Transparent Optimization of Parallel File System I/O via Standard System Tool Enhancement

Paul Z. Kolano
NASA Advanced Supercomputing Division, NASA Ames Research Center
M/S 258-6, Moffett Field, CA 94035 U.S.A.
paul.kolano@nasa.gov

***Abstract*—Standard system tools employed by users on a daily basis do not take full advantage of parallel file system I/O bandwidth and do not understand associated idiosyncrasies such as Lustre striping. This can lead to non-optimal utilization of both the user's time and system resources. This paper describes a set of modifications made to existing tools that increase parallelism and automatically handle striping. These modifications result in significant performance gains in a transparent manner with maximum speedups of 27x, 15x, and 31x for parallelized cp, tar creation, and tar extraction, respectively.**

I. Introduction

In a typical HPC workflow, users invoke a variety of standard system tools to transfer data onto the system and prepare it for processing before analyzing it on large numbers of CPUs and preparing/retrieving the results. User data is typically stored on a parallel file system that is capable of serving large numbers of clients with high aggregate bandwidth. The standard tools used to manipulate the data, however, are not typically written with parallel file systems in mind. They either do not employ enough concurrency to take full advantage of file system bandwidth and/or leave files in a state that leads to inefficient processing during later access by parallel computations. This results in non-optimal utilization of both the user's time and system resources.

This paper presents a set of modifications to standard file manipulation tools that are optimized for parallel file systems. These tools include highly parallel versions of cp and tar that achieve significant speedups on several file systems as well as Lustre-specific versions of bzip2, gzip, rsync, and tar that have embedded knowledge of Lustre striping to ensure files are striped appropriately for efficient parallel access. These tools can be dropped into place over standard versions to transparently optimize I/O whenever the user would normally invoke them resulting in greater productivity and resource utilization. These modifications will be discussed as well as detailed performance numbers on various parallel file systems.

This paper is organized as follows. Section II discusses the addition of stripe-awareness into commonly used tools. Section III describes high performance modifications to the cp and tar utilities. Section IV details related work. Finally, section V presents conclusions and future work.

II. Stripe-Aware System Tools

Parallel file systems such as CXFS [20], GPFS [18], and Lustre [19] utilize parallel striping across large numbers of disks to achieve higher aggregate performance than is possible on a single-disk file system. Unlike CXFS and GPFS, however, Lustre striping must be specified explicitly before a file is first written. The stripe count determines how many Object Storage Targets (OSTs) a file will be divided across, which can significantly impact I/O performance. A greater number of OSTs provides more available bandwidth but also produces greater resource contention during metadata operations. Striping cannot be changed after a file is created without copying the file in its entirety so should be set carefully.

Figure 1 shows the time to perform the four basic metadata operations on 10,000 files while varying the stripe count using an otherwise idle Lustre file system. As can be seen, the impact of striping can be significant even when file contents are not being read or written. The greatest impact moving between 1 and 64 stripes was on the stat operation with a slowdown of 9.8x while the unlink operation was least impacted with a slowdown of 2.9x. The open and create operations slowed down by 8.4x and 5.9x, respectively.

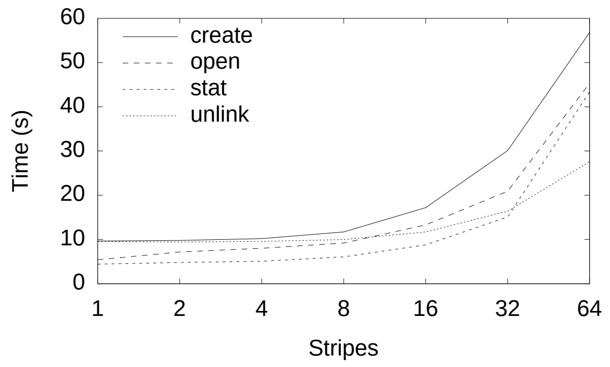

Fig. 1. Run time of Lustre metadata operations on 10k files

To better understand the read/write effects of striping, I/O performance was measured by running parallel instances of the dd command on varying numbers of 3.0 GHz dual quad-core Xeon Harpertown front-ends while reading or writing disjoint blocks of the same file to /dev/null or from /dev/zero, respectively, via direct I/O. File sizes ranged from 1 GB to 64 GB with stripe counts varying from 1 to 64. Figures 2, 3, 4, and 5 show the times to write files of each size and stripe count on 1, 2, 4, and 8 hosts, respectively, while Figures 6, 7, 8, and 9 show the corresponding read times. As can be seen, write times consistently decrease as the stripe count increases for all file sizes and numbers of hosts with more significant drops as the number of hosts increases. In the read case, however, read times only decrease up to a certain number of stripes after which they increase again, but larger stripe counts become more effective as the number of hosts increases.

If the stripe count of the directory containing a newly created file has ben explicitly set, that file will be striped according to the parent directory settings. Otherwise, it will be striped according to the default striping policy. As just shown, however, different file sizes may behave better with different stripe counts. A high default value causes small files to waste space on OSTs and generates an undesirable amount of OST traffic during metadata operations. A low default value results in significantly reduced parallel performance for large files and imbalanced OST utilization. Users can also explicitly specify the stripe count for files and directories. Many users, however, may not know about striping, may not remember to set it, or may not know what the appropriate value should be. Directory striping may be set but the typical mixtures of large and small files mean that users face the same dilemma as in the default case. Even when specified correctly, carefully striped files may revert to the default during manipulation by common system tools.

Since neither approach is optimal, a new approach was developed that utilizes stripe handling embedded into common system tools to stripe files dynamically according to size as users perform normal activities. This allows the default stripe count to be kept low for more common small files while larger files are transparently striped across more OSTs as they are manipulated. The tools selected for modification were based on typical HPC workflows in which a user (1) remotely transfers data to the file system using *scp, sftp, rsync, bbftp, gridftp, etc.*, (2) prepares data for processing using *tar -x, gunzip, bunzip2, unzip, etc.*, (3) processes data on compute resources using arbitrary code, (4) prepares results for remote transfer using *tar -c, gzip, bzip2, zip, etc.*, and (5) remotely retrieves results from the file system. The key to the approach is that by the third step, the input data will have hopefully been striped appropriately in order for processing to achieve the highest I/O performance. The fifth step can be ignored as data is being written to an external file system outside the local organization. Additional tools occur in other common activities such as administrators copying data between file systems to balance utilization using *cp, rsync, etc.*, users copying data between file systems (e.g. home/backup directory to scratch space) using *cp, rsync, etc.*, or users retrieving data from archive systems using *scp, sftp, rsync, bbftp, gridftp, etc.*

Adding stripe-awareness to existing tools is a straightforward process. Since striping needs to be specified at file creation, the first step is to find instances of the open() call that use the O_CREAT flag and determine if the target file is on Lustre using statfs(). If so, the projected size of the file is computed and used to determine the desired stripe count. Finally, the open() call is switched to the Lustre API equivalent, llapi_file_open(), with the given count. Determining the

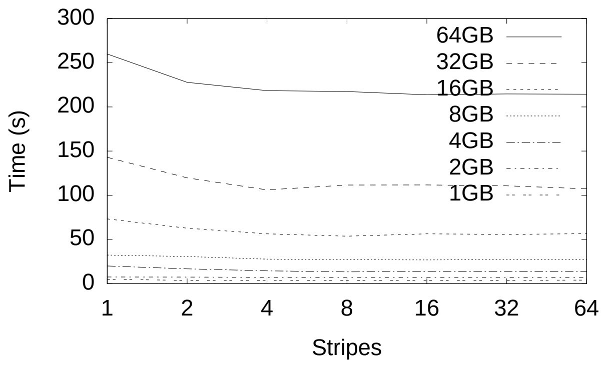

Fig. 2. 1-host dd write

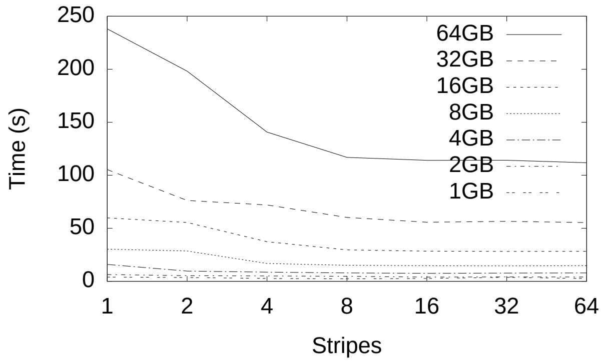

Fig. 3. 2-host dd write

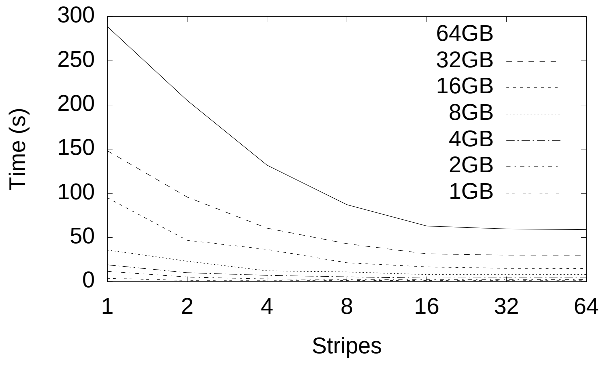

Fig. 4. 4-host dd write

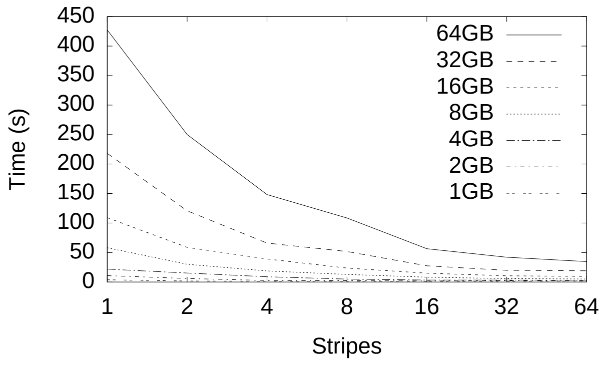

Fig. 5. 8-host dd write

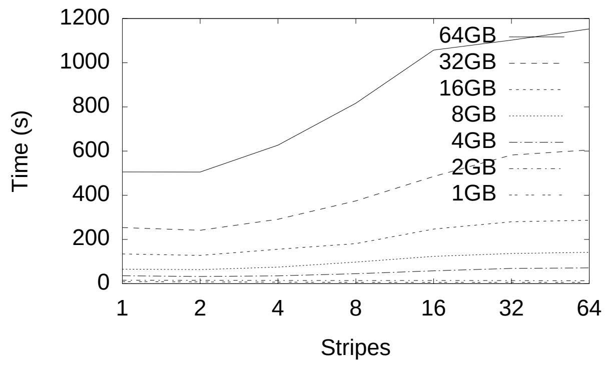

Fig. 6. 1-host dd read

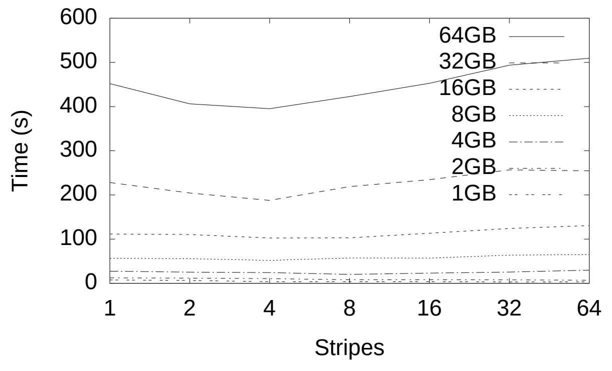

Fig. 7. 2-host dd read

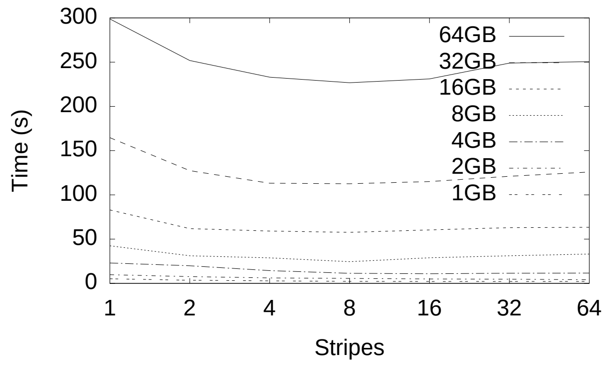

Fig. 8. 4-host dd read

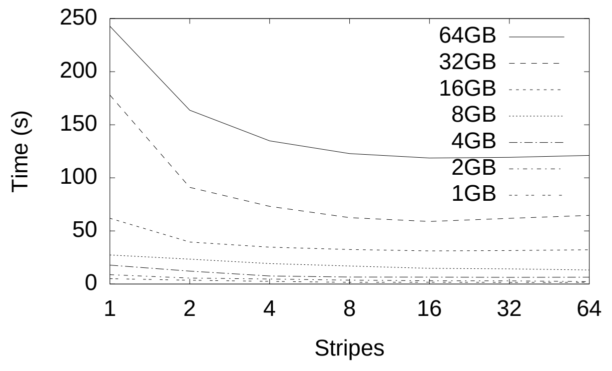

Fig. 9. 8-host dd read

projected size may involve greater complexity for applications such as tar where the target size may be the sum of many individual file sizes.

From the figures, an approximate function for computing optimal stripe count can be extrapolated by examining the points of diminishing or negative returns. Based on this limited data, the optimal write stripe count is projected to be $\frac{size \cdot hosts}{8GB}$ while the optimal read stripe count is projected to be $\frac{size \cdot hosts}{16GB}$. Since the number of hosts that users will run on is unknown, a value must be chosen to represent anticipated usage (e.g. from historical job logs). Armed with these formulas, a set of stripe-aware tools has been implemented to cover the basic workflow/other activities of archival/extraction via tar, compression/decompression via bzip2/bunzip2 and gzip/gunip, local transfer via cp and rsync, and remote transfer via rsync. This set of tools is known as Retools (**Re**striping **Too**ls for **Lus**tre), which will be available as open source shortly [17].

Figures 10, 11, 12, and 13 show the execution times of invoking the regular and stripe-aware versions of bzip2/bunzip2, gzip/gunzip, rsync, and tar, respectively, on single-striped files of varying size with 1 GB files used for the tar cases. To better approximate the stripe count desired for the local environment, the logs of over 900,000 jobs from 2012 were analyzed to find the average number of nodes used, where nodes roughly correspond to hosts in the previous formulas. The average number of nodes was found to be 12 with a time adjusted average (with nodes scaled by job run time) of 24. Based on these numbers, striping was chosen to optimize reads at a projected job size of 16 nodes, which works out to 1 stripe per GB. The performance for stripe-aware cp and an alternate version of stripe-aware tar will be given in Section III. As can be seen, modest performance gains were achieved in 32 of 49 cases due to the benefit of writing to a higher number of stripes. Only 8 of 17 losses were greater than 5%, all but one of which were from the bunzip2 case. It is not known why bunzip2 performs so poorly with increased striping.

Note that any performance gains by these tools are simply beneficial side effects of the main goal of increasing I/O performance during resource-intensive parallel jobs. To understand the types of gains that may be achieved during such jobs, the mpi-tile-io benchmark [16] was used to simulate an I/O access pattern that may be seen in practice. This benchmark tests the performance of the file system using noncontiguous tiled access, which is found in tiled displays and various numerical applications. Figure 14 shows the results of running mpi-tile-io on varying numbers of CPUs (8 per node) across different numbers of stripes with the tile size fixed at 1 GB per CPU. As can be seen, at higher numbers of CPUs and lower stripe counts, contention for the underlying OSTs becomes a significant issue and each CPU is only able to get its data at a fraction of the possible I/O rate.

The figure also shows the stripe counts predicted to be optimal from the previous dd results and the counts resulting from the local configuration of 1 stripe per GB. With the exception of the anomalous 1 CPU, 2 stripe case, the configured value is an average of 5% away from ideal with a maximum difference

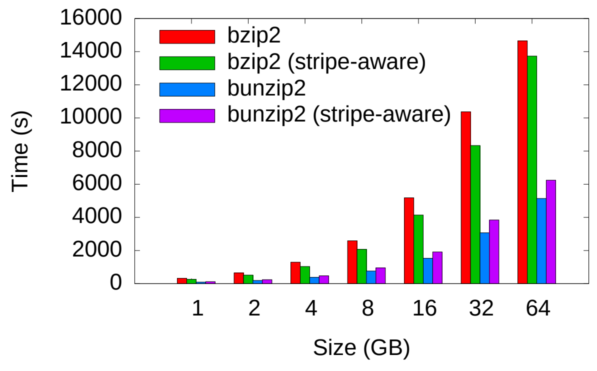

Fig. 10. Stripe-aware bzip2

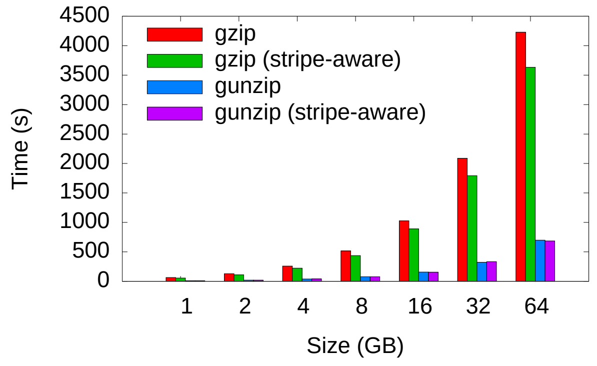

Fig. 11. Stripe-aware gzip

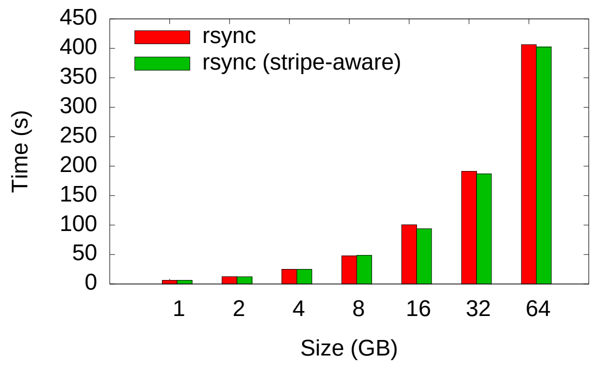

Fig. 12. Stripe-aware rsync

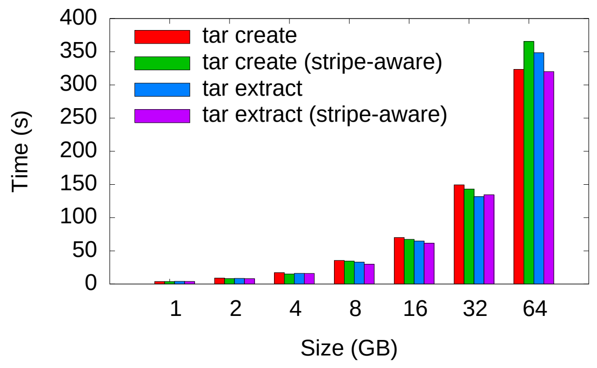

Fig. 13. Stripe-aware tar

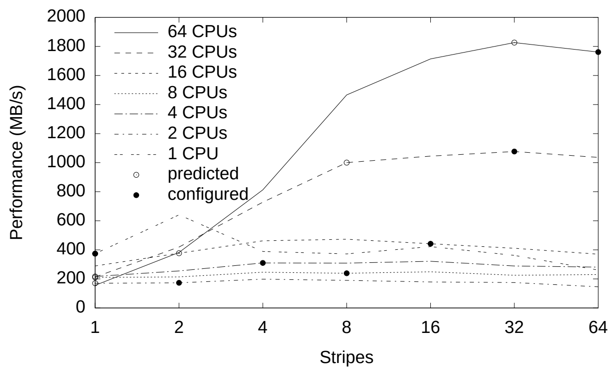

Fig. 14. Read performance of mpi-tile-io with a 1 GB tile per CPU

of 13%. The predicted value is an average of 15% away from ideal with a maximum difference of 32%. It should be noted that domain experts would likely achieve the highest possible performance using manual striping as they understand exactly how the data will be used. For the majority of users that may not understand all the intricacies of striping, however, the transparent optimization provided by stripe-aware tools will likely provide a significant performance boost.

III. High Performance System Tools

Stripe-awareness is a good first step towards transparent I/O optimization. The default stripe count can be kept low for more common small files resulting in reductions to OST contention and wasted space. Large files will automatically use more stripes as they are manipulated by these tools allowing user computations to achieve higher performance and OST utilization to be kept in better balance. The tools themselves, however, achieve only modest performance gains with I/O rates still nowhere near the raw speed of the file system.

Several issues prevent standard tools from achieving higher performance. They typically use a single thread of execution, which cannot keep single system I/O bandwidth fully utilized. They rely on the operating system buffer cache, which can become a bottleneck with multiple threads. They use sequential reads/writes instead of overlapping them. Finally, they operate on one host, where single system bottlenecks limit maximum I/O. Adding parallelism is significantly more complex than adding stripe-awareness. This section describes high speed variants of cp and tar that have been designed to provide I/O performance more in line with that of parallel file systems.

Performance was measured on CXFS, GPFS, and Lustre file systems, each attached to a different compute system. CXFS tests were run on a shared-memory SGI Altix 4700 consisting of 1.6 GHz dual-core Itanium 2 Montecito processors with 2 GB memory per core. GPFS tests were run on IBM iDataPlex nodes consisting of two 2.8 GHz quad-core Xeon Nehalem processors with 24 GB memory per node connected via DDR Infiniband. Lustre tests were run on SGI ICE nodes consisting of two 3.0 GHz quad-core Xeon Harpertown processors with 8 GB memory per node connected via DDR Infiniband.

CXFS and Lustre tests were run across two different file systems but only a single GPFS file system was available. GPFS scaling results were likely decreased due to double resource consumption. The GPFS instance was a pre-production file system, however, with no other activity on it. All CXFS and Lustre instances were in production, but Lustre was measured at known near-idle periods. The activity on CXFS was at unknown levels during testing but likely to have been fairly idle given the level of compute node activity.

A. *High Performance Cp*

Cp is the primary data movement application on Linux/Unix operating systems and is often involved in large file operations such as administrators balancing out file systems or users copying data between scratch and archive file systems. The standard single-threaded implementation of cp from GNU coreutils [5], however, cannot fully utilize parallel file system bandwidth due to limited concurrency, causing these operations to take much longer than needed. A high performance version of cp allows the speed of these operations to be increased transparently through more effective I/O utilization.

Mcp is a multi-threaded modification of the coreutils cp command using OpenMP [2] and is available as open source [14]. Mcp was previously introduced by the author [9] with results on Lustre but additional results on CXFS and GPFS are shown here to demonstrate its general applicability to different parallel file systems. Mcp is stripe-aware so can also be used as a fast restriping tool as files on Lustre must be copied to be restriped.

In general, copying regular files is an embarrassingly parallel task since files are completely independent from one another. The processing of the directory hierarchy containing the files, however, must ensure that a file's parent directory exists and is writable when the copy begins and must have its original permissions and ACLs when the copy completes. Mcp operates using two types of threads. A single traversal thread behaves like cp except when a regular file is encountered. In cp, regular files are copied immediately whereas in mcp, the traversal thread instead creates a copy task and pushes it onto a semaphore-protected task queue. It then pops an open queue to wait for the file to be opened before setting permissions and ACLs and continuing on to the next file.

The second type of thread is a worker thread with multiple instances. Each worker pops a task from the task queue, opens the file, and pushes a notification onto the open queue allowing the traversal thread to continue. Once the file has been opened, directory permission and ACL changes made by the traversal thread can no longer affect the worker's access to the file. The worker then performs the copy and pops another task when complete.

The original cp uses buffered I/O, which can exhibit poor buffer cache utilization since file data is read once, but never accessed again. This increases CPU workload by the kernel and decreases performance of other I/O as it thrashes the buffer cache. To address this problem, mcp supports both direct I/O, which allows read and/or writes to skip the buffer cache entirely, and posix_fadvise(), which informs the kernel that data can be released after it is read and/or written.

Figure 15 shows the performance of copying 64 1 GB files on a single node for varying numbers of threads on different file systems. Direct I/O performance was not gathered on GPFS as it was significantly worse than the other two schemes during initial testing. As can be seen, however, direct I/O on CXFS and Lustre resulted in significant gains with more than a 7x improvement in both 8-thread cases. GPFS had the lowest overall performance gain of 16%, but there was less margin for gain as the original cp performed exceptionally well at 4x faster than cp on Lustre and 6x faster than cp on CXFS. Fadvise() provided a small gain/loss on CXFS/GPFS, respectively, but improved Lustre much more where buffering saw a negative impact at higher thread counts.

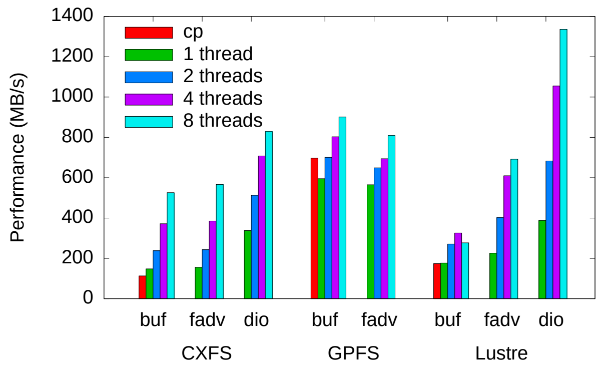

Fig. 15. Buffer-managed copy performance

Multi-threading increases overall parallelism, but parallelism within each thread can be increased even more via double buffering. Using asynchronous I/O, the read of the next file block can be overlapped with the write of the previous block. This theoretically reduces the time to process each block from time(read) + time(write) to max(time(read), time(write)). Figure 16 shows double buffering results where direct I/O on Lustre had the greatest gains, improving an average of 52% up to 10x cp on 8 threads. Direct I/O on CXFS improved an average of 25% up to 4 threads, but decreased 4% at 8 threads indicating that perhaps I/O limits were being reached on the compute system. Fadvise() on GPFS improved an average of 18% up to 1.3x cp on 8 threads, while on CXFS and Lustre, performance improved slightly up to 4 threads but decreased at 8 threads 12% and 4%, respectively.

On GPFS and Lustre, tests used a single node of a distributed memory system, hence subject to that node's physical resource limitations. To overcome these limits, mcp supports multi-node operation where the aggregate resources of multiple nodes can be used to carry out the same transfer. Nodes are divided into a single manager node that runs the traversal thread and multiple worker nodes. All nodes run multiple worker threads along with a communication thread to handle the distribution of tasks between the traversal thread on the manager node and worker threads on other nodes via TCP or MPI. The manager TCP/MPI thread waits for messages

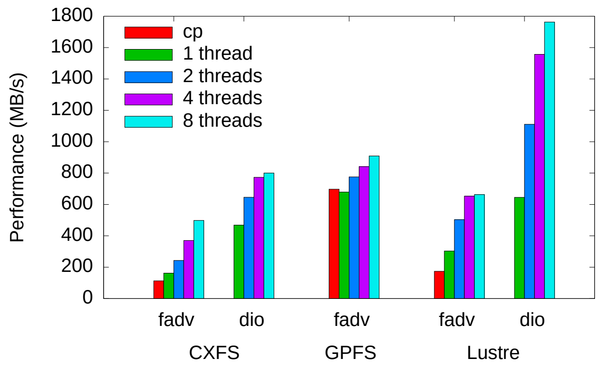

Fig. 16. Double-buffered copy performance

from the worker TCP/MPI threads, after which it pops a task from the manager task queue and sends it back to the worker node, who pushes it onto the local task queue. Worker threads operate as normal by popping tasks off their local queue and pushing notifications of file opens onto the local open queue, which are sent back to the manager via the TCP/MPI thread.

Figure 17 shows the results of using multiple nodes for the same copy. Very significant performance gains are achieved on Lustre with fadvise() actually overtaking direct I/O on 16 nodes at 27x cp compared to 24x cp for direct I/O. Larger gains are also seen on GPFS at 3.4x cp on 16 nodes.

With less files than threads or a few large files, threads can become imbalanced. To evenly distribute workload across threads/nodes, mcp supports split processing of files so multiple threads can operate on different portions of the same file. In this mode, the traversal thread may add multiple tasks for each file corresponding to different portions of the file at different offsets that can be processed in parallel.

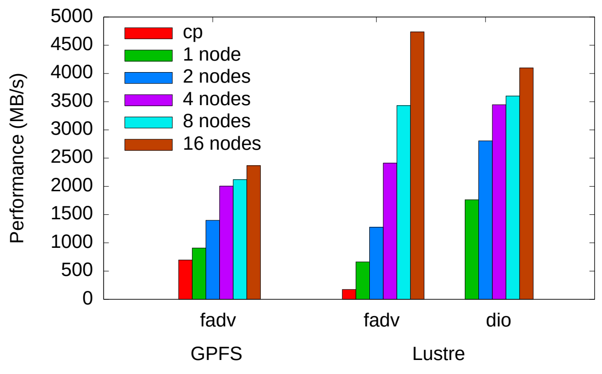

Fig. 17. Multi-node copy performance

Figure 18 shows the copy of a single 64 GB file across multiple threads. The profiles of the CXFS and Lustre single node cases are very similar with an initial gain followed by almost no gain as the number of threads increases. On CXFS, the maximum is similar to the multi-file case, but on Lustre, there appears to be a bottleneck accessing the same file with multiple threads on the same node that more than halves multi-file performance with both direct I/O and fadvise(). If 1 thread per node is used instead of 1 thread per cpu, however, the results are significantly different at 19x and 12x cp on 16 nodes/threads for direct I/O and fadvise(), respectively. It is likely that Lustre locking mechanisms are enforcing sequential access to the file between threads on the same node as the maximum achieved in the single file 8-thread cases is almost identical to that of the 1-thread cases in the multi-file scenario.

GPFS scaled similarly in both the thread per cpu and thread per node cases up to 3.6x cp on 16 nodes/threads. Note that the GPFS file system for this particular test was different from previous tests. The split file test used a production GPFS file system with an unknown amount of other activity.

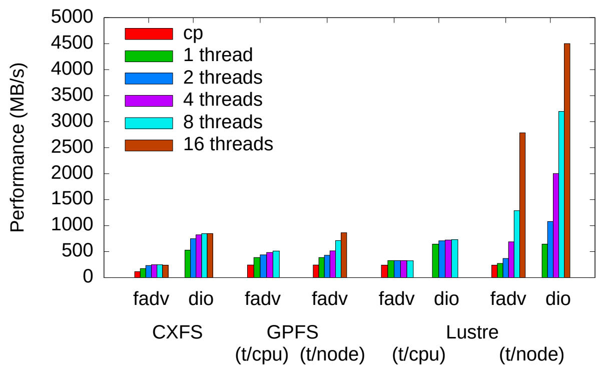

Fig. 18. Split-file copy performance

B. High Performance Tar

One of the most commonly used tools in basic user workflows is tar, which is used to consolidate multiple input/output files before transfer to/from the system. Tar was originally intended for inherently sequential tape access, hence is sequential by design. Since it is now more commonly used on standard file systems, however, these design choices leave it with the same types of inefficiencies that plague cp.

While the tar source code could have been optimized in the same manner as the cp code was for mcp, the tar code is quite a bit larger and more complex than cp, which itself took a significant effort to parallelize. Instead, it was realized that the sequential nature of tar files allow them to be easily parallelized by leveraging existing mcp work. Namely, each file within a standard tar archive is stored contiguously surrounded by header and padding blocks. Hence, creations/extractions can be achieved with partial copies to/from a given offset in the tar archive from/to a given file.

To support this functionality, mcp was given the ability to copy any offset and length of the source file to any offset of the target, and on Lustre, the ability to specify the target stripe count as a multiple of the length. Additionally, mcp was given the ability to read a list of the source/target files to copy from stdin instead of having to specify them on the command line. Then a Perl version of tar from the Archive::Tar module was modified such that instead of copying file data itself, it creates a list of partial copies and feeds them to mcp, which is then spawned on as many hosts as specified. During creation, a (suitably striped) tar skeleton is written with appropriate headers and padding. In the extraction case, file locations within the existing archive are determined.

The drawback to this approach is that the resulting program is not completely drop-in compatible with the original (tar) as are all the other modifications discussed thus far. Only basic creation/extraction options and the POSIX ustar format [7] are supported. Any unsupported options or formats cause the default tar program to be invoked instead.

Figure 19 shows the results of creating and extracting a tar file consisting of 64 1 GB files. Note that GPFS and Lustre used front-end systems with the previous configuration instead of dedicated PBS nodes. Also, GPFS tests were run on the shared production file system with unknown activity mentioned in the mcp split file tests. Lustre achieved creations at 15x tar and extractions at 31x tar on 8 hosts. CXFS had the next highest gains with creations at 6.3x tar and extractions at 9.3x tar on 8 cpus. Finally, GPFS achieved creations at 3.2x tar and extractions at 3.6x tar on 8 hosts.

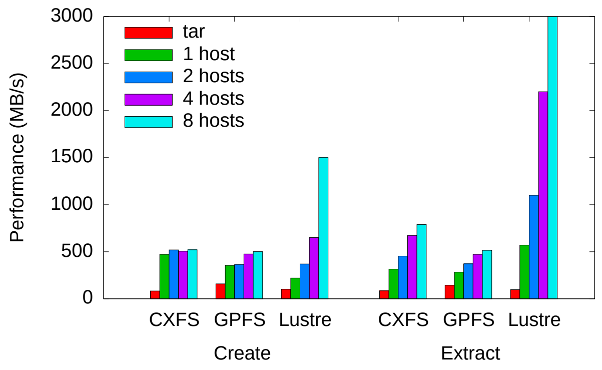

Fig. 19. Multi-host parallel tar performance

IV. Related Work

There are a variety of efforts related to this paper. SGI's cxfscp [21] is a multi-threaded copy tool that supports direct I/O and achieves results similar to mcp on shared-memory systems, but offers minimal benefit on cluster architectures. Streaming parallel distributed cp (spdcp) [12] has similar goals as mcp and achieves very high performance on clustered file systems using MPI to parallelize transfers of files across many nodes. Like mcp, spdcp can utilize multiple nodes to transfer a single file. The spdcp designers made the conscious decision to develop from scratch, however, instead of using GNU coreutils as a base, whereas mcp started with coreutils to support all available cp options and to take advantage of known reliability characteristics. Mcp can also use a TCP model as well as MPI to support a larger class of systems.

There are several related multi-threaded programs for the Windows operating systems. RichCopy [6] supports multi-threading in addition to the ability to turn off the system buffer, which is similar to mcp's direct I/O option. MTCopy [11] operates in a similar manner as mcp with a single file traversal thread and multiple worker threads. MTCopy also has the ability like mcp to split the processing of large files amongst multiple threads.

Ong et al. [15] describe the parallelization of cp and other utilities using MPI. Their cp command, however, was designed to copy one file to many nodes whereas mcp was designed to allow many nodes to copy parts of the same file. Desai et al. [1] use a similar strategy to create a parallel rsync utility that can synchronize files across many nodes at once. HPSS Tar [4] optimizes tar by allowing archives to be created/extracted directly to/from HPSS, bypassing local storage. It uses multi-threading, buffer management, and HPSS network striping capabilities to significantly increase tar performance. The pltar utility [13] uses MPI to parallelize archive creation and extraction and achieves very high performance on Lustre file systems.

There are several studies of Lustre performance. Simms et al. [22] measured the sustained local/remote Lustre I/O performance using varying stripe counts with results focused on optimizing stripe count for different blocks sizes instead of for different file sizes. Fahey et al. [3] investigate basic Lustre I/O performance characteristics and find minimal benefit to having more stripes than writers, which supports optimum stripe counts based on number of hosts. Yu et al. [23] investigate stripe size performance and stripe count metadata cost, confirming the high metadata overhead of small files with large stripe counts. Finally, Laros et al. [10] examine Lustre performance and observe drops during single file I/O when compared to multi-file I/O as was observed earlier.

V. Conclusions and Future Work

This paper has described modifications made to standard system tools commonly found in user workflows to better support parallel file systems. Various tools were enhanced with knowledge of Lustre striping to improve parallel I/O performance in computational jobs as well as reducing contention, wasted space, and imbalances on OSTs. The cp and tar tools were further modified with greatly enhanced parallelism for significant performance gains on several parallel file systems. By deploying these tools in place of their standard counterparts, users will transparently achieve better I/O utilization and increased performance by simply using tools as normal.

Figure 20 shows a summarized view of the times a typical workflow might take before the availability of the tools described in this paper and after, where data of a given size is transferred in, extracted with tar, and run through some parallel application on an appropriate number of CPUs (only the I/O portion shown), with the resulting output data (assumed to be the same size as the input for simplicity) tarred up and transferred back to the origin. As can be seen in the middle part of the figure, although the optimizations reduce execution time considerably, remote transfer time can still end up dominating the workflow as network speeds are typically much slower than disk speeds. In other work by the author, a tool called Shift [8] was introduced that can automatically parallelize remote transfers as well. The right side of the figure

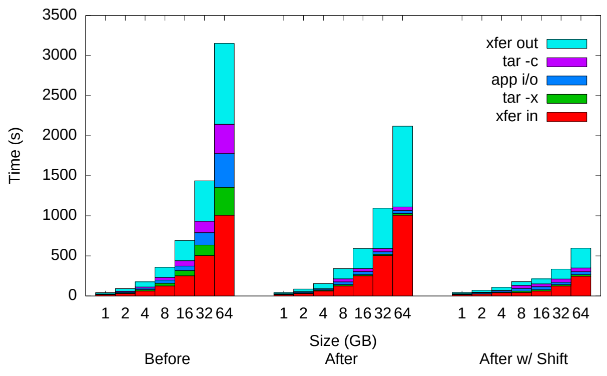

Fig. 20. Summary of performance gains

shows a fully optimized workflow with transfer times taking advantage of Shift's greatly enhanced performance. As can be seen, when taken together, these tools allow workflows to operate at much higher throughputs than default tools provide, thereby increasing the resource utilization of both users and systems with no extra effort on the users' part.

There are a variety of directions for future research. Other standard tools should be made stripe-aware including zip, pigz, and pbzip2 for archival/compression and scp, sftp, bbftp, and gridftp for file transfer. Additional tools should be investigated to see which can be parallelized and which would benefit the most. Finally, other I/O access patterns should be studied to further validate the benefits of automatic striping.

VI. Acknowledgments

This work is supported by the NASA Advanced Supercomputing Division under Task Number ARC-013 (Contract NNA07CA29C) with Computer Sciences Corporation.

References

[1] N. Desai, R. Bradshaw, A. Lusk, E. Lusk: MPI Cluster System Software. 11th European PVM/MPI Users' Group Meeting, Sept. 2004.

[2] L. Dagum, R. Menon: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering, vol. 5, no. 1, Jan.-Mar. 1998.

[3] M. Fahey, J. Larkin, J. Adams: I/O Performance on a Massively Parallel Cray XT3/XT4. 22nd IEEE Intl. Parallel and Distributed Processing Symp., Apr. 2008.

[4] Gleicher Enterprises: HPSS Tar Man Page. http://www.mgleicher.us/GEL/htar/htar_man_page.html.

[5] GNU Core Utilities. http://www.gnu.org/software/coreutils/manual/coreutils.html.

[6] J. Hoffman: Utility Spotlight: RichCopy. TechNet Magazine, Apr. 2009.

[7] IEEE Computer Society: Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7. IEEE Standard 1003.1-2008, Dec. 2008.

[8] P.Z. Kolano: High Performance Reliable File Transfers Using Automatic Many-to-Many Parallelization. 5th Wkshp. on Resiliency in High Performance Computing, Aug. 2012.

[9] P.Z. Kolano, R.B. Ciotti: High Performance Multi-Node File Copies and Checksums for Clustered File Systems. 24th USENIX Large Installation System Administration Conf., Nov. 2010.

[10] J.H. Laros, L. Ward, R. Klundt, S. Kelly, J.L. Tomkins, B.R. Kellogg: Red Storm IO Performance Analysis. 9th IEEE Intl. Conf. on Cluster Computing, Sept. 2007.

[11] Y.S. Li: MTCopy: A Multi-threaded Single/Multi File Copying Tool. CodeProject article, May 2008. http://www.codeproject.com/KB/files/Lys_MTCopy.aspx.

[12] K. Matney, S. Canon, S. Oral: A First Look at Scalable I/O in Linux Commands. 9th LCI Intl. Conf. on High-Performance Clustered Computing, Apr. 2008.

[13] K.D. Matney, G. Shipman: Parallelism in System Tools. 52nd Cray User Group Conf., May 2010.

[14] Multi-Threaded Multi-Node Utilities. http://mutil.sourceforge.net.

[15] E. Ong, E. Lusk, W. Gropp: Scalable Unix Commands for Parallel Processors: A High-Performance Implementation. 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Sept. 2001.

[16] Parallel I/O Benchmarking Consortium. http://www.mcs.anl.gov/research/projects/pio-benchmark.

[17] Restriping Tools for Lustre. http://retools.sourceforge.net.

[18] F. Schmuck, R. Haskin: GPFS: A Shared-Disk File System for Large Computing Clusters. 1st USENIX Conf. on File and Storage Technologies, Jan. 2002.

[19] P. Schwan: Lustre: Building a File System for 1,000-node Clusters. 2003 Linux Symp., Jul. 2003.

[20] L. Shepard, E. Eppe: SGI InfiniteStorage Shared Filesystem CXFS: A High-Performance, Multi-OS Filesystem from SGI. Silicon Graphics, Inc. white paper, 2004.

[21] Silicon Graphics Intl.: Cxfscp Man Page. http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=man&fname=/usr/share/catman/a_man/cat1m/cxfscp.z.

[22] S.C. Simms, G.G. Pike, D. Balog: Wide Area Filesystem Performance using Lustre on the TeraGrid. 2nd TeraGrid Conf., Jun. 2007.

[23] W. Yu, J.S. Vetter, H.S. Oral: Performance Characterization and Optimization of Parallel I/O on the Cray XT. 22nd IEEE Intl. Parallel and Distributed Processing Symp., Apr. 2008.