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], Alok Choudhary[3]

[1] DOE NNSA/ASCR Flash Center, Astronomy & Astrophysics, University of Chicago, Chicago IL

[2] Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL

[3] Center for Ultra-scale Computing and Information Security, Northwestern University, Evanston, IL

E-mail: {robl,rross}@mcs.anl.gov, {cdaley,dubey}@flash.uchicago.edu, {wkliao,kgao,choudhar}@ece.northwestern.edu

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 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. While these optimizations required a detailed understanding of both the FLASH application and the I/O system software, this work demonstrates how collaboration between application and computer science groups can magnify each others efforts.

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.

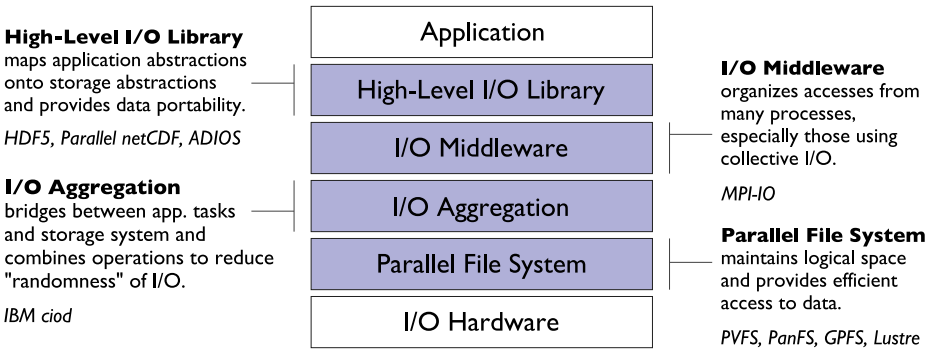


Figure 1. The I/O software stack presents numerous opportunities for optimization.

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. FLASH provides an Adaptive Mesh Refinement (AMR) grid using a modified version of the PARAMESH [3] package and a Uniform Grid (UG) to store Eulerian data. FLASH also contains implementations of parallel I/O using either HDF5 [4] or Parallel netCDF (PNetCDF) [5] 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 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 [6], the Parallel Virtual File System (PVFS) [7] [8], experimental MPI-IO implementations [9], and MPI datatype processing [10] [11]. In fact, download figures from 2007 show that 28% of FLASH downloads are for parallel I/O related studies [12]. 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. First, real scientific codes can achieve high I/O rates with today's I/O stack. Second, 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.

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. We introduce the target BG/P platform and the chosen FLASH test application in Section 3. Sections 4, 5, and 6 cover the performance improvement techniques we applied and their benefit. Finally, we summarize the work and discuss lessons learned during this I/O project in Section 7.

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 which 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.

The existence of blocks is a feature of both the FLASH uniform grid and PARAMESH. In the PARAMESH case, individual blocks may also refine to increase the resolution at specific regions of the computational domain. A refined block containing $16 \times 16 \times 16$ cells produces 2 (1D), 4 (2D) or 8 (3D) child blocks, each with $16 \times 16 \times 16$ cells but half the cell spacing. The child blocks are completely contained within the volume of the parent block.

2.1. Memory Layout

The cell-centered data for cells of blocks assigned to the current processor is stored in a 5-D array of double precision typed data, named `unk`. It is allocated once in the PARAMESH package at the start of the FLASH run and has the same size in each MPI process. It is also allocated once in the uniform grid implementation with the additional simplification of 1 block per processor. 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 extent of the array is given in Equation 1.

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

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 four-dimensional 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 five-dimensional 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 a 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 10 write calls for checkpoint files and 3 write calls for plot files. This approach works well, especially if the application requests collective I/O. It also maps well to the programming interfaces provided by high-level I/O libraries – one variable per function call.

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 ([13], [6]), 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: one, bypassing the temporary buffer by selecting memory regions directly and two, altering the file format so that all application variables reside in a single large 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. 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 3 different datatypes to select data for standard and experimental layouts for both

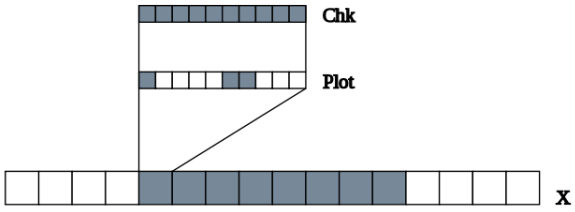


Figure 2. Data extracted from a single PARAMESH block in a 1D simulation with NUNK_VARS=10 (10 mesh variables), nPlotVar=3 (3 mesh variables for visualization), NXB=8 (8 internal cells) and NGUARD=4 (4 guard cells)

.

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 are shown in Table 1.

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

All three derived datatypes incorporate the same pattern of guardcell exclusion. Since this is a simple, regular pattern, it is described using a `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 in PARAMESH.

Note that the construction of these memory descriptions provides a good example of collaboration. The I/O experts can educate 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. I/O Experiments

All experiments are performed on the IBM Blue Gene/P, Intrepid, at Argonne National Laboratory (ANL)[14]. This Blue Gene installation has 160K cores, each operating at 850MHz 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 ([15]). 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 (further details in [16]). 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. We run the application in 3D and use 16^3 cells per block. Each block consists of 10 mesh variables, and the problem size is controlled by adjusting the global number of blocks. Because our interests focused on I/O behavior, we choose to advance only 4 time steps 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 by setting the minimum mesh refinement level equal to the maximum mesh refinement level using parameter values of `lrefine_min = lrefine_max = 5`. This means that 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 $(2^3)^{L-1}$ leaf blocks at level L and there are $\sum_{n=1}^{L}(2^3)^{n-1}$ total blocks up to and including level L, where level L is any fully refined level. In our case of L=5, a single block at the base level mesh creates an oct-tree mesh with $4,096$ leaf blocks and $4,681$ total blocks. This means that we can easily control the total block count by adjusting the number of base level blocks in each coordinate direction using the parameters `nblockx`, `nblocky` and `nblockz`. The experiments are arranged in this way so that we can perform a weak scaling study with powers of 2 core count. The actual parameter values used and the number of generated blocks are shown in Table 3.

All three parameter sets are used in our weak scaling studies that quantify the benefits of collective I/O (Section 4) and examine the performance implications of altering the on-disk format (Section 5). The core count is selected to give approximately 32 leaf blocks per core, which is a problem size per core that is representative of current simulations this application group runs on Intrepid.

Table 2. Experiment parameter options

nblockx	nblocky	nblockz	total blocks	leaf blocks
1	2	2	18,724	16,384
2	2	4	74,896	65,536
4	4	4	299,584	262,144

The third parameter set (describing 299,584 total blocks) is used in our strong scaling study which investigates the Parallel-NetCDF nonblocking optimizations (Section 6). The experiments at high core counts place fewer blocks per core than typical simulations (e.g. only 4 leaf block per core at 65,536 cores). Nonetheless, it is an important regime to explore: the International Exascale Software Roadmap [17] suggests future architectures may have orders of magnitude less 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 amounts 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 10 mesh variables. Plot file I/O writes only selected variables of interest (in these experiments, the 1st, 6th, and 7th 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 plotfile 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 ([18], [19]). However, it was not initially possible to use the collective mode on Intrepid because the output data was silently corrupted in a non-deterministic fashion. Additionally, the error could not be reproduced on any other platform. We found that the ROMIO MPI-IO library was re-using an internal datatype representation incorrectly, which resulted in HDF5 files containing corrupted data. The bug was reported to IBM, and recent BlueGene drivers (since V1R4) have eliminated the bug.

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 [16]. This parameter is varied in Figure 3 to show the impact of collective I/O optimizations during weak scaling experiments. Note, FLASH3 only uses PNetCDF library in collective mode and so we choose to show PNetCDF performance measurements in later sections.

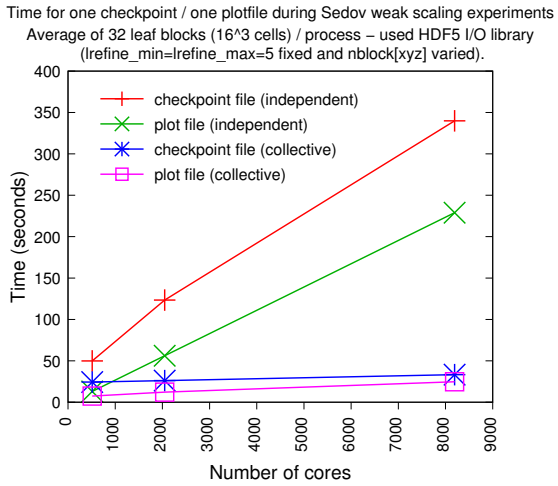


Figure 3. Impact of collective I/O optimizations on the time to write checkpoint files and plot files when using the HDF5 library. Independent I/O exhibits poor scalability, even at scales representing a fraction of the entire machine.

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.

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 optimizations with FLASH using the HDF5 library [20]. Here, the authors found that a FLASH application run on 8,192 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 [21] and

ASCI White Frost [18].

Even with the large performance gains collective I/O provides for standard FLASH, storage performance studies [15] suggest 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 that 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 tradeoffs 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 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 tradeoff. Each function call represents a relatively expensive I/O operation. All other factors aside, if the goal is to achieve highest I/O performance a better approach would describe the entire application I/O pattern and then execute a single call [22]. Placing all mesh variables into a single larger variable, as in the experimental file layout approach, achieves this operation combining, and as will be shown 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 different FLASH binary that selects type B for checkpoint files and type C for plot files, 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. The one notable exception is the time to write plot files with HDF5 library. Here, the type conversion from double precision in memory to single precision in file prevents collective I/O optimizations. HDF5 tries to limit the number of buffer copies. When source and destination have different types, HDF5 must fall back to a slower approach. This does

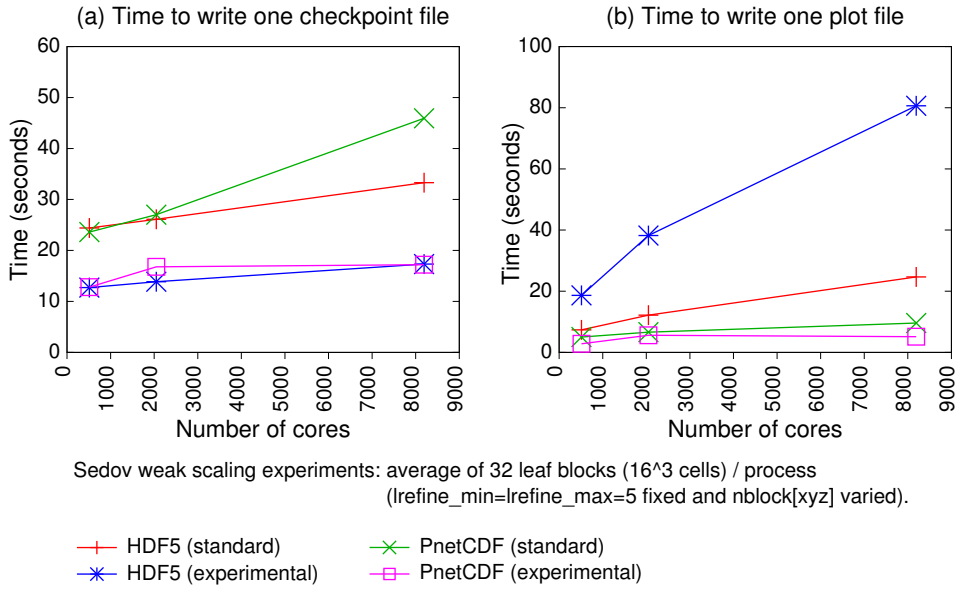
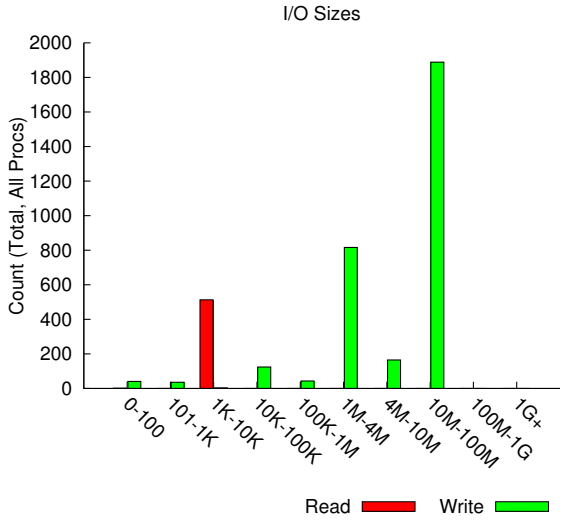


Figure 4. (a) Impact of file layout on the time to write a checkpoint file (a) and a plot file (b).



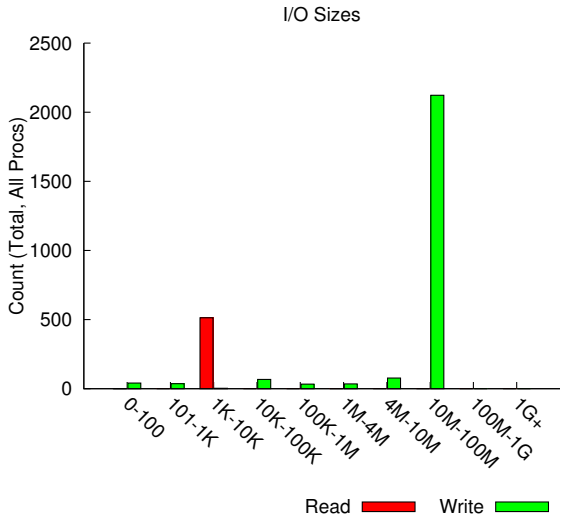
Most Common Access Sizes

access size	count
16777216	1886
4194304	763
1072	512
8388608	120

Figure 5. Selected Darshan high level statistics for standard file layout.

not affect the PNetCDF implementation because PNetCDF allocates extra buffers to transform the data before using MPI-IO to transfer data to file (background in [11]).

It is possible to quantify how the file accesses change by using the Darshan tool [23] developed at ANL. Darshan is a library that captures information about usage of MPI-IO and POSIX functions. It uses the MPI 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 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 while there are write



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.

operations less than 10MB in size, they account for a small portion of the total. We record an additional 208 16MiB-sized accesses for the experimental file layout, which appear to replace some of the 763 4MiB accesses from the standard file layout. We know 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 the two-phase optimization is operating efficiently. We will discuss the MPI-IO library in greater detail in Section 6.1.

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

6. Nonblocking I/O with the Standard File Layout

The experimental layout in Section 5 is less convenient for post processing tools because all mesh data is 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 [24], Visit [25], 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 [26]. 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 [27]. 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 strong scaling experiments up to 65,536 cores on Intrepid. Each experiment wrote out five checkpoint files containing ten double-precision variables and associated annotations and five plotfiles containing three single-precision variables and associated annotations. Each checkpoint file is 92 GiB and each plotfile is 14 GiB. 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 type B and C for the experimental file layout tests.

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 Figure 7 the experimental format approach and the nonblocking interface to the standard format approach do have essentially identical performance.

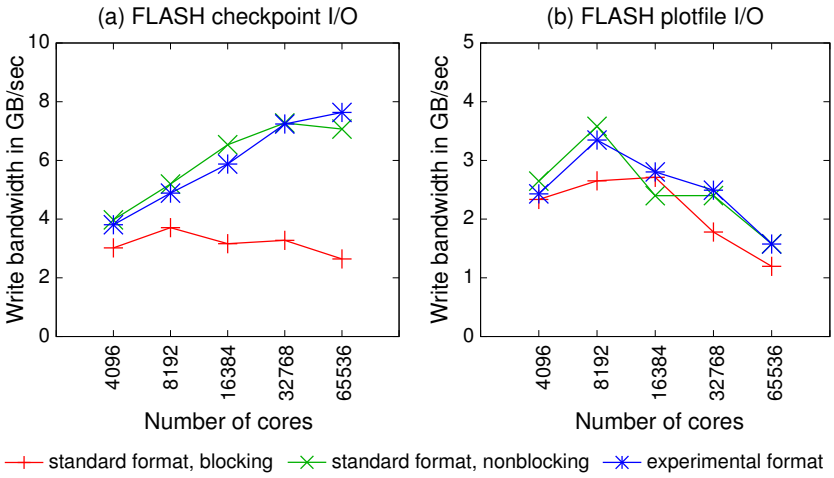


Figure 7. (a) Strong scaling results for FLASH checkpoint writes. Aggregating multiple operations, either by changing file layout or by using the PNetCDF nonblocking interface, greatly enhances strong scaling ability. (b) Strong scaling results for FLASH plotfile writes. The small amount of I/O per process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

Checkpoint data per MPI process and MPI-IO aggregator

Processors	4,096	8,192	16,384	32,768	65,536
Aggregators	128	256	512	1,024	2,048
Data per aggregator					
blocking	73.6 MiB	36.8 MiB	18.4MiB	9.0 MiB	4.5 MiB
nonblocking	736 MiB	368 MiB	184MiB	89.6 MiB	44.8 MiB

Table 3. Amount of data per aggregator (I/O processes) for checkpoint I/O under strong scaling. The nonblocking API combines multiple operations and in concert with MPI-IO collective I/O optimizations sends larger request sizes to the file system.

Plotfile data per MPI process and MPI-IO aggregator

Processors	4,096	8,192	16,384	32,768	65,536
Aggregators	128	256	512	1,024	2,048
Data per aggregator					
blocking	37 MiB	18 MiB	9.3 MiB	4.7 MiB	2.3 MiB
nonblocking	110 MiB	55 MiB	27 MiB	14 MiB	7.0 MiB

Table 4. Amount of data per aggregator (I/O process) for plotfile I/O under strong scaling. Without the nonblocking write-combining, average request size for plotfile I/O is quite small.

Why does the standard, one variable at a time approach fail to scale? We have fixed the total amount of I/O. At 4,096 MPI processors, each process contributes 2.3 MiB of data per variable. At 65,536 processors, that amount goes down to 146 KiB of data per variable. Even after the MPI-IO layer applies I/O aggregation, the I/O aggregators still make request of about 4 MiB (Table 3). The nonblocking I/O approach and the experimental file layout both result in larger I/O request sizes. These two approaches also result in less synchronization. Instead of ten rounds of collective I/O, the alternate approaches perform only a single round.

Plotfile I/O only further exacerbates the "data per process" problem. The plotfile 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 runs, plotfile I/O requires only 1.1 MiB per process at 4,096 processes and 73 KiB per process once scaled up to 65,536 processes. At the highest scale, even MPI-IO's I/O aggregation optimization cannot bring the average request size higher than about 2.3 MiB, as shown in Table 4. The storage system on Intrepid performs best with large I/O requests – on the order of megabytes. While alternate approaches leave us well short of the I/O rates achieved by checkpoint I/O, these approaches still increase the average size of the I/O operation seen by the storage system, boosting rates by about 40% over the variable-at-a-time technique.

6.1. Detailed Overhead Analysis

From prior experience we know the two-phase I/O strategy, while a powerful optimization technique for collective I/O, does impose some overhead. The data exchange phase among processes requires a significant amount of network communication before carrying out the final well-formed I/O operation [28]. At these scales it is reasonable to ask how much overhead each portion of the collective I/O code path contributes. The two-phase method ROMIO implements selects a subset of processes to serve as "I/O aggregators". These processes will carry out I/O on behalf of all processes. Next, the library examines the I/O requests of all processes participating in the collective I/O call. The library then splits up the file into "file domains" and assigns these file domain to the aggregators.

The three main sources of overhead for a collective write operation are

Metadata: Every process exchanges information about their portion of the I/O request. Processes then decompose the file into file domains and construct an I/O plan or schedule.

Exchange: The first phase of "two-phase". Every process knows how data is laid out in memory and where it must go in the file. Processes send data to the responsible I/O aggregator.

Writing: The second phase of "two-phase". With a full buffer of (now) contiguous data, the aggregators perform the actual I/O.

We instrumented the MPI-IO library to capture timing information around these major parts of the collective I/O code path. The FLASH application generates datasets in the standard layout in both cases. We are interested in what, if any, additional processing overhead we incur when passing the more complex requests types generated by the Parallel-NetCDF nonblocking interface down to the MPI-IO layer. In Figure 6.1, we show the total time for the collective `MPI_File_write_all` call and the percentage of time the MPI library spent in the three main sections.

The "Exchange" phase on BlueGene is implemented with a highly-tuned `MPI_Alltoallv`. The "Writing" phase is entirely limited by performance of the I/O storage infrastructure. For this application we appear to have reached the optimization limits. Alternate file domain partitioning schemes such as Persistent File Domains [29] will only have a small impact on performance. If we were determined to apply further optimizations to this workload, we would evaluate write-behind caching strategies or other techniques to offload I/O processing from the application.

7. 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

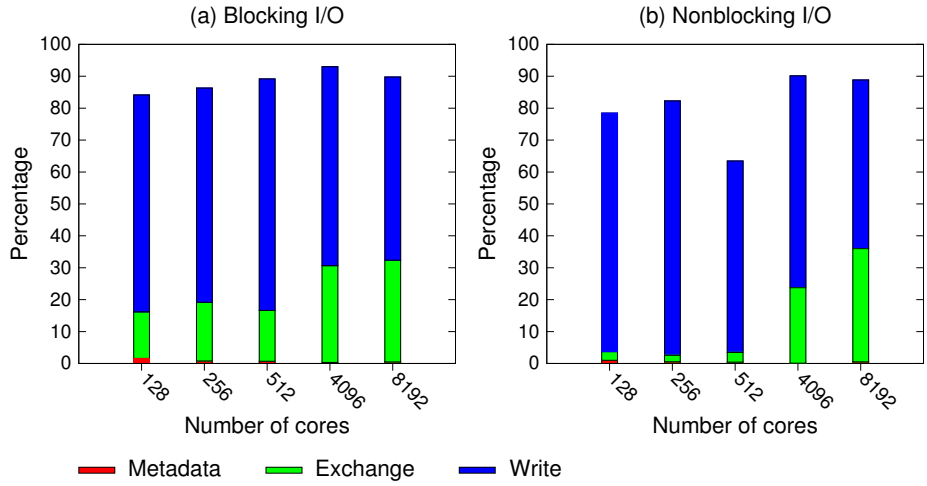


Figure 8. Percentage of time MPI-IO calls spend in major sources of overhead. I/O efficiency more than offsets the slight increase in relative overhead from processing the PNetCDF nonblocking-generated requests to the MPI-IO library.

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 our 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 improves 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 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. The fix means that I/O is much less

of a bottleneck in FLASH applications and is typically less than 10% of total runtime in production simulations using up to 132K processors.

8. Acknowledgments

We wish to 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 resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This work was supported in part by the U.S. Dept. of Energy under Contract DE-AC02-06CH11357, SCIDAC SDM Center grant no. DE-FC02-07ER25808 and DE-SC0001283, ASCR DE-SC0005309, and ASCR DE-SC0005340, the US National Science Foundation (NSF): HECURA CCF-0621443, CCF-0938000, SDCI OCI-0724599, ST-HEC CCF-0444405, CNS-0551639, and IIS-0536994.

Bibliography

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

[2] K. Antypas, A. C. Calder, A. Dubey, R. Fisher, M. K. Ganapathy, J. B. Gallagher, L. B. Reid, K. Riley, D. Sheeler, and N. Taylor. Scientific applications on the massively parallel bg/l machine. In H. R. Arabnia, editor, *Proceedings of the 2006 International Conference on Parallel & Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems & Applications*, pages 292–298, November 2006.

[3] P. MacNeice, K.M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.

[4] HDF5. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[5] Parallel netCDF. `http://www.mcs.anl.gov/parallel-netcdf/`.

[6] Robert Ross, Daniel Nurmi, Albert Cheng, and Michael Zingale. A case study in application i/o on linux clusters. In *in Proceedings of SC2001*, pages 1–17, 2001.

[7] A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In William Gropp, Rob Pennington, Dan Reed, Mark Baker, Maxine Brown, and Rajkumar Buyya, editors, *Proceedings of IEEE Cluster*, pages 405–414. IEEE Computer Society, 2002.

[8] Avery Ching, Alok Choudhary, Wei keng Liao, and Neil Pundit. Evaluating i/o characteristics and methods for storing structured scientific data. In *In Proceedings of the International Parallel and Distributed Processing Symposium*, 2006.

[9] Weikuan Yu and Jeffrey S. Vetter. Parcoll: Partitioned collective i/o on the cray xt. In *ICPP*, pages 562–569, 2008.

[10] Robert Ross, Neill Miller, and William Gropp. Implementing fast and reusable datatype processing. In *In EuroPVM/MPI*, pages 404–413. Springer Verlag, 2003.

[11] Robert Ross, Robert Latham, William Gropp, Ewing Lusk, and Rajeev Thakur. Processing MPI datatypes outside MPI. *Lecture Notes in Computer Science*, September 2009.

[12] Anshu Dubey, Lynn B. Reid, Klaus Weide, Katie Antypas, Murali K. Ganapathy, Katherine Riley, Daniel J. Sheeler, and A. Siegal. Extensible component based architecture for flash, a massively parallel, multiphysics simulation code. *CoRR*, abs/0903.4875, 2009.

[13] FLASH I/O benchmark. `http://flash.uchicago.edu/~zingale/flash_benchmark_io/`.

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

[15] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *Proceedings of Supercomputing*, November 2009.

[16] FLASH3.2 User Guide. `http://flash.uchicago.edu/website/codesupport/flash3_ug_3p2.pdf`.

[17] Jack Dongarra and Pete Beckman et al. The international exascale software roadmap. *to appear in International Journal of High Performance Computer Applications*, 25(1), 2011.

[18] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, , and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003*, November 2003.

[19] MuQun Yang and Quincey Koziol. Using collective io inside a high performance io software package HDF5. Technical report, National Center for Supercomputing Applications, 2006.

[20] Heike Jagode, Shirley Moore, Dan Terpstra, Jack Dongarra, Andreas Knpfer, Matthias Jurenz, Matthias S. Mller, Wolfgang E. Nagel, and Technische Universitt Dresden Germany. Trace-based performance analysis for the petascale simulation code flash, 2009.

[21] C.M. Chilan, M. Yang, A. Cheng, and L. Arber. Parallel i/o performance study with hdf5, a scientific data package, 2006.

[22] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1998.

[23] Philip Carns and Robert Latham and Robert Ross and Kamil Iskra and Samuel Lang and Katherine Riley. 24/7 Characterization of Petascale I/O Workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS)*, New Orleans, LA, September 2009.

[24] Quickflash. `http://quickflash.sourceforge.net`.

[25] VisIt. `https://wci.llnl.gov/codes/visit/`.

[26] Kui Gao, Wei keng Liao, Alok Choudhary, Robert Ross, and Robert Latham. Combining I/O Operations for Multiple Array Variables in Parallel NetCDF. In *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage, held in conjunction with the the IEEE Cluster Conference, New Orleans, Louisiana*, September 2009.

[27] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[28] Avery Ching, Alok Choudhary, Kenin Coloma, Wei Keng Liao, Robert Ross, and William Gropp. Noncontiguous access through MPI-IO. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.

[29] Kenin Coloma, Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, Rajeev Thakur, and Lee Ward. A new flexible MPI collective I/O implementation. In *Proceedings of the IEEE Conference on Cluster Computing (Cluster 2006), Barcelona, Spain*, September 2006.