High Performance Multi-Node File Copies and Checksums for Clustered File Systems*

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

Abstract

Mcp and msum are drop-in replacements for the standard cp and md5sum programs that utilize multiple types of parallelism and other optimizations to achieve maximum copy and checksum performance on clustered file systems. Multi-threading is used to ensure that nodes are kept as busy as possible. Read/write parallelism allows individual operations of a single copy to be overlapped using asynchronous I/O. Multi-node cooperation allows different nodes to take part in the same copy/checksum. Split file processing allows multiple threads to operate concurrently on the same file. Finally, hash trees allow inherently serial checksums to be performed in parallel. This paper presents the design of mcp and msum and detailed performance numbers for each implemented optimization. It will be shown how mcp improves cp performance over 27x, msum improves md5sum performance almost 19x, and the combination of mcp and msum improves verified copies via cp and md5sum by almost 22x.

1 Introduction

Copies between local file systems are a daily activity. Files are constantly being moved to locations accessible by systems with different functions and/or storage limits, being backed up and restored, or being moved due to upgraded and/or replaced hardware. Hence, maximizing the performance of copies as well as checksums that ensure the integrity of copies is desirable to minimize the turnaround time of user and administrator activities. Modern parallel file systems provide very high performance for such operations using a variety of techniques such as striping files across multiple disks to increase aggregate I/O bandwidth and spreading disks across multiple servers to increase aggregate interconnect bandwidth.

To achieve peak performance from such systems, it is typically necessary to utilize multiple concurrent readers/writers from multiple systems to overcome various single-system limitations such as number of processors and network bandwidth. The standard cp and md5sum tools of GNU coreutils [11] found on every modern Unix/Linux system, however, utilize a single execution thread on a single CPU core of a single system, hence cannot take full advantage of the increased performance of clustered file system.

This paper describes mcp and msum, which are drop-in replacements for cp and md5sum that utilize multiple types of parallelism to achieve maximum copy and checksum performance on clustered file systems. Multi-threading is used to ensure that nodes are kept as busy as possible. Read/write parallelism allows individual operations of a single copy to be overlapped using asynchronous I/O. Multi-node cooperation allows different nodes to take part in the same copy/checksum. Split file processing allows multiple threads to operate concurrently on the same file. Finally, hash trees allow inherently serial checksums to be performed in parallel.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the test environment used to obtain performance numbers. Section 4 discusses the various optimization strategies employed for file copies. Section 5 details the additional optimizations employed for file checksums. Section 6 describes how adding checksum capabilities to file copies decreases the cost of integrity-verified copies. Finally, Section 7 presents conclusions and related work.

2 Related Work

There are a variety of efforts related to the problem addressed by this paper. SGI ships a multi-threaded copy program called cxfscp [25] with their CXFS file system [27] that supports direct I/O and achieves significant performance gains over cp on shared-memory systems, but

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

offers minimal benefit on cluster architectures. Streaming parallel distributed cp (spdcp) [17] 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.

Ong et al. [20] describe the parallelization of cp and other utilities using MPI. The cp command described, however, was designed to transfer the same file to many nodes as opposed to mcp, which was designed to allow many nodes to take part in the transfer of the same file. Desai et al. [9] use a similar strategy to create a parallel rsync utility that can synchronize files across many nodes at once. Peer-to-peer file sharing protocols such as BitTorrent [6] utilize multiple data streams for a single file to maximize network utilization from low bandwidth sources and support parallel hashing where the integrity of each piece may be verified independently.

High performance remote file transfer protocols such as bbFTP [3] and GridFTP [1] use multiple data streams for portions of the same file to overcome single stream TCP performance limitations. GridFTP additionally supports striped many-to-many transfers to aggregate network and I/O bandwidth. HPN-SSH [22] is a high performance version of SSH that achieves significant speedups using dynamically adjusted TCP receive windows. In addition, HPN-SSH incorporates a multi-threaded version of the AES counter mode cipher that increases performance further by parallelizing MAC and cipher operations on both the sender and receiver.

There are several related multi-threaded programs for the Windows operating systems. RichCopy [14] 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 [15] 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. HP-UX MD5 Secure Checksum [13] is an md5sum utility that uses multi-threading to compute the checksums of multiple files at once. Unlike msum, however, it cannot parallelize the checksum of a single file.

A variety of work uses custom hardware to increase checksum performance. Deepakumara et al. [8] describe a high speed FPGA implementation of MD5 using loop unrolling. Campobello et al. [4] describe a technique to generate high performance parallelized CRC checksums in compact circuits. CRCs are fast but are unsuitable for integrity checks of large files.

In general, checksums are not easily parallelizable since individual operations are not commutative. A general technique, used by mcp and msum, is based on Merkle trees [18], which allow different subtrees of hashes to be computed independently before being consolidated at the root. A similar approach is described by Sarkar and Schellenberg [23] to parallelize any hash function using a predetermined number of processors, which was used to create a parallel version of SHA-256 call PARSHA-256 [21]. Fixing the number of processors limits achievable concurrency, however, so mcp and msum instead use a predetermined leaf size in the hash tree, which allows an arbitrary number of processors to operate on the same file.

The underlying file system and hardware determine the maximum speed achievable by file copies and checksums. High performance file systems such as Lustre [26], CXFS [27], GPFS [24], and PVFS [5] utilize parallel striping across large numbers of disks to achieve higher aggregate performance than can be achieved from a single-disk file system.

3 Test Environment

All performance testing was carried out using dedicated jobs on the Pleiades supercluster at NASA Ames Research Center, which was recently ranked as the sixth fastest computer system in the world [29] with peak performance of 1.009 PFLOPs/s. Pleiades currently consists of 84,992 cores spread over 9472 nodes, which are connected by DDR and QDR Infiniband. There are three types of nodes with different processor and memory configurations. The nodes used for testing consist of a pair of 3.0 GHz quad-core Xeon Harpertown processors with 6 MB cache per pair of cores and 1 GB DDR2 memory per core for a total of 8 GB per node.

All file copies were performed between Lustre file systems, each with 1 Metadata Server (MDS) and 8 Object Storage Servers (OSS) serving 60 Object Storage Targets (OST). Based on the IOR benchmark [12], the source file system has peak read performance of 6.6 GB/s while the destination file system has peak write performance of 10.0 GB/s. Since copies can only progress at the minimum of the read and write speeds, the peak copy performance of this configuration is 6.6 GB/s. Checksums were performed on the same source file system, hence peak achievable checksum performance is also 6.6 GB/s. Both file systems had zero to minimal load during testing.

Two test cases are used throughout the paper. One case consists of 64 1 GB files while the other consists of a single 128 GB file. Both sets of files were generated from an actual 650 GB user data set. Before any tests could

be done, it was necessary to choose a Lustre stripe count for the files that determines how many OSTs they are striped across. Table 1 shows the performance of cp for the two cases at the default (4 OSTs) and the maximum (60 OSTs) stripe counts. As can be seen, the 64 file case performs best at the default stripe count while the single file case performs best at the maximum. In the 64 file case, the maximum stripe count yields too much parallelism as every OST has to be consulted for every file. In the single file case, the default stripe count yields too little parallelism as large chunks of the file will reside on the same OST, which limits how much I/O bandwidth is available for the copy.

All operations in the remainder of the paper will use the default stripe count for the 64 file case and the maximum stripe count for the single file case. The corresponding cp performance of 174 MB/s for the 64 file case and 240 MB/s for the single file case represent the baseline that the various optimizations throughout the remainder should be compared against.

tool	stripe count	64x1 GB	1x128 GB
cp	default	174	102
cp	maximum	132	240

Table 1: Copy performance (MB/s) vs. stripe count

4 File Copy Optimization

4.1 Multi-Threaded Parallelism

In general, copying regular files is an embarrassingly parallel task since files are completely independent from one another. The processing of the hierarchy of directories containing the files, however, must be handled with care. In particular, a file's parent directory must exist and must be writable when the copy begins and must have its original permissions and ACLs when the copy completes.

The multi-threaded modifications to the cp command of GNU coreutils [11] utilize three thread types as shown in Figure 1 implemented via OpenMP [7]. A single *traversal thread* operates like the original cp program, but when a regular file is encountered, a copy task is pushed onto a shared task queue instead of performing the copy. Mutual exclusivity of all queues discussed is provided by semaphores based on OpenMP locks. Before setting properties of the file, such as permissions, the traversal thread waits until an open notification is received on a designated open queue, after which it will continue traversing the source tree.

One or more *worker threads* wait for tasks on the task queue. After it receives a task, each worker opens the source and target files, pushes a notification onto the open queue, then reads/writes the source/target until done. When stats are enabled, the worker pushes the task (with embedded stats) onto a designated stat queue and then waits for another task. The stat queue is processed by the *stat thread*, which prints the results of each copy task.

Table 2 shows the performance of multi-threading for varying numbers of threads. As can be seen, multi-threading alone has some benefit in the many file case up to 4 threads, after which the kernel buffer cache most likely becomes a bottleneck. For the single file case, multi-threading alone has no benefit since all but one thread sit idle while the file is being transferred. This case will be addressed in the next section.

tool	threads	64x1 GB	1x128 GB
mcp	1	177	248
mcp	2	271	248
mcp	4	326	248
mcp	8	277	248

Table 2: Multi-threaded copy performance (MB/s)

4.2 Single File Parallelization

As seen in the previous section, a number of files less than the number of threads results in imbalanced utilization and correspondingly lower performance. To evenly distribute workload across threads, mcp supports split processing of a single file so that multiple threads can operate on different portions of the same file. Figure 2 shows the processing by the traversal thread and worker threads when split processing is added. The main difference is that the traversal thread may add a number of tasks up to the size of the file divided by the split size and worker threads will seek to the correct location first and only process up to split size bytes.

Table 3 shows the performance of multi-threaded copies of a single large file when different split sizes are used. As can be seen, performance is increased from the unsplit case, but only minimal speedup is seen as the number of threads increases. In Section 4.5, however, significant benefits will be shown when splitting over multiple nodes. In addition, the table shows very little difference between the performance at different split sizes indicating that overhead from splitting is minimal. Since there is minimal difference, a split size of 1 GB will be used throughout the remainder of the results in the paper.

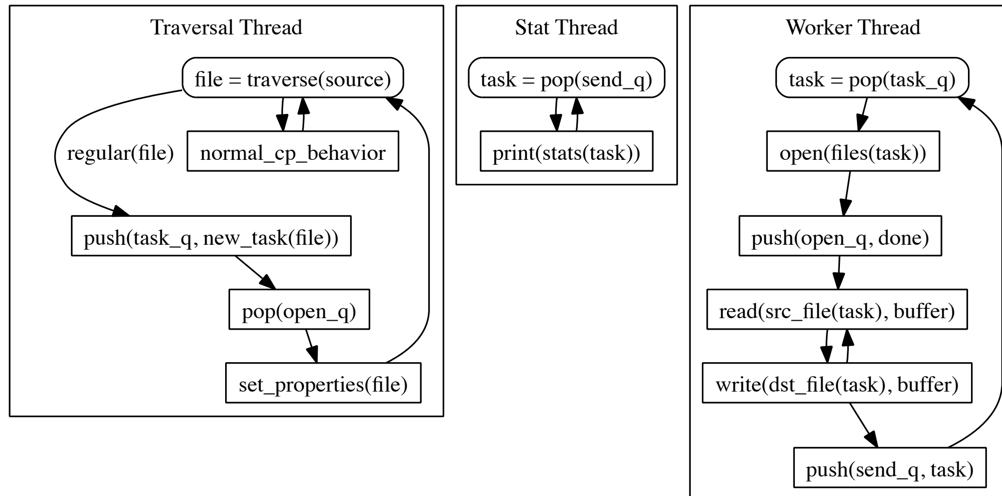

Figure 1: Multi-threaded copy processing

tool	threads	split size	1x128 GB
mcp	2	1 GB	286
mcp	2	16 GB	296
mcp	4	1 GB	324
mcp	4	16 GB	322
mcp	8	1 GB	336
mcp	8	16 GB	336

Table 3: Split file copy performance (MB/s)

4.3 Buffer Management

As witnessed in the Section 4.1, increasing the number of threads yields minimal gains at a certain point. One issue is that file copies generally exhibit poor buffer cache utilization since file data is read once, but then 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 two buffer cache management approaches.

The first approach is to use file advisory information via the posix_fadvise() function, which allows programs to inform the kernel about how it will access data read/written from/to a file. Since mcp only uses data once, it advises the kernel to release the data as soon as it is read/written. The second approach is to skip the buffer cache entirely using direct I/O. In this case, all reads and writes go direct to disk without ever touching the buffer cache.

Table 4 shows the performance of multi-threaded copies when fadvise and direct I/O are utilized with different buffer sizes. As can be seen, performance increases significantly for both cases. Direct I/O achieves about double the performance of fadvise for a single node, but as will be seen in Section 4.5, the performance difference decreases as the number of nodes increases. From this point forward, 128 MB buffers will be used to maximize performance, although this size of buffer is impractical on multi-user systems due to memory limitations. More reasonable 4 MB buffers, however, have been found in testing to achieve a significant fraction of the performance of larger buffers.

4.4 Read/Write Parallelism

In the original cp implementation, a file is copied through a sequence of blocking read and write operations across each section of the file. Through the use of double buffering, it is possible to exploit additional parallelism between reads of one section and writes of another. Figure 3 shows how each worker thread operates in double buffering mode. The main difference is with the write of each file section. Instead of using a standard blocking write, an asynchronous write is triggered via aio_write(), which returns immediately. The read of the next section of the file cannot use the same buffer as it is still being used by the previous asynchronous write, so a second buffer is used. During the read, a write is also being performed, thereby theoretically reducing the original time to read each section from time(read) + time(write) to max(time(read), time(write)). After the read completes, the worker thread blocks until the write is finished (if not already done by that point) and the next cycle begins.

Table 5 shows the copy performance of double buffering for each buffer management scheme across a varying number of threads. As can be seen, double buffering increases the performance of the 64 file case across all

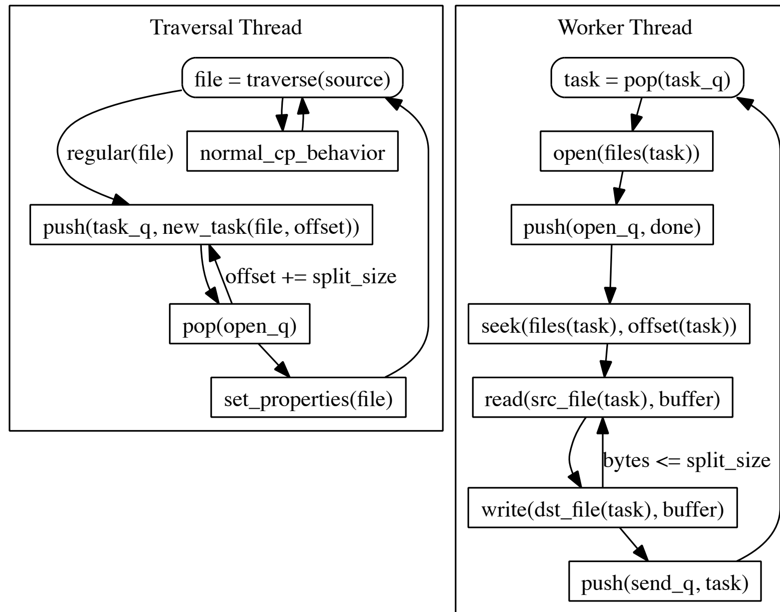

Figure 2: Split file copy processing

numbers of threads. The single file case, however, yields minimal benefit with the exception of the 1 thread case. It is clear a bottleneck exists in the single file case from a single node, but further investigation is needed to determine the exact cause. Double buffering is enabled in all remaining copy results.

4.5 Multi-Node Parallelism

While the results in Table 5 show significant speedup compared to the original cp implementation, it is still a fraction of the peak performance of the file system, hence it is unlikely that a single node can ever achieve the maximum. For this reason, mcp supports multi-node parallelism using both TCP and MPI models. Only the TCP model will be discussed as it is the more portable case and many of the processing details are similar.

In the multi-node TCP model, one node is designated as the *manager node* and parcels out copy tasks to *worker nodes*. The manager node is the only node that runs a traversal thread and stat thread. Both types of nodes have some number of worker threads as in the multi-threaded case. In addition, each node runs a *TCP thread* that is responsible for handling TCP-related activities, whose behavior is shown in Figure 4. The manager TCP thread waits for connections from worker TCP threads. A connection is initiated by a worker TCP thread whenever a worker thread on the same node is idle. If the worker previously completed a task, its stats are forwarded to the manager stat thread via the manager TCP thread. In all cases, the manager thread pops a task from the task queue and sends it back to the worker TCP thread, where it is pushed onto the local task queue for worker threads.

TCP communication introduces security concerns, especially for copies invoked by the root user. Integrity concerns include lost or blocked tasks, where files may not be updated that are supposed to be, replayed tasks where files may have changed between legitimate copies, and/or modified tasks with the source and destination changed arbitrarily. The main confidentiality concern is that contents of normally unreadable directories may be revealed if tasks are intercepted on the network or falsely requested from the manager. Finally, availability can be disrupted by falsely requesting tasks and/or by normal network denials of service.

To protect against TCP-based attacks, all communication is secured by Transport Layer Security (TLS) with Secure Remote Password (SRP) authentication [28]. TLS [10] provides integrity and privacy using encryption so tasks cannot be intercepted, replayed, or modified over the network. SRP [30] provides strong mutual authentication so worker nodes will only perform tasks from legitimate manager nodes and manager nodes will only reveal task details to legitimate worker nodes.

Table 6 shows the copy performance for different numbers of total threads spread across a varying number of nodes. As can be seen, multi-node parallelism achieves significant speedups over multi-threading alone, especially for the single file case. For the same number of total threads, performance increases as the number of

tool	threads	buffer size (MB)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp	1	32	216	383	227	408
mcp	1	64	219	401	226	411
mcp	1	128	226	388	204	415
mcp	2	32	360	689	319	670
mcp	2	64	372	723	317	696
mcp	2	128	402	683	313	723
mcp	4	32	541	1065	330	679
mcp	4	64	575	1039	327	699
mcp	4	128	610	1055	331	721
mcp	8	32	653	1185	332	685
mcp	8	64	681	1223	328	718
mcp	8	128	692	1336	328	743

Table 4: Buffer cache managed copy performance (MB/s)

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp	1	303	645	329	645
mcp	2	503	1111	329	709
mcp	4	653	1557	327	725
mcp	8	663	1763	325	731

Table 5: Double buffered copy performance (MB/s)

nodes increases as there is greater aggregate bandwidth and less resource contention. Direct I/O achieved the highest performance using 16 nodes and a single thread, while fadvise was best in the cases with the largest number of nodes and threads. While fadvise performed significantly worse than direct I/O in earlier sections, it actually surpasses direct I/O in some of the larger 64 file cases and achieved the fastest overall performance at 4.7 GB/s.

5 File Checksum Optimization

5.1 Multi-Threaded Parallelism

The greater the amount of data copied, the greater the possibility for data corruption [2]. The traditional approach to verifying integrity is to checksum the file at both the source and target and ensure that the values match. Checksums are inherently serial, however, so many of the techniques of the previous sections cannot be applied to any but the most trivial checksum algorithms.

Instead of parallelizing the algorithms themselves, serial algorithms are utilized in parallel through the use of Merkle (hash) trees [18] as mentioned previously. This functionality is implemented in a modification to the md5sum command of GNU coreutils called msum. Note that the use of hash trees makes multi-threaded msum unsuitable for verifying standard hashes. Hence, the main purpose of msum is to verify the integrity of copies within the same organization or across organizations that both use msum. This limitation is necessary for performance, however, as most standard hashes cannot be parallelized.

Msum uses a processing model that is similar to the mcp model shown in Figure 1. The msum traversal thread, however, is based on md5sum functionality with correspondingly less complexity. Figure 5 shows the processing by the msum stat thread (which has become the *stat/hash thread*) and worker threads. After copying their portion of the file, worker threads also create a hash tree of that portion, which is embedded in the task sent back to the stat/hash thread through the TCP threads. The stat/hash thread computes the root of the hash tree when all portions have been received.

Table 7 shows the performance of msum across varying numbers of threads and buffer management schemes. Note that msum utilizes libgcrypt [16] to enable support for many different hash types besides MD5, hence performance is not strictly comparable between the md5sum implementation and msum. As can be seen, significant performance gains are achieved by multi-threading even without buffer management. Direct I/O yields sizable gains while the gains by fadvise are more minimal.

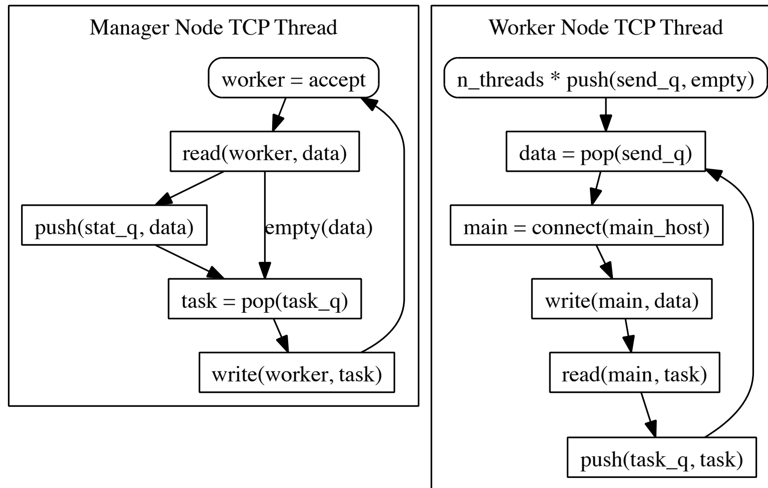

Figure 4: Multi-node copy processing

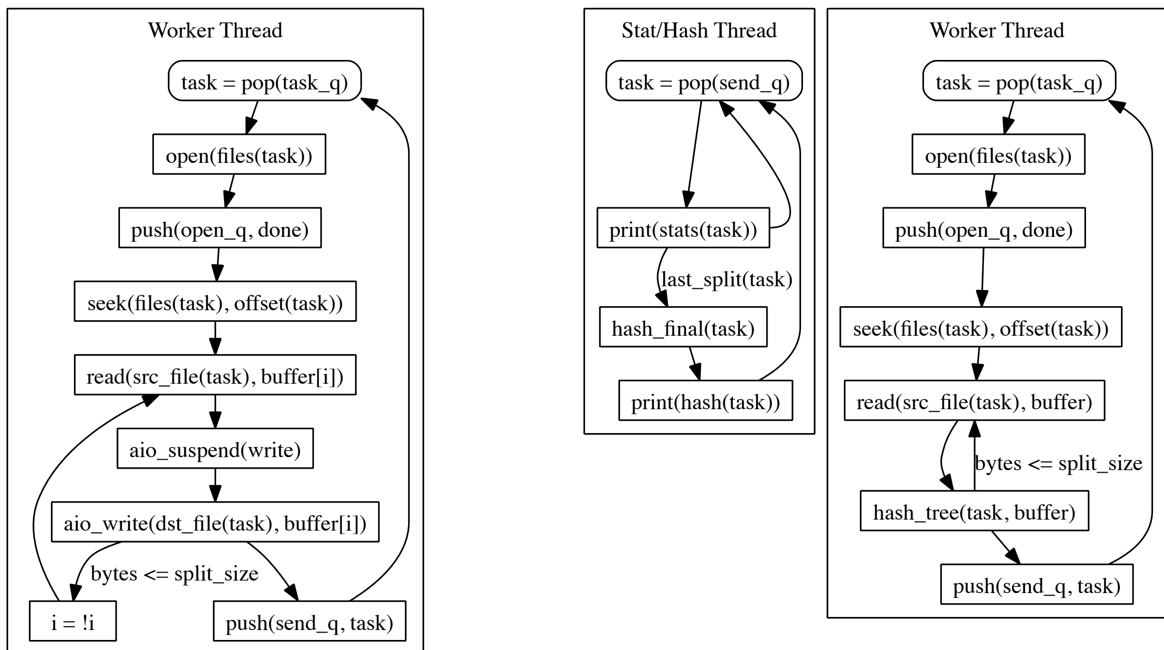

Figure 3: Double buffered copy processing

Figure 5: Multi-threaded checksum processing

5.2 Read/Hash Parallelism

Like the original cp implementation, the original md5sum implementation uses blocking I/O during reads of each section of the file. Double buffering can again be used to exploit additional parallelism between reads of one section and the hash computation of another. Figure 6 shows how each worker thread operates in double buffered mode within msum. In this mode, an initial read is used to seed one buffer. When that read completes, an asynchronous read is triggered via aio_read() into the second buffer. During this read, the hash of the first buffer is computed, after which the buffers are swapped and execution proceeds to the next section of the file after blocking until the previous read completes.

Double buffering theoretically reduces the original time to process each section of the file from time(read) + time(hash) to max(time(read), time(hash)) with best performance achieved when the time to read each sec-

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp	2	2	1	578	1161	273	1080
mcp	4	2	2	969	1673	379	1248
mcp	4	4	1	1119	2074	689	2001
mcp	8	2	4	1256	1857	426	1239
mcp	8	4	2	1818	2996	1068	2316
mcp	8	8	1	2058	3213	1289	3196
mcp	16	2	8	1276	2807	451	1226
mcp	16	4	4	2398	3446	1187	2208
mcp	16	8	2	3187	3599	1787	3723
mcp	16	16	1	3474	**4098**	2786	**4501**
mcp	32	4	8	2411	2957	1189	2142
mcp	32	8	4	3430	3459	2257	3706
mcp	32	16	2	4510	4011	3110	3930
mcp	64	8	8	3216	3346	2253	3626
mcp	64	16	4	**4735**	4011	3620	3914
mcp	128	16	8	–	–	**3824**	4400

Table 6: Multi-node copy performance (MB/s)

tool	threads	64x1 GB	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB	1x128 GB (fadvise)	1x128 GB (direct i/o)
md5sum	1	309	–	–	286	–	–
msum	1	278	284	330	263	265	349
msum	2	541	536	625	378	385	483
msum	4	906	903	1092	570	626	698
msum	8	886	908	1355	508	692	711

Table 7: Multi-threaded checksum performance (MB/s)

tion is the same as the time to hash each section. Table 8 shows the performance achieved by double buffering within msum for each buffer management scheme across a varying number of threads. As can be seen, double buffering increases the performance of all the 64 file cases except the 8 thread direct I/O case and all the single file cases except the 8 thread fadvise case. Double buffering is enabled in all remaining checksum results.

5.3 Multi-Node Parallelism

Msum supports the same TCP and MPI models as mcp for multi-node parallelism. TCP threads behave identically to those shown for mcp in Figure 4. Table 9 shows the checksum performance for different numbers of total threads spread across a varying number of nodes. As can be seen, multi-node parallelism achieves significant speedups over multi-threading alone. As was the case with mcp, performance generally increases for the same number of total threads as the number of nodes increases as there is greater aggregate bandwidth and less resource contention.

Both fadvise and direct I/O achieved the highest performance with 16 nodes and 2 threads in the 64 file case and with 16 nodes and 8 threads in the single file case. Once again, fadvise began to yield higher performance than direct I/O in some of the larger cases and once again had the highest overall performance at 5.8 GB/s. Note that this is 88% of peak of the file system and includes hashes as well as reads.

6 Verified File Copy Optimization

6.1 Buffer Reuse

In a typical integrity-verified copy, a file is checksummed at the source, copied, and then checksummed again at the destination to gain assurance that the bits at the source were copied accurately to the destination. This process normally requires two reads at the source since the checksum and copy programs are traditionally separate so each must access the data independently. Adding

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
msum	1	428	489	461	520
msum	2	811	973	462	522
msum	4	926	1647	662	766
msum	8	936	1315	613	776

Table 8: Double buffered checksum performance (MB/s)

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
msum	2	2	1	821	928	471	603
msum	4	2	2	1522	1834	832	939
msum	4	4	1	1487	1744	820	1027
msum	8	2	4	1819	2845	1298	1330
msum	8	4	2	2837	3122	1454	1798
msum	8	8	1	2844	3130	1808	2225
msum	16	2	8	1649	2979	1165	1076
msum	16	4	4	3218	3689	1891	1944
msum	16	8	2	4820	5292	3148	3654
msum	16	16	1	4770	4957	3248	3397
msum	32	4	8	3248	3719	1759	1936
msum	32	8	4	4664	4183	4640	4256
msum	32	16	2	**5812**	**5613**	4533	4856
msum	64	8	8	4114	3680	4256	3579
msum	64	16	4	5543	5131	4595	5114
msum	128	16	8	–	–	**5192**	**5227**

Table 9: Multi-node checksum performance (MB/s)

checksum functionality into the copy portion eliminates one of the reads to increase performance. Mcp incorporates checksums for this reason. This processing is similar to Figure 5 except the buffer is written between the read and the hash computation.

Table 10 shows the performance of copying with checksums for varying numbers of threads and different buffer management schemes. As was the case with the standard copy results in Table 4, direct I/O outperforms fadvise on a single node with the 64 file case achieving better results than the single file case.

6.2 Read/Hash Parallelism

The double buffering improvements of Section 5.2 were incorporated into mcp's checksum functionality with processing similar to Figure 6 with an additional write after the hash. Ideally, both the read of the next section and the write of the current section could be performed while the hash of the current section was being computed. This approach was implemented, but did not behave as expected, possibly due to concurrency controls within the file system. Further investigation is warranted as this would provide an additional increase in performance. Table 11 shows the performance increases achieved with double buffering during copies with checksums. As can be seen, performance increases in all but the 8 thread 64 file fadvise case.

6.3 Multi-Node Parallelism

Table 12 shows the multi-node performance of copies incorporating checksum functionality. Peak performance of just under 4.0 GB/s was achieved with 8 nodes and 4 threads in the 64 file direct I/O case.

Table 13 is a composite view of Tables 5, 6, 8, 9, 11, and 12 that shows the performance of integrity-verified copies using the traditional checksum + copy + checksum versus a copy with embedded checksum + checksum. As can be seen, performance is better in almost every case with only a few scattered exceptions. Both fadvise and direct I/O achieve verified copies over 2 GB/s with 16 nodes and 2 threads in the 64 file case.

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp (w/ sum)	1	156	224	92	201
mcp (w/ sum)	2	294	428	152	376
mcp (w/ sum)	4	503	770	216	510
mcp (w/ sum)	8	629	1102	266	602

Table 10: Copy with checksum performance (MB/s)

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp (w/ sum)	1	190	290	104	222
mcp (w/ sum)	2	356	558	171	400
mcp (w/ sum)	4	561	966	235	560
mcp (w/ sum)	8	626	1498	275	671

Table 11: Double buffered copy with checksum performance (MB/s)

7 Conclusions and Future Work

Mcp and msum provide significant performance improvements over standard cp and md5sum using multiple types of parallelism and other optimizations. Tables 14, 15, and 16 show the maximum speedups obtained at each stage of optimization for copies, checksums, and integrity-verified copies, respectively. The relative effectiveness of each optimization is difficult to discern as they build upon each other and would have different peak speedups if applied in a different order. The total speedups from all improvements, however, is significant. Mcp improves cp performance over 27x, msum improves md5sum performance almost 19x, and the combination of mcp and msum improves verified copies via cp and md5sum by almost 22x. These improvements come in the form of drop-in replacements for cp and md5sum so are easily used and are available for download as open source software [19].

There are a variety of directions for future work. Currently, only optimized versions of cp and md5sum have been implemented from GNU coreutils. Optimized versions of the coreutils install and mv utilities should also be implemented as they would immediately benefit from the same techniques. In general, other common single-threaded utilities should be investigated to see if similar optimizations can be made.

Another area of study is to determine if mcp can be made into a remote transfer utility. While it currently can only be used for copies between local file systems, mcp already contains network authentication processing in the multi-node parallelization. In addition, most of the other techniques would be directly applicable to a high performance multi-node striping transfer utility. The missing component is a network bridge between the local read buffer and remote write buffer. The buffer reuse optimizations to checksums can be used directly to support integrity-verified remote transfers.

Although not discussed, mcp and msum both have the ability to store intermediate hash tree values within file system extended attributes. The purpose of this feature is to allow file corruption to be detected and precisely located over time in persistent files. The use of extended attributes has been found to be impractical, however, when the hash leaf size is small since only some file systems such as XFS support large extended attribute sizes and read/write performance of extended attributes is suboptimal. Further investigation is required to determine if greater generality and higher performance can be achieved using a mirrored hierarchy of regular files that contain the intermediate hash tree values.

References

[1] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster: The Globus Striped GridFTP Framework and Server. ACM/IEEE Supercomputing Conf., Nov. 2005.

[2] L.N. Bairavasundaram, G.R. Goodson, B. Schroeder, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau: An Analysis of Data Corruption in the Storage Stack. 6th USENIX Conf. on File and Storage Technologies, Feb. 2008.

[3] BbFTP. http://doc.in2p3.fr/bbftp.

[4] G. Campobello, G. Patane, M. Russo: Parallel CRC Realization. IEEE Trans. on Computers, vol. 52, no. 10, Oct. 2003.

[5] P.H. Carns, W.B. Ligon, R.B. Ross, R. Thakur: PVFS: A Parallel File System for Linux Clusters. 4th Annual Linux Showcase and Conf., Oct. 2000.

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp (w/ sum)	2	2	1	375	554	197	439
mcp (w/ sum)	4	2	2	682	1028	257	779
mcp (w/ sum)	4	4	1	714	1028	380	833
mcp (w/ sum)	8	2	4	1075	1756	398	1106
mcp (w/ sum)	8	4	2	1304	1815	611	1396
mcp (w/ sum)	8	8	1	1387	1858	722	1545
mcp (w/ sum)	16	2	8	1185	2506	617	1568
mcp (w/ sum)	16	4	4	2000	2716	825	1905
mcp (w/ sum)	16	8	2	2362	3032	1151	2233
mcp (w/ sum)	16	16	1	2439	2858	1319	2274
mcp (w/ sum)	32	4	8	2166	2809	907	2215
mcp (w/ sum)	32	8	4	3124	**3952**	1494	2318
mcp (w/ sum)	32	16	2	3229	3595	1973	3088
mcp (w/ sum)	64	8	8	2139	3147	1525	2693
mcp (w/ sum)	64	16	4	**3275**	3739	2353	**3277**
mcp (w/ sum)	128	16	8	–	–	**2481**	3183

Table 12: Multi-node copy with checksum performance (MB/s)

[6] B. Cohen: Incentives Build Robustness in BitTorrent. 1st Wkshp. on Economics of Peer-to-Peer Systems, Jun. 2003.

[7] 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.

[8] J. Deepakumara, H.M. Heys, R. Venkatesan: FPGA Implementation of MD5 Hash Algorithm. 14th IEEE Canadian Conf. on Electrical and Computer Engineering, May 2001.

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

[10] T. Dierks, E. Rescorla: The Transport Layer Security (TLS) Protocol Version 1.2. IETF Request for Comments 5246, Aug. 2008.

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

[12] R. Hedges, B. Loewe, T. McLarty, C. Morrone: Parallel File System Testing for the Lunatic Fringe: the care and feeding of restless I/O Power Users. 22nd IEEE / 13th NASA Goddard Conf. on Mass Storage Systems and Technologies, Apr. 2005.

[13] Hewlett Packard: HP-UX MD5 Secure Checksum A.01.01.02 Release Notes. Sept. 2007. `http://docs.hp.com/en/5992-2115/5992-2115.pdf`.

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

[15] 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`.

[16] Libgcrypt. `http://www.gnupg.org/documentation/manuals/gcrypt`.

[17] 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.

[18] R.C. Merkle: Protocols for Public Key Cryptosystems. 1st IEEE Symp. on Security and Privacy, Apr. 1980.

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

[20] 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.

[21] P. Pal, P. Sarkar: PARSHA-256 – A New Parallelizable Hash Function and a Multithreaded Implementation. 10th Intl. Wkshp. on Fast Software Encryption, Feb. 2003.

[22] C. Rapier, B. Bennett: High Speed Bulk Data Transfer Using the SSH Protocol. 15th ACM Mardi Gras Conf., Jan. 2008.

[23] P. Sarkar, P.J. Schellenberg: A Parallel Algorithm for Extending Cryptographic Hash Functions. 2nd Intl. Conf. on Cryptology in India, Dec. 2001

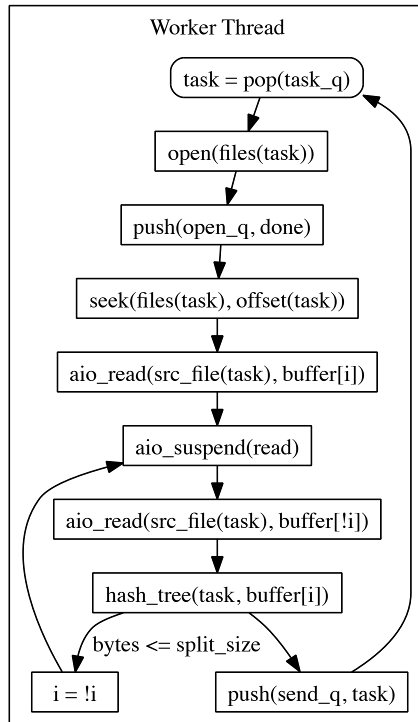

Figure 6: Double buffered checksum processing

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

[25] 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`.

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

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

[28] D. Taylor, T. Wu, N. Mavrogiannopoulos, T. Perrin: Using the Secure Remote Password (SRP) Protocol for TLS Authentication. IETF Request for Comments 5054, Nov. 2007.

[29] TOP500 Supercomputing Sites, Jun. 2010. `http://www.top500.org/lists/2010/06`.

[30] T. Wu: The Secure Remote Password Protocol. 5th ISOC Network and Distributed System Security Symp., Mar. 1998.

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
md5sum + cp + md5sum	1	1	1	100		90	
msum + mcp + msum	1	1	1	125	177	135	185
mcp (w/ sum) + msum	1	1	1	131	182	84	155
msum + mcp + msum	2	1	2	224	338	135	190
mcp (w/ sum) + msum	2	1	2	247	354	124	226
msum + mcp + msum	2	2	1	240	331	126	235
mcp (w/ sum) + msum	2	2	1	257	346	138	254
msum + mcp + msum	4	1	4	270	538	164	250
mcp (w/ sum) + msum	4	1	4	349	608	173	323
msum + mcp + msum	4	2	2	426	592	198	341
mcp (w/ sum) + msum	4	2	2	470	658	196	425
msum + mcp + msum	4	4	1	446	613	257	408
mcp (w/ sum) + msum	4	4	1	482	646	259	459
msum + mcp + msum	8	1	8	274	478	157	253
mcp (w/ sum) + msum	8	1	8	375	700	189	359
msum + mcp + msum	8	2	4	527	805	257	432
mcp (w/ sum) + msum	8	2	4	675	1085	304	603
msum + mcp + msum	8	4	2	796	1026	432	647
mcp (w/ sum) + msum	8	4	2	893	1147	430	785
msum + mcp + msum	8	8	1	840	1052	531	825
mcp (w/ sum) + msum	8	8	1	932	1165	515	911
msum + mcp + msum	16	2	8	500	973	254	373
mcp (w/ sum) + msum	16	2	8	689	1361	403	638
msum + mcp + msum	16	4	4	962	1201	526	674
mcp (w/ sum) + msum	16	4	4	1233	1564	574	962
msum + mcp + msum	16	8	2	1372	1524	836	1225
mcp (w/ sum) + msum	16	8	2	1585	1927	842	1386
msum + mcp + msum	16	16	1	1414	1544	1025	1233
mcp (w/ sum) + msum	16	16	1	1613	1812	938	1362
msum + mcp + msum	32	4	8	970	1141	505	666
mcp (w/ sum) + msum	32	4	8	1299	1600	598	1033
msum + mcp + msum	32	8	4	1388	1303	1144	1351
mcp (w/ sum) + msum	32	8	4	1870	2032	1130	1500
msum + mcp + msum	32	16	2	1767	1651	1311	1500
mcp (w/ sum) + msum	32	16	2	**2075**	**2191**	1374	1887
msum + mcp + msum	64	8	8	1254	1187	1094	1198
mcp (w/ sum) + msum	64	8	8	1407	1696	1122	1536
msum + mcp + msum	64	16	4	1748	1564	1405	1546
mcp (w/ sum) + msum	64	16	4	2058	2162	1556	**1997**
msum + mcp + msum	128	16	8	–	–	1546	1639
mcp (w/ sum) + msum	128	16	8	–	–	**1678**	1978

Table 13: Multi-node verified copy performance (MB/s)

origin	optimization	peak speedup
cp	multi-threading	1.9
multi-threading	split files	1.4
split files	posix_fadvise	2.5
spit files	direct I/O	4.8
posix_fadvise	double buffering	1.3
direct I/O	double buffering	1.6
double buffering	multiple nodes	7.1
cp	all	27.2

Table 14: Summary of copy optimizations

origin	optimization	peak speedup
md5sum	multi-threading	2.9
multi-threading	split files	2.2
split files	posix_fadvise	1.4
split files	direct I/O	1.5
posix_fadvise	double buffering	1.7
direct I/O	double buffering	1.6
double buffering	multiple nodes	6.2
md5sum	all	18.8

Table 15: Summary of checksum optimizations

origin	optimization	peak speedup
md5sum + cp + md5sum	multi-threaded + split files + buffer management + buffer reuse	6.1
multi-threaded + split files + buffer management + buffer reuse	double buffering	1.2
double buffering	multiple nodes	10.7
md5sum + cp + md5sum	all	21.9

Table 16: Summary of verified copy optimizations