A User-Friendly Approach for Tuning Parallel File Operations

Robert McLay, Doug James, Si Liu, John Cazes, and William Barth
Texas Advanced Computing Center
The University of Texas at Austin
Austin, TX 78712
Email: mclay@tacc.utexas.edu

Abstract—The Lustre file system provides high aggregated I/O bandwidth and is in widespread use throughout the HPC community. Here we report on work (1) developing a model for understanding collective parallel MPI write operations on Lustre, and (2) producing a library that optimizes parallel write performance in a user-friendly way. We note that a system’s default stripe count is rarely a good choice for parallel I/O, and that performance depends on a delicate balance between the number of stripes and the actual (not requested) number of collective writers. Unfortunate combinations of these parameters may degrade performance considerably. For the programmer, however, it’s all about the stripe count: an informed choice of this single parameter allows MPI to assign writers in a way that achieves near-optimal performance. We offer recommendations for those who wish to tune performance manually and describe the easy-to-use T3PIO library that manages the tuning automatically.

I. INTRODUCTION

As large-scale computer systems continue to support larger jobs and more users, their parallel file systems must remain stable and scale with the increased load. The user community must adapt their applications to scale as well. One common approach, serial I/O from one task on behalf of all other tasks, does not scale efficiently. A second popular approach, simultaneous serial I/O from all tasks, each writing to its own file, may create serious performance problems on Lustre-based file systems. The learning curve for implementing better approaches can be steep, however, and users transitioning to parallel I/O techniques such as those provided by MPI-IO are often disappointed with the results. Lustre in particular is quite sensitive to inappropriate tuning parameters. In addition, a single user’s application can adversely impact I/O performance for other users and affect the stability of the file system itself.

This report is the first in a series on two related initiatives designed to help improve the state of the practice from the perspective of the user: a systematic study of the factors affecting collective write performance on Lustre-based systems, and a user-friendly software tool that dramatically simplifies the problem of tuning production codes. The software tool, which we call the T3PIO library, implements in a user-friendly way the recommendations and heuristics described in this paper.

In this section we motivate what follows with the results of two experiments that hint at the potential value of performance tuning (especially user-friendly auto-tuning). Sections II and III set the stage with some general technical background and a brief survey of the relevant literature. Section IV describes the effect of three tuning parameters on collective write performance, and includes a careful treatment of some counter-intuitive, important effects that we believe are not well

Fig. 1. Test code with and without T3PIO auto-tuning (Stampede scratch, direct MPI-IO, nodes=80, tasks=1280, average performance over 10 reps).

Fig. 2. WRF 2.5km CONUS Benchmark with and without T3PIO auto-tuning (Stampede scratch, HDF5, 4.4 GB, 16 tasks per node). Effective rate reflects WRF self-reported write times.
known within the computational research community. Section V describes the mechanisms available for setting the key tuning parameters, including the T3PIO library. Section VI, the heart of the paper, describes our recommended heuristics for optimizing collective write performance. Section VII is a brief look at other factors affecting performance, while section VIII summarizes our major conclusions and plans for the future.

Figures 1 and 2 demonstrate the effectiveness of the auto-tuning capabilities of the T3PIO library. The first figure shows performance as a function of file size for a test code writing files ranging from 1 to 100 GB. The test code uses direct MPI-IO calls to write to a single file a global 3D array decomposed across a three dimensional processor grid, a representative I/O pattern typical of many production codes. Here and elsewhere the reported rate is the net effective throughput, reflecting times spent in all I/O operations including the open and close functions (the latter may consume 30% or more of the total time because it may be involved in ensuring that earlier writes have been completed). Figure 2 shows the effects of using T3PIO in the Weather Research and Forecasting (WRF) Model [26] to tune its parallel HDF5 I/O. The experiment is a standard public benchmark problem (Single domain, 2.5km CONUS benchmark runs on June 4, 2005); see [27] for download and more information. Despite the relatively small 4.4 GB file spread over 4-16 nodes (16 MPI tasks per node), the benefit of the T3PIO library is undeniable.

The performance improvements in both experiments are due almost entirely to T3PIO’s choice of stripe count. This is a key point of this paper: parallel write performance depends heavily on an appropriate choice of this parameter. The default stripe count on a typical Lustre file system (usually 1, 2 or 4) is an appropriate choice for routine and typical file operations, but many computational researchers do not realize that this default is usually not an appropriate choice for collective parallel I/O.

As we discuss below, there is another sense in which one can argue that performance tuning is all about the stripe count. Performance depends on a delicate balance between the number of stripes and the actual number of collective writers assigned by the Message Passing Interface (MPI) library to manage the write on behalf of other MPI tasks. Unfortunate combinations of these parameters may degrade performance considerably. Moreover, the programmer does not have direct control over the number of writers. We demonstrate below, however, that an informed choice of stripe count allows the MPI stack to assign writers in a way that achieves near optimal performance. Our primary goal is to help the computational research community understand the relevant concepts and exploit them to optimize their parallel file operations.

II. BACKGROUND

This study focuses on Lustre [15], a parallel object-based file system in widespread use, developed to support large-scale operations on modern supercomputers. Lustre attains high I/O performance by simultaneously striping a single file across multiple Object Storage Targets (OSTs) that manage the system’s spinning disks. It is sometimes helpful to oversimplify a bit and imagine that the number of stripes associated with a file represents the number of disks over which the system stores the file. Reality is actually more complicated: each OST is typically a Redundant Array of Independent Disks (RAID) working together in a way that acts like a single spinning drive. In any case, stripe size, usually measured in MegaBytes (MB), is the size of one block of data written to a single OST.

MPI-IO, a part of the MPI2 standard [11], is the specification that describes the interface a programmer uses to accomplish parallel file operations using functions in the MPI library. Programmers can do so by calling MPI-IO functions directly, or through a library (e.g. HDF5 [6] or one of the various versions of NetCDF [25]). MPI-IO file operations can be either collective or independent.1 In collective write operations (the focus of this paper), all MPI tasks in the communicator participate in the call and coordinate their activity with one another. Designated tasks (formally called aggregators, but also called writers) collect data from tasks that need to write to disk, and issue the write request on behalf of all the tasks. A common and generally effective strategy is using a single writer for each node. It is worth observing that the role of each such writer is much broader than simply managing the write activity of MPI tasks on its own node.

ROMIO [1], developed by Argonne National Lab and included in MPI libraries based on MPICH [12] and OpenMPI [14], is the most common implementation of MPI-IO. MPI-IO uses a data structure called the MPI_Info object to specify stripe count, stripe size, and other parameters; the ROMIO interface to Lustre passes this information to the file system itself.Achieving optimal performance requires balancing the I/O bandwidth of the OSTs with the bandwidth available to the designated writers that deliver data to the file system on behalf of all MPI tasks.

We ran most experiments reported here on TACC’s Stampede [18] system, a Linux cluster composed of 6400 dual-socket nodes connected via FDR Infiniband. Stampede’s scratch file system has 348 OSTs. We also ran tests on Stampede’s recently retired work file system (48 OSTs), as well as TACC’s new Global Shared File System (GSFS). The GSFS, managed as a system we call Stockyard [16], has 672 OSTs and connects via LINET routers to multiple clusters; it currently serves as the work file system for both Stampede and TACC’s new Maverick visualization cluster [19]. We conducted additional tests on TACC’s Lonestar [17] system, a Linux cluster with 1888 dual socket nodes connected via QDR Infiniband. Lonestar’s scratch file system has 90 OSTs. In all cases, we varied the experimental conditions to give us confidence in our results, running parameter studies with a variety of file sizes (20 MB to 1 TB), multiple compiler versions (Intel 13 on Stampede, Intel 12 on Lonestar), MPI stacks (mvapich2/1.9a2 and IMPI 4.1 on Stampede, mvapich2/1.6 on Lonestar), and Lustre versions (2.4.1 on Stampede, 1.8.6 on Lonestar). We also varied the programming language (C++ and Fortran 90), data structures (3D arrays vs. 1D arrays of structures), and libraries (HDF5 vs. direct calls to MPI-IO). All experiments involved collective writing to a single file.

1Here “independent” specifically refers to a type of communication involving calls to MPI-IO functions. In particular, we distinguish between independent MPI-IO parallel file operations and file activity that does not involve the MPI-IO library (e.g. multiple MPI tasks each writing separate POSIX output files using direct, essentially serial calls).
III. RELEVANT WORK IN THE LITERATURE

The performance model, recommendations, and library described here represent a synthesis of our own experiments, inspection of the ROMIO source code, and the cumulative experience of the larger community as reported in the literature. We highlight here the references that we have found most helpful in planning our experiments, understanding our results, and characterizing our recommendations.

There has been a great amount of research focusing on developing and evaluating strategies to optimize I/O performance on parallel file systems. Much of this work involves tuning the middleware libraries, the MPI-IO libraries, the parallel file systems, or a combination of these components.

For collective write operations, one of the most widespread techniques is two-phase collective I/O [3], which is now an integral part of the ROMIO library [1]. More details about the design and development of this approach appear in [22], [23], and the references therein.

This approach has been extended and optimized. Data sieving [21], [23] reduces the number of I/O requests to the file system by loading contiguous chunks of data to temporary buffers instead of reading several separate pieces. Local file view [20] and split writing [28] optimize I/O performance by writing to separate subfiles in parallel and combining them after the fact. Two-level striping [13] and hierarchical striping [28] reduce overlapping by introducing extra levels of striping and reorganizing the communication. Client-side file caching reduces the amount of data transfer between application processes and I/O devices [8], [9]. Dickens and Logan [4] is particularly important for those wishing to understand the technical details behind the “chasms” phenomenon that we discuss below. They describe an important new communication pattern that is also now part of ROMIO.

Several other related research studies evaluating I/O performance of various applications and platforms also enlighten our research. Behzad et al. [2] report on an approach to autotuning that is decidedly different from our own. Others have engaged in performance studies, especially on Cray systems, that have led to a maturing body of knowledge regarding best practices. We have found [5], [7], and [29] particularly useful.

IV. FACTORS AFFECTING PERFORMANCE

There are several factors that control the write performance of an application on a target architecture. Because practitioners will typically choose the number of nodes and MPI tasks, the compiler, and MPI stack for reasons other than I/O performance, we treat these factors as givens rather than opportunities for I/O tuning. It is also important to remember that parallel file systems are shared resources: jobs running on the system compete with each other for access to the same components of Lustre.

Beyond these givens, however, there are at least three “knobs” available for tuning collective I/O performance on Lustre-based systems: stripe count, stripe size, and the number of writers (aggregators). We begin by observing the general effect of each of these parameters on collective write performance.

Figure 3 shows the typical effect of stripe count on write performance for a 20 GB HDF5 file.2 For files large enough to justify parallel I/O, performance generally increases with stripe count, approaching a peak at or near $s = N$ (where $s$ is the stripe count and $N$ is the number of nodes), and reaching a peak no later than a small multiple of $N$ (generally $4N$ on Stampede and $6N$ on Lonestar for modest node counts). Peak performance typically occurs sooner for small files and later for larger ones. For very small files (see section VII), peak may occur well before $s = N$. Clearly the most notable characteristic of this graph, however, is the presence of “chasms” at unfortunate choices of $s$. Fortunately, these chasms are completely deterministic, predictable, and avoidable. Understanding when and why they occur (and how to avoid them) requires that we first address the effect of writers on performance.

Writers

Figure 4 shows the effect of the requested number of writers on performance for a 50 GB file. Again, the behavior is typical, and performance generally increases with the number of writers. The stair steps in the graph, however, hint at a key point that is easy to miss: the programmer does not have direct control over the actual number of aggregators assigned to execute a given MPI collective write. The programmer can use any of several mechanisms to specify a quantity called $cb\_nodes$ (“cb” stands for “collective buffer”), but the ROMIO documentation makes clear that this quantity represents the maximum number of writers ROMIO can assign. The quantity $cb\_nodes$ defaults to $N$, the number of nodes. It is one of several upper bounds that ROMIO applies before determining the actual number of writers. A second important bound is the maximum number of writers per node, which defaults to 1. Thus, there are in place two defaults that limit the maximum number of writers to $N$.

---

2Here and elsewhere, the figure depicts the maximum, average, and minimum values of results from at least five identical experiments.
However, there is more going on. Once ROMIO applies these limits and determines an upper bound (call it \( w_{\text{max}} \)) on the actual number of writers, the code then sets the actual \( w \) to be the largest integer less than or equal to \( w_{\text{max}} \) that is a multiple or divisor of the stripe count \( s \) (the details are in the source file ADIOT_LUSTRE_Get_striping_info.c, a part of the ROMIO layer that interfaces directly with the Lustre file system). This is why we see stair steps in the plot.

This calculation is subtle and important enough that some examples are in order. Consider, for example, the 40-node experiment shown in Figure 3. If the stripe count is \( s = 80 \), ROMIO will set \( w \) equal to the largest divisor of 80 less than or equal to the node count, so the result is \( w = 40 \). If \( s = 100 \), the largest such divisor is \( w = 25 \). If \( s = 43 \), or any other prime number greater than \( N = 40 \), then ROMIO will set \( w = 1 \), a most unfortunate choice.

Exploring the Chasms

We can now see when and how chasms occur. The deepest chasms occur when the stripe count \( s \) is a prime number greater than the node count \( N \); in such cases, \( w = 1 \). Less serious chasms occur when \( s > N \) and \( s \) has only small divisors below \( N \) (e.g. small multiples of prime numbers). Chasms cannot occur when \( s \leq N \), and do not occur when \( s \) is a multiple of \( N \). Together, these facts make it easy to choose \( w \) in a way that gives ROMIO the freedom to select \( w \) to optimize a collective write.

Stripe Size

Figure 5 shows the effect of stripe size on collective write performance. The result suggest strongly that stripe size has little effect on performance on our test systems. On experiments involving files as large as 1 TB, we have observed little benefit to deviating from the system default stripe size. For very small files (below 1 GB), we have sometimes observed a small benefit from slightly larger stripe sizes (up to about 16 MB), but the improvements are modest at best (and the evidence inconclusive).

V. SETTING THE KNOBS

Manual Control

Section IV makes clear that collective write performance depends critically on setting stripe count to an optimal value. While the results we have outlined suggest it is less important to exercise control over (max) writers and stripe size, there may be circumstances under which one may wish to tune these parameters as well. What mechanisms are available to do so?

Two of the parameters, stripe count and stripe size, are properties of Lustre files. The striping parameters on a new file depend on the properties of the directory in which the file is created. One can set a directory’s stripe count and stripe size from the command line using the \texttt{lfs setstripe} command. This affects all files created in the directory after issuing the command. It does not affect files that already exist. Unless you are moving a file across file systems, the \texttt{mv} command has no effect on striping. Copying the file with \texttt{cp} can change the file’s striping. The \texttt{lfs getstripe} command gives striping information on existing files.

MPI allows the programmer/analyst to pass values (or at least hints) to ROMIO for all three parameters through the \texttt{MPI_Info} object, which can store key-value pairs associated with these and several other parameters. One can do so in source code, of course. Another mechanism is a “hints” configuration file: by default, ROMIO reads the contents of /etc/romio-hints; the environment variable \texttt{ROMIO_HINTS} can specify another filename/location. See [24] for more information.

The hints file provides full control and advanced capabilities to administrators and individual users with specialized needs, and the options available go well beyond those discussed in this paper. One can exercise explicit control over the hostnames (nodes) on which ROMIO can assign writers. This is a capability we sometimes use at TACC to prevent the MPI stack from assigning writers to tasks running on Stampede’s Intel Xeon Phi Many Integrated Core (MIC) coprocessors. It is also easy to override ROMIO defaults to allow more than
one writer per node, but we have not yet observed any benefit from doing so.

T3PIO

T3PIO [10], a library written by the first author, provides a user-friendly means for C, C++, and Fortran 90 programmers to tune Lustre for collective write operations based on MPI-IO (including HDF5 [6] as well as NetCDF4, NetCDF, and PnetCDF). T3PIO controls parallel I/O performance by tuning stripe count, stripe size, and max writers; it does so by defining entries in the MPI_Info object.

Programmers use the library by making a single function call to (1) specify explicitly the desired stripe count, stripe size, and/or number of writers, or (2) direct the library to set these parameters automatically to near-optimal settings. One can also pass to the library optional information such as file size. The call to auto-tune from Fortran (C/C++ is similar) can be as simple as this:

```fortran
integer :: info ! info object
call t3pio(MPI_COMM_WORLD, info, "./"
```

Programmers then take advantage of the specified settings by using the MPI_Info object in their write-related function calls (typically replacing the MPI_INFO_NULL object with the newly-defined MPI_Info object).

Programmers can also elect to use T3PIO to control the tuning manually. The function argument list provides one mechanism for doing so, but this is not necessary: T3PIO also allows users to control the tuning using environment variables.

VI. Optimizing Performance

The results of our studies suggest near-optimal tuning heuristics that are simpler than we originally expected. We are assuming, of course, that the goal is to maximize collective MPI-IO write performance subject to the givens (node count, file size, architecture, etc.), and that the file is large enough to justify parallel file operations (more on this in section VII). At the very least, these heuristics will avoid poor combinations of stripe count vs. MPI writers, and can serve as an excellent starting point for additional application-specific tuning.

Resource Limits

One final consideration remains: practical limits on the number of stripes available to any single file. Clearly, the total number of OSTs on the file system is a constraint. A second limitation is built into the Lustre software: versions 2.1 and below limit the number of stripes for any given file to 160 (this limit is 2000 in versions 2.2 and above). Another potential consideration is good citizenship: some practitioners advocate restricting stripe counts to one-half or two-thirds of the total available OSTs to reduce contention and adverse impact on other users’ jobs. Thus, for any given write operation, there is a practical limit $s_{\text{max}}$ to the number of stripes available. This limit resembles

$$s_{\text{max}} = \min(c s_{\text{total}}, L),$$

where $s_{\text{total}}$ is the total number of OSTs available on the file system, $L$ is the Lustre-imposed limit (160 or 2000), and $c$ is the fraction of $s_{\text{total}}$ that one is willing to use for any single file. There are certainly other reasonable alternative ways to think about $s_{\text{max}}$.

Tuning Heuristics

The tuning heuristics now become remarkably easy to summarize:

- Choose an appropriate stripe count less than or equal to $s_{\text{max}}$. If possible, set $s$ equal to a small multiple of $N$. If this is not possible (because $s_{\text{max}} \leq N$), set $s = s_{\text{max}}$.
  - A good rule of thumb is to try (in order), candidate values of $s$ like 6$N$, 5$N$, 4$N$, 3$N$, 2$N$, or 1$N$, stopping at the first value that is less than or equal to $s_{\text{max}}$.
  - T3PIO currently begins its search at 4$N$, but we are currently examining more sophisticated approaches for selecting the threshold multiple at which to begin the search.
- Let ROMIO manage writers. Selecting $s$ properly avoids any issues with chasms, and in fact gives ROMIO the freedom to set the number of writers to a near-optimal value.
- Leave stripe size at the system default. There is no a priori reason to expect changes to the default stripe size to result in performance improvements.

The auto-tuning model in T3PIO as currently implemented sets the three knobs to values that reflect the spirit of these heuristics. We will continue to modify T3PIO to reflect the latest results of our performance studies.

VII. Other Factors Affecting Performance

The parameter space governing Lustre performance is huge. We do not pretend that we have completely characterized Lustre performance, or that it is even possible to do so. We have, however, conducted additional experiments to begin to gain some insight into other factors affecting performance. We briefly summarize our current observations here.

File Size

Write performance (effective rate) is a strong function of file size: all else being equal, writes to larger files tend to achieve higher bandwidth. Above a certain threshold file size, however, the optimal choice of tuning parameters appears to be surprisingly insensitive to file size. For small files, the heuristics we describe in this paper do not apply.

Figure 6 depicts an extreme example: writing a 20 MB HDF5 file from 10 nodes (when $s = 10$ this amounts to writing a single 2 MB stripe from each of the 10 assigned writers). The graph is qualitatively different from that in figure 3; the file is clearly too small to benefit from striping and/or multiple writers. The bump at $s = 11$ is more or less the opposite of the chasm effect: at $s = 11$, ROMIO assigns a single writer, and performance across 11 stripes is similar to the $s = 1$ performance. Performance variation across the 10 runs is higher here than in experiments involving larger files; this is typical of other small write operations.
Figure 7 shows performance vs. stripe count for a slightly larger case: a 1 GB file spread across 10 nodes. Here we are seeing behavior that is beginning to resemble canonical performance: the write operation does benefit from striping. Peak performance above $s = 6$ is uneven; average performance is more or less level at larger stripe counts, but we again observe a bump in performance at $s = 11$, when ROMIO assigns a single writer. One key observation: despite its small size, the heuristic $s = N = 10$ is likely to produce near-optimal performance.

We cannot yet characterize precisely the threshold below which the performance model and heuristics are no longer valid. The threshold undoubtedly depends on (at least) absolute file size, size per node, and size per actual writer. We can report, however, that the heuristics produce near-optimal performance even when the file size per node is as small as a few hundred megabytes.

T3PIO accepts file size as an optional input argument, but it does not yet generate tuning parameters that depend on file size. We anticipate introducing threshold tests in near-term revisions, and more sophisticated dependencies as our performance model matures.

Architecture

We have conducted a wide range of experiments on four Lustre file systems on both the Stampede and Lonestar Linux clusters. While there are other systems to test and experiments to conduct, we report that performance characteristics are consistent across these systems, and the heuristics appear to be valid. We have not yet studied Cray systems; see [5] for a careful performance study on Cray-based systems.

Figure 8 illustrates the similarities nicely. In this 64 GB Lonestar scratch experiment, performance as a function of stripe count closely resembles the typical behavior shown in figure 3. In this experiment, performance increases steadily with stripe count; here peak performance does not occur until $s = 6N$ (a bit higher than is typical on Stampede). Note the chasm phenomenon described in section IV. Despite the qualitatively similar features, however, this figure does have its own distinct signature, inviting a more careful look at the impact of the characteristics of both the file system and the cluster itself.

To begin understanding those differences, we took a step back and looked at the performance of a single aggregator writing to multiple stripes. The results surprised us: the single-writer performance signatures on Stampede scratch (Figure 9), GSFS on Stampede (figure 10), and Lonestar scratch (Figure 11) are dramatically different. We are not yet prepared to explain these differences. We do, however, expect that analyzing single-writer behavior will lead to additional insights into parallel write performance.

One tantalizing pair of experiments has us both intrigued and puzzled. Figure 12 shows a 50 GB HDF5 write from 8 nodes to Stampede scratch. As expected, performance is good at $s = N = 8$. However performance is reliably, repeatably 20-25% better at $s = 9$, a stripe count that we would normally expect to observe for smaller files.
avoid. At this stripe count, ROMIO assigns 3 writers; clearly 3 writers and 3 stripes per writer is a better choice here than 8 writers and 1 stripe per writer. At first glance, this appears to be consistent with the 3 stripes per writer bias on Stampede scratch suggested by figure 9. We repeated the experiment on the GSFS, expecting (based on figure 10) no such 3-to-1 bias. On GSFS, however, we observed the same effect: $s = 9$ was measurably superior to $s = N = 8$. The one-writer GSFS signature does not explain this; clearly there is more work to do.

**Independent MPI-IO**

The pros and cons of collective vs. independent MPI-IO are outside the scope of this paper, and we have not yet studied MPI-IO independent file operations in a serious way. We have, however, run a few experiments to explore an obvious question: how do the heuristics described in this paper affect the performance of a parallel MPI-IO write performed in independent (rather than collective) mode?

Early indications suggest that the heuristics produce near-optimal performance even for MPI-IO independent writes to a single file. Preliminary results as shown in Figure 13 are typical. In this experiment performance increases with stripe count, peaks at $s = 5N$, then slowly diminishes. There is one important new characteristic, however: because the concept of collective writers is meaningless for independent MPI-IO, there are no chasms, and no need to avoid unfortunate choices of stripe count.

**VIII. SUMMARY**

Appropriate tuning can substantially improve I/O performance for collective MPI-IO write operations. Our experiments to date suggest simple heuristics that make it easy to determine near-optimal settings, and the T3PIO library provides a user-friendly mechanism that makes it possible to achieve those settings (or specify your own) with a single function call. The key tuning parameter is stripe count: an appropriate choice of stripe count gives ROMIO the freedom to assign collective writers in a near-optimal way.
There is much more to be done, however, and the parameter space is vast. We are looking to extend our work in a number of important directions: performance on other architectures; factors affecting file operations on very large files; and performance of other file operations (e.g., reads and parallel independent writes). One especially important issue is the stripe count at which peak performance occurs: can we characterize file systems in a way that allows us to predict with confidence the best multiple of node count to target when selecting $s$? We also plan to study the difficult but important question of contention: when and how do one user’s file operations adversely affect other users, and how can we mitigate the problem?

We are especially excited about our work enhancing the functionality and usability of the T3PIO library. We are nearing completion on a particularly promising project that will be the topic of an upcoming paper: a proof-of-concept demonstration that embeds T3PIO directly inside ROMIO itself. Once complete, this experimental version of ROMIO will allow users to opt-in to T3PIO using an environment variable, and exploit its full capabilities without the need to modify (or even recompile) even a single line of source code.

Our ultimate goal is simple to describe but challenging to achieve: we hope to help improve the state of the practice for all users so we can all be more effective stewards of the precious computational resources at our disposal.

REFERENCES

[19] ——, “Visualization Resources (Maverick),” tacc.utexas.edu/resources/visualization.