Measuring I/O Performance of Lustre and the Temporary File System for Tradespace Applications on HPC Systems

Leonard Kosta
Department of EE & CS
US Military Academy
West Point, NY
leonard.kosta@usma.edu

Harrison Hunter
Department of E & CE
Mississippi State University
Mississippi State, MS
rhh132@msstate.edu

Glover George
Information Technology Lab
US Army ERDC
Vicksburg, MS
glover.e.george@erdc.dren.mil

Andrew Strelzoff
Information Technology Lab
US Army ERDC
Vicksburg, MS
andrew.strelzoff@erdc.dren.mil

Suzanne J. Matthews*
Department of EE & CS
US Military Academy
West Point, NY
suzanne.matthews@usma.edu

ABSTRACT

Tradespace analysis is an important part of design and modeling simulations in the military, in which thousands of design parameters are made available for detecting potential areas for performance enhancement in expensive equipment such as combat and transport vehicles. In pursuit of larger, more accurate tradespaces, engineers port existing serial design and evaluation codes onto ERDC's supercomputers. This results in a significant number of producer-consumer, file-mediated codes running simultaneously on Lustre, the distributed file system underlying our HPC systems. In this paper, we compare the performance of Lustre to Linux `tmpfs`, a shared memory file system that was designed for interprocess data transfer within a single node. Our results suggest that when conducting many file I/O operations in close succession, `tmpfs` offers tremendous speedup over Lustre-controlled disk storage. For tradespace applications, our results lend credence to a strategy of scheduling and organizing disparate codes in such a way to leverage `tmpfs`. We believe our results will help guide engineers at other national labs and scientists without HPC expertise to more easily port their I/O heavy serial codes to HPC systems.

CCS Concepts

•**Software and its engineering** → *Distributed systems organizing principles;*

Keywords

Tradespace Design, I/O Performance, Lustre, Temporary File System

*Corresponding Author

ACM SE '17 April 13-15, Kennesaw, GA USA

ACM ISBN 978-1-4503-5024-2/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3077286.3077326

1. INTRODUCTION

One key challenge in modeling and simulation in the military is the orchestration of large combinations of pre-existing software packages into a single executable pipeline capable of generating a large "tradespace" of design options, with thousands of design and performance parameters. A tradespace in this context is all system inputs, intermediate values, and outputs organized as a large table of potential design options. A central goal is to port serial tradespace codes to HPC systems; however, there are numerous challenges involved. In cases where the source code is available, rewriting complex serial code to efficiently use MPI is often infeasible, given requirements deadlines. In cases where the serial source code is not available, lab developers must resort to workarounds to enable code to execute in parallel.

The challenges that developers face with porting tradespace codes at the US Army Engineer Research & Development Center (ERDC) are not unique. Labs across the country work with conceptual design teams who have traditionally performed tradespaceing using Excel or similar workflows. HPC and distributed-shared memory/MPI expertise is generally non-existent in these early stages of the platform acquisition workflow. In addition, the linking of codes written in one language (for example, C# to FORTRAN) that need to interact with others often necessitates file-mediated communication. Where this file I/O dominates over the computational time, any effort to accelerate this operation directly impacts the overall turnaround time for simulations.

For a tradespace consisting of many design variables and millions of iterations, tens of millions of small, temporary files have to be written. Reading and writing from disk is roughly 70 times slower than reading and writing data from RAM. This speed penalty is expected and reasonable for long-running "process time dominated" software such as high fidelity physics packages, but is problematic when running large parameter sweeps or design of experiments massively parallel on large clusters. In this case, multiple runs of the composite application executing on thousands of processes simultaneously flood the distributed file system (Lustre) causing dominating delays in data read-write performance.

The Lustre file system [7] is a parallel, distributed file system designed to accommodate concurrent I/O operations of large magnitude. It is used in 70 of the Top 100 supercomputers on the Top500 (top500.org) list worldwide [3]. In Lustre, all file metadata is handled by a dedicated metadata server (MDS). Until version 2.5 (released in late 2013), Lustre only allowed for a single MDS per file system. While it is now possible to use multiple MDS servers, many machines still use a single MDS due to the cost of adding and maintaining additional servers. Consequently, programs running on these machines can potentially encounter a major bottleneck when it comes to opening lots of files [1].

The alternative is to offload as many of these small I/O operations to a faster, temporary location such as a compute node's shared memory partition, such as `tmpfs`. Splitting the design parameter input space of a tradespace into multiple chunks, and running the single threaded software on each of these chunks yet on thousands of cores is potentially a more feasible option given time and expertise constraints. In this case, we posit that small I/O operations should be written to `tmpfs`, rather than Lustre, in order to create more efficient tradespace applications.

To test our hypothesis, we ran our experiments on Topaz [2], one of ERDC's machines. Its local Lustre file system has a single operating MDS. Our tests were performed in a live environment with other users contributing to I/O load. Our results suggest that writing to `tmpfs` is up to $30,000$ times faster than Lustre. This suggests that for programs that use files as a medium of inter-process communication, scheduling interdependent, disparate serial codes to use `tmpfs` instead of Lustre will yield significant performance improvement. In a broader sense, our research provides useful data for HPC novices seeking to find ways to avoid the Lustre I/O bottleneck in their programs.

2. RELATED WORK

Many researchers identify the Lustre bottleneck, though few present end-user options for avoiding it. Alam *et. al.* [1] discuss the potential bottleneck caused by Lustre's metadata server. They lay out hardware and networking experiments to tune the metadata server's performance. Larkin and Minga [5] acknowledge the bottleneck and explain that the metadata server can only handle a finite number of operations before failing. They suggest adding more metadata servers to accommodate this load, but do not identify the point at which this may become necessary nor provide any other potential solutions. Spitz and Koehler [8] investigate the various causes for Lustre failure. They enumerate issues in both hardware and software, including Lustre's metadata performance. Ihara *et. al.* [4] use benchmarking tools to test methods to improve the speed of Lustre's MDS. They specifically studied the impact of directory structure and file size have on metadata operations. These works provide insight into the causes of Lustre's scalability troubles, but do not suggest alternatives to using Lustre for file-mediated communication.

This paper presents early research towards quantifying the slowdown generated by floods of simultaneous writes on Lustre. For ERDC and the Department of Defense, our research enables us to increase the efficiency of large-scale tradespace applications with relatively little overhead. For HPC novices trying to quickly port serial code to an HPC system, our research highlights potential performance issues of their own applications when using Lustre-based HPC systems.

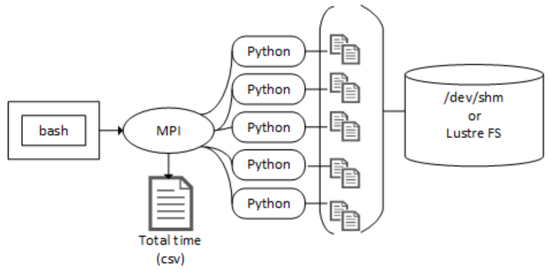


Figure 1: Overview of execution of test suite

3. EXPERIMENTAL METHODS

Figure 1 depicts an overview of our process. To simulate an application flooding Lustre with simultaneous small writes, we write a Python script that creates an arbitrary large number of small files. This enables us to conduct a virtual "stress test" on the Lustre metadata server, and identify any patterns in Lustre's behavior under a load. For our experiments, we created eight-byte files consisting of the single digit 0. Choosing a file size this small allows us to put the primary load on Lustre's metadata server, enabling us to observe exactly how well the metadata server performs without data generation being a confounding variable. Writing at least one byte as data also helps us determine if the file creation was a success.

To simulate a parallel program, the script employs the `mpi4py` Python module and Topaz's native SGI implementation of MPI. The script creates files in parallel while running on different cores and nodes. We also create a bash script which spins off multiple runs of our Python program. Every time the script finishes, it appends its total run time to a CSV file. This total time is the maximum of the individual process run times.

We conduct our experiments on a subset of Topaz [2], a $3,456$-node SGI cluster hosted by ERDC. Each node consists of 36 Intel Xeon E5-2699v3 Haswell cores at 2.3 GHz each. Each node has 117 GB of main memory. Topaz has two "work" file systems of 6.2 PB running Lustre 2.5, each with a single operating MDS. All experiments were conducted on the "live" system, with other users simultaneously using the cluster. This enables us to study the performance on these file systems under normal load. We also run our scripts on the local `tmpfs` of these nodes.

4. RESULTS AND DISCUSSION

We test Lustre and `tmpfs` on various parameter sets. Our initial "small" jobs write up to $1,000$ files on each process. In our "medium" set of experiments, we max out at about $10,000$ per process. We lastly conduct a final "large" run consisting of up to $30,000$ files per process. For each experiment we varied the number of requested processes. For each specification, we request a commensurate number of cores, so that each process runs on a separate core.

4.1 Performance of Lustre

Figure 2 and 3 show our results of our experimentation with Lustre with a "small" number $(1 \ldots 1,000)$ of writes. This specification is run with 36, 72, and 108 cores, resulting in a maximum of approximately $100,000$ file write requests. During the first run (Figure 2), Lustre scales fairly well with

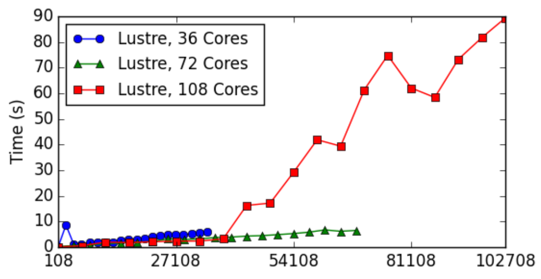


Figure 2: Lustre performance when conducting a small number of file write operations, run 1.

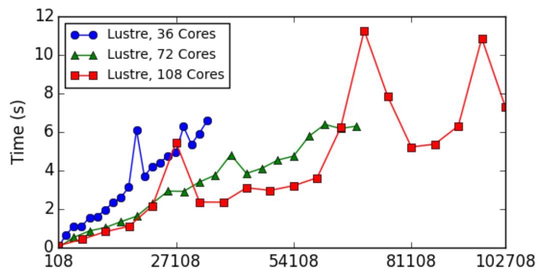
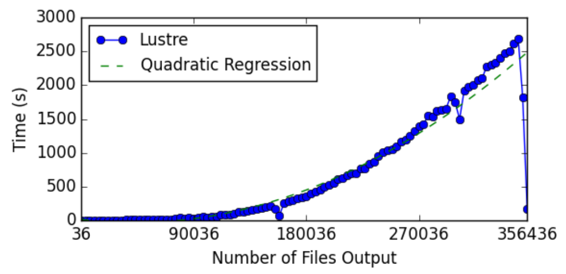


Figure 3: Lustre performance when conducting a small number of file write operations, run 2.



[t]

Figure 4: Lustre performance when conducting a medium number of file write operations.

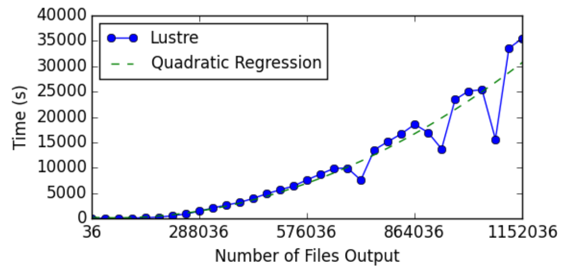


Figure 5: Lustre performance when conducting a large number of file write operations.

36 and 72 cores, but experiences significant latency with 108 cores. Repeated executions at this scale display high variance both between and within runs (Figure 3). For example, for the second run, Lustre took less than 12 seconds to write on the order of $100,000$ files, as compared to almost 90 seconds in the first. Yet, Lustre still experiences periods of latency albeit at a smaller scale. We posit this difference is due to activity of other users on the HPC system. Considering the relatively small number of files written, the tests were susceptible to interference from other metadata operations on the system. All operations requiring metadata access pass through the metadata server [9]. Increased traffic at the metadata server could have delayed the file write operations induced by our test suite [5].

When we transition to our "medium" scale experiments, it becomes impractical to submit jobs utilizing more than 36 cores to the HPC, due to job queuing and time restrictions. Therefore, only 36 cores were requested. In this iteration, the number of files produced per process was increased to a maximum of $10,000$, resulting in a peak of $360,000$ file write operations sent to the metadata server. Figure 4 illustrates our results. We note the high correlation ($R^2 = 0.8932$) between the data and a curve of best-fit suggests that on Lustre the time taken to write files to disk storage varies roughly quadratically with the number of file write operations conducted. While a request to write thousands of files takes seconds, a request to write hundreds of thousands of files can take an hour or longer.

We note that toward the end of the run, I/O request time declines significantly. Examining the output files from these points showed that some were empty rather than filled with the expected 0 digit. We believe that this was caused by an error at the metadata server, or from exceeding the maximum number of pending metadata operations allowed by the metadata server [8]. We do not believe that this data was influenced significantly by the activity of other users on the system due to the small variance and high correlation with a curve of best fit.

Finally, we vary the number of output files per process from $1 \ldots 32,000$ for "large" experiments. Once again, we request 36 cores. This results in a maximum of over $1,000,000$ file write requests being sent to the metadata server, which took almost ten hours to process (Figure 5). Again, the general trend of a quadratic relationship between the independent and dependent variables held ($R^2 = 0.9243$). As in the "medium" experiments, there were points where the time taken to complete the file write operations sharply declined. Once again, some of these write operations fail; however, the failures did not occur at the same number of file write operations supplied.

4.2 Performance of Temporary File System

As with Lustre, the performance of writing to `tmpfs` was assessed multiple times on "small", "medium", and "large" scales. Figure 6 shows our results. Writing to the temporary file system is very efficient. The difference between small and medium scale tests is negligible. At no point during the climb to $1,000,000$ file write requests did writing to `tmpfs` take more than 5 seconds to finish execution. Servicing these file I/O requests require approximately linear time ($R^2 = 0.9964$), and does not result in data loss. We believe the perceived linear growth is due to file I/O operations being treated as memory operations in `tmpfs` [6]. Figure 7 shows the speedup of writing to `tmpfs` over Lustre. Speedup varies from 100 on small numbers of files to $30,000$ on extremely large number of file writes.

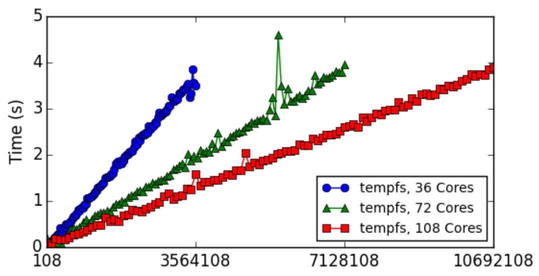


Figure 6: `tmpfs` performance when conducting a very large number of file write operations.

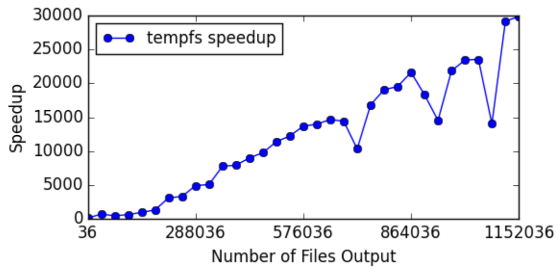


Figure 7: Speedup of tmpfs over Lustre on large data sets.

.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we experimentally demonstrate the performance discrepancy of large numbers of write operations on Lustre compared to the Linux Temporary File System (`tmpfs`). We note that the relative speedup provided by `tmpfs` continues to increase commensurate to the number of write operations. In contrast, Lustre slows down and occasionally fails to execute file writes properly as the number of write requests grow large.

There are several implications of our results. For large-scale applications that perform many intermediate write operations (such as is required for tradespaces), designing code that writes intermediates to `tmpfs` instead of Lustre can lead to significant speedups. While it may be obvious to experienced HPC developers that writing to shared memory will be faster than a networked file system, it is not readily obvious to novice HPC users trying to quickly port interdependent, file-mediated serial code to an HPC cluster. This is especially true for scientists from other disciplines who do not possess in-house computer scientists with HPC expertise. Furthermore, HPC laboratories operating on tight deadlines often do not have the time or resources rewrite serial code to use shared memory effectively, or may not have access to the source code at all.

Our results led ERDC to re-schedule tradespaces codes to better leverage `tmpfs`. Using shell scripting and other parallel wrapping techniques, interdependent serial codes are scheduled together on a single node, enabling maximum use of `tmpfs`. Thus, multiple nodes have large numbers of interdependent serial codes using `tmpfs`, with Lustre used to mediate file communication between nodes. Due to the limited memory of `tmpfs`, using Lustre is not completely avoidable. However, organizing tradespace applications in this way significantly reduces the impact of the Lustre MDS bottleneck, and simultaneously improves the performance of large-scale tradespace applications. This enables developers to focus on ongoing development, and near-seamlessly port their code to HPC without concerning themselves with advanced parallel programming techniques. This in turns lowers the boundaries to HPC. We stress our results are not only relevant to ERDC, but to conceptual design research and development staff throughout the DoD.

Future work will examine the effect of splitting large numbers of writes over many directories. We also plan to explore whether the same patterns found for write-intensive jobs hold true for read-intensive jobs.

6. ACKNOWLEDGMENTS

This research was made possible by the U.S. Army Engineer Research and Development Center's Information Technology Laboratories and the DoD High Performance Computing Modernization Program. The first two authors are undergraduate student researchers. The last three authors are faculty advisors at ERDC and USMA respectively. The opinions expressed in this work are solely of the authors, and do not necessarily reflect those of the US Army, ERDC, the US Military Academy, or the Department of Defense.

7. REFERENCES

[1] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel I/O and the metadata wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, pages 13–18, New York, NY, USA, 2011. ACM.

[2] DoD Supercomputing Resource Center. High performance computing systems. Internet Website, last accessed, July 2016. https://www.erdc.hpc.mil/hardware/index.html.

[3] J. Franklin. Lustre 2.8.0 released at lug 2016 and declared general availability. *OpenSFS Administration*, 3 2016.

[4] S. Ihara. Lustre metadata fundamental benchmark and performance. Powerpoint Presentation, last accessed, July 2016. http://www.eofs.eu/fileadmin/lad2014/slides/03_Shuichi_Ihara_Lustre_Metadata_LAD14.pdf.

[5] J. Larkin and A. Minga. Optimizing I/O performance for lustre. Powerpoint Presentation, last accessed, June 2016. https://erdc.hpc.mil/docs/Tips/OptimizingIOPerformanceForLustre.pdf.

[6] R. Love. *Linux Systems Programming: Mapping Files into Memory*, chapter 4. O'Reilly Publishing Inc., 2007.

[7] Seagate Technology LLC. Lustre. Internet Website, last accessed, 2016. http://lustre.org/.

[8] C. Spitz and A. Koehler. Tips and tricks for diagnosing lustre problems on cray systems. In *Proceedings of the Cray User Group 2011*. Cray Inc, 2011.

[9] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. Understanding lustre filesystem internals. Technical Report ORNL/TM-2009/117, National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee, April 2009.