Parallel Synchronization of Multi-Pebibyte File Systems

Andy Loftus
National Center for Supercomputing Applications
University of Illinois
Urbana, IL 61820
Email: aloftus@illinois.edu

***Abstract*—When challenged to find a way to migrate an entire file system onto new hardware while maximizing availability and ensuring exact data/metadata duplication, NCSA found that existing file copy tools were insufficient to accomplish the task in a timely manner. So they set out to create a new tool. One that would scale to the limits of the file system and provide a robust interface to adjust to the dynamic needs of the cluster. The resulting tool, Psync (Parallel Sync), effectively manages numerous syncs running in parallel. As a synchronization tool, it correctly duplicates both contents and metadata of every inode from the source file system to the target file system including special files such as soft and hard links, and additional metadata such as ACLs and Lustre stripe information. Psync supports dynamic scalability features to add and remove worker nodes on the fly and robust management capabilities that allow an entire run to be paused and unpaused or stopped and restarted. Psync has been run successfully on hundreds of nodes with multiple processes each. Spreading the load of synchronization across thousands of processes results in relatively short runtimes for an initial copy and even shorter runtimes for successive runs (to update a previous sync with new and changed files). This paper will present the overall design of Psync and its use as a general purpose tool for copying large amounts of data quickly and efficiently.**

I. Introduction

As file systems grow into the pebibyte range, the time to manage the data increases as well. Tasks that used to take minutes now take hours and those that took hours now take days. Building file systems of this size requires parallelism and likewise the tools to manage them need to be parallel in order to finish in a reasonable amount of time. In terms of runtime, copying an entire file system is certainly the most time consuming management task. Accomplishing this in under 24 hours is possible, in theory, if enough parallelism is used and if the filesystem can handle the load. However, standard file synchronization tools are not parallel, hence the need for new tools to manage these tasks. This paper introduces a tool that can perform a file system synchronization in parallel, achieving a copy of multiple pebibytes of data in a relatively short period of time (determined by the scale of parallelism available).

A. Conventions

Given some possible confusion about the spelling of "filesystem" vs. "file system", a note about the difference is warranted. The spelling "filesystem", one compound word, refers to the technical implementation of the underlying data storage mechanism. For example: ext4, lustre, and gpfs are all examples of "filesystems". The spelling, "file system", as two words, refers to the actual data that is stored and its relevant directory structure.

Being that this paper is about file syncronization, there is always a source file and a target file. To reduce wordiness, the word "source" may be dropped from "source file" when the meaning is clear. If the word "file" is used without specifying whether it is the source or the target, assume the source file is being referenced.

II. Motivation

The driving factor for the creation of psync arose from the need to update firmware on the filesystem hardware serving the Blue Waters supercomputer at NCSA. The nature of the update would require a destructive reformat of the filesystem. Additionally, the update procedure would take several days to complete, which is an unacceptable length of time to let the entire machine sit idle. The resulting plan was to synchronize all the data from the file system to a second location, so that the supercomputer could continue to operate while the hardware was updated. This process would then be repeated to move the data back and repeated again on the other file systems in a rolling fashion till all the hardware was updated. Given the high cost of an idle supercomputer, the amount of downtime must also be minimized. These two tasks, synchronizing data and minimizing downtime, drive the goals of psync.

III. Design

A. Synchronizing Data

Goal one, synchronizing data between two file systems, is already solved by rsync. However, rsync does not meet the second goal of minimizing downtime because it is single threaded and would take multiple weeks to complete a sync of this size. One solution is to run multiple rsyncs in parallel, though care must be taken to ensure each rsync operates on non-overlapping parts of the file system. Breaking the file system into useful chunks usually requires some forehand knowledge of the layout of the file system, which makes this approach difficult and unique to each file system. Rsync also suffers in performance when it encounters a directory containing millions of files and also for a directory with many

large files. Psync addresses these issues by running one rsync per file. This may seem counter-productive given the expense of forking a new rsync process for each file. However, the startup time is constant no matter how many processes are run in parallel. The rsync startup cost is inversely proportional to the number of workers while the files processed per unit of time is directly proportional to the number of workers.

B. Minimizing Downtime

Goal two, minimizing downtime, is addressed in two ways: by fast copying and minimizing the total amount of data to copy. Fast data copy is accomplished by copying files in parallel, which includes walking the directory tree, in parallel, to find the files that need to be copied. Copying files in parallel is discussed above. Walking the directory tree in parallel is accomplished by operating on each directory independently, without recursing. Starting with the top level directory, every sub-directory becomes a new directory scan that will be handled by some other worker in parallel.

The second part of minimizing downtime is to minimize the amount of data to copy. Copying less data should, quite obviously, complete in less time and thus require a shorter downtime. This is realized by copying as much data as possible ahead of time while the file system is still live. Similar to rsync, the first psync run will copy every file since the target is empty, but successive runs will copy only new and/or changed files. Also during successive syncs, files deleted from the source will be deleted from the target. Finally, during the actual downtime, the filesystem is quiesced and a final synchronization will bring the target file system into perfect sync with the source file system.

C. Distributed Task Queue

With the overall job of file system synchronization broken down into small, individual tasks, the remaining chore is to run all the tasks. This is the role of the task queue. The task queue is a centralized data structure that holds the individual tasks and provides an interface to pull tasks for working and push tasks for future work. The task queue also ensures that each task is executed only once. Psync defines workers that query the task queue to get work. Some tasks may generate new tasks, such as when a "directory scan" discovers new sub-directories. New tasks are pushed onto the queue where they will eventually be pulled and executed by a worker. At the start of psync, a single task is pushed onto the queue, containing the top level source and target directories. A single worker will execute the task. As the top level directory is scanned, each file will be pushed onto the queue as a new "file sync" task. If the top level directory contains subdirectories, each subdirectory will be pushed onto the queue as a new "directory scan" task. These two task types, "file sync" and "directory sync", are the core part of psync. Any worker can perform either task. This allows both parts of the synchronization, walking the directory tree and file sync, to happen in parallel.

IV. Design Challenges

Designing a fast file synchronization tool requires a focus on correctness and performance. Each are equally important in making the development effort worthwhile.

A. Correctness

One of the requirements for the data move on the Blue Waters machine was to preserve hardlinks. The algorithm to do this is as follows:

1) Get the source file FID.
2) Check the target filesystem for a file (in a pre-determined temp directory) with a name matching the source file FID.
3) If a temp file does not exist, copy the source file to the temp file.
4) Create a hardlink on the target filesystem linking the target filename to the temp file.

Following this algorithm, only the first instance of a file with multiple hard links will be physically copied to the target filesystem. Any additional hard links found on the source file system will only be linked to the appropriate name on the target file system. It should be noted that there is no initial check if the file has multiple hard links. The algorithm is applied to all files. The reason for this is, when syncing a live file system, additional hard links may be created at any point in time. If a file was copied without first creating a temp file, and then the source file had additional names linked to it, the first file copy on the target file system would point to a different inode than the rest of the link names that were copied using the temp file method. It is also important to note that the temp file copy algorithm described above is almost exactly how rsync copies files. The difference is that rsync copies to a random temp file and, after linking the target filename, deletes the temp file. Since psync is already managing the temp file creation, rsync is invoked with the "in-place" option.

The advantage to using rsync for file and directory copies is that rsync already has the capability to sync the associated metadata. When operating on Lustre, psync retrieves the stripe count of the source item (file or directory) and pre-creates the target item with the appropriate stripe count.

The final step of correctly copying files is to verify that the target file exactly matches the source file. Checksums are calculated for both the source and the target files using md5. Psync logs matching checksums with INFO severity and mismatched checksums with WARNING severity.

B. Rsync Performance

Most of the performance gains are a direct result of operating in parallel. A technical detail that bears mention is the performance of rsync on large files. Rsync uses relatively small block sizes for copying files, resulting in exponentially increasing transfer times as file size increases. To deal with this, files larger than 512 MiB are copied using the standard linux tool 'dd' with a 4 MiB transfer block size. The amount of time saved during file copy far outweighs the added time cost of the extra process startup.

V. Implementation

Psync is written in Python and uses Celery[1] for its task queue. Celery provides a framework for creating the workers and tasks. Celery also handles the details of interacting with the task queue. The task queue itself is an AMQP broker, and while Celery supports several, RabbitMQ [2] is the only fully AMQP compliant broker. Redis [3] is used as a centralized log system.

VI. Task Design

There are currently four tasks defined in psync: *syncdir*, *syncfile*, *synchardlink*, and *syncdirmeta*. The two primary tasks are *syncdir* and *syncfile*, which do the majority of the work. The latter two tasks handle special cases. Each task will be discussed in turn.

A. Syncdir

Syncdir operates as follows:

1) List all entries in the source directory
2) List all entries in the target directory
3) Delete any target entries that no longer exist in the source directory
4) Process all entries of type dir
    a) Create target subdirectory (if needed)
    b) Enqueue subdirectory as a new *syncdir* task
5) Process all entries of type file (including special files)
    a) Enqueue each entry as a new *syncfile* task
6) Enqueue directory as a new *syncdirmeta* task.

B. Syncfile

The *syncfile* task uses the following algorithm:

1) If target ctime is newer than source ctime:
    a) Return
2) If sizes or mtimes don't match:
    a) If file is large, invoke dd
    b) Invoke rsync
    c) Return
3) If any other metadata values dont match
    a) Invoke rsync

C. Synchardlink

The *synchardlink* task is identical in function to the *syncfile* task. It differs only in that instances of this task get posted to a special queue. The need for a separate queue for hardlinks is to prevent a race condition from occuring when a new hardlink is first being created on the target filesystem. If multiple *syncfile* tasks, working on different filenames sharing the same inode, start close enough in time to each other, each task will attempt to create the tempfile. One will succeed and the others will fail. By processing hardlinks serially, this race condition is avoided.

D. Syncdirmeta

The purpose of the *syncdirmeta* task is to sync just the metadata of a directory inode. Similar to *synchardlink*, this task is also processed on a separate queue. The need for a special queue is to ensure that directory metadata is correct after the directory stops changing. Each time a file or subdirectory is added (or removed) from a parent directory, the parent directory's inode changes. Since the creation of the files and subdirectories happens in other tasks in parallel, the mtime of the parent directory will be wrong unless the directory entry is sync'd last. The queue that holds *syncdirmeta* tasks is processed last to ensure that directories have the correct mtime at the end of the sync.

VII. Performance Analysis

Psync was tested on the Blue Waters supercomputer using Cray XE6 compute nodes. Each node contains two AMD Interlagos model 6276 CPUs running at 2.3 GHz and 64 GiB of RAM. The compute nodes connect to LNET routers via the Cray Gemini high speed network, which runs at an average speed of 6 GiB/s. The LNET routers use QDR inifiniband to connect to Cray Sonexion 1600 filesystem appliances.

During a recent outage, a test psync run copied the Home file system to the Projects file system. The test started out with a small number of nodes and processes and ramped up to a final count of 2384 processes on 398 nodes.[1]

Since the test was run during an outage, psync was the only activity on the file system. Using completed inodes as measure of progress, the sync moved 2.25% of the file system per hour, resulting in a 44 hour runtime. In terms of data transfer rate, this equates to 1.5 GiBps and 757 inodes per second. The latter of these two measurements is more useful since the primary bottleneck is the single MDS server keeping up with metadata requests. This is shown in Figure 1., where the metadata transactions per second from psync is fully within the expected maximum range for the MDS (as calculated by mdtest during acceptance testing.) Another measure of MDS load is the disk %util reported by the *sar* command. For this psync run, the disks on the MDS were consistently at or above 98% utilized.

During a successive psync run, when an intial copy already exists on the target, far fewer nodes are required to push the MDS server to it's peak load in terms of transactions per second and disk %-utilization. Continuing the test run from earlier, a successive psync was run, again from the Home file system to the Projects file system. Due to time constraints, the re-sync was not able to run during the outage. The re-sync was run with 800 processes on 100 nodes and was run on a live filesystem. Using metrics of *avg file create time* and *avg ls time*, it was determined that more processes would degrade interactive file system response time too much. Even at this small (relative to the initial sync) number of processes, the entire re-sync took 16 hours, which is 6.25% per hour, or

[1]The first 200 nodes ran 4 processes each and the last 198 nodes ran 8 processes per node.

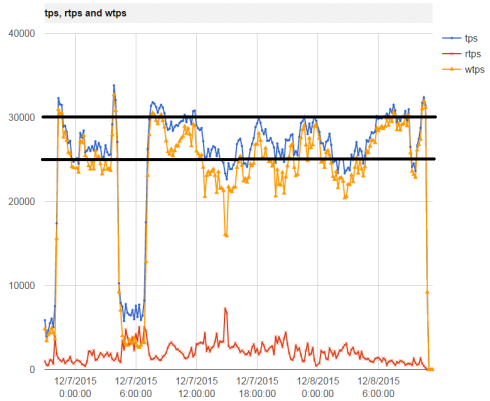

Fig. 1. The two horizontal lines show the range of maximum transactions per second (25K-30K) for file creates on the MDS as calculated using mdtest during acceptance testing for Blue Waters. The graph labels are "wtps" (write transactions per second), "rtps" (read transactions per second), and "tps" (combined transactions per second).

2083 inodes per second. Remember that most of those inodes are simply being checked and very few are actually new or changed files that require an actual data transfer.

VIII. Limitations

Psync was initially written to synchronize an entire file system and as such requires to be run with full superuser privileges. At NCSA, the normal job schedulers do not allow jobs to run as root, which is why psync was designed to use its own task queue for scheduling work. Therefore, setting up and running psync requires shell access to the nodes it will run on. This is usually outside of the normal work flow for the typical cluster. When using cluster compute nodes they should be removed from the normal scheduling process to prevent jobs from running on those nodes.

Psync is also still very specific to Lustre. While all filesystem interaction is provided by an external module, there are no modules currently written to support filesystems other than Lustre.

IX. Future Work

Psync is still a very young project and there are many areas where it could be improved and extended. What follows is a short, but not exhaustive list of possible areas for improvement.

A potential performance enhancement would be to interact with the filesystem using efficient, low level libraries instead of command line tools. On Lustre, invoking the command line tool *lfs getstripe* causes the specified file to be opened twice, which is expensive since it involves communication to both the MDS and any OSSs involved. Getting the stripe information for a file or directory without requiring communication with the OSS or additional MDS messages could have far reaching impact since this is done for every file that is copied and every directory that has changed.

As touched upon in the Limitations section, psync could be adapted to function on other filesystem types. The only requirement to do so would be to provide a module that provides functions for interacting with the filesystem. Two examples of additional filesystem modules are a generic Posix module and a GPFS module.

One of the pain points of using Lustre is choosing a useful stripe count for files. Psync could very easily be adjusted to review source file stripe count and write the new, target file with a different stripe count. The new stripe count would be based on a size-per-stripe threshold passed as a customizable parameter. Additionally, using the same file system as both the source and the target would allow psync to be used as a general restriping tool.

As of the current writing of psync, the checksums of the source and target files are calculated as part of the *syncfile* task. However, this has a slight potential for error since the data read for either file could come from the worker nodes local cache. If this happens, the data that was actually written to disk is never really checked. A better approach is to enqueue the checksum as an additional task so that another node will calculate the checksums. With a different node executing the checksum task, that node will not have any data cached and will get the actual file data that was written to disk.

A second possible improvement resulting from performing checksums as their own task is the possibility for improved performance by isolating checksum tasks to machines best suited to the task. Checksum calculation tasks could be run on machines with fast CPU's or even GPU's while file transfer tasks could be run on machines with slower CPU's but fast network and fast filesystem access.

Acknowledgment

This work was funded by the National Science Foundation in conjunction with the Blue Waters project and NCSA. Special thanks go to Alex Parga (NCSA) for the hardlink solution and countless other contributions. Andy Loftus would also like to thank Chad Kerner (NCSA) and David McMillan (CRAY).

References

[1] Celery: Distributed Task Queue (http://www.celeryproject.org/)
[2] RabbitMQ - Messaging that just works (https://www.rabbitmq.com/)
[3] Redis (http://redis.io/)