Storage Challenges at Los Alamos National Lab

John Bent
john.bent@emc.com

Gary Grider
ggrider@lanl.gov

Brett Kettering
brettk@lanl.gov

Adam Manzanares
adamm@lanl.gov

Meghan McClelland
meghan@lanl.gov

Aaron Torres
agtorre@lanl.gov

Alfred Torrez
atorrez@lanl.gov

Abstract

There yet exist no truly parallel file systems. Those that make the claim fall short when it comes to providing adequate concurrent write performance at large scale. This limitation causes large usability headaches in HPC.

Users need two major capabilities missing from current parallel file systems. One, they need low latency interactivity. Two, they need high bandwidth for large parallel IO; this capability must be resistant to IO patterns and should not require tuning. There are no existing parallel file systems which provide these features. Frighteningly, exascale renders these features even less attainable from currently available parallel file systems. Fortunately, there is a path forward.

1 Introduction

High-performance computing (HPC) requires a tremendous amount of storage bandwidth. As computational scientists push for ever more computational capability, system designers accommadate them with increasingly powerful supercomputers. The challenge of the last few decades has been that the performance of individual components such as processors and hard drives as remained relatively flat. Thus, building more power supercomputers requires that they be built with increasing numbers of components. Problematically, the mean time to failure (*MTTF*) of individual components has over remained relatively flat over time. Thus, the larger the system, the more frequent the failures.

Traditionally, failures have been dealt with by periodically saving computational state onto persistent storage and then recovering from this state following any failure (*checkpoint-restart*). The utilization of systems is then measured using *goodput* which is the percentage of computer time that is spent actually making progress towards the completion of the job. The goal of system designers is therefore to maximize goodput in the face of random failures using an optimal frequency of checkpointing.

978-1-4673-1747-4/12/$31.00 2012 IEEE

Determining checkpointing frequency should be straight-forward: measure MTTF, measure amount of data to be checkpointed, measure available storage bandwidth, compute checkpoint time, and plug it into a simple formula [3]. However, measuring available storage bandwidth is not as straightforward as one would hope. Ideally, parallel file systems could achieve some consistent percentage of the hardware capabilities; for example, a reasonable goal for a parallel file system using disk drives for storage would be to achieve 70% of the aggregate disk bandwidth. If this were the case, then a system designer could simply purchase the necessary amount of storage hardware to gain sufficient performance to minimize checkpoint time and maximize system goodput. However, there exist no currently available parallel file systems that can provide any such performance level consistently.

2 Challenges

Unfortunately, although there are some that can, there are many IO patterns that *cannot* achieve any consistent percentage of the storage capability. Instead, these IO patterns achieve a consistently low performance such that their percentage of hardware capability diminishes as more hardware is added! For example, refer to Figures 1a, 1b, and 1c, which show that writing to a shared file, *N-1*, achieves consistently poor performance across the three major parallel file systems whereas the bandwidth of writing to unique files, *N-N*, scales as desired with the size of the job. The flat lines for the N-1 workloads actually show that there is no amount of storage hardware that can be purchased: regardless of size, the bandwidths remain flat. This is because the hardware is not at fault; the performance flaw is within the parallel file systems which cannot incur massively concurrent writes and maintain performance. The challenge is due to maintaining data consistency which typically requires a serialization of writes.

An obvious solution to this problem is for all users to always perform N-N file IO in which every process writes to a unique file. This approach does not come without

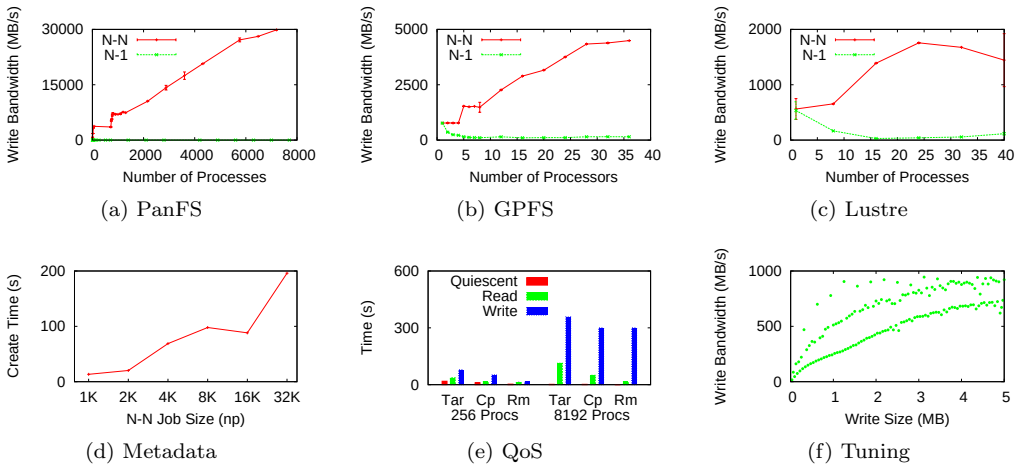

Figure 1: **Usability Challenges.** *These graphs address the key usability challenges facing today's users of HPC storage systems. The top three graphs demonstrate the large discrepancy between achievable bandwidth and scalability using N-N and N-1 checkpoint patterns on three of the major HPC parallel file systems. The bottom left graph shows the challenge of metadata creation at large job size, the bottom middle shows how the notion of interactivity is a cruel joke when small file operations are contending with large jobs performing parallel IO, and finally, the bottom right graph shows the reliance on magic numbers that plagues current parallel file systems.*

trade-offs however. One is a performance limitation at scale and the other is a reduction in usability as will be discussed later in Section 3.

The system problem is the massive workload caused by by large numbers of concurrent file creates when each process opens a new file. Essentially this causes the same exact problem on parallel file systems as does writing in an N-1 pattern: concurrent writes perform poorly. In this case, the concurrent writing is done to a shared directory object. These directory writes are handled by a metadata server; no current production HPC parallel file system supports distributed metadata servers. As such, large numbers of directory writes are essentially serialized at a single metadata server thus causing very large slow-downs during the create phase of an N-N workload as is shown in Figure 1d.

3 Implications for Usability

This causes large usability headaches for LANL users. All of the large computing projects at LANL are well-aware of, and dismayed by, these limitations. All have incurred large opportunity costs to perform their primary jobs by designing around these limitations or paying large performance penalties. Many create archiving and analysis challenges for themselves by avoiding writes to shared objects by having each process in large jobs create unique files. Some have become parallel file system experts and preface parallel IO by doing complicated queries of the parallel file system in order to rearrange their own IO patterns to better match the internal data organization of the parallel file system.

3.1 Tuning

Many users have learned that parallel file systems have various *magic numbers* which correspond to IO sizes that achieve higher performance than other IO sizes. Typically these magic numbers correspond with various object sizes in the parallel file system ranging from a disk block to a full RAID stripe. The difference between poorly performing IO sizes and highly performing IO sizes is shown in Figure 1f which was produced using LBNL's PatternIO benchmark [8]. Also, this graphs seems to merely show that performance increases with IO size, a closer examination shows that there are many small writes that perform better than large writes. In fact, a close examination reveals three distinct curves in this graph: the bottom is IO sizes matching no magic numbers, the middle is for IO sizes in multiples of the object size per storage device, and the upper is for IO sizes in multiples of the full RAID stripe size across multiple storage devices.

The implication of this graphs is that those users who can discover magic numbers and then use those magic numbers can achieve much higher bandwidth than those users who cannot. Unfortunately, both discovering and exploiting magic numbers is difficult and often intractable. Magic numbers differ not only on different parallel file systems (*e.g.* from PanFS to Lustre) but also on different installations of the same file system. Tragically, there is no simple, single mechanism by which to

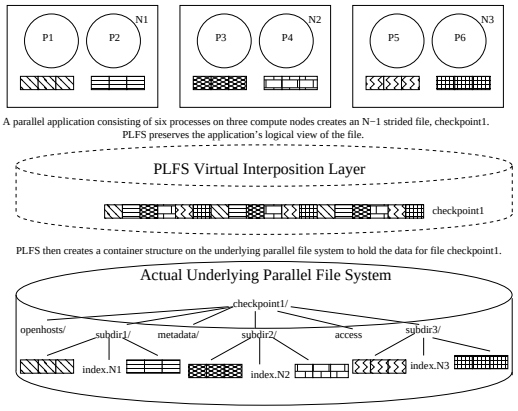

Figure 2: **PLFS Architecture.** *This figure shows how PLFS maintains the user's logical view of a file which physically distributing it into many smaller files on an underlying parallel file system.*

extract magic numbers from a file system.

We have a user at LANL who executes initialization code which first queries *statfs* to determine the file system *f_type* and then, based on which file system is identified, then executes different code for each of the three main parallel file systems to attempt to discover the magic number for that particular installation. Once discovering this value, the user then reconfigures their own, very complicated, IO library to issue IO requests using the newly discovered magic number. Of course, most users would not prefer to jump through such hoops, and frankly, many users should not be trusted with low-level file system information. Not because they lack intelligence but because they lack education; they are computational scientists who should not be expected to become file system experts in order to extract reasonable performance from a parallel file system.

Of course, even if all users could easily discover magic numbers, they could not all easily apply them. For example, many applications do adaptive mesh refinement in which the pieces of the distributed data structures are not uniformly sized: neither in space nor in time. This means that users looking for magic numbers will need some sort of complicated buffering or aggregation. An additional challenge is that magic numbers are not as easy as merely making the individual IO operations be of a particular size; they must also be correctly aligned with the underlying object sizes. So not only must users attempt to size operations correctly, they must also attempt to align them correctly as well. There are other approaches to address this problem such as collective buffering [10] in MPI-IO. As we will show later in Section 4.1, collective buffering is beneficial but is not a complete solution.

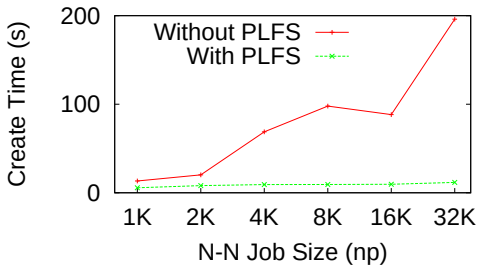

Figure 3: **Addressing Metadata Challenge.** *This graph shows how distributed metadata keeps create rates manageable at large scale.*

3.2 Quality of Service

Finally, although checkpoint-restart is a dominant activity on the storage systems, obviously it is not the only activity. Computational science produces data which must then be explored and analyzed. As the output data is stored on the same storage system which services checkpoint-restart, data exploration and analysis workloads can content with checkpoint-restart workloads. As is seen in Figure 1e, the checkpoint-restart workloads can wreak havoc on interactive operations. In this experiment, the latency of small file operations, such as untarring a set of files, copying that same set of files, and then removing the files, was measured during periods of quiescence and then compared to the latency of those same operations when they were contending with large parallel jobs doing a checkpoint write and a restart read. The most painful latency penalties are seen when the operations contend with a 8192 process job doing a checkpoint write.

4 Path Forward

There are many emerging technologies, ideas, and potential designs that offer hope that these challenges will be addressed in time for the looming exascale era.

4.1 PLFS

Our earlier work in SC09 [2] plays a prominent role in our envisioned exascale IO stack. That work showed how PLFS makes all N-1 workloads achieve the performance of N-N workloads and also how PLFS removes the need for tuning applications to the underlying system (*i.e.* in PLFS, every number is a magic number!). Those results will not be repeated here but suffice it to say that they eliminate the challenges show in Figures 1a, 1b, 1c, and 1f. From a usability perspective, PLFS is an important contribution: in addition to removing the need for IO tuning,

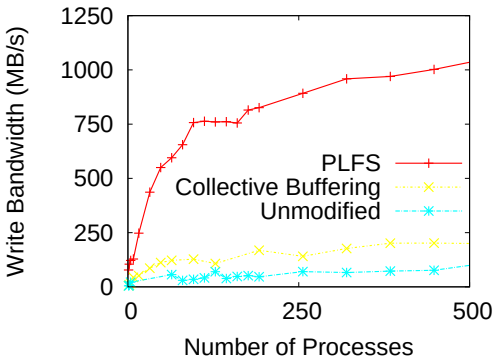

Figure 4: **Collective Buffering.** *This graph shows that collective buffering may not be sufficient for many workloads.*

PLFS is transparently accessible by *unmodified* applications using either POSIX IO or the MPI IO libraries.

Note that collective buffering [10] is another approach to dealing with the thorny problem of magic numbers. Figure 4 shows that, for one particular workload, collective buffering is an improvement over an unmodified approach to IO but underperforms the bandwidth obtainable using PLFS. In fairness, however, we are not collective buffering experts and perhaps collective buffering could be tuned to achieve much higher bandwidth. Ultimately though, our usability goal is to remove file system and parallel IO tuning from the user's purview.

Figure 2 shows the architecture of PLFS and how it preserves the user's logical view of a file while physically striping the data into many smaller files on an underlying parallel file system. This effectively turns all large parallel workloads into N-N workloads. Of course, as we saw in Figure 1d, even N-N workloads suffer at very large scale. Additionally, we know that this performance degradation is due to an overloaded metadata server which will destroy interactive latency as we saw in Figure 1e.

Borrowing ideas from GIGA+ [7], PLFS now addresses these challenges as well. Recent versions of PLFS (since 2.0) can stripe the multiple physical files comprising a logical file over multiple file systems. In the case where each file system is served by a different metadata server, this distributes metadata load very effectively as can be seen in Figure 3 which is the same as Figure 1d but with an added line showing how metadata distribution within PLFS can remove metadata bottlenecks. Note that the workload shown was run using an N-N workload. Although PLFS was originally designed for N-1 workloads, this new functionality will allow PLFS to address metadata concerns for all exascale checkpoint workloads.

5 Redesigning the IO Stack

PLFS has proven to be a very effective solution for current IO challenges: it allows all workloads to easily achieve a consistently high percentage of the aggregate hardware capability.

PLFS is not sufficient however to solve the looming exascale IO challenges before us. Recent work [9] shows that the checkpointing challenge is becoming increasingly difficult over time. The checkpoint size in the exascale era is expected to be around 32 PB. To checkpoint this in thirty minutes (a decent rule of thumb) requires 16 TBs of storage bandwidth. Economic modeling shows that current storage designs would require an infeasible 50% of the exascale budget to achieve this performance.

5.1 Burst buffer

We must redesign our hardware stack and then develop new software to use it. Spinning media (*i.e.* disk) by itself is not economically viable in the exascale era as it is priced for capacity but we will need to purchase bandwidth. Additionally, the storage interconnect network would be a large expense. Thus far, we have required an external storage system for two main reasons: one, sharing storage across multiple supercomputers improves usability and helps with economies of scale; two, embedding spinning media on compute nodes decreases their MTTF.

Our proposal is to make use of emerging technologies such as solid state devices, *SSD*. This media is priced for bandwidth and for low latency so the economic modeling shows it is viable for our bandwidth requirements. Additionally, the lack of moving parts is amenable to our failure models and allows us to place these devices within the compute system (*i.e.* not on the other side of the storage network). Unfortunately, being priced for bandwidth means these devices cannot provide the storage *capacity* that we require. We still require our existing disk-based parallel file systems for short-term capacity needs (long-term capacity is served by archival tape systems not otherwise discussed here).

We propose adding these devices as a new layer in our existing storage hierarchy between the main memory of our compute nodes and the spinning disks of our parallel file systems; we call this interposition of SSD a *burst buffer* as they will absorb very fast bursts of data and serve as an intermediate buffer to existing HPC parallel file systems. This is not a new idea and is commonly suggested as a solution to the well-known latency gap between memory and disk. Our proposal however is how to specifically incorporate these burst buffers into the existing HPC storage software stack.

5.2 E Pluribus Unum

Our envisioned software stack incorporates many existing technologies. The SCR [6] software is a perfect candidate for helping schedule the burst buffer traffic and to enable restart from neighboring burst bufers within the compute nodes. However, we envision merging SCR and PLFS to allow users to benefit from PLFS's capability to handle both N-1 and N-N workloads and to allow use by unmodified application.

We have already add PLFS as a storage layer within the MPI IO library. This library has many important IO optimizations in addition to collective buffering described earlier. One such optimization is available using *MPI_File_set_view*. This is an extremely nice feature from a usability perspective. This is clear when we consider what computational scientists are doing: they stripe a multi-dimensional data structure representing some physical space across a set of distributed processors. Dealing with these distributed multi-dimensional data structures is complicated enough without even considering how to serialize them into and out of persistent storage. *MPI_File_set_view* lesses this serialization burden; by merely describing their distribution, the user then transfers the specific serialization work to the MPI IO library.

Note that other data formatting libraries such as HDF [1], Parallel netCDF [4], SCR , and ADIOS [5] provide similar functionality and have proven very popular as they remove computer science burdens from computational scientists. These data formatting libraries are the clear path forward to improve usability of HPC storage. However, they will not work in their current form on burst buffer architectures. We envision adding our integrated PLFS-SCR storage management system as a storage layer within these data formatting libraries just as we have done within the MPI IO library. A key advantage of a tight integration between PLFS-SCR and these data formatting libraries is that semantic information about the data can be passed to the storage system thus enabling semantic retrieval.

5.3 Co-processed data analysis

There are two key features of our proposal that enable *co-processed* data analysis. The first is that the burst buffer architecture embeds storage much more closely to the compute nodes which drastically reduces access latencies for both sequential and random accesses. The second is that because the data has been stored using data formatting libraries, semantic retrieval of data is possible. This means that we can more easily attempt to co-locate processes within the analysis jobs close to the burst buffers containing the desired data. Finally, even when the data is not available on a local burst buffer, we can take advantage of the low-latency interconnect network between the compute nodes to transfer data between burst buffers as needed.

6 Conclusion

In this proposal, we have described how current usability of HPC storage systems is hampered by two main challenges: poor performance for many large jobs, and occasional intolerably slow interactive latency. We have offered PLFS as a solution for these challenges on today's systems.

Finally, we point out the inability of PLFS to address exascale challenges by itself. We then offer a proposal for integrating PLFS with a burst buffer hardware architecture PLFS and a set of other existing software packages as one path towards a usable and feasible exascale storage solution.

References

[1] The HDF Group. http://www.hdfgroup.org/.

[2] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[3] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[4] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. *SC Conference*, 0:39, 2003.

[5] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24, New York, NY, USA, 2008. ACM.

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[7] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. GIGA+: Scalable Directories for Shared File Systems. In *Petascale Data Storage Workshop at SC07*, Reno, Nevada, Nov. 2007.

[8] Rajeev Thakur. Parallel I/O Benchmarks. http://www.mcs.anl.gov/ thakur/pio-benchmarks.html.

[9] B. Schroeder and G. Gibson. A large scale study of failures in high-performance-computing systems. *IEEE Transactions on Dependable and Secure Computing*, 99(1), 5555.

[10] R. Thakur and E. Lusk. Data sieving and collective i/o in romio. In *In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.