O libraries see essentially the same workload regardless of adaptivity at the application level.

In our studies quantifying the benefits of collective I/O (section 4) and examining the performance implications of altering the on-disk format (section 5), we adjust the parameters to provide approximately 32 leaf blocks per core. This problem size, on a per core basis, is representative of current simulations this application group runs on Intrepid.

Our study of the Parallel-NetCDF nonblocking optimizations (section 6) uses about 11 blocks per core—the smallest number of data per core the FLASH team would ever consider using in a production run. We use a smaller number of data per core for several reasons: as a pragmatic matter, I/O research groups have only limited CPU hours on leadership class machines; as we scale our experiments to larger core counts we need to constrain the total number of CPU hours consumed for I/O benchmarking. Furthermore, increasing the per-core working set is too easy a way to achieve higher I/O rates. This smaller working set draws into greater focus the overheads of the I/O library and MPI-IO middleware layers. In experiments with much larger working sets, the time spent transferring bytes to storage dominates any other overheads. Scalability studies with small per-core working sets also help us prepare for future machine designs: the International Exascale Software Roadmap [15] suggests that future architectures may have orders of magnitude lower memory per core than present architectures. Applications will thus not be able to merely increase the problem size to achieve higher I/O rates, and will instead require I/O strategies and novel programming APIs and models that can successfully deal with small numbers of data per processor.

The I/O in each step consists of checkpoint files for restart purposes and plot files for analysis. A checkpoint file is a dump of the complete state of a running application, including mesh data in double precision and, if included, particles. A plot file is a user-selected subset of mesh variables stored in single precision. In these experiments checkpoint I/O writes all ten mesh variables. Plot file I/O writes only selected variables of interest (in these experiments, the first, sixth and seventh variables). In both the checkpoint and plotfile cases, for post-processing convenience, the application creates a single file containing all output variables, a layout we call the *standard* file layout.

The FLASH log file records timings for an initial setup phase and a subsequent simulation phase. We configured FLASH to write one checkpoint and one plot file after each of four timesteps. Graphs of these experiments report the time spent in checkpoint or plot file I/O averaged over the four iterations.

4. Enabling collective I/O optimizations

As mentioned earlier, FLASH can make use of either the HDF5 or Parallel-NetCDF high-level I/O libraries. Both APIs support collective I/O. Collective I/O interfaces were first enabled in FLASH 3.1; repeated experiments have demonstrated the benefits of collective I/O to the FLASH access pattern [20, 21]. However, it was not initially possible to use the collective mode on Intrepid because the output data were silently corrupted in a non-deterministic fashion. Additionally, the error could not be reproduced on any other platform.

To truly appreciate the challenge of this corruption issue, we need to provide some additional details of the MPI-IO implementation on Intrepid. It was our good fortune to have access to the source code to the BlueGene MPI library. We also had FLASH developers providing ROMIO experts simplified test cases.

We eventually found the problem in the datatype processing part of the MPI-IO library. MPI-IO uses MPI datatypes to describe non-contiguous accesses in memory and in file. Such non-contiguous accesses arise naturally when dealing with scientific data (e.g. a sub-cube of a multi-dimensional array) or when dealing with an I/O library's file format (e.g. describing lists of data blocks stored in an HDF5 file).

The MPI-IO implementation on Intrepid is based on the portable ROMIO MPI-IO library [22] (as are nearly all MPI-IO implementations). ROMIO processes MPI datatypes by 'flattening' them, or constructing a list of offset-length pairs. These flattened representations are in turn stored inside the library so as to avoid paying the cost of generating the representation should a type be used again.

We found that the ROMIO MPI-IO library was assigning the wrong flattened representation to one of these MPI datatypes, re-using a flattened representation from a prior I/O request. This mismatch between datatype and representation resulted in HDF5 files containing corrupted data. Earlier testing failed to find this problem as those tests did not repeatedly set a different file view. The high-level I/O library (HDF5 in this case) does set multiple file views each time a different variable is accessed in the file. While identifying the problem took a great deal of time, the eventual fix was quite simple: fix ROMIO to clean up datatype representations once they are no longer used. We used our own patched version of the MPI-IO library containing this fix until the Intrepid library was upgraded to V1R4.

With collective I/O data corruption fixed, the FLASH team could once again complete simulations in a reasonable amount of time. We present a comparison between collective I/O and independent I/O to emphasize the importance of this optimization for high I/O rates. Furthermore, MPI-IO collective I/O provides the foundation for the other optimizations we investigated: the lack of a correct and efficient MPI-IO implementation hindered our ability to achieve high I/O performance until the collective I/O bug was fixed.

FLASH3 can use the HDF5 library in collective or independent mode through a runtime parameter `useCollectiveHDF5` in the `flash.par` parameter file [19]. This parameter is varied in figure 3 to show the impact of collective I/O optimizations during weak scaling experiments. Note that FLASH3 only uses the PNetCDF library in collective mode and so we choose to show PNetCDF performance measurements in later sections.

The results in figure 3 clearly show that collective I/O optimizations improve write performance and that the improvement is more significant at higher core counts. The collective I/O optimization involves merging I/O requests from multiple processes into fewer larger requests from a subset of processes. The underlying file system delivers higher performance with larger, contiguous request sizes. Consolidating I/O traffic down to a subset of 'I/O aggregators' also reduces the number of processes simultaneously writing to the file system. Further, the MPI-IO library will align writes to file system block boundaries, reducing lock contention [23]. These factors all contribute towards collective I/O taking one tenth the time of the independent I/O case at 8192 MPI processes. At larger process counts, the discussion for this workload is not about improving I/O rates but rather allowing I/O to complete in anything resembling an acceptable amount of time.

These observations are consistent with a recent study on Jaguar at Oak Ridge National Laboratory (a Cray XT4 (at the time of the paper) with a Lustre parallel file system) which also investigated collective I/O

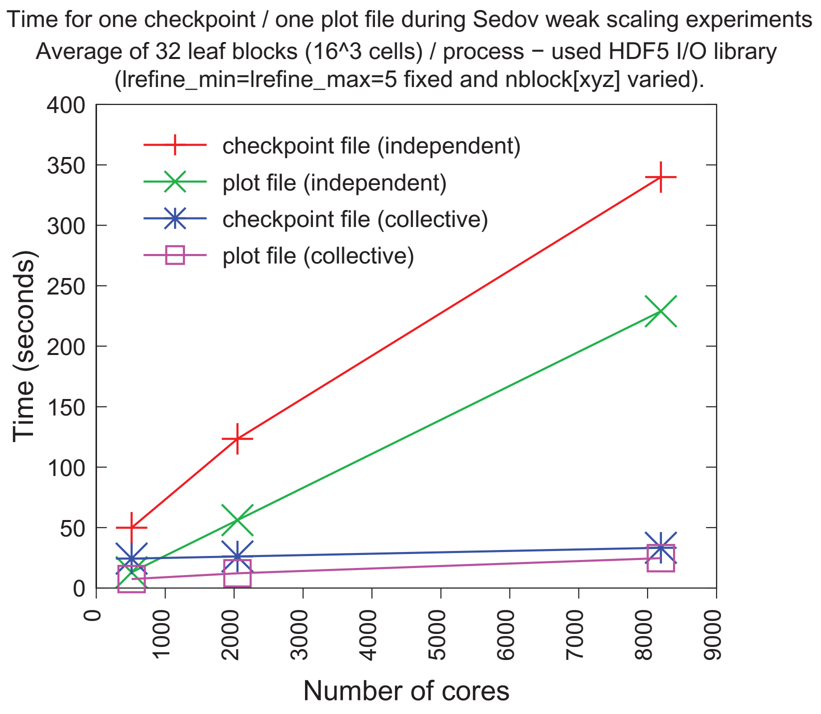


Figure 3. Impact of collective I/O optimizations on the time to write checkpoint files and plot files when using the HDF5 library. Independent I/O exhibits poor scalability, even at scales representing a fraction of the entire machine. Collective I/O reduces the time for these operations by an order of magnitude.

optimizations with FLASH using the HDF5 library [24]. Here, the authors found that a FLASH application run on 8192 cores produced a checkpoint file 2.5 times faster with collective I/O, and 4.6 times faster when the collective I/O was combined with striping the file across all 144 I/O servers (Object Storage Targets (OSTs)). Similar studies also show performance improvement from using collective I/O with FLASH applications on NCAR Bluesky and uP [25] and ASCI White Frost [20].

To generalize this scenario somewhat, as applications ran on petascale systems they frequently encountered difficulty in achieving performance through collective I/O. Not unreasonably, these groups took the pragmatic approach of abandoning collective I/O and adopting alternate approaches (Carns *et al* [26] show the prevalence of file-per-process on a production machine; Bent *et al* [27] describe how to implement file-per-process while presenting a single shared file to the application). However, the involvement of the I/O community frequently results in improvements benefiting all applications [28]. Beyond improvements specific to FLASH, we hope from this and other recent works that applications re-consider the benefits of collective I/O and high-level I/O libraries.

Even with the large performance gains collective I/O provides for standard FLASH, other storage performance studies [18] suggest that the standard FLASH I/O approaches achieve only half of the theoretical peak. We hypothesize that we can attribute some portion of this missing performance to the fact that we make one high-level I/O call per variable, and that if we could perform all I/O in a single write we would get back some of the missing performance. The next two sections document the tricks we applied to further improve I/O performance, and the trade-offs those approaches offer us.

5. Changing the FLASH file layout

When determining the file layout a scientific application will use, a developer will consider several factors. The high-level I/O libraries that FLASH uses offer an API tailored for single-variable access. For example, the parameters to HDF5's `H5DWrite` function describe a memory region and a file region associated with

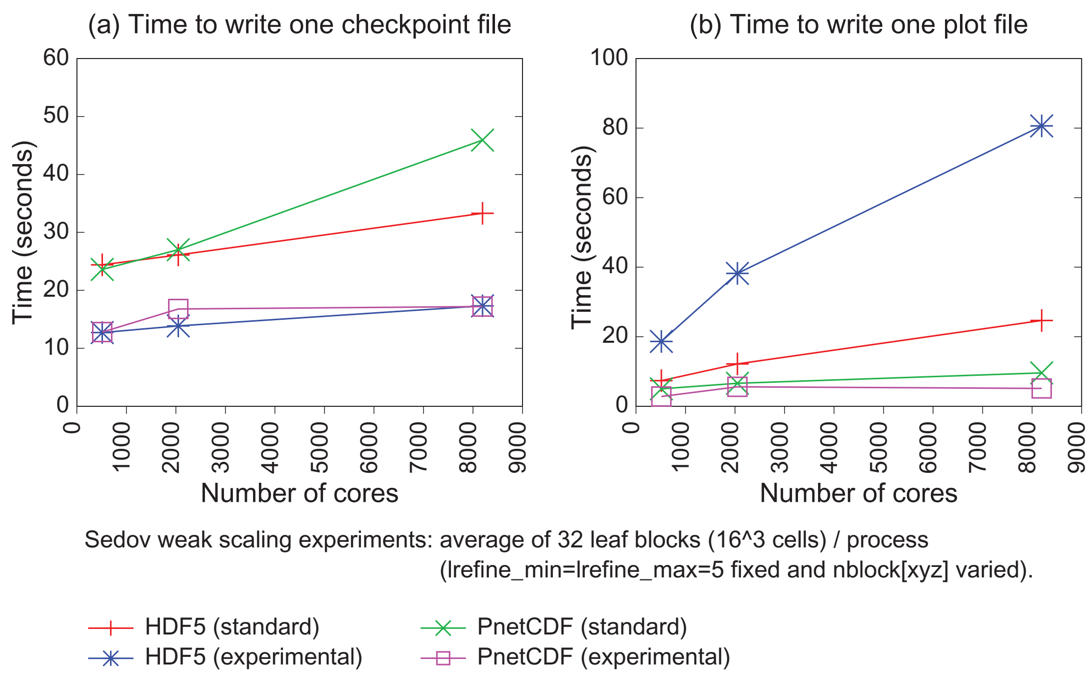


Figure 4. (a) Impact of file layout on the time to write a checkpoint file (a) and a plot file (b).

a single HDF5 variable or dataset. The Parallel-NetCDF `ncmpi_put_vara_double_all` function likewise writes (subarray) data into a specific variable. Separating application data into individual variables on disk offers a straightforward implementation. Furthermore, other tools in the scientific workflow, such as those for visualization or analysis, might find a file layout where each application variable is stored as a separate object easier to manage, and potentially more self-descriptive.

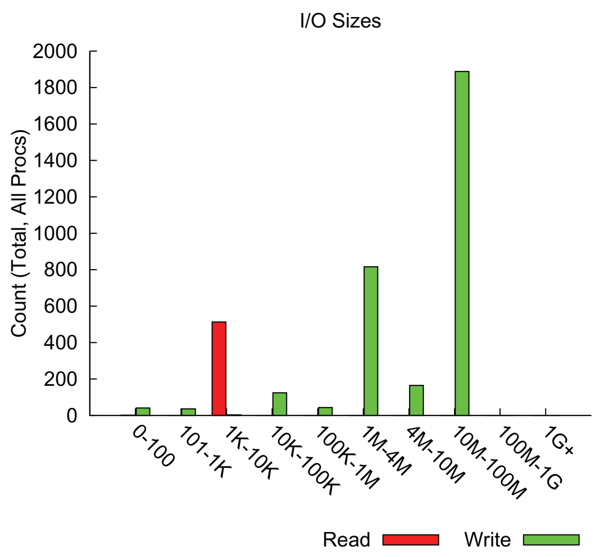
The standard file layout approach (storing application data in multiple library objects), however, offers a slight performance trade-off. Each function call represents a relatively expensive I/O operation. All other factors aside, if the goal is to achieve the highest I/O performance a better approach would describe the entire application I/O pattern and then execute a single call [29]. If the application places all mesh variables into a single I/O library object, as in the experimental file layout approach, then a single I/O library call could be issued to service all application variables instead of N separate calls. Experiments confirm that this approach does improve performance.

Figure 4 shows the average time to write a file for the standard and experimental file layouts for checkpoint and plot files. In this figure, the standard file layout measurements are obtained using the traditional FLASH approach of copying data into a temporary buffer. The experimental file layout measurements are obtained using a datatype memory selection (as described in table 1).

The results generally show that the experimental file layout reduces the time to write checkpoint files and plot files by half (HDF5) to one-third (PNetCDF). This is because the single library call transfers a larger quantity of data and thus gives further opportunity for the MPI-IO library to optimize the file accesses.

One notable exception is the time to write plot files with HDF5 library. When writing plot files, FLASH writes double-precision floating point data out as single precision. The HDF5 library detects this type conversion from double precision in memory to single precision in file and disables collective I/O optimizations, falling back to independent I/O. This type difference does not affect the approach using PNetCDF because PNetCDF allocates extra buffers internally, casts all data to the destination type and then passes that intermediate buffer to the MPI-IO library to transfer data to the file (background in [10]). At these scales and with this amount of I/O being written to disk, the additional memory copies in the PNetCDF library add no measurable overhead.

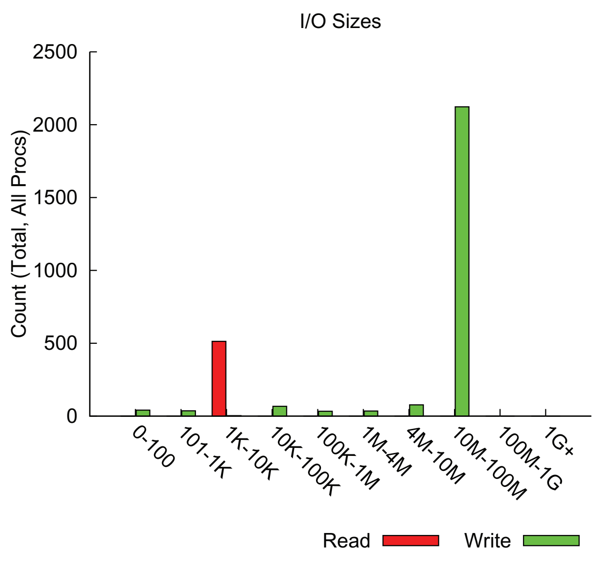
It is possible to quantify how the file accesses change by using the Darshan tool [30] developed at ANL. Darshan is a library that captures information about the usage of MPI-IO and POSIX functions. It uses the MPI



Most Common Access Sizes

access size	count
16777216	1886
4194304	763
1072	512
8388608	120

Figure 5. Selected Darshan high-level statistics for the standard file layout. The 512 1072-byte reads should be ignored as they stem from initially reading a configuration file.



Most Common Access Sizes

access size	count
16777216	2094
1072	512
8388608	66
74896	44

Figure 6. Selected Darshan high-level statistics for experimental file layout. The 512 1072-byte reads should be ignored as they stem from initially reading a configuration file.

profiling interface to monitor MPI-IO functions and wrapper functions inserted using GNU linker to monitor POSIX functions. We show the most relevant statistics when using the PNetCDF library for standard file layout in figure 5 and experimental file layout in figure 6. The improvement in average I/O operation size is most apparent in the histogram, showing that while there are write operations less than 10 MB in size, they account for a small portion of the total. We record an additional 208 16 MiB-sized accesses for the experimental file layout, which appear to replace some of the 763 4 MiB accesses from the standard file layout. We know that the MPI-IO library on BlueGene uses a two-phase collective buffering optimization which by default uses a 16 MiB intermediate buffer. Seeing a large number of 16 MiB access sizes strongly suggests that the two-phase optimization is operating efficiently.

The small file accesses stem from reading the FLASH parameter file and the PARAMESH parameter file and also writing to the FLASH log file. These small operations have a negligible impact on performance at this scale and so have not been scrutinized. The Darshan summaries do suggest that a more scalable approach for reading in this parameter information might be needed for future levels of scalability.

To summarize, the organization of data in a file can have surprising implications for computational scientists. We were able to reduce the time to produce a checkpoint or plot-file from one-half to one-third of the original time. The Darshan tool confirmed our hypothesis that this reduction in time came from a more favorable I/O workload—a larger number of requests for large blocks and a smaller number of short blocks. These improvements come at the cost of a significant change to the FLASH science workflow: a new file format would require updating a host of analysis and visualization tools. In the next section we consider an alternate approach providing all the performance benefits of the alternate file format without actually needing to change the file format.

6. Nonblocking I/O with the Standard File Layout

The experimental layout in section 5 is less convenient for postprocessing tools because all mesh data are stored in the same array. This means that the tools must perform strided reads to extract data for a single mesh variable, e.g. density. This represents a significant trade-off between the write performance and read performance. In addition to the performance trade-off, the Flash Center already has a huge quantity of data laid out in the standard file format and many tools expecting this file format. Examples of such tools include quickflash [31], Visit [32] and custom applications that create simulation movies of galaxy cluster mergers and buoyancy-driven turbulent nuclear combustion.

Ideally, there would be a mechanism that would allow us to combine write operations and give us improved write performance while maintaining the established file layout. A recent extension to the Parallel-netCDF API allows exactly that [33]. The nonblocking Parallel-NetCDF API offers similar semantics as that of MPI nonblocking routines. A caller posts one or more nonblocking operations, passing in a buffer that cannot be modified until a subsequent test for completion indicates that the operation has completed. Typically, these interfaces are used to overlap computation with I/O or communication, but in this case, Parallel-NetCDF uses the interface to combine all posted operations into one larger, more efficient operation in a model similar to that used by Bulk Synchronous Parallel [34]. The write-combining optimization in Parallel-NetCDF provides all of the benefits of the experimental file layout (describing the entire operation with a single request), while retaining the established file layout for compatibility and convenience.

To demonstrate the performance impact of our new approaches, we performed weak scaling experiments up to 65 536 cores on Intrepid. Each experiment wrote out four checkpoint files containing ten double-precision variables and associated annotations and four plot files containing three single-precision variables and associated annotations. The same FLASH binary is used for all experiments and is configured to select grid data using type A for the write-combining tests and types B and C for the experimental file layout tests.

These experiments run at eight times more MPI processors than the experiments presented earlier in this paper. The larger scale necessitated a smaller per-process working size so as to limit overall CPU-hour consumption to a manageable size. Even with the smaller per-process size of 11 blocks per core, at the largest scales we are producing checkpoint files of 229 GiB and plot files of 35 GiB.

As discussed in earlier sections, these two approaches to improving I/O performance—altering the FLASH file layout or using the Parallel-netCDF write-combining optimization—should have the same I/O characteristics. In both cases, the high-level I/O library can issue a single I/O operation to the file system encompassing data from multiple application variables at once. In figures 7 and 8, the experimental format approach and the nonblocking interface to the standard format approach do have essentially identical performance. Larger requests and fewer syncronization points yield three to four times better performance, even at these much larger scales. The unexpected performance drop-off for checkpoint writes at 64 K cores warrants further study.

Why does the standard, one-variable-at-a-time approach fail to scale? Consider the checkpoint case. In this weak scaling study, each processor contributes 374 480 bytes per variable. The MPI-IO layer will apply an optimization called 'I/O aggregation', where the library designates a subset of MPI processes to perform I/O on behalf of all processors. The default setting on BlueGene has one aggregator for every 32 MPI processes.

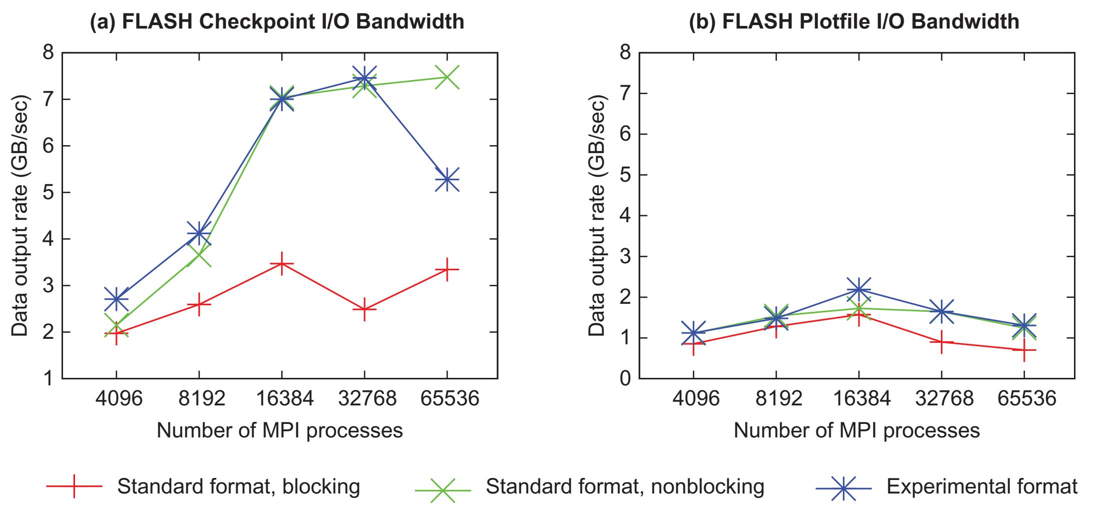


Figure 7. (a) Weak scaling bandwidth results for FLASH checkpoint writes. Aggregating multiple operations, either by changing file layout or by using the PNetCDF nonblocking interface, greatly enhances the scaling ability. (b) Weak scaling results for FLASH plotfile writes. The small amount of I/O per process at these scales prevents high bandwidth rates, but even so, operation aggregation offers a 40% gain in bandwidth.

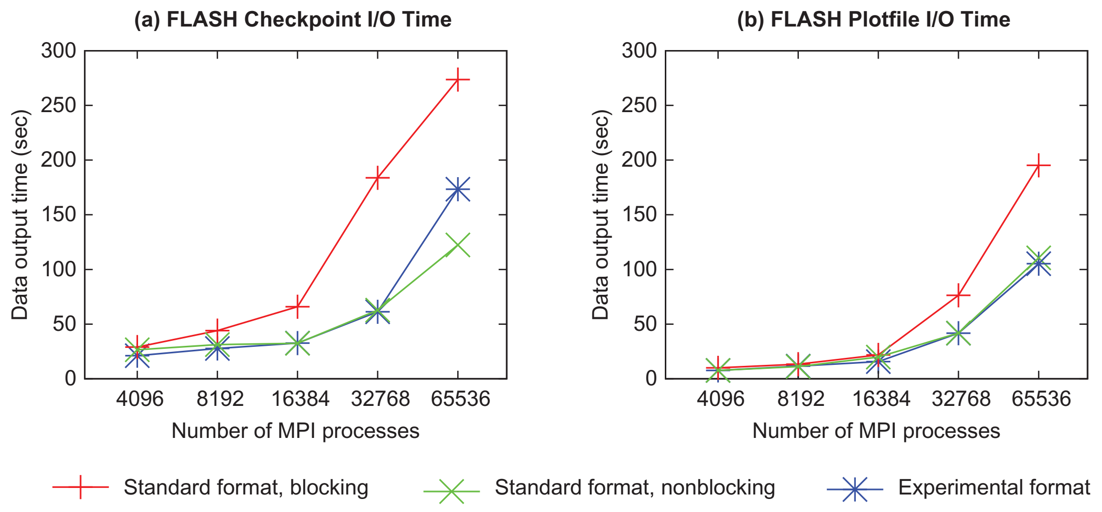


Figure 8. Scaling results of time spent in I/O for the checkpoint (a) and plot file (b).

Thus the I/O aggregators make requests of about 11 MiB. Because checkpoint I/O under the nonblocking I/O approach and the experimental file layout combines ten application variables, the average request size increases by ten times. These two approaches also result in less synchronization. Instead of ten rounds of collective I/O, the alternate approaches perform only a single round.

Plot file I/O only further exacerbates the ‘data per process’ problem. The plot file case writes fewer variables (three, in this case) and those variables are stored in a smaller 4-byte floating point type, instead of the full 8-byte precision used in the checkpoint case. For these plot file runs, each process contributes only 187 KiB of data per variable. MPI-IO’s I/O aggregation optimization brings the average request per variable up to just under 6 MiB. Three plot file variables will result in each I/O aggregator writing 17 MiB.

The MPI-IO library uses a temporary buffer to stage I/O as part of its two-phase optimization. If an aggregator has to process more I/O than this buffer, it breaks the I/O into pieces big enough to store into the scratch buffer and issues multiple writes. The default buffer size on BlueGene is 16 MiB. If we re-run the experiment with a 64 MiB intermediate buffer (big enough to hold all checkpoint and plot file data) we can further improve performance. We did not have enough time to conduct this experiment, however.

The storage system on Intrepid performs best with large I/O requests—of the order of megabytes. These alternate approaches yield plot file I/O performance short of the I/O rates achieved by checkpoint I/O. However, by increasing the average size of the I/O operation seen by the storage system, these approaches boost I/O rates by about 40% over the one-variable-at-a-time technique.

Regardless of the approach, reducing the number of collective operations sees benefits from two sources: increasing the request size and reducing the amount of syncronization. While we can use tools such as the Darshan statistics tool to demonstrate the change in I/O request size, we would also like to be able to quantify the overhead of the two-phase collective I/O optimizations, specifically the portion of time spent in I/O versus the amount of time spent re-arranging the data into a more I/O friendly workload. Darshan provides a good first step; however, it overstates the cost of the communication phase in some configurations, especially the ones like BlueGene where the ratio of compute nodes to I/O aggregators is so high. We have engaged the Darshan developers to come up with a new metric to accurately capture this cost.

7. Related work and future directions

In this work, we applied and evaluated MPI collective I/O and Parallel-NetCDF non-blocking optimizations, at which point we deemed I/O performance sufficiently addressed as far as the FLASH application was concerned. However, the parallel I/O research community has developed many techniques for improving application I/O performance. Even though we did not apply these techniques to FLASH, application groups looking for improved I/O performance should consider these approaches.

At the lowest levels of the I/O software stack research has shown benefits from interfaces allowing a richer description of the I/O workload. The NCIO benchmark is developed and derived from the I/O pattern of earlier FLASH development that produces a high degree of non-contiguity in file access pattern [35]. The optimizations proposed in this work include the list I/O and datatype I/O. Both approaches use concise representations for a large number of non-contiguous requests to reduce the data transfer between file system clients and servers. In this work, the optimizations were implemented in the PVFS2 file system. While PVFS2 is available on the Intrepid machine, the BlueGene I/O forwarding layer unfortunately does not expose the list I/O and Datatype I/O interfaces.

In addition to the MPI-IO optimizations discussed in this work, the MPI-IO layer has been host to further optimizations. Collaborative client-side file caching, a user-space distributed caching layer in MPI-IO, was proposed in [36]. It enables small request aggregation in the cache buffers so that they are later flushed to the file system in order to achieve higher bandwidths.

Various file domain partitioning methodologies in MPI-IO implementation were studied and proposed to align I/O requests with the file system locking and striping configurations in order to minimize the possible lock conflicts [37]. Significant performance improvements have been reported on parallel computers with Lustre and GPFS parallel file systems. The BlueGene MPI-IO library does already implement the GPFS-favorable ‘block aligned’ optimization. Any BlueGene systems running the Lustre file system, however, will need to modify their MPI-IO library.

I/O delegation is proposed as an optimization for MPI-independent I/O [38]. It employs separate MPI processes to perform I/O for the application processes. This dedicated caching process can aggregate small requests, in some ways achieving the benefits of collective I/O without as much communication or synchronization overhead. Large applications like FLASH using MPI-independent I/O can in some configurations outperform collective I/O. This and other research suggests that a ‘set aside process’ facility would be helpful on systems of BlueGene’s scale, but such a facility has not been developed or deployed at this time.

One additional Parallel-NetCDF optimization we did not evaluate was subfiling, proposed in [39]. This approach divides large arrays into smaller ones and saves each subarray to a separate file. Since PnetCDF stores data in the self-describing netCDF file format, subfiling can operate transparently to the users. In addition, dividing write requests among more files reduces the number of processes competing the file system locks and hence can achieve higher performance. In the FLASH case, we were able to optimize this workload fairly closely to peak performance (given the per-client working set sizes) and decided we would face diminishing returns if we were to pursue additional optimizations.

8. Conclusions

This work demonstrates the potential for improving I/O rates in computational science applications through several means. A detailed understanding of the I/O software stack and the storage architecture of the Intrepid machine coupled with an equal level of familiarity with the FLASH data model allowed this collaboration to quickly experiment with altering file formats and novel programming interfaces to reduce checkpoint times for FLASH.

In this paper we show that collective I/O optimizations are important to the write performance of FLASH checkpoint files and plot files. However, using collective I/O optimizations alone may not be enough for the increasing I/O demands of FLASH where the scientists want to use finer-resolution grids, larger numbers of particles and more frequent file output. We demonstrate that further optimization is possible by changing the file layout, and we show that writing all mesh variables to the same dataset can improve write performance significantly. Here, many I/O library writes are replaced with a single I/O library write which gives the MPI-IO library more opportunity for optimization. The low-level impact of this change is monitored using the Darshan library and we find that a larger portion of file accesses involve *big* data transfers.

The file layout change yields higher performance during the simulation phase, but will require updating other tools in the analysis workflow to understand this new file format. The new format would also turn reads of a single mesh variable into a strided read, potentially slowing down read performance significantly. This leads us to experiment with the nonblocking write feature of PNetCDF, which allows us to retain the standard FLASH file layout. We find that this approach gives us performance similar to the experimental FLASH file layout, while maintaining compatibility with existing analysis applications.

We have demonstrated that collaboration between application developers and I/O consultants is essential, especially when there are bugs below the application layer. There are many layers of abstraction in the I/O software stack, and application developers do not have the time or expertise to resolve these problems. In this case study, the collective I/O bug remained an open problem for over a year and prevented running certain science problems at larger scales. Since the fix, collective I/O has been used in all FLASH production simulations on BlueGene/P. In a recent simulation of the ‘Deflagration to Detonation Transition’ (DDT) model of a Type Ia supernova, approximately 7% of total wallclock time was spent in I/O. This simulation was run on 65 536 cores for just under 12 h and produced five checkpoint files, 17 plot files and 74 particle files at a total size of 7.4 TiB. The additional optimization approaches we have discussed reduce the further time spent in I/O and will become more important on future architectures.

Acknowledgments

We thank all the past contributors to the FLASH code. The software described in this work was in part developed by the DOE-supported ASC/Alliance ASC/Flash Center at the University of Chicago under grant B523820. This research used the resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the US Department of Energy under contract DE-AC02-06CH11357. This work was supported in part by the US Department of Energy under contract DE-AC02-06CH11357, SCIDAC SDM Center grant numbers DE-FC02-07ER25808 and DE-SC0001283, ASCR DE-SC0005309 and ASCR DE-SC0005340, the US National Science Foundation: HECURA CCF-0621443, CCF-0938000, SDCI OCI-0724599, ST-HEC CCF-0444405, CNS-0551639 and IIS-0536994.

References

[1] Fryxell B, Olson K, Ricker P, Timmes F X, Zingale M, Lamb D Q, MacNeice P, Rosner R, Truran J W and Tufo H 2000 FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes *Astrophys. J. Suppl. Ser.* **131** 273–334

[2] Antypas K, Calder A C, Dubey A, Fisher R, Ganapathy M K, Gallagher J B, Reid L B, Riley K, Sheeler D and Taylor N 2006 Scientific applications on the massively parallel bg/l machine *Proc. 2006 Int. Conf. Parallel and Distributed Processing Techniques and Applications & Conf. on Real-Time Computing Systems & Applications* ed H R Arabnia pp 292–8

[3] *HDF5* http://hdf.ncsa.uiuc.edu/HDF5/

[4] *Parallel netCDF* http://www.mcs.anl.gov/parallel-netcdf/

[5] Ross R, Nurmi D, Cheng A and Zingale M 2001 A case study in application I/O on linux clusters *Proc. SC2001* pp 1–17

[6] Ching A, Choudhary A, Liao W-K, Ross R and Gropp W 2002 Noncontiguous I/O through PVFS *Proc. IEEE Cluster* ed W Gropp, R Pennington, D Reed, M Baker, M Brown and R Buyya (IEEE Computer Society) pp 405–14

[7] Ching A, Choudhary A, Liao W K and Pundit N 2006 Evaluating i/o characteristics and methods for storing structured scientific data *Proc. Int. Parallel Distributed Processing Symp.* doi: 10.1109/IPDPS.2006.1639306

[8] Yu W and Vetter J S 2008 Parcoll: partitioned collective I/O on the Cray Xt *ICPP* pp 562–9

[9] Ross R, Miller N and Gropp W 2003 Implementing fast and reusable datatype processing *EuroPVM/MPI Conf.* (Berlin: Springer) pp 404–13

[10] Ross R, Latham R, Gropp W, Lusk E and Thakur R 2009 Processing MPI datatypes outside MPI *Lecture Notes in Computer Science* 42–53

[11] Dubey A, Reid L B, Weide K, Antypas K, Ganapathy M K, Riley K, Sheeler D J and Siegal A 2009 Extensible component based architecture for flash, a massively parallel, multiphysics simulation code *CoRR* abs/0903.4875

[12] Palmer B, Koontz A, Schuchardt K, Heikes R and Randall D 2011 Efficient data IO for a parallel global cloud resolving model *Environ. Model. Softw.* **26** 1725–35

[13] Chen J H *et al* 2009 Terascale direct numerical simulations of turbulent combustion using s3d *Comput. Sci. Dis.* **2** 015001

[14] Fu J, Min M, Latham R and Carothers C D 2011 Parallel I/O performance for application-level checkpointing on the blue gene/p system *Cluster Computing (CLUSTER), 2011 IEEE Int. Conf.* pp 465–73

[15] Dongarra J *et al* 2011 The International exascale software project roadmap *Int. J. High Perform. Comput. Appl.* **25** 3–60

[16] *FLASH I/O Benchmark* http://flash.uchicago.edu/∼zingale/flash benchmark io/

[17] *ALCF Computing Resources* http://www.alcf.anl.gov/resources/storage.php

[18] Lang S, Carns P, Latham R, Ross R, Harms K and Allcock W 2009 I/O performance challenges at leadership scale *Proc. of the Conf. on High Performance Comp. Networking, Storage and Analysis* 40:1–40:12

[19] *FLASH3.2 User Guide* http://flash.uchicago.edu/website/codesupport/flash3 ug 3p2.pdf

[20] Li J, Liao W K, Choudhary A, Ross R, Thakur R, Gropp W, Latham R, Siegel A, Gallagher B and Zingale M 2003 Parallel netCDF: a high-performance scientific I/O interface *Proc. SC2003* 39

[21] Yang M and Koziol Q 2006 Using collective IO inside a high performance IO software package—HDF5 *Technical Report* National Center of Supercomputing Applications

[22] Thakur R, Gropp W and Lusk E 1999 On implementing MPI-IO portably and with high performance *Proc. 6th Workshop on I/O in Parallel and Distributed Systems* (New York: ACM) pp 23–32

[23] Yu H *et al* 2006 High performance file I/O for the BlueGene/L supercomputer *Proc. 12th Int. Symp. on High-Performance Computer Architecture (HPCA-12)* 187–96

[24] Jagode H, Knüpfer A, Dongarra J, Jurenz M, Müller M S and Nagel W E 2009 Trace-based performance analysis for the petascale simulation code FLASH 428–39

[25] Chilan C M, Yang M, Cheng A and Arber L 2006 Parallel I/O performance study with HDF5, a scientific data package

[26] Carns P, Harms K, Allcock W, Bacon C, Lang S, Latham R and Ross R 2011 Understanding and improving computational science storage access through continuous characterization *Mass Storage Systems and Technologies, IEEE/NASA Goddard Conf.* pp 1–14

[27] Bent J, Gibson G, Grider G, McClelland B, Nowoczynski P, Nunez J, Polte M and Wingate M 2009 PLFS: a checkpoint filesystem for parallel applications *Proc. Conf. on High Performance Computing Networking, Storage and Analysis, SC '09* (New York: ACM) pp 21:1–21:12

[28] Howison M, Koziol Q, Knaak D, Mainzer J and Shalf J 2010 Tuning HDF5 for lustre file systems *Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS)*

[29] Thakur R, Gropp W and Lusk E 1998 Data sieving and collective I/O in ROMIO *Proc. Seventh Symp. on the Frontiers of Massively Parallel Computation* (Los Alamitos, CA: IEEE Computer Society Press) pp 182–9

[30] Carns P, Latham R, Ross R, Iskra K, Lang S and Riley K 2009 24/7 Characterization of petascale I/O workloads *Proc. First Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS)* (*New Orleans, LA, September 2009.*)
[31] Quickflash http://quickflash.sourceforge.net
[32] VisIt https://wci.llnl.gov/codes/visit/
[33] Gao K, Liao W K, Choudhary A, Ross R and Latham R 2009 Combining I/O operations for multiple array variables in parallel NetCDF *Proc. Workshop on Interfaces and Architectures for Scientific Data Storage, held in conjunction with the IEEE Cluster Conf. (New Orleans, Louisiana)*
[34] Valiant L G 1990 A bridging model for parallel computation *Commun. ACM* **33** 103–11
[35] Ching A, Choudhary A, Liao W K, Ward L and Pundit N 2006 Evaluating I/O characteristics and methods for storing structured scientific data *Proc. 20th Int. Conf. on Parallel and Distributed Processing, IPDPS'06* (Washington, DC: IEEE Computer Society) p 69
[36] Liao W K, Coloma K, Choudhary A and Ward L 2007 Cooperative client-side file caching for MPI applications *Int. J. High Perform. Comput. Appl.* **21** 144–54
[37] Liao W K and Choudhary A 2008 Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols *Proc. 2008 ACM/IEEE Conf. on Supercomputing, SC '08* (Piscataway, NJ: IEEE Press) pp 3:1–3:12
[38] Nisar A, Liao W K and Choudhary A 2008 Scaling parallel I/O performance through I/O delegate and caching system *Proc. 2008 ACM/IEEE Conf. on Supercomputing, SC '08* (Piscataway, NJ: IEEE Press) pp 9:1–9:12
[39] Gao K, Liao W-K, Nisar A, Choudhary A, Ross R and Latham R 2009 Using subfiling to improve programming flexibility and performance of parallel shared-file I/O *Proc. 2009 Int. Conf. on Parallel Processing, ICPP '09* (Washington, DC: IEEE Computer Society) pp 470–7