A first look at scalable I/O in Linux commands

Ken Matney[1], Shane Canon[1], and Sarp Oral[1]

Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN, 37831

Abstract. Data created from and used by terascale and petascale applications continues to increase, but our ability to handle and manage these files is still limited by the capabilities of the standard serialized Linux command set. This paper introduces the Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) efforts towards providing parallelized and more efficient versions of the commonly used Linux commands. The design and implementation details as well as performance analysis of an in-house developed distributed parallelized version of the *cp* tool, *spdcp* is presented. Tests show that our *spdcp* utility can achieve 73 times more performance than its serialized counterpart. In addition, we introduce current work to extend this approach to other tools.

1 Introduction

Users of HPC systems with parallel file systems still rely on legacy serial tools to perform many day-to-day operations. Parallel file systems such as Lustre and GPFS are today capable of delivering hundreds of Gigabytes per second (GB/s) in aggregate bandwidth, but standard serial-based Linux utilities cannot harness this capability. For example, making a backup copy of checkpoint files, compressing output, or creating a *tar* file of results typically is carried out with standard Linux tools. Consequently, users are limited to the performance that can be sustained by a single node for these tasks. Thus, the user is not able to take advantage of the extensive capabilities of the parallel file system. The Center for Computational Sciences at Oak Ridge National Laboratory has begun working on tools to address this issues. In this paper we will describe the approach used in developing these tools and present some early performance results. We will also discuss work in progress and future plans.

2 Motivation

The National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory operates a number of the most powerful computer systems used for open research [1] [2]. The flagship system, Jaguar, is a Cray XT4 with over 20,000 cores and 40 TB of memory. It is configured with a parallel filesystem with nearly 1 PB of disk capacity and over 40 GB/s of file system bandwidth. The system uses the Lustre file system [3]. The Lustre file system aggregates distributed storage units into one logical file system. Files are striped transparently by the file system across multiple storage targets to aggregate both capacity and bandwidth. As a result, users can achieve high throughput to storage for critical I/O operations such as writing or reading a checkpoint file. Applications such as the Gryokinetic Tokamak Code (GTC) have demonstrated over 10 GB/s of aggregate bandwidth. However, many day to day operations fail to achieve even a small fraction of this capability because the underlying utilities such as *cp*, *bzip2*, and *tar* must be confined to a single node.

A fully striped file (a single file striped across all storage targets) can be written at over 20 GB/s on a Jaguar file system. However, using *cp* to copy this file between two local Lustre file systems might only sustain 200 MB/s. As a result, while it might have taken around 50 seconds to create a 1 TB checkpoint file, it would take more than 80 minutes to make a copy of the file. The user would likely encounter similar problems when compressing and uncompressing files, creating a *tar* file, or other operations that rely on serial-based tools.

From discussions with our users, it has become evident that these bottlenecks in day-to-day operations are the source of some very real barriers to productivity and that there was a clear and growing need for parallel versions of these common tools. Furthermore, if a generalized framework could be created for parallelizing many of these common tasks, it could be extended to other use cases. Fortunately, many of these tools lend themselves to parallelization with very clear ways to decompose the the input domain. We chose to focus on those utilities that would quickly provide the most benefit to our user community.

3 Approach

There are some limiting factors in parallelizing Linux commands. First, the source data must be randomly accessible. Data from a checkpoint file in a file system is an example, while data from a socket or pipe is not. Second, the dataset must reside on multiple independent physical devices. Since performance improvement is based on parallel I/O, accessing multiple independent physical devices concurrently increases the achievable aggregate bandwidth.

There are two types of parallelization that can be exploited. First, there is the parallelism associated with processing multiple files simultaneously. Second, there is the parallelism associated with using multiple processors to map cooperatively the data of a single file. Obviously, the gain from the use of the latter is dependent on how well the file has been distributed across multiple servers and if the work can be easily decomposed.

Another critical factor to performance is the size of the data buffers that are employed. Like most file systems, parallel file systems prefer large buffers. For example, Lustre file system achieves best performance with 1 MB buffers. Parallel file systems are typically more sensitive to buffer sizes since these file systems rely on networks to transport data from the storage servers to the clients. Furthermore, byte range locking is typically used to insure consistency. Larger buffers require less overhead in managing these locks, resulting in better performance.

Since the details of how to decompose the work depends on the specific command targeted, each command has to be examined individually. However, the basis of algorithms for performing I/O in parallel remains the same. In addition, a method for communicating between the various participating processors must be established. While system specific low-level protocols such as Portals on a Cray XT or Verbs on an InfiniBand cluster might provide the best performance, they lack portability. Therefore, MPI is used to ensure portability while sacrificing some degree of performance. Our parallelized utilities can easily be ported and compiled for most parallel systems.

While a Lustre file system was used in the development and testing of the initial implementation, these techniques can be applied to other parallel file systems. In certain cases, Lustre-specific calls to query the layout of the data are used to improve efficiency. However, good performance and efficiency can still be achieved without these Lustre specific calls.

Lustre is a POSIX compliant, object-based file system composed of three components:

MetaData Server A single MetaData Server (MDS) per filesystem that stores and manages Lustre file metadata, such as filenames, directories, permissions, striping pattern, and file layout.

Object Storage Target One or more Object Storage Targets (OSTs) are block devices that actually store the file data. OSTs are managed by the Object Storage Servers (OSSs). At any given configuration there can be one or more OSTs controlled by a given OSS.

Client Client(s) access and use the data. Lustre provides all clients with standard POSIX semantics and concurrent read and write access to the files in the filesystem.

Currently, Lustre uses an enhanched version of *ext3* file system on MDS and OSTs to store Lustre file data. Lustre achieves high read and write performance by distributing the file data over multiple OSTs. This is known as *striping*. The number of OSTs that a file is striped across is known as *stripe count*. With striping, the maximum file size is not limited by the size of a single block device, and the aggregate I/O bandwidth scales with the number of OSSs. A more detailed description of Lustre file system is beyond the scope of this document. Interested readers are encouraged to read [3].

The Linux *cp* utility was selected as the first tool for parallelization, as it is a commonly used function, and the decomposition is simple since the mapping of input data to output data is direct. Consequently, there are almost no dependencies between the individual threads carrying out the copy. The parallel version of *cp* is termed *spdcp* for streaming parallel distributed *cp*. Currently, *spdcp* only works on Lustre file system, but our future plans involve extending it to other file systems, such as GPFS. We are in the process of publicly releasing the *spdcp* source code under an open source license.

4 Prototype for a Parallel Distributed Copy

In preparing the prototype, there are two possible ways in which to proceed. The first is to take the source for GNU cp and modify it. The second is to write the function from scratch. It is unlikely that a patch to rework *cp* could make it into the mainstream given the amount of changes that are needed to parallelize it. Therefore we chose to implement a new copy command starting from scratch. However, we tried to preserve many of the command-line options and general behaviour of *cp*.

The overall design consists of several components. A diagram of the components is shown in Fig. 1. The base component is the "launch process" which invokes the MPI-based components. In addition to launching the MPI job, it also performs a number of other operations, as described below. The "rank 0 process" in the MPI job is designated as a master. It is responsible for managing the work. A number of slave processes are responsible for copying the file data from source to target. How this work is distributed across the slave nodes is described below.

There are a number of design considerations to be made. First, the prototype needs to be aware of the parallel characteristics of source file(s). It needs to be able to acquire these attributes for source file(s) and set these on target file(s). Next, it needs to be aware of the available resources. That is to say, if the Linux command is not run within the context of a batch job, it needs to spawn a batch job and request appropriate resources.

Another design choice was to decide how meta-data operations would be decomposed. Currently, Lustre employs a single Metadata Server (MDS) for a file system. Consequently, having multiple clients interact with the MDS may not improve performance and may even reduce it. Therefore, the prototype performs many of the meta-data specific tasks in the launch process. For example, the Linux command that launches the MPI job, performs the search for source file(s), acquires both Linux meta-data and Lustre meta-data for these, and sends all of this information to MPI master via a pipe. Furthermore, this process creates the target directory hierarchy before sending the list of files to the MPI based components. This avoids duplication of effort and race conditions, e.g., multiple processes requesting creation of the same target directory.

Finally, the launch process handles correctly setting timestamps on target directories when needed. The advantage to this strategy may not be obvious. Since the launch process has already has traversed the source hierarchy, it only needs to retain a list of the directories and their meta-data. The launch process must allow the MPI job to complete so that it can ensure any updates to the access time are not overwritten by any of the slave processes.

The prototype employs a variable strategy for decomposing work to determine the number of clients to employ in copying each file. It makes this determination based on a performance prediction model of the dataset.

For small files or files with only a single Lustre stripe, the entire operation is carried out by a single slave node. For files that are distributed over multiple stripes, the work is distributed across a subset of processes. The master process waits until the appropriate number of slave processes are available and then schedules the copy operation across the subset. A "team leader" is selected within the subset. The team leader ensures that the target file has been created with the appropriate Lustre meta-data parameters, such as the stripe count and stripe width.

If the typical file meta-data (modification date, etc.) is to be an exact copy of the original, then all of the team members report to the team leader that they have completed all of their I/O requests. Otherwise, the team members report directly back to the master node for their next assignment. Likewise, after the team members report back to their team leader for completion notification, they await further instructions from the master node. The team leader reports to the master node to indicate that the copy has completed and the team members are are ready for the next assignment.

The techniques described above allow the load on the target OSTs to be managed. By instructing the prototype command to use only a specified number of processors for the parallel part, in conjunction with specifying the buffer width, we can ensure that the ideal number of clients are participating in the copy operation for a given file. Contention can still arise from other copy threads having stripes that overlap on the same OST. However, preventing this would increase the complexity and likely provide only marginal improvements in performance.

The prototype implementation of *spdcp* strives to mimic the standard *cp* command that users are familiar with. The intent is to create a drop in replacement for *cp* that users can easily employ in their existing scripts.

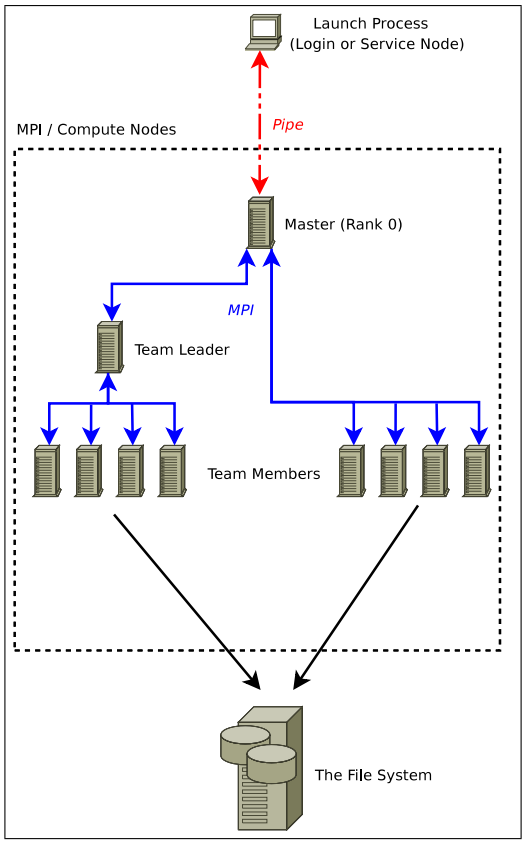

Fig. 1. Diagram of the components used in the parallel distributed copy. All compute nodes access the file system. The number of team members used for a source input file depends on the source file Lustre stripe pattern.

However, some additional command-line options have been added to control aspects of the parallel execution of the utility. For example, there are options to control the number of tasks and buffer sizes. Furthermore, since our environment requires submitting a batch job to run a parallel job, the utility can transparently submit itself to the batch queue. Consequently there are options related to the batch submission as well. A sample execution is shown in Fig. 2.

5 Performance

A series of performance measurements were carried out on *spdcp* tool. Three reference data sets were created in order to measure the performance of the *spdcp* tool. The first data set (*workload 1*) consisted of 2400 files, each of size 100 MB. This is representative of files typically created by a modeling application which later are analyzed or visualized. The second data set (*workload 2*) consisted of 10 files, each of size 24000 MB. This is representative of a checkpoint which is done to a shared file. The third data set (*workload 3*) consisted of 1200 files of size 100 MB and 5 files of size 24,000 MB. This was done to demonstrate the ability to efficiently copy a non-uniform data set.

The Linux *c*p command was used to establish baseline performance. Then, we evaluated the performance at various scales in order to understand the scaling behavior for the prototype. These measurements were performed on a 3500 socket Cray XT3 system against its local Lustre file system. The Lustre file system consisted of 80 OSTs served by 20 Object Storage Servers (OSSs). The backend storage was provided by 10 couplets of DDN 8500 [10]. This file system has been measured using the IOR [11] benchmark to sustain over 10 GB/s on a file-per-process run.

```
spdcp -s 16 -r /source/directory/ /target/directory/
```

```
spdcp -h
Usage:  spdcp [options] SRC    DEST

            or

        spdcp [options] SRC... DIRECTORY

        Copy file SRC to file DEST or list of files SRC... to directory
        DIRECTORY, replicating Lustre stripe information where possible.
        Copy is performed in parallel by distributed clients using MPI
        message passing for synchronization and control.  When compute
        node resources are accessible only in batch mode, command will
        stage job and retain control until job finishes.  The following
        options offer control over command:

            -h        Print this message (disables copy)
            -V        Print command and agent versions (disables copy)
            -d        Use dummy form (disables copy, prints targets)
            -v        Increase verbosity level (maximum 2)
            -p        Preserve mode, ownership, and timestamps
            -r, -R    Copy recursively
            -c        Reduce OST count at destination to source usage
            -n        Do not offset initial OST at destination
            -b  {F}   Increase I/O request size by a factor of F
            -s  {M}   Employ M parallel clients
            -A  {P}   If spawning batch job, charge run to project, P
            -w  {T}   If spawning batch job, limit walltime to T seconds
            -q  {Q}   If spawning batch job, direct to batch queue, Q
```

Fig. 2. Sample execution of *spdcp* (*Top*). The total number of clients requested is identified by the *-s* switch. Note that, this number also includes the "master (or rank 0) node." The *spdcp* help menu (*Bottom*).

As can be seen in Fig. 3, *spdcp* achieves good parallel speedup. The data exhibit a certain amount of variation because they were obtained during the course of normal production operation of the Cray XT3. It should be noted that the stock Linux *cp* utility achieved 324 MB/s, 126 MB/s, and 177 MB/s for *workload 1*, *workload 2*, and *workload 3*, respectively.

In terms of peak performance, as can be seen in Fig. 3, the *workload 2* achieves the best performance with *spdcp*, at around 9300 MB/s. This is a 73x performance increase compared to the Linux *cp* utility. The peak performance is 7300 MB/s for *workload 1*. This is 22x speed up compared to the Linux *cp* utility. For workload 3 the peak is at approximately 9100 MB/s; a 51x speed up over the Linux *cp* utility.

Also, as can be seen in Figure 3, the peak performance is obtained at 160 to 256 clients. However, from a practical point of view, the scaling of performance levels off at around 100 clients. This makes sense given that the number of clients and OSTs are roughly equivalent. Consequently, the OSTs have nearly reached their peak bandwidth. This is further demonstrated by the fact that the aggregate bandwidth is 73% to 93% of the peak bandwidth as measured by IOR.

6 On-going work

The parallel implementation of the copy utility is just the first step in a broader initiative to create a suite of parallelized tools. Towards this end, we have started to create a framework to generalize the approaches used

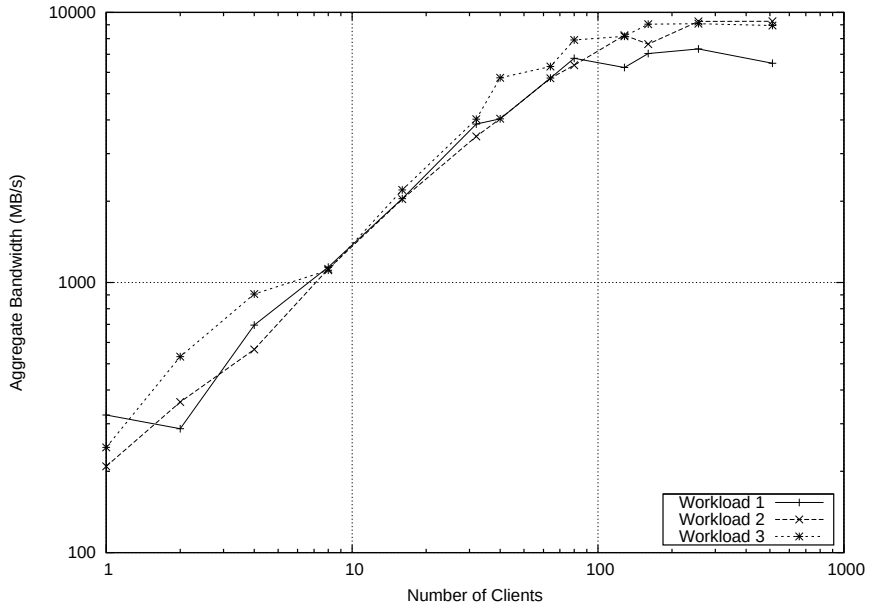

Fig. 3. *spdcp* performance for clients up to 512. The *workload 1* is composed of large files, *workload 2* is composed of small files, and *workload 3* is a mix of large and small files. The stock Linux *cp* utility achieved 324 MB/s, 126 MB/s, and 177 MB/s for *workload 1*, *workload 2*, and *workload 3*, respectively (not shown on the figure).

in *spdcp* so that they can easily be applied to other common utilities. The *spdcp* utility does not currently use the framework, but may be re-implemented using the framework in the near future. This framework, which is called *spdframe*, has already been used for compression and decompression of *bzip2* files [12]. This presents slightly more difficulty than the copy tool, as the decomposition for decompression is more difficult. Preliminary tests show that our *bzip2* implementation is promising and under right configurations (*e.g.* 64 processors with a 20 MB file) it can achieve 15 times more performance for compression compared to its serialized version on. Future work will focus on applying the framework to *tar* and other common file based utilities.

While we are focusing on applying the framework to common tools, the framework lends itself to other uses as well. The framework provides an easy way for users to apply a function over multiple files in parallel. So, for example, a user could easily apply the framework to perform a parallel grep on a set of files.

7 Related work

Increasing the performance of common Linux utilities gathered some attention from the research community over the years. William Gropp and Ewing Lusk [4] have first realized the limitations of legacy serial UNIX utilities in parallel environments. They introduced several parallel versions of commonly used UNIX utilities with parallel *rsh* as the underlying parallel synchronization and communication mechanism. As a follow up to their work, Emil Ong, Ewing Lusk, and William Gropp developed the MPI-based version of their parallelized UNIX utilities [5]. However, there is a clear distinction between our goal and theirs. The target for Gropp and Lusk was increase efficiency by executing the same command with the same argument list and parameters in parallel over multiple independent nodes with independent operating systems and file systems. In many aspects, they have implemented SIMD-like versions of the common UNIX tools. However, our approach departs from theirs as our goal was to increase the efficiency of a single execution a given Linux utility by parallelizing and distributing its workload over multiple worker/compute nodes, all sharing a common file system, but independent OSes.

Jeff Gilchrist and Aysegul Cuhadar [7] introduced two parallelized versions of BWT-based *bzip2* nblock-sorting file compressor, namely *pbzip2* and *mpibzip2*. The *pbzip2* is a thread-parallel version of *bzip2* for use

on shared memory machines. It produces compatible but larger archives compared to the original *bzip2*. The *mpibzip2* is an MPI-based parallel implementation of the *bzip2* block-sorting file compressor for clusters. The *bzip2smp* program is another parallelized version of the *bzip2* compressor [8]. It is specifically targeted for SMP systems. It is very cache-dependant and does not perform well with hyperthreaded systems. It is similar to *pbzip2* in nature, but unlike *pbzip2*, *bzip2smp* supports compression from *stdin*.

Conclusion

Increasing parallelism in file systems pave the way for processing larger datasets in shorter times. However, while capabilities for generating larger datasets are constantly increasing, our tools for handling and managing such files, still remain serial and limited in performance.

The Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) has started an initiative for providing high-performance, parallel versions of commonly used Linux commands. The *cp* command was our starting point. We have developed and implemented a MPI-based batch-processing capable parallel version of the standard *cp* command. Tests show that, our version can achieve 73 times more performance over its standard serialized counterpart.

Also, this paper introduces our efforts towards developing a parallelized distributed version of the *bzip2* command. The implementation follows a framework, which if successful, will be used for developing and parallelizing other Linux commands.

Acknowledgments

The authors would like to thank the staff and colleagues who have contributed material to this paper. Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

About the Authors

Ken Matney is a researcher in the Technology Integration Group which is part of the National Center for Computational Sciences at Oak Ridge National Lab. He can be reached by E-Mail: matneykdsr@ornl.gov. Shane Canon is the Group Leader for Technology Integration Team. He can be reached by E-Mail: canonrs@ornl.gov. Sarp Oral is a researcher in the Technology Integration Group which is part of the National Center for Computational Sciences at Oak Ridge National Lab. He can be reached by E-Mail: oralhs@ornl.gov.

References

1. National Center for Computational Sciences. Web Page http://nccs.gov.
2. Top500 Supercomputer sites - November 2007 list. Web Page http://www.top500.org/list/2007/11.
3. Cluster File Systems, Inc. Lustre manual. Web page. http://www.lustre.org/manual.html.
4. William Gropp and Ewing L. Lusk. Scalable Unix tools on parallel processors In *Proceedings of the Scalable High-Performance Computing Conference*, pp. 56-62, 1994.
5. Emil Ong, Ewing L. Lusk, and William Gropp. Scalable Unix Commands for Parallel Processors: A High-Performance Implementation In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing*, pp. 410-418, 2001.
6. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm *Technical Report 124, Digital Systems Research Center*, 1994.
7. Jeff Gilchrist and Aysegul Cuhadar. Parallel Lossless Data Compression Based on the Burrows-Wheeler Transform In *21st International Conference on Advanced Networking and Applications (AINA '07)*, pp. 877-884, 2007.
8. Web Page http://bzip2smp.sourceforge.net/
9. R. S. Canon and H. Sarp Oral. A Center-wide File System using Lustre. In *CUG Proceedings*, 2006.
10. DataDirect Networks. Web Page http://datadirectnetworks.com/
11. Hedges *et al* Parallel file system testing for the lunatic fringe: the care and feeding of restless I/O power users In *IEEE Mass Storage Systems and Technologies Proceedings*, 2005
12. Julian Seward. The bzip2 and libbzip2 official homepage. Web Page http://sources.redhat.com/bzip2