FCP: A Fast and Scalable Data Copy Tool for High Performance Parallel File Systems

Feiyi Wang, Verónica G. Vergara Larrea, Dustin Leverman, Sarp Oral
National Center for Computational Sciences
Oak Ridge National Laboratory, Oak Ridge, TN, USA
Email:{fwang2, vergaravg, leverman, oralhs} at ornl.gov

Abstract—**The design of HPC file and storage systems has largely been driven by the requirements on capability, reliability, and capacity. However, the convergence of large-scale simulations with big data analytics have put the data, its usability, and management back on the front and center position. One of the most common and time consuming data management tasks is the transfer of very large datasets within and across file systems.**

In this paper, we are introducing the FCP tool, a file system agnostic copy tool designed at the OLCF for scalable and high-performance data transfers between two file system endpoints. It provides an array of interesting features such as adaptive chunking, checksumming on the fly, checkpoint and resume capabilities to handle failures, and preserving stripe information for Lustre file system among others. It is currently available on the Titan supercomputer at the OLCF. Initial tests have shown that FCP has much better and scalable performance than traditional data copy tools and it was capable of transferring petabyte-scale datasets between two Lustre file systems.

I. Introduction

The design of HPC file and storage systems has largely been driven by the requirement on capability, reliability, and capacity. Data manageability and usability often take the back seat in the grand scheme of options and restrictions. However, the convergence of large-scale simulations with big data analytics have put the data, its usability, and management back on the front and center position. Among many tasks related to data management, data integrity and file system profiling are paramount to file system users and operators. Few, if any, production-ready tools can take advantage of a cluster-based infrastructure to truly tackle large scale datasets as often seen in HPC environments. This technical deficiency prompted the design and implementation of a suite of efficient and scalable file system tools to help with everyday user and administrator needs.

In this paper, we will introduce the FCP tool, as a part of a file system tool suite under development at Oak Ridge Leadership Computing Facility (OLCF). The FCP tool is a file system agnostic tool designed for scalable and high-performance data transfers between two file system endpoints. In addition, it provides an array of interesting features such as adaptive chunking, checksumming on the fly, checkpoint and resume capabilities to handle failures, and options of preserving stripe information for Lustre file systems. The FCP tool is currently available on the Titan supercomputer environment at the OLCF. Our initial tests have shown that FCP has much better performance and scalability than traditional data copy tools and that it is capable of transferring petabyte-scale datasets between two Lustre file systems. The FCP tool is built on top of pCircle, a parallelization engine, and uses MPI to take advantage of existing infrastructure and maximize the efficiency of the transfer.

There are multiple efforts published in the literature aiming to increase the efficiency and ease use of handling large data sets. Among these, *streaming parallel distributed copy (spdcp)* [1] was designed from scratch and uses MPI to parallelize data transfers between two POSIX file systems using multiple nodes. *spdcp* was file system agnostic, but optionally was able to preserve and replicate the source's data set Lustre striping pattern on the target file system. MCP [2], [3] functions in a similar fashion, however, it uses the GNU coreutils as a basis and provides *cp*-like command line interface to the users.

Ong, Lusk, and Gropp [4] also investigated the same topic and they presented a family of parallelized everyday Linux commands using MPI, called *Parallel Unix Commands*. *Parallel copy (ptcp)* was one of the proposed tools in the family. *ptcp* was designed as a tool to replicate a set of files from a local file system to a selected set of nodes. Other work can also be found in the literature, such as [5], [6], which focuses on providing an efficient mechanism for transferring data sets between a local parallel file system and a remote HPSS archive system. Both efforts [5], [6] catalog the data sets, analogue to the Unix *tar* utility, before transferring to the HPSS. While [6] uses a single node with multiple threads to achieve high performance, [5] uses MPI over multiple nodes to achieve even a higher tar and copy aggregate performance to HPSS.

Several other works are built on top of widely-adopted tools such as *rsync* [7], [8], [9]. One potential issue with this approach is that *rsync* was designed in an era in which network bandwidth was a limited and premium resource. The delta calculation allows only the data difference per file to be transferred over the network to achieve the higher performance and greater efficiency. This is not the case with most data centers where the internal network bandwidth is often over-provisioned comparing to disk bandwidth. On top of it, with the rysnc approach the underlying patching mechanism does not apply the data difference to the destination file "in-place." The rsync utility rather reconstructs the destination file which is equivalent to reading the old data, modifying it in memory, and writing the new data to the destination file. From the I/O perspective, this read-modify-write operation sequence can be expensive. We evaluated the rsync algorithm as one of the supported transfer modes in the FCP (instead of brute-force overwrite method which we currently employ), and our experiments showed with large-scale (particularly large file size) data sets, the rysnc method is not efficient.

Our work differs from existing works in multiple fronts. First, we organize and parallelize the transfers over adaptively defined *chunks* instead of files between two POSIX endpoints, using the parallel tree walk algorithm discussed in [10]. By parallelizing over chunks, we can achieve even greater degrees of parallelism and performance. Second, previous effort such as *spdcp* adopted a master-slave approach where a single master coordinates the copy tasks, whereas FCP is completely decentralized for better load balancing and scalability. Our work is both motivated and derived from [11] but written from scratch in Python for greater flexibility without sacrificing performance.

II. Design, Implementation, and Optimization

Much of user-level file system activities can boil down to a tree walk to visit each node of interest and perform operations along the way. Tree walking is a trivial operation and there are well-understood graph algorithms for this purpose. For scalability to billions of nodes, however, we need a parallelized solution in order to distribute the workload. Lafon's work [10] forms the foundation, FCP [12] can be thought as a derivative solution. This section provides a high-level overview of the core ideas as well as engineering optimizations we have done to make FCP a viable tool in a production environment.

A. Workload Parallelization Engine

There are three core building blocks pertaining our implementation of workload parallelization. The first of which is the *work-stealing pattern* load balancing technique, in which, an idle worker requests (or steal) work from other busy workers. This method is well-known and can be found in implementations such as Intel's Cilk Plus and Intel's Thread Building Blocks (TBB) [13]. This pattern is particularly important for large scale data transfers, in that, when scanning and working on a file system as large as Spider 2, with 50 millions directories and half a billion files, any single process might generate contention or even get stuck on particular directory tree. An evenly distributed workload will suffer from the slowest worker syndrome, which inevitably slows down the overall progress and reduces efficiency. The *work-stealing pattern* allows FCP to overcome this bottleneck and *self-balance* the system to maximize the efficiency.

A second building block is on *distributed termination*. This is a fundamental problem in distributed computing. Its goal is to determine if distributed computation has completed. It is a non-trivial task because no process has complete knowledge of the global state, and you can not assume a global time either. There are many algorithms to attack this problem and our current implementation is the Dijkstra-Scholten algorithm[14] and summarized as the following:

1) The system in consideration is composed of N machines, $n_0, n_1, \ldots, n_{N-1}$, logically ordered and arranged as a ring. Each machine can be either white or black. All machines are initially colored as white.
2) A **token** is passed around the ring. Machine n's next stop is $n+1$. A token can be either white or black. Initially, machine n_0 is white and has the white token.
3) A machine only forwards the token when it is passive (no work).
4) Any time a machine sends work to a machine with lower rank, it colors itself as black.
5) Both initially at n_0, or upon receiving a token:
   a) if a machine is white, it sends the token unchanged.
   b) if a machine is black, it makes the token black, makes itself white, and forwards the token.

The algorithm itself is not particularly difficult to implement. However, coupled with the messaging and work-stealing pattern does make the system behavior non-deterministic and a challenge to debug at scale.

B. Asynchronous Progress Report: A Compromise

Generally speaking, we need to handle two type of nodes when walking a file system tree: files and directories. For files, we need to decide how to break them up (chunking) for more parallelism, and for directories, we scan all its entries. The most logical and efficient approach would be doing the *parallel walk*, *chunking*, and *copying* at the same time. To summarize:

1) Given a file node, break it into chunks, and put each chunk into the local work queue.
2) Given a chunk, perform copy operation.
3) Given a directory, scan and put each file *or* directory into the local work queue.

As we can see, this is an efficient solution that intermixes the copy operation with the scanning operation. The problem with this approach is more of a *usability* issue: the end user has no idea how far the copy has progressed. The best feedback information that can be given is how many files and directories have been scanned thus far. This is often not enough when users are copying a large dataset, which could take tens of hours or more to transfer. A percentage value is more comforting. It also implies we need to have an idea of not only how many total files/directories there are, but also periodic gathering of summarization information from all processes involved. For MPI-based messaging, this means a collective operation in which all processes have to stop processing the local queue and report its status, which is a suboptimal solution.

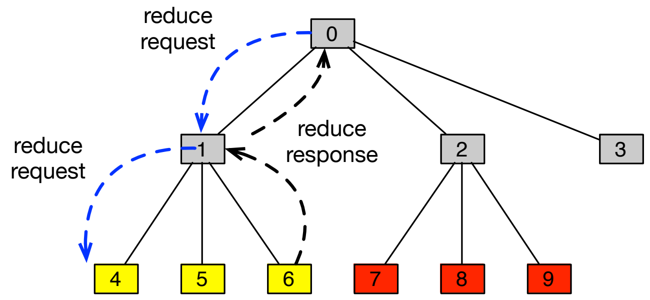

Figure 1. Async notification of work progress

This is where non-blocking collective operations come to the rescue (though we are not relying on the presence of MPI-3). We first form a k-degree tree out of all ranks. Each intermediate node is responsible of sending a query (asynchronous) and collecting the status report from its children. It only relays the summary information to its parent when all of its children have reported. The summary information is ready when the root node receives the report. This process is illustrated in Figure 1.

This mechanism allows each process to proceed on its work without a global stop and synchronization step, thus improving the overall efficiency and performance of the transfer. The user-configurable progress report interval is a soft guidance. The actual output might differ due to its asynchronous report nature, but it rarely matters in practice. However, though the percentage report from a full scan of the tree provides a better user experience, it compromises on efficiency.

C. *Adaptive Chunking*

Conventional file system tools are not particularly adept at handling large files. In a production HPC storage and file system, large files are a norm rather than an exception. A recent profiling study using *fprof* [15] for OLCF Spider 2 system shows a distinct dichotomy of a very large number of small files and a relatively small number of large files, shown in Figure 2. More specifically, even though 85% or more files are less than 256 KiB, they are less than 1% of total file size. This indicates that systems such as Spider 2 are hosting few number of very big files. In fact, a recent tally shows at least 18,000 files with size above 128 GiB

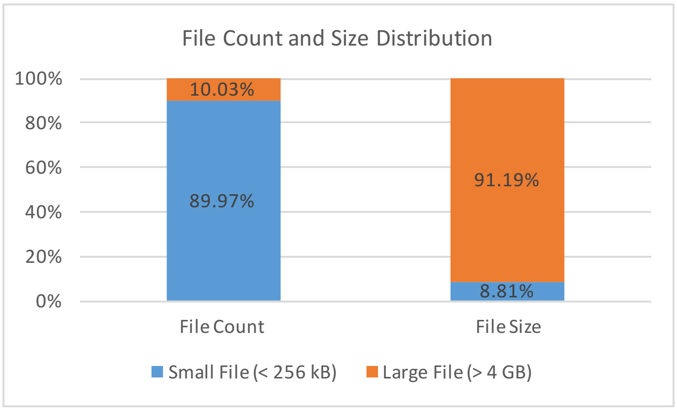

Figure 2. Profiling Atlas File System At Scale (500 million files plus 43 million directories)

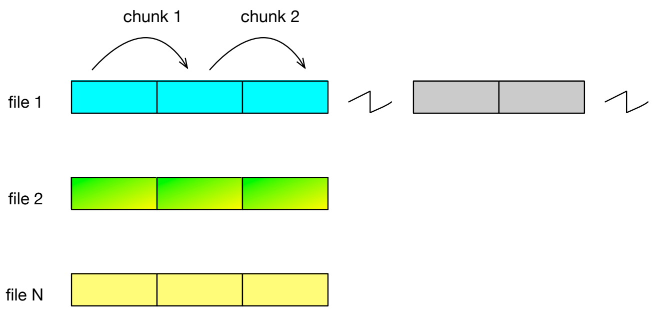

Figure 3. Breaking Large File Into Chunks

each, and a few hundred with size 4 TiB and beyond. For data transfers of such dataset, chunking, i.e, breaking up each file into a sequence of chunks as shown in Figure 3, is absolutely necessary to increase the parallelism.

Each chunk object is self-contained, in the sense that it carries all the necessary offset information for the processes to handle them independently even when the workload has migrated from one process to another. Regarding to the size of the chunk, there are at least two design considerations. First, if we make the chunk small, parallelism increases at the cost of both processing and metadata overhead associated with each chunk. The return of the parallelism will diminish as the chunk size decreases. If we increase the chunk size too much, we might miss out on the opportunity for increasing parallelism for files with size just below the chunking threshold. On the other hand, making the chunk size completely file dependent has consequences on checksumming as well, which we will discuss in the next section. Sufficient to say, there is no one size that fits all. Our current design allows the user to specify a chunk size of their choice based on knowledge of data size distributions. By default, FCP will parallel scan all files under transfer and adaptively pick a chunk size that represents the balance of total chunks and total file size.

D. Parallel Checksumming and Verification

One of the major concerns of curating large scientific dataset is the integrity, both during the data generation and after the data movement. The *bit rotten* syndrome has been observed by users at OLCF facility before. Conventional checksumming tools are file-based, whereas scientific users prefers a single signature for the entire dataset. It is obviously unwieldy to carry hundreds or even thousands individual signatures for large datasets. To that end, FCP provides *on-the-fly* checksumming and *dataset- based* checksumming as an option.

A parallel block checksumming can *piggyback* along the copy task in the sense that the block of data has already been read into the memory from the disk. All we need is a slice of compute time for the checksum. More importantly, the checksum itself as a result can be used for **verification**. However, the verification phase must happen *after* the copy job is completed. At that point, we can re-read the data from the destination and checksum it against original checksums we have computed and saved.

We also consider the option of checksumming before the copy job completes, i.e. as soon as the block data are written. The problem lies in the fact that without a data sync or close operation, we cannot be certain that the subsequent read will reflect the data residing on the disk. Yet, such sync operation for each block at this frequency can be dangerously expensive. Therefore, the design is to carry out this task in a separate phase, after the copy job is completely finished. Ideally, we would like to *overlap copy, checksumming, and verification* operations. However, the nature of distributed operation implies we do not have the global awareness to know if a file has been completely copied until the entire copy is done. There might be some further optimizations we could do in this area, but this remains to be explored.

E. Checkpointing and Restart

Large scale data transfer takes time. OLCF provides a dedicated cluster known as Data Transfer Nodes (DTN) that handles the data movement. However, this dedicated resource has to be shared among all facility users. Let us assume a medium-size (e.g. 200 TB) data transfer job, allocated with a default set of 4 transfer nodes. We can further assume each transfer node can do roughly 1 GB/s. Then, the job would require at least 14 hours to finish. There are many scenarios under which a transfer job can fail midway and *resume* capability is truly preferred. In the case of single thread operations, this would have been trivial as there is a centralized view on how much work has been done thus far. In distributed computation, this is no longer the case. FCP is designed just as any large scale scientific application running on an HPC cluster: it periodically writes out checkpoint files for a snapshot of the work completed (or remaining to be completed). In the event of failures, users can restart the job and resume the data transfer.

This is more or less a conceptually straightforward process with two caveats. The first is related to *when* it fails. Since FCP walks the tree prior to the actual copy stage, if a failure occurs in the middle of the tree walking stage, we do not have enough information to do a snapshot yet, and the tree walk cannot be restarted. The second caveat is related to choosing an in-core or out-of-core solution for scalability, which we will elaborate further in the next section.

F. Extreme Scalability: In-Core and Out-of-Core

FCP was originally designed as an in-core solution: all metadata are kept in memory. There are at least two advantages when every thing are kept in core: the obvious one is performance; secondly, it makes the checkpointing easy. At any point of time, the local work queue and its queuing items represent a snapshot of the remaining works. However, in the case of extreme scale datasets, we start to face memory pressure. The memory usage comes primarily from two sources:

- metadata information such as path, mode, size, uid and gid, etc. collected during the tree walk.
- local work queue, which holds source, destination, checksum, offset etc.

Each metadata object takes about 112 bytes. For a 500 million objects, this requires an estimated 56 GB. Each chunk object takes about 64 bytes. The length of the local work queue is dynamic: it depends on both the size of the files (how many chunks are needed) and the number of files within a directory. One of the worst case scenarios are large flat directories with millions of files underneath. To curtail the size of the local work queue, we can perform a small optimization by inserting newly scanned file objects to the front of the queue, and newly scanned directory objects to the tail of the queue, as shown in Figure II-F. This effectively limits the queue size (order wise) to be the largest number of files within a directory.

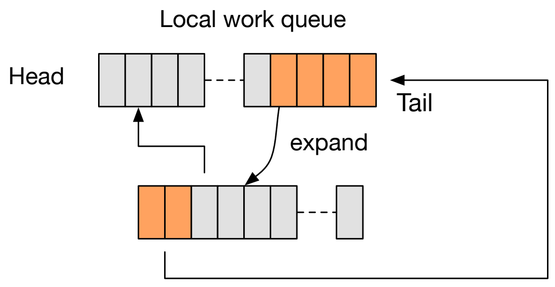

Figure 4. Local work queue head-tail optimization

If one can assume that the directory tree is largely balanced and a reasonable number of nodes are used for transfer, then the memory pressure will be lower. Short of this assumption, we will have to consider the extreme case of a single node with limited memory transferring extreme scale datasets. In this case, we would have to resort to an out-of-core solution, i.e., persisting metadata information as well as queue to a backend persistent storage such as a key-value or database store. One of the issues we encountered was the complicated handling of checkpointing: the database state no longer represents the last known state of remaining work items. For example, there is no guarantee that the last n work items from persistent store have been completed, if the failure occurs in the middle of the transfer.

III. Evaluation

Before exposing a new tool to users, all features must be thoroughly tested for performance, functionality, and usability. We completed a performance evaluation of the tool, and in addition verified the functionality of FCP's Lustre stripe preservation and checksumming features.

All the experiments conducted during the evaluation were run on the OLCF's data transfer node (DTN) cluster. The DTN cluster is comprised of 8 nodes each with an 8-core 1.8 GHz Intel Xeon CPU E5-2603 processor, and is tuned specifically for wide-area network data transfers. The DTNs are connected to Spider 2 via FDR InfiniBand interconnect which provides a peak bandwidth of 6.75 GB/s.

A. Single-Node Performance Testing

To verify the performance of FCP, we chose to use our eight scheduled data transfer nodes (DTNs), which have the center-wide Spider 2 file systems mounted. We identified four data sets to use for our testing:

- Data set 1: Single 1TB file of zeros
- Data set 2: Single 100GB file of zeros
- Data set 3: One-thousand 100MB files of zeros
- Data set 4: Mixed small file data set (real data with 100k files with sizes ranging from 4KB - 100MB)

To eliminate caching effects from the storage system, we pre-created the data sets for all of the runs and dumped the caches on the storage servers. After each of the runs of the scheduled DTNs, we would drop all caches before and after each run as part of the job submission script.

Many tools are used to move data around within the OLCF, namely: `cp`, `rsync`, DCP [11], and FCP. We ran tests using all four of these tools to move all four of the data sets mentioned above. Results from our tests are shown in Table I and Fig. 5.

In every single-client test case except for the small file case, FCP was the fastest, substantially outperforming all other tools. FCP was about 8 seconds slower in the transfer of the small file dataset, which could be attributed to the very large number of files that were involved in the tree walk step.

Table I
Single Lustre 2.7 client performance (Average)

Dataset Description	cp	rsync	dcp (np 8)	fcp (np 8)
Single 1TB file	3556s	7474s	2871s	1653s
Single 100GB file	193s	545s	203s	123s
1000x 100MB files	290s	574s	212s	107s
Mixed small files	436s	321s	76s	84s

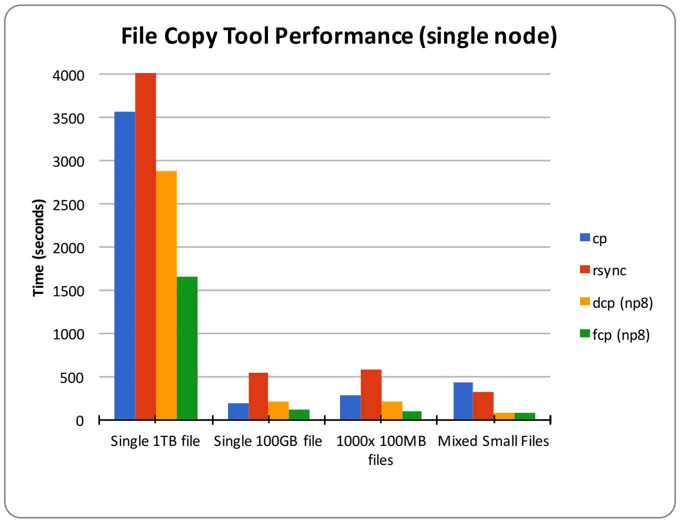

Figure 5. Average transfer time single Lustre 2.7 client performance

B. Multi-node performance on the OLCF's Spider 2 Lustre file system

To better understand the performance of FCP on parallel file systems, we designed tests that used different Lustre file stripe counts and ran across 1, 2, 4, and 8 nodes on the OLCF's DTN cluster. The same data sets used in section III-A were used for all the multi-node performance tests.

The default stripe count on Spider 2, the Lustre parallel file system at the OLCF, is set to 4 and was chosen to balance between small and large I/O operations. In addition to the default stripe count of 4, copy operations with 1, 32, and 512 stripe counts were conducted. Figures 6 to 9 show the performance results obtained from FCP transfers completed for all test data sets with varying stripe counts.

As Fig. 6 shows, the performance of FCP scales up as more nodes are used for the transfer. This is particularly evident in the single large file (1TB) case, in which, the copy operation using 8 DTN nodes is approximately 6X faster than using a single node. Furthermore, the results show that for single 1 TB file transfers it is best to use a stripe count of 32, and as many transfer nodes as available. The speedup observed starts to tail off between the 4 and 8 node cases, which indicates that for files in the 1 TB - 10 TB range, adding more than 8 nodes to the transfer is not necessary. For cases with larger files, for example in the 10s - 100s of TB range, it is recommended to use a larger stripe count.

Another thing to note from Fig. 6 is the 1 stripe case, in which the transfer speed does not improve as more nodes are added. Similar results are observed from the medium file size transfers conducted, in which a single 100 GB file was copied under the same configurations described above. Like with the 1 TB file case, FCP performance in the 100 GB case is fastest when 4 nodes and 32 stripes are used.

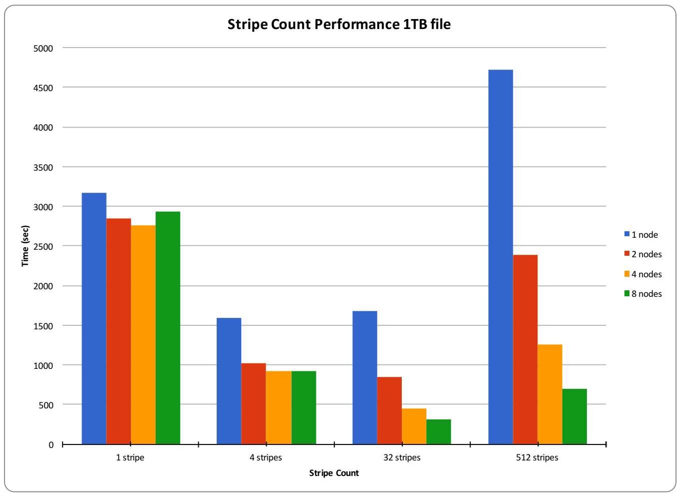

Figure 6. Average transfer time multi-client performance of a single 1 TB file

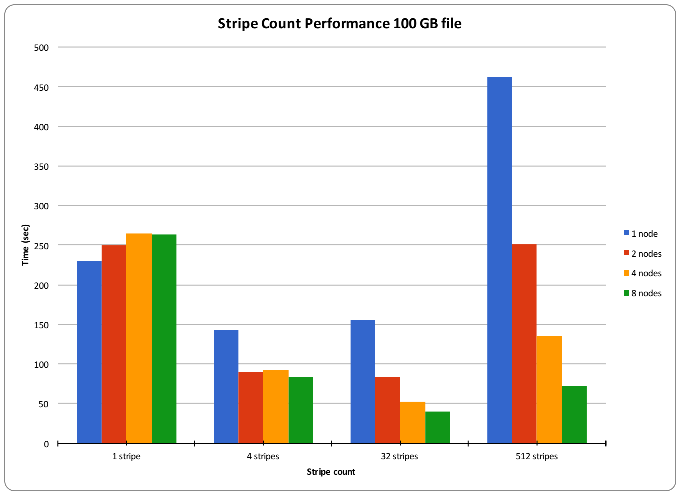

Figure 7. Average Transfer Time Multi-client Performance of a single 100 GB file

The results from multi-node tests with data sets 3 and 4 indicate that for many small files, on the order of 100 MB or less, FCP has similar performance with 1, 4, and 32 stripes. In both cases, the transfer speed continues to improve with the addition of more nodes, and the best performance is obtained when using a single stripe or the default stripe count of 4.

For the mixed small files test case (Data Set 3), using 512 stripes was a challenge. When files are small, spreading each file across that many stripes results is generally not recommended. In addition, as we can see in Fig. 8, the transfer speed on a single node using 512 stripes is approximately 5X slower than a transfer of the same data set when using a 32 stripes. Although performance does improve when more nodes are used, the best measurement we obtained with 512 stripes for data set 3 was about 2X slower than the 32 stripe count case.

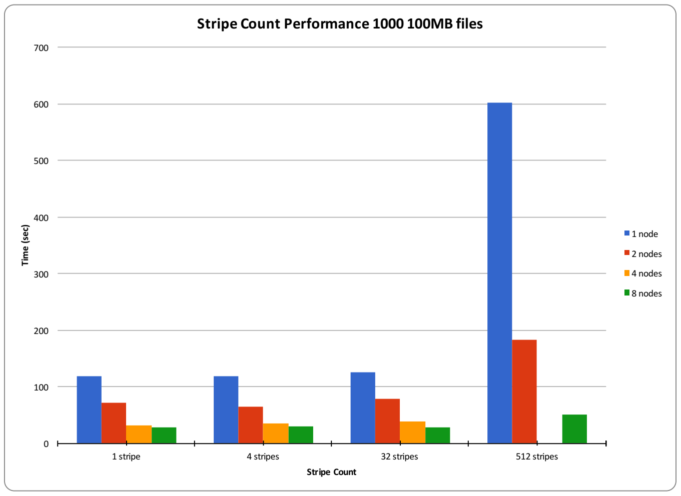

Figure 8. Average transfer time multi-client performance of 1,000 x 100 MB files

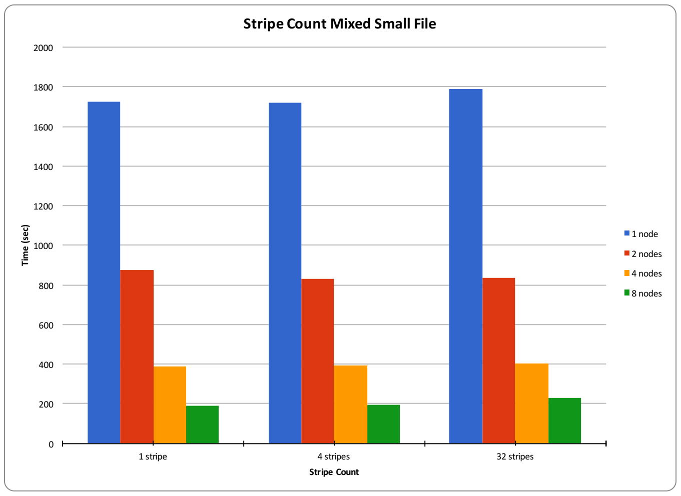

Figure 9. Average transfer time multi-client performance of mixed small files

The multi-node tests allowed us to measure the scalability of FCP. Our results show that the performance obtained is dependent on the type of data set being copied. It has also helped us identify guidelines that could be helpful for the user community when attempting to transfer data sets.

C. *Testing checksum feature*

As section II-D describes, FCP is capable of verifying the integrity of a data set *on the fly*. To test the correct functionality of the checksumming feature, we used the data sets described in III-A in conjunction with an in-house tool designed to corrupt files. The following scenarios were benchmarked for each data set:

- Measure a baseline performance with the checksum feature disabled.
- With the checksum feature turned on, transfer the data set and verify the transfer was successful.
- With the checksum feature enabled, start a transfer, corrupt the data, and verify that FCP detects the data corruption.

For each scenario, a single node with 8 processes per node was used, and the transfer time and transfer rate were measured. The results from the checksum verification transfers are shown in Figures 10 and 11.

As expected, enabling the checksum feature results in a performance penalty. The greatest penalty is observed in data set 3 at about 2X runtime increase. Interestingly, in data set 4, the mixed small files case, the performance of FCP is less impacted by enabling the checksumming. The performance penalty in that case is approximately 15%. This could be attributed to the fact that checksumming a small file is not as expensive as checksumming a larger file. In the large file scenarios, we see approximately a 200% increase in transfer time for the 100 GB case, and about a 60% in the 1 TB case. In addition, the results show about a 10% penalty when FCP is transferring and verifying a corrupted file.

The experiments show that FCP is able to accurately detect when a transfer has been corrupted. However, because the checksum operation is expensive, users should expect longer transfer times. The impact of the checksumming step could potentially be mitigated by using additional nodes. Future work will further expand the test plan to include multi-node checksumming verification tests.

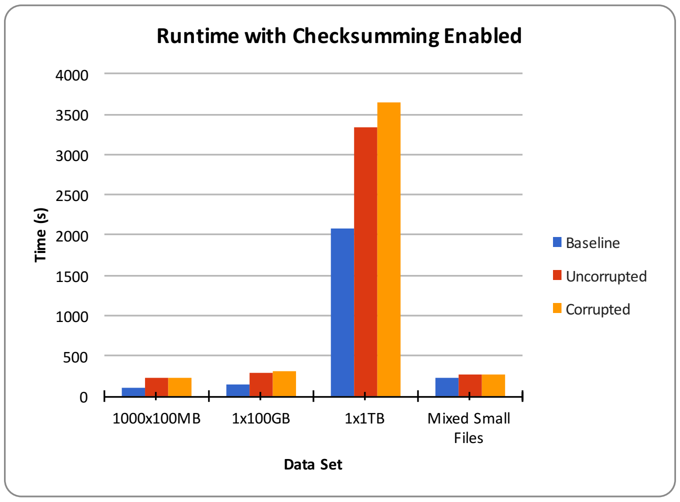

Figure 10. Average transfer time with checksumming feature enabled

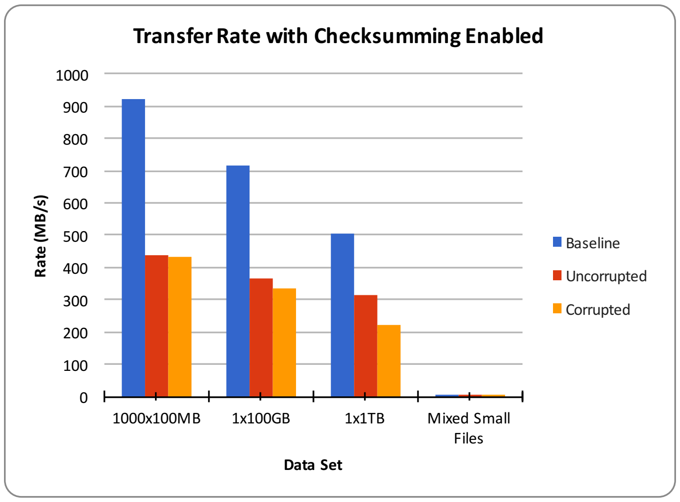

Figure 11. Average transfer rate with checksumming feature enabled

IV. Conclusion

This paper introduced a fast and scalable data copy tool known as FCP. Different from several past efforts of building parallelism on top of existing tools such as `rsync`, FCP is based on the principle of *work-stealing pattern* for parallel workload distribution and balancing. It takes advantage of ubiquitous MPI in clustered computing environments to go beyond single node limitation for scalability. FCP offers a multitude of features such as on-the- fly checksumming, checkpoint and restart that makes it a viable and attractive option for large scale data movement.

Based on the experimental results, FCP outperforms commonly used data copy tools, such as `cp`, `rsync`, and DCP, in all data sets used except the mixed small files scenario. Through detailed testing, we have verified that FCP is able to complete transfers obtaining good performance, while preserving the stripe count in Lustre file systems. The results obtained here can serve as guidance for FCP users to allow them to choose the best parameters for their transfers on Lustre file systems.

Acknowledgment

The authors would like to thank Suzanne Parete-Koon, Jason Hill and Matt Belhorn for their contributions to the design of the FCP test plan, and its execution. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

References

[1] K. Matney, S. Canon, and S. Oral, "A first look at scalable I/O in linux commands," in *Proceedings of the 9th LCI In-*

ternational Conference on High-Performance Clustered Computing, 2008.

[2] P. Z. Kolano and R. Ciotti, "High performance multi-node file copies and checksums for clustered file systems," in *LISA*, 2010.

[3] P. Z. Kolano, "Transparent optimization of parallel file system i/o via standard system tool enhancement," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1963–1970.

[4] E. Ong, E. Lusk, and W. Gropp, "Scalable unix commands for parallel processors: A high-performance implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2001, pp. 410–418.

[5] G. K. D. Matney Sr and G. Shipman, "Parallelism in system tools," in *Proceedings of the Cray User Goup conference (CUG 2011)*, 2011.

[6] G. Enterprises, "Hpss tar man page."

[7] M. Stearman, "Sequoia data migration experiences," in *Lustre User Group (LUG) 2013*, 2013.

[8] A. Loftus, "Parallel synchronization of multi-pebibyte file systems," in *Proceedings of the 2nd International Workshop on the Lustre Ecosystem: Enhancing Lustre Support for Diverse Workloads*, 2016.

[9] R. Wagner, "Lustre data mover: Because file systems are rooted trees and rsync must die," in *Lustre User Group (LUG) 2016*, 2016.

[10] J. LaFon, S. Misra, and J. Bringhurst, "On distributed file tree walk of parallel file systems," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2012, pp. 1–11.

[11] File utilities for distributed systems. [Online]. Available: http://fileutils.io/

[12] F. Wang. (2016) Fcp: A fast and scalable data copy tool. [Online]. Available: https://rawgit.com/olcf/pcircle/master/man/fcp.8.html

[13] M. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2013.

[14] W. Fokkink, *Distributed Algorithms: An Intuitive Approach*. MIT Press, 2013.

[15] F. Wang. (2016) A parallel file system profiler. [Online]. Available: https://rawgit.com/olcf/pcircle/master/man/fprof.8.html