A Case Study in Application I/O on Linux Clusters

Robert Ross Daniel Nurmi
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA
{rross, nurmi}@mcs.anl.gov

Albert Cheng
Scientific Data Technologies Group, NCSA
University of Illinois at Urbana-Champaign
Champaign, IL 61820, USA
acheng@ncsa.uiuc.edu

Michael Zingale
Center on Astrophysical Thermonuclear Flashes
University of Chicago
Chicago, IL 60637, USA
zingale@flash.uchicago.edu

Abstract

A critical but often ignored component of system performance is the I/O system. Today's applications demand a great deal from underlying storage systems and software, and both high-performance distributed storage and high level interfaces have been developed to fill these needs.

In this paper we discuss the I/O performance of a parallel scientific application on a Linux cluster, the FLASH astrophysics code. This application relies on three I/O software components to provide high-performance parallel I/O on Linux clusters: the Parallel Virtual File System, the ROMIO MPI-IO implementation, and the Hierarchical Data Format library. Through instrumentation of both the application and underlying system software code we discover the location of major software bottlenecks. We work around the most inhibiting of these bottlenecks, showing substantial performance improvement. We point out similarities between the inefficiencies found here and those found in message passing systems, indicating that research in the message passing field could be leveraged to solve similar problems in high-level I/O interfaces.

1 Introduction

Today's scientific applications run on hundreds or thousands of processors and rely on system software to pass structured data between processes and store this data for later consumption. No single software component can provide all the functionality necessary for these applications to operate. What we have seen, instead, is the growth of system software packages that provide some subset of the functionality necessary,

(c) 2001 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

Table 1: Software stack examined in this work

FLASH Application
HDF5
ROMIO
PVFS

including message passing components and I/O libraries. In many cases these packages themselves rely on other components in order to operate.

Supporting high-performance I/O in parallel computing environments has proven to be a difficult task, especially with the competing demands of application-driven interfaces and high performance. System software packages now exist that provide various pieces of this I/O puzzle; together these packages create a stack that provides a high-level application-oriented interface at the top and high-performance parallel I/O storage at the bottom. However, because of the wide variety of packages, hardware, applications, and configuration parameters, performance through such a hierarchy can vary.

In this work we examine the performance of one such collection of system software in the context of the FLASH I/O benchmark, which simulates the I/O pattern of an important scientific application. We find that initially the application performance falls far below our expectations, and we detail the process of identifying the largest performance bottleneck and overcoming it. This process in turn has revealed areas for future improvement in the system software.

The remainder of this paper is organized as follows. In Section 2 we discuss the system software used in this work. In Section 3 we cover the FLASH application and the I/O benchmark derived from it. Section 4 describes initial performance of the FLASH I/O benchmark, the instrumentation process, the discovery of the major performance bottleneck, and our final performance numbers. In Section 5 we draw conclusions and outline our plans for future work.

2 I/O System Software

This work centers on the combined use of three I/O system software packages to provide both a convenient API and high performance I/O access. The Parallel Virtual File System (PVFS), the ROMIO MPI-IO implementation, and the Hierarchical Data Format (HDF5) application interface together provide the I/O functionality needed by the FLASH code. The organization of these components into a software stack is shown in Table 1, with the FLASH application at the highest level and PVFS at the lowest level. In this section we discuss some of the most important features of these packages.

2.1 Parallel Virtual File System

The Parallel Virtual File System (PVFS) [3] is a parallel file system designed for Linux clusters. It is intended to fill two roles, both providing a high-performance parallel file system for these clusters that anyone can obtain and use for free, and serving as an infrastructure for future I/O research. PVFS has been used to create multiterabyte distributed file systems, and aggregate bandwidths to PVFS file systems have been shown to exceed 3 Gbytes/sec when using large numbers (e.g., 128) of I/O servers with modest I/O hardware.

PVFS is implemented using a client-server architecture. A collection of user-space processes together provide a cluster-wide consistent name space and store data in a striped manner across multiple nodes in the cluster. Messages between clients and servers are passed over TCP/IP, and all PVFS file system data is stored in files on file systems local to the processes. Two mechanisms are provided for client-side access to the system. First, a user library of I/O calls, called `libpvfs`, provides a low-level interface to the PVFS servers similar to the UNIX I/O interface [11]. Applications may link to this library to directly access PVFS file systems from user space, avoiding kernel file system overhead on the client side. Other I/O libraries can also take advantage of this interface to obtain access to PVFS file systems [3, 4, 15]. Second, a Linux kernel module is provided that implements client-side kernel support for PVFS file systems. This module, along with an accompanying process, allows PVFS file systems to be "mounted" in the same manner as an NFS or local file system, merging the PVFS file system into the local directory hierarchy. Through this mechanism existing binaries may manipulate data on PVFS file systems or even be executed off a PVFS file system.

In the context of this work, PVFS provides a high-performance I/O infrastructure for storage of application checkpoint and output data. Higher-level I/O interfaces will build on this by linking to the `libpvfs` library, which allows user-level access to PVFS.

2.2 ROMIO MPI-IO Implementation

The MPI message passing interface specification provides a de facto standard for message passing in parallel programs. The MPI-2 specification, which builds on the successful MPI-1 specification, includes a section on I/O commonly referred to as MPI-IO [10, 13]. Just as MPI has become a popular choice for message passing in parallel applications, the MPI-IO interface has become a prominent low-level interface for I/O access in parallel applications.

ROMIO is one implementation of the MPI-IO interface [17]. It is unique in that it implements an abstract I/O device (ADIO) layer that aids in porting ROMIO to new underlying I/O systems. The success of this design is evident in the number of systems for which ROMIO provides MPI-IO support, including HP, IBM, NEC, SGI, and Linux.

In Linux clusters ROMIO can be configured to operate on top of PVFS, providing applications using the MPI-IO interface direct access to PVFS file systems [3]. This allows applications to utilize this high-performance I/O option without constraining the application programmers to using the PVFS interface. Some application programmers, however, desire even more high-level interfaces for data storage that more easily map to the application data structures. For these users ROMIO fills a different role; it provides a standard interface on which higher-level I/O interfaces may be built. We discuss one such interface in the following section.

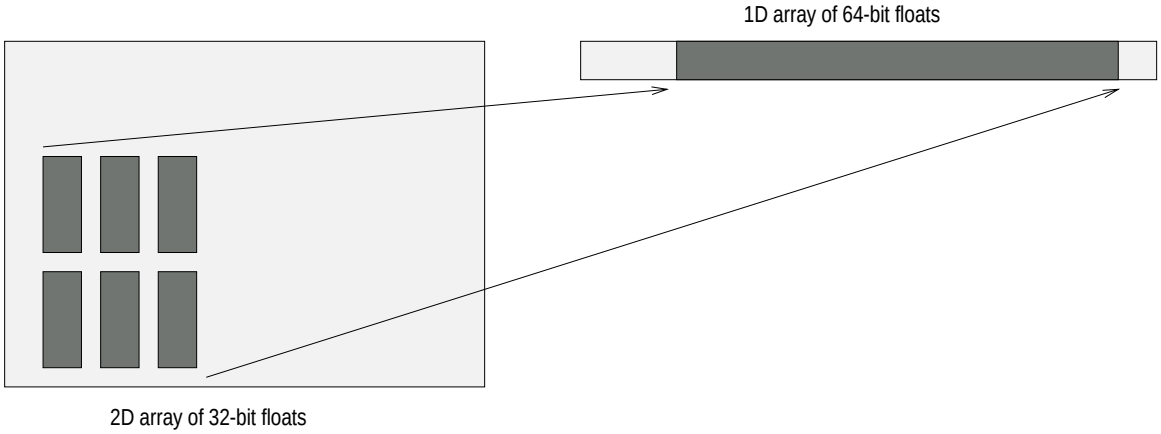

Figure 1: Example of I/O selection operation in HDF5. A regular series of blocks from a 2D array of 32-bit floats is written to a contiguous sequence of 64-bit floats at a certain offset in a 1D array.

2.3 Hierarchical Data Format 5

HDF5 is an I/O library for high performance computing and scientific data management. HDF5 can store large numbers of large data objects, such as multidimensional arrays, tables, and computational meshes, and these can be mixed together in any way that suits a particular application. HDF5 supports cross platform portability of the interface and corresponding file format, as well as ease of access for scientists and software developers. HDF5 and its predecessor HDF4 are de facto standards in use by many scientific and engineering applications, including the NASA Earth Observing System and DOE ASCI projects.

The main conceptual building blocks of HDF5 are the "dataset" and the "group." An HDF5 dataset is a multidimensional array of elements of a specified datatype. Datatypes can be atomic (integers, floats, and others) or compound (like C structs). HDF5 groups are similar to directory structures in that they provide a way to explicitly organize the datasets in an HDF5 file. The FLASH code, examined later in this work, uses datasets as the basic record structure for storing data. It also makes use of compound datatypes to improve I/O performance when more than one variable is being accessed at a time.

When reading or writing an HDF5 dataset, an application describes two datasets: a source dataset and a destination dataset. These can have different sizes and shapes and, in some instances, can involve different datatypes. When an application writes a subset from the source to the destination, it specifies the subset of the source data that is to be written and the subset of the destination that is to receive the data. The only restriction on this operation is that the two subsets contain the same amount of data; like the datasets they can have entirely different shapes. Figure 1 illustrates an I/O involving datasets of different sizes, shapes, and datatypes. These subsets are selected by defining *hyperslabs*, which are similar in many ways to MPI derived datatypes.

The HDF5 library contains a "virtual file layer" (VFL), a public API for doing low level I/O, so that drivers can be written for different storage resources. One of these VFL implementations is written to use MPI-I/O, making it possible to link to ROMIO/PVFS. This in turn means that any application that uses HDF5 for I/O can take advantage of the ROMIO/PVFS implementation.

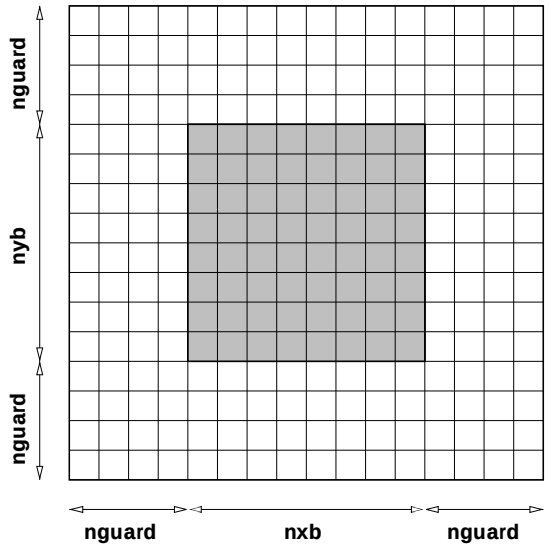

Figure 2: Structure of a single AMR block. The shaded region in the center is where the solution is updated. The perimeter of guardcells is used to hold information from neighboring blocks for the hydrodynamics algorithms.

When performing I/O in a multilayered configuration such as the one we are discussing here, it is useful to have a mechanism for passing information from one layer to another. To this end, the HDF5 API provides a "property list," which is a list of specifications, guidelines, or hints to be used by the library itself or to be passed on by the VFL to lower layers. Since different platforms vary a great deal in how they do I/O, the choice of properties often depends on which platform is being used. Hence, an application may set different properties in different computing environments. Property lists are exploited in this application for directing certain MPI-IO operations, such as the use of collective I/O. Since the set of valid properties is extendable, we expect to discover new uses of this capability as we carry out studies such as this one.

3 FLASH and the FLASH I/O Benchmark

The FLASH code [7] is an adaptive mesh, parallel hydrodynamics code developed to simulate astrophysical thermonuclear flashes in two or three dimensions, such as Type Ia supernovae, Type I X-ray bursts, and classical novae. It solves the compressible Euler equations on a block-structured adaptive mesh and incorporates the necessary physics to describe the environment, including the equation of state, reaction network, and diffusion.

FLASH is written in Fortran 90 and uses MPI for interprocess communication. The target platforms are the ASCI machines (Blue Pacific at LLNL, ASCI Red at SNL, and Nirvana at LANL) and Linux clusters (Chiba City at ANL and C-plant at SNL). The code scales well up to thousands of processors [2] and has been used for numerous production runs.

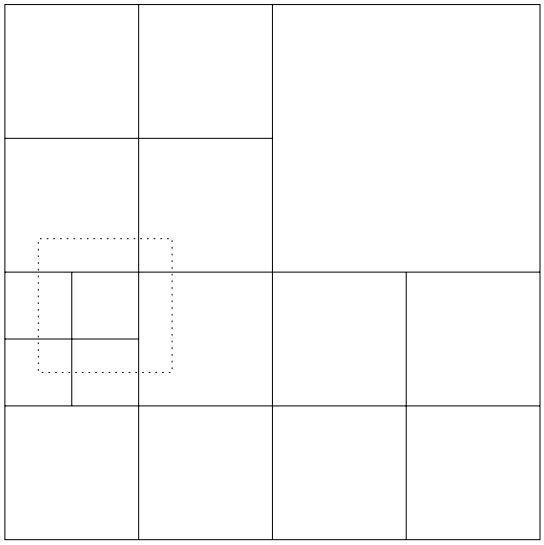

Figure 3: A simple two-dimensional domain showing the AMR block structure. The dotted box around one of the finest blocks shows the region encompassed by the guardcells.

The physical problems we are interested in studying involve a large range of length scales, up to 13 orders of magnitude. To increase the range of length scales we can follow in a simulation, FLASH employs adaptive mesh refinement (AMR) through the PARAMESH package [12]. AMR allows the code to put more resolution at complex flow features, such as shock fronts and material interfaces, and to follow smooth flows with coarser resolution. The typical domain shape is rectangular because there are no complicated boundaries in the problems of interest. A structured grid is ideal with this geometry, allowing for direct addressing of data blocks and more efficient use of cache-based machines.

The computational domain is divided into small blocks, each containing eight computational zones in each coordinate direction. Additionally, each block has a perimeter of four guardcells to hold the state variables of the neighboring blocks for use in the hydrodynamics algorithm (see Figure 2). In each zone, we typically store around 24 variables (e.g., density, velocities, energy, and pressure).

Figure 3 shows a simple two-dimensional computational domain with four levels of refinement. We restrict blocks to have at most a one level of refinement difference with their neighbors. When a block is refined, it is cut in half in each direction, creating four new child blocks in two dimensions or eight new child blocks in three dimensions. Finer blocks have twice the spatial resolution of their coarser neighbors. The dashed box around one of the finer blocks in the figure indicates the region that the guardcells cover. All of these blocks, regardless of their physical dimensions, have the same logical layout: eight blocks in each coordinate direction and four guardcells on the border. The block hierarchy is managed by an oct-tree, relating the parents and their children through the levels of refinement. Load balancing is achieved by passing a work-weighted space filling curve through the blocks, which produces a one-dimensional ordering of the blocks. Blocks are then partitioned based on this ordering so that equal work is placed on each processor. Typically 50-100 blocks are placed on each processor.

A typical large production run of FLASH will generate around 0.5 Tbytes of data, distributed between 1,000 plotfiles and 100 checkpoint files. In early production runs on the ASCI platforms, I/O took as much as half of the total run time on 1,024 processors. This amount of time was considered excessive and could be better spent furthering the calculation, hence the importance in speeding up the I/O.

With this problem in mind, the FLASH I/O benchmark was created to test the I/O performance of FLASH independently of the entire code. It sets up the same data structures as FLASH/PARAMESH and fills them with dummy data. I/O is then performed through one of a variety of interfaces; for this work parallel HDF5 is used.

The FLASH I/O benchmark performs three separate performance tests: checkpoint, plotfile without corners, and plotfile with corners. Checkpoint data is used to restart the application in the event of failure during a run. This checkpoint file needs to store all the variables (excluding the guardcells), the tree structure, and some small additional data including the current simulation time, the current time step, and the number of steps. The computational blocks account for $> 95\%$ of the data written during each checkpoint. A total of 24 separate I/O operations, one per variable, is used to write all the computational block data during the checkpoint operation.

The last two tests measure the time to create plotfiles, which are used for visualization of a run. These have the same format as checkpoint files, but with fewer variables stored. Additionally, the variables are stored with reduced precision (4-byte reals instead of 8-byte reals), which is adequate for visualization work. These blocks are also written with a separate I/O operation per variable. The "plotfile with corners" test stores plotfile data as a 9x9x9 interpolated block rather than the normal 8x8x8 block, as this aids in the visualization phase. The creation of the interpolated block during the run costs a small amount of time, but it avoids an extra postprocessing step. These plotfiles are stored with each variable in a separate record, just as in the checkpoint case.

The variables in checkpoint and plotfiles are written in this manner for a number of reasons. First, the subset of variables we choose to store in a plotfile may not be adjacent. Hence, we have to either pack the data together in a large buffer or write each variable individually. Additionally, the data needs to be interpolated to the corners for some plotfiles, which also requires a buffer to store the interpolated result. Either way, creating a buffer that holds all of the variables that we wish to store in one write would use a lot of memory. Finally, after the simulation is completed, a great deal of analysis is performed on this output data, typically by looking at one variable at a time. Storing each variable separately greatly decreases the amount of time it takes to extract a single variable from the data file.

Since single variables are extracted from the array of blocks, the values are not contiguous in memory. This extraction is performed by the HDF5 library using its hyperslab functionality. The HDF5 hyperslab functionality serves two purposes: it avoids code in the application to perform packing, and it eliminates the need for a separate pack buffer. A hyperslab is defined by the application, which describes the variable desired, excluding guardcells, and this hyperslab is used by the HDF5 library to extract the appropriate values during the write process.

4 Performance Results

The performance results presented here use PVFS on the Chiba City [5] cluster at Argonne National Laboratory. At the time of our experiments, the cluster was configured as follows. There were 256 nodes used for computation, each with two 500 MHz Pentium III processors, 512 Mbytes of RAM, a 100 Mbits/sec Intel EtherExpress Pro Fast Ethernet network card, and a Myrinet card. The nodes were running Linux 2.4.2 and using version 1.4.1pre2.2 of the GM (Myrinet) driver. All MPI communication was performed over the Fast Ethernet using the latest development version of MPICH. All PVFS data transfer took place over the Myrinet network. In all tests two processes were run on each node in order to take full advantage of the dual processors.

In addition to the compute nodes, Chiba City has eight storage nodes. All the data presented here is from runs using these nodes to form an eight-node PVFS file system. These nodes are single-processor 500 MHz Pentium III machines with 512 Mbytes of RAM and Myrinet cards. Each of these has an Adaptec AIC-2940 Ultra Wide SCSI controller attached to an IBM EXP15 disk array with four SCSI disks organized into a single logical block device. The disk array has a maximum bandwidth of 40 Mbytes/sec (because of hardware limitations). A ReiserFS [14] file system is built on top of this and used by the I/O server for local data storage. The Bonnie [1] I/O benchmark reported 35.9 Mbytes/sec write bandwidth for a single such file system with 94% CPU utilization. This test was performed with a 1 Gbyte file. This test indicates that we have an aggregate I/O bandwidth of 287 Mbytes/sec to local storage on these nodes.

Since PVFS currently uses TCP for all communication, we measured the performance of TCP over the Myrinet network using the `ttcp` test, version 1.11 [16]. The `ttcp` program reported a bandwidth of 65.5 Mbytes/sec using a 1 Kbyte buffer with 91% CPU utilization. This gives us an aggregate network bandwidth of 524 Mbytes/sec to the I/O servers.

While this I/O system is in some sense meager for a cluster of this size, we will see that its performance is not the most significant bottleneck for this application.

First we describe a series of simple tests designed to give us an indication of the maximum performance we could expect from the PVFS file system under ideal conditions, using each of the three system software interfaces. Following our examination of the system software layers, we discuss the FLASH I/O benchmark. We include an in-depth examination of the major performance bottleneck encountered and a discussion of how this problem was corrected.

4.1 Write Performance through Interface Layers

In Figure 4 we examine the performance of concurrent write access through the various interface layers. The first of these applications uses the PVFS library calls directly to perform I/O operations. The second uses MPI-IO calls through the ROMIO library, which in turn map to PVFS library calls. The final test application uses the HDF5 interface library, which in turn maps the operations into MPI-IO calls. In all cases the application creates a new file, synchronizes, writes concurrently to the file, synchronizes, closes the file, and exits. Figure 5 shows the write pattern; in every case compute processes concurrently write single contiguous data blocks to a data file. The amount of data written by each node was chosen to be 7 Mbytes,

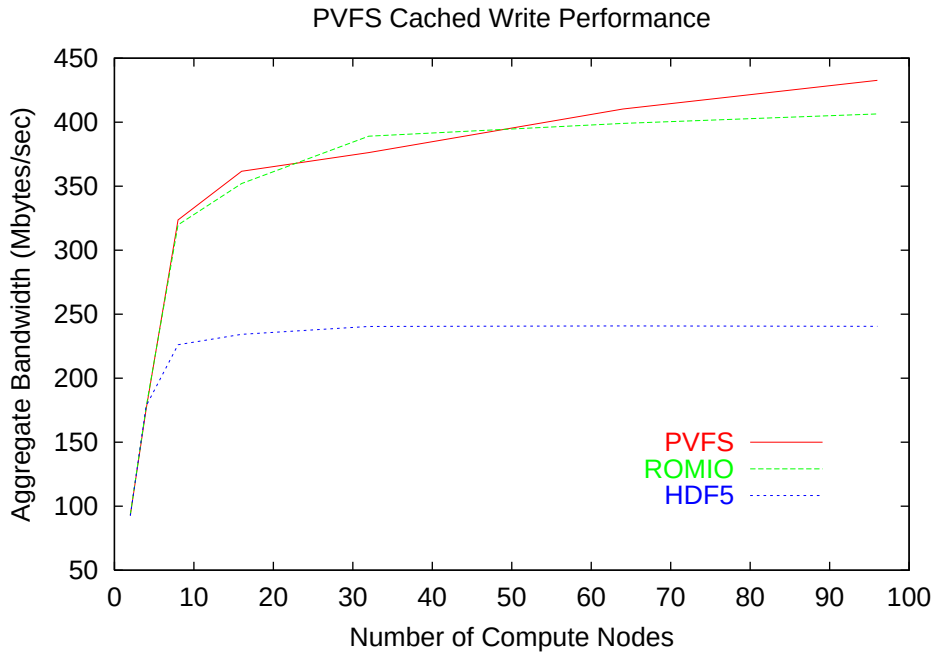

Figure 4: Comparison of I/O write performance through interfaces

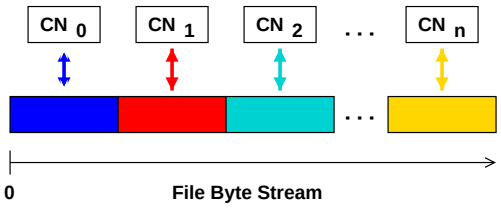

Figure 5: Access pattern for performance testing

Table 2: Initial FLASH I/O Benchmark Performance

	Checkpoint			Plotfile (no corners)			Plotfile (corners)		
# of procs	Size (MBytes)	Time (sec)	MB/sec	Size	Time	MB/sec	Size	Time	MB/sec
64	486.2	73.6	6.6	40.7	12.6	3.23	57.9	16.0	3.6

which is the approximate amount of data written by a node during the checkpoint process. This helps give us an upper bound on performance of the system during this phase of the test application.

We see that performance quickly ramps up to approximately 325 Mbytes/sec in the PVFS and ROMIO cases. After this the aggregate increases slowly, appearing to peak out at around 400-420 Mbytes/sec for both interfaces. With this access pattern there appears to be no appreciable overhead for using ROMIO rather than PVFS. The aggregate performance seen here is higher than the aggregate performance of the physical disks on the servers. This indicates that file system caching is having a beneficial effect on the I/O servers. Thus these numbers are not a "true" measure of sustained I/O throughput, but rather an indicator of the performance an application would see if accessing PVFS in a similar manner. In the case of HDF5 writes we see significantly lower peak performance, peaking at approximately 240 Mbytes/sec. This can be attributed to misalignment of accesses within the HDF5 file. Since HDF5 stores metadata inside the file, the actual file data does not necessarily reside on appropriate boundaries. This results in a higher variance in access times between individual processes, and since our test uses the worst-case result in determining collective performance, we see a significant performance drop.

We know that the FLASH I/O benchmark writes out approximately the same amount of data, but it does so with multiple writes per processor (one per variable, plus additional application metadata). Thus we would expect the FLASH performance to fall below these HDF5 performance values.

4.2 FLASH I/O Benchmark Performance

As discussed previously, the FLASH I/O benchmark uses the parallel HDF5 interface for writing out checkpoint and plotfile data. This interface in turn uses the ROMIO MPI-IO implementation to write data to a PVFS file system.

The performance of packages such as HDF5 can be architecture and application dependent because it is easy to overlook peculiarities of specific systems or applications when designing a portable package. For example, it is known that the use of collective MPI-IO operations under the parallel HDF5 interface is beneficial only in some situations. We found an example of an application-dependent inefficiency when we first ran the FLASH I/O benchmark. Table 2 shows the performance; aggregate bandwidth falls far below our expectations, reaching less than 3% of the previously measured peak write performance!

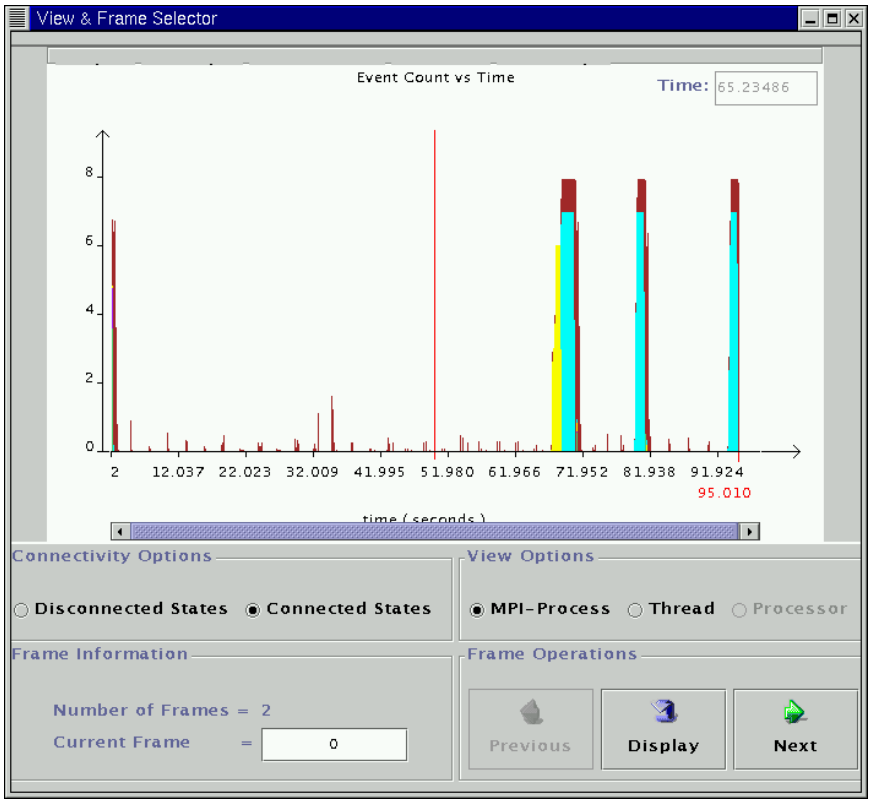

Figure 6: Jumpshot preview of FLASH I/O benchmark

4.2.1 Tracking Down the Performance Bottleneck

We have chosen to describe in detail the process of tracking down our performance problems, emphasizing the use of profiling and visualization of process execution. We feel that the results of this profiling are unexpected enough to warrant this discussion and highlight the benefits of these types of tools.

We decided to examine the application performance in more detail using the Jumpshot visualization tool [19] and the MPE logging components from the MPICH package [8]. Jumpshot provides two main views of logs: a preview and a state view. In the preview, Jumpshot shows a count of the number of processes inside instrumented states during the course of application execution. In the state view, Jumpshot shows a time line for each process, with a single line on a black field indicating the process was not in an instrumented region, and colored boxes with white borders indicating that the process was in a known state. While a tool such as the `gprof` execution profiler might have provided enough information to come to the same conclusions, it is incapable of accurately showing the patterns of calls that are immediately obvious from our visualizations.

By default MPE logs all MPI calls including the MPI-IO calls (in the latest development version). What we would expect is that the majority of the time all processes would be inside an MPI-IO call in the FLASH I/O benchmark, since the program is not performing any real computations. For an eight-node run, this would be shown in the preview with solid bars across the preview reaching up near the eight process mark

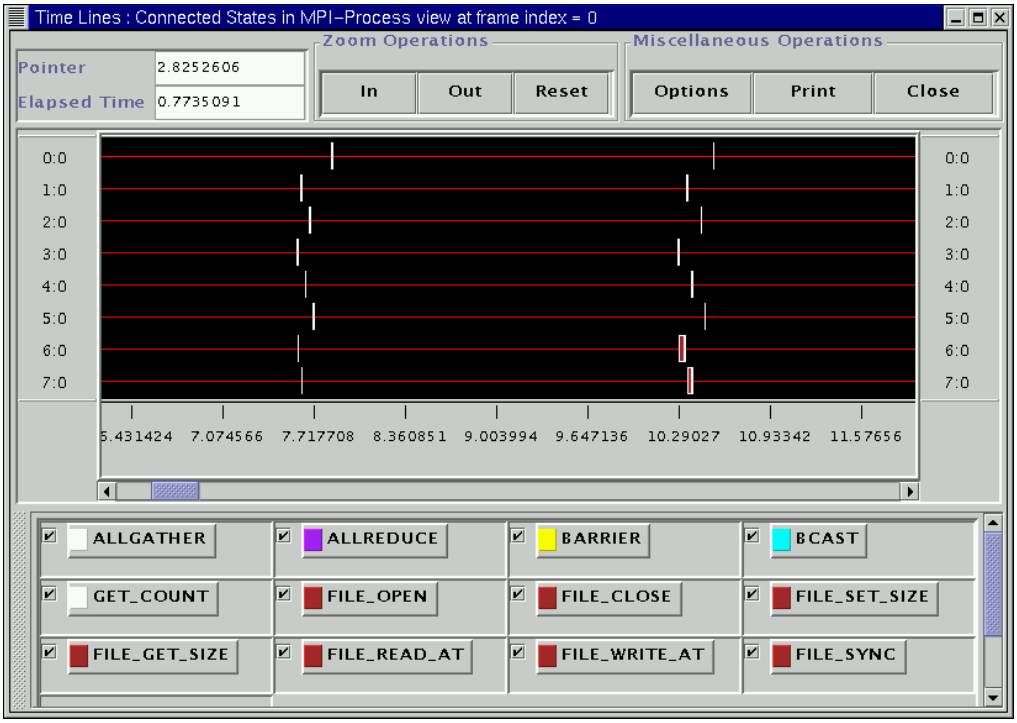

Figure 7: Trace of FLASH I/O benchmark – MPI calls

indicating that all eight processes were in a logged state. However, what we actually see from the preview (Figure 6) of an eight-node run is entirely different. We do see three regions of intense MPI activity late in execution that correspond to the periods at the end of the three I/O phases. The code spends a great deal of time outside of the MPI calls during the I/O phases, however, and this unaccounted-for time is what is hindering performance. These regions are the ones that are mostly empty, with small dark bars representing `MPI_File_write_at()` calls.

In Figure 7 we see the state view of the region around seven seconds into the eight-node run; during this time the application is writing the checkpoint file. In this time line view the boxes are our `MPI_File_write_at()` calls; in some cases they are so narrow that the actual dark (red) color of the box is not noticeable. The application is spending approximately 0.04 seconds in the write operation with gaps of around 2.7 seconds between writes during which nothing is logged (i.e., the application was not in an MPI call). We can immediately rule out the PVFS and ROMIO components as the source of this performance degradation based on this information. To further support this claim, we instrumented the FLASH benchmark to log all calls to `H5Dwrite()` by adding MPE logging calls to the FLASH code. This quick instrumentation showed us that nearly all of our lost time was spent in H5Dwrite() calls, as seen in Figure 8.

We next instrumented the HDF5 code in the same manner, focusing on the internals of `HDF5write()`. We eventually determined that there were two operations, a gather of variables from memory and a scatter into the file, that were consuming the majority of this lost time. Figure 9 shows the log viewer zoomed in at approximately seven seconds again, this time viewing these gather (white) and scatter (gray/cyan) times. For each `MPI_File_write_at()` (barely visible following the box representing the scatter), which consumes

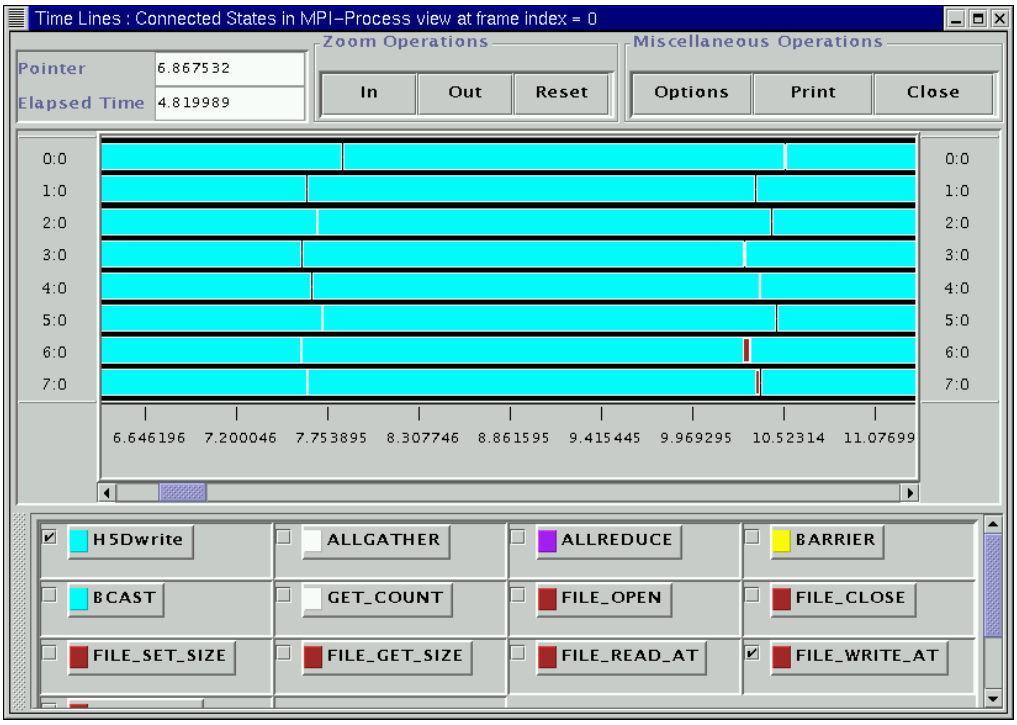

Figure 8: Trace of FLASH I/O benchmark – time in `H5Dwrite()`

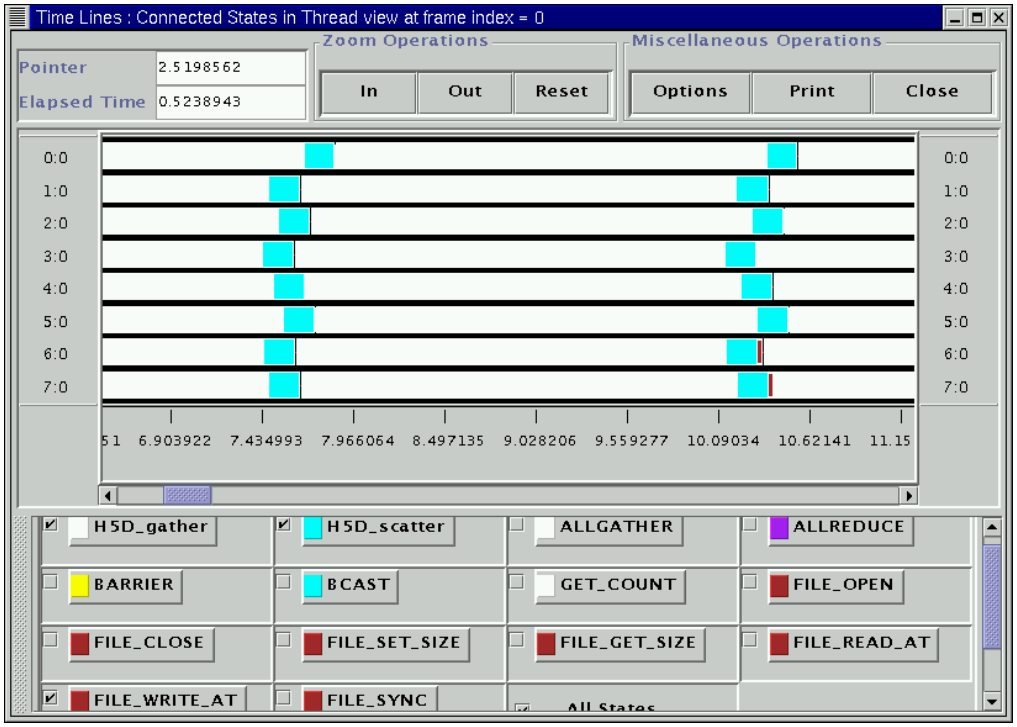

Figure 9: Trace of FLASH I/O benchmark – time in gather and scatter

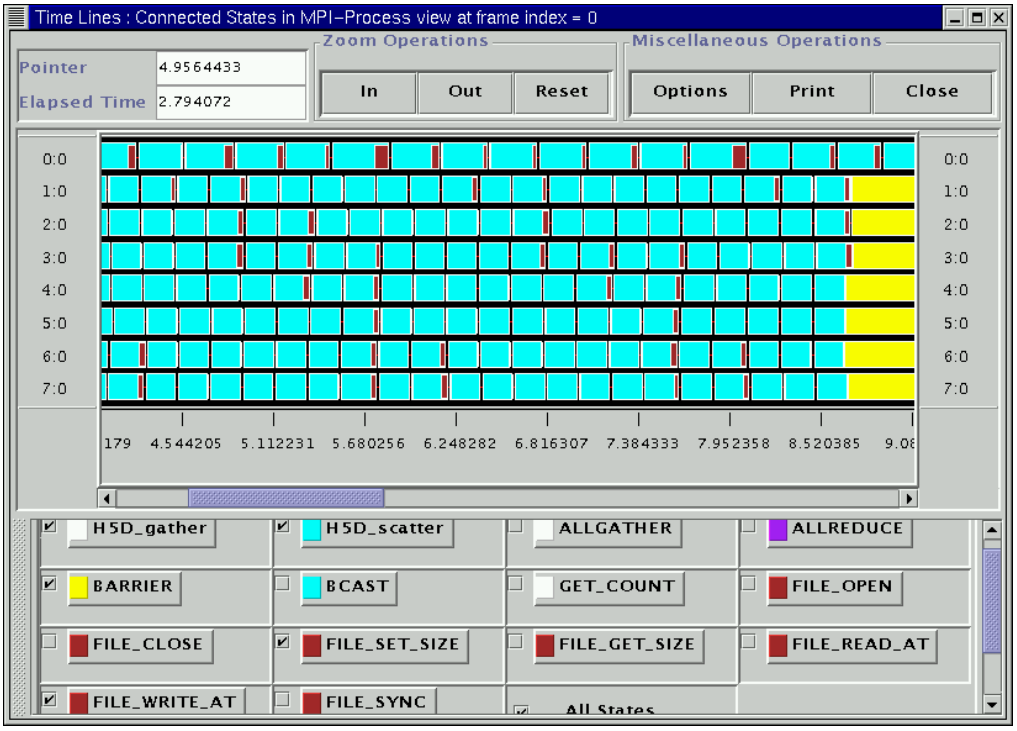

Figure 10: Trace of FLASH I/O benchmark – gather avoided

around 0.04 seconds, each process spends approximately 2.5 seconds in the gather operation and 0.2 seconds in the scatter code, including the write operation.

4.2.2 Regaining Performance

We determined that the gather function could be avoided by eliminating the use of a "hyperslab" description on the memory (a feature of HDF5 exploited by the application), instead packing the variables into a contiguous buffer within the benchmark itself. This requires an additional buffer for use in packing and a small amount of additional code in the application to perform the packing operation prior to the write. This modification reduces the memory available for storing blocks by a small amount and is somewhat inconvenient from the programming standpoint. However, this change also results in our checkpoint time on 64 nodes dropping from 73.6 to 11.6 seconds, making it well worth the small additional cost. Figure 10 shows why; the gather operation now takes an negligable amount of time (the large white regions seen in Figure 9 are no longer present), leaving only the scatter (gray/cyan) and `MPI_File_write_at()` (dark gray/red) taking significant amounts of time during the write phase. In fact, the parallel portion of the checkpoint write, which used to continue until roughly 65 seconds into the run (as seen in Figure 6), instead completes at around 9 seconds into the run, and all but process 0 have entered a barrier (light gray/yellow on right side). The same procedure was used to improve the plotfile storage as well. Table 3 summarizes the performance of the modified benchmark. While our performance reaches only approximately 23% of peak HDF5 performance, these numbers compare favorably with runs on other platforms [6].

Table 3: Modified FLASH I/O Benchmark Performance

	Checkpoint			Plotfile (no corners)			Plotfile (corners)		
# of procs	Size (MBytes)	Time (sec)	MB/sec	Size	Time	MB/sec	Size	Time	MB/sec
32	243.1	9.34	26.0	20.4	3.08	6.63	29.0	3.41	8.51
64	486.2	11.7	41.7	40.7	4.20	9.70	57.9	4.43	13.1
128	972.4	21.9	44.4	81.4	6.25	13.0	115.8	6.55	17.7
192	1458.6	26.8	54.4	122.1	8.36	14.6	173.7	8.80	19.7
256	1945.1	33.9	57.4	162.9	10.65	15.3	231.5	11.37	20.4

We still see that only a small portion of the application time is spent actually writing data. The scatter operations as a whole take around five times as long as the writes in the eight process case. This plays a large role in our reaching only approximately 18 % of our ideal performance. Unfortunately, the use of hyperslabs in the scatter operation is buried in the HDF5 implementation and is not trivial to avoid. Figure 10 also shows us an additional inefficiency; while the writes for processes 1–7 are almost in lock-step with the gather time eliminated, writes for process 0 are out of alignment with respect to the others. This sort of information, which is obvious when viewed in this manner and can be an important indicator of performance problems, would not have been accessible from `gprof` or similar tools that provide only statistical information.

Why does the gather (and to a lesser extent the scatter) take so long? It turns out that HDF5 implements its hyperslab processing operations recursively. Just as implementations of datatype processing have been recently drawing attention in the realm of message passing as a key area for performance improvement [18, 9], hyperslab processing in HDF5 deserves similar attention.

5 Conclusions and Future Work

In this work we have presented a collection of system software components and shown how these components can be used together to provide high-performance I/O for an important class of applications. Using a benchmark derived from one such application, we examined the performance of the components using a freely available instrumentation library and found that in our environment some inefficiencies were apparent. Working with developers at all levels, we identified and overcame the worst of these inefficiencies through software modifications in order to reach a more acceptable level of functionality.

This work showcases the importance of solid interfaces to system software components. Interfaces such as UNIX I/O, MPI-IO, MPI, and HDF5 and protocols such as TCP allow us to create components that can be used as building blocks for solutions such as the one shown here and that will operate on top of numerous hardware configurations.

Moreover, this work points out the importance of efficient datatype processing, not only in message passing systems, but also in high-level I/O interfaces. Having uncovered this opportunity for improvement in HDF5, we intend next to examine related datatype processing work in the context of hyperslab processing in order to design a more efficient mechanism for both the gather and the scatter operations in HDF5.

Availability

All the software used in this work is freely available in source form. Source code, compiled binaries, documentation, and mailing-list information for PVFS are available from the PVFS Web site:
`http://www.parl.clemson.edu/pvfs/`

The HDF Web site provides documentation, presentations, and source for the HDF package:
`http://hdf.ncsa.uiuc.edu`

Information and source code for the ROMIO MPI-IO implementation are available at the ROMIO Web site:
`http://www.mcs.anl.gov/romio/`

The FLASH I/O benchmark Web site provides source code and performance results from a number of platforms:
`http://flash.uchicago.edu/~zingale/flash_benchmark_io/`

Documentation and source code for the MPE library, which is part of the MPICH project, is available at the MPICH Web site:
`http://www.mcs.anl.gov/mpi/mpich/`

Acknowledgements

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the Department of Energy under Grant No. B341495 to the Center for Astrophysical Thermonuclear Flashes.

References

[1] Tim Bray. Bonnie file system benchmark. `http://www.textuality.com/bonnie/`.

[2] A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, G. Henry, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Truran, and M. Zingale. High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In *Proceedings of SC2000*, November 2000.

[3] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[4] Matthew Cettei, Walter Ligon, and Robert Ross. Support for parallel out of core applications on beowulf workstations. In *Proceedings of the 1998 IEEE Aerospace Conference*, March 1998.

[5] Chiba City, the Argonne scalable cluster. `http://www.mcs.anl.gov/chiba/`.

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

[7] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

[8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[9] William Gropp, Ewing Lusk, and Deborah Swider. Improving the performance of MPI derived datatypes. In *MPIDC*, 1999.

[10] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[11] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[12] 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, 2000.

[13] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997. http://www.mpi-forum.org/docs/docs.html.

[14] ReiserFS. http://www.reiserfs.org.

[15] Hakan Taki and Gil Utard. MPI-IO on a parallel file system for cluster of workstations. In *Proceedings of the First IEEE International Workshop on Cluster Computing*, 1999.

[16] Test TCP. ftp://ftp.arl.mil/pub/ttcp/.

[17] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[18] Jesper Larsson Traff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In *PVM/MPI 1999*, pages 109–116, 1999.

[19] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.