A Bloom Filter Based Scalable Data Integrity Check Tool for Large-scale Dataset

Sisi Xiong
University of Tennessee Knoxville
Knoxville, TN, USA
Email: sxiong@vols.utk.edu

Feiyi Wang
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: fwang2@ornl.gov

Qing Cao
University of Tennessee Knoxville
Knoxville, TN, USA
Email: qcao1@utk.edu

***Abstract*—Large scale HPC applications are becoming increasingly data intensive. At Oak Ridge Leadership Computing Facility (OLCF), we are observing the number of files curated under individual project are reaching as high as 200 millions and project data size is exceeding petabytes. These simulation datasets, once validated, often needs to be transferred to archival system for long term storage or shared with the rest of the research community. Ensuring the data integrity of the full dataset at this scale is paramount important but also a daunting task. This is especially true considering that most conventional tools are serial and file-based, unwieldy to use and/or can't scale to meet user's demand.**

To tackle this particular challenge, this paper presents the design, implementation and evaluation of a scalable parallel checksumming tool, `fsum`, which we developed at OLCF. It is built upon the principle of parallel tree walk and work-stealing pattern to maximize parallelism and is capable of generating a single, consistent signature for the entire dataset at extreme scale. We also applied a novel bloom-filter based technique in aggregating signatures to overcome the *signature ordering* requirement. Given the probabilistic nature of bloom filter, we provided a detailed error and trade-off analysis. Using multiple datasets from production environment, we demonstrated that our tool can efficiently handle both very large files as well as many small-file based datasets. Our preliminary test showed that on the same hardware, it outperforms conventional tool by as much as 4x. It also exhibited near-linear scaling properties when provisioned with more compute resources.

I. Introduction

Data has overwhelmed the digital world in terms of volume, variety and velocity [1]. Similarly, the scale of scientific data requirement in HPC environment is also growing at unprecedented pace. As an example, at Oak Ridge Leadership Computing Facility (OLCF), the file and storage system designed to serve 2008 Jaguar machine (known as Spider 1) provided 10 petabytes usable space. Four years later, Spider 2 (for primarily Titan supercomputer) started provisioning 32 petabytes to the users [2]. There are 50 million directories and half a billion files in the system, this is barely under control with active purging policy in place. We are also observing that a single project can produce as much as 200 million files in the short span of time. As such, the storage, management and analysis of such amount of data present familiar yet difficult extreme-scale data challenges.

Another aspect of the scientific dataset is its value: it can take millions of million compute hours to yield the dataset. The results often need to be sanitized, validated, and archived for long term storage, and/or shared with scientific community for further analysis. Ensuring its integrity during the usefulness of data life-cycle is paramount important. This life-cycle may include activities such as copying data from one file system to other, moving from one site/facility to another. However, it is not uncommon to see user reporting of missing, and/or corrupted files. Also, silent data corruption has been reported in many production systems [3].

One of primary approaches that counters file corruption and provide data integrity is through checksumming [4]. By comparing previous and current checksums, one can detect whether the content has changed. Conventionally, the checksumming tools are both serial and file based. This is where the conventional checksum tools fall short as far as extreme-scale dataset in HPC is concerned. Profiling of our current production file system clearly shows the bi-modal trend: We have over 60% of small files that are less than 1 MiB in size, but we also have tens of thousands of files that are over 4 TiB, hundreds of them are as large as 32 TiB each.

In both scenarios existing serial and file based checksum method are not efficient to handle either millions of files or files as large as many terabytes. In best cases, they take an excruciatingly long time to process, or fail outright due to the size of the data. Hence, a parallel checksumming approach is truly needed and preferred. On the other hand, with millions of files, file-based checksums are cumbersome and lack usability. Therefore, a single dataset level signature is more desirable.

In this paper, we present a novel, parallel dataset checksumming approach, `fsum`. More specifically, we first break files into chunks of reasonable size, and calculate chunk-level checksums in parallel. Next, we develop a bloom filter based method to aggregate all chunk-level checksums to yield the single dataset-level checksum. Due to bloom filters' probabilistic nature of error, we provided a detailed analysis on the trade-offs, and we demonstrated its scalability with representative datasets from our current production environment.

The rest of the paper are organized as the following: we present the design and implementation of `fsum` in Section II. The experiment results and related works are shown in Section III and Section IV. We conclude the paper in Section V.

II. Design and Implementation

In this section, we discuss three aspects of the design that differentiate `fsum` from other solutions. First is about the need for chunk-level checksum versus file-level checksum. Then, we focus on work-stealing pattern and the consequent need for aggregating (and sorting) *all* chunk-level checksum to generate a single, consistent signature. Finally, we show the bloom filter-based algorithm as an option to solve the *ordering* issue discussed earlier. We argue that it can greatly improve the scalability, at the cost of probabilistic error. And finally, we discuss the trade-offs on CPU, memory, error probability in details.

A. File-level and chunk-level checksum

Conventional checksums are file based. However, in HPC environment, it is not uncommon for end users to create very large files. A recent profiling study using `fprof` [5] for OLCF Spider 2 Lustre file system shows a bi-modal distribution: A majority of total number of files are really small, e.g. less than 1 MiB, and a small percentage of files are really big: hundred of thousands files with size 4 TiB and beyond, some can be as large as 32 TiB. With traditional checksumming tools, which are most likely serial and file based, it can take many hours to finish processing them. We also ran into cases when a program just hangs with super large input size. Given this file distribution, breaking up a large file into a sequence of chunks is absolutely necessary.

Each data chunk object is self-contained, in the sense that it carries all the necessary offset information for each process to handle them independently even when the chunk has been transferred from one process to another. Regarding the chunk size, we have to compromise on two fronts: if we make the chunk small, parallelism increases at the cost of both processing and metadata overhead associated with each chunk 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. By default, our checksumming approach will parallel scan all files and adaptively pick a chunk size that represents a balance of total chunks and total file size.

B. Parallel work-stealing pattern

After chunking, we utilize the work stealing pattern [6], [7] to distribute all the chunks to multiple processes to generate chunk-level checksums in parallel. In particular, an idle process sends out a work request, or "steals" work from a busy process, which equally distributes its local work queue to one or more requesters. We implement the work stealing pattern using MPI. Initially, the root process (rank 0) puts the files and/or directories that are intended to be checksummed into its local work queue, while other processes have empty local work queue. As `fsum` proceeds, all work gets spread out among multiple processes.

A major characteristic of the work stealing pattern is its randomness. When executing `fsum` multiple times, using the same input dataset but different or the same number of processes, the work distribution among all processes are different, in the sense that a chunk might be assigned to a different process. Therefore, a process calculates a different set of chunk-level checksums. However, since we take the same dataset as input, the union of checksum sets from all the processes are identical.

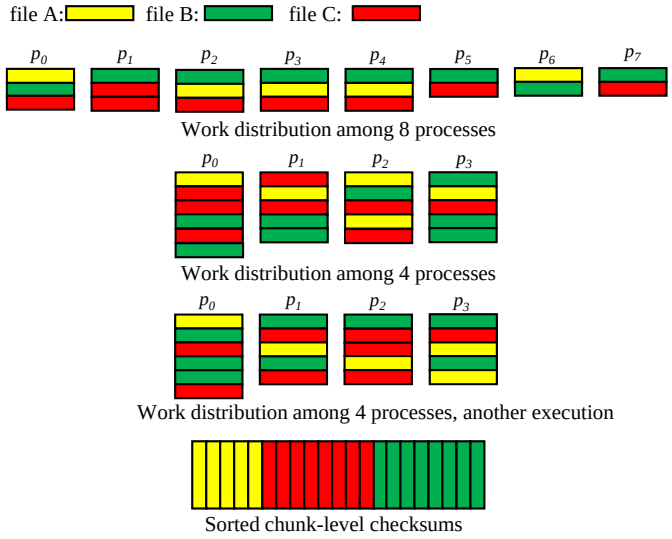

Fig. 1. Random work distribution using different/same number of processes during different executions

For example, as shown in Figure 1, for a dataset that contains three files, which have been split into multiple chunks, the work distribution using 4 and 8 process are obviously different. Furthermore, during two executions of `fsum` using both 4 processes, the work distribution are still different. In other words, we need a method that can generate a single and consistent signature regardless the non-deterministic nature of the algorithm and how user will run it.

C. Bloom filter based signature generation

We discussed earlier about the unwieldiness of having file-based signatures for a large dataset with millions of files. Our goal is to generate a single and consistent dataset-level signature regardless the random nature of work-stealing pattern and how user launch the parallel process. We compare two design options here: sorting-based and bloom-filter based.

1) Design principles and the sorting-based approach: The dataset-level signature should be only dataset content dependent. In other words, two signatures generated by two executions of `fsum` based on the same input dataset should be identical. However, the randomness of the work stealing pattern leads to the fact that the chunk-level checksums are generated in different processes and in different orders. Besides, a desirable approach should avoid storing all the chunk-level checksums on a single process/node, which might face memory pressure and become the bottleneck.

One straightforward solution is to concatenate all chunk-level checksums to a hash list, and calculate the top hash based on it. Another possible approach is to use a Merkle

tree [8] to generate the root hash. However, as mentioned before, the checksums are generated in different orders during multiple executions of `fsum`. Therefore, a sorting is necessary. However, it is not scalable since the at some point, the root process has to gather checksums from all the processes and store them in memory as a hash list or a Merkle tree. In extreme cases, distributed sort or external sort needs to be performed, which inevitably increases the complexity of the checksum aggregation.

2) Bloom filter: Due to drawbacks of sorting, we present a bloom filter based approach to aggregate the chunk-level checksums. A bloom filter is an array of m bits that are initialized to all 0s, and its functionality is to identify membership, i.e., whether an element is in a set. The bloom filter supports two basic operations: insertion and query. By inserting an element, k hashed positions, which are calculated based on the element, out of m bits are set to 1. By querying an element, k hashed positions are tested against 1, and any 0 findings indicate that the element is not in the set. There are two types of errors: false negative errors never happen to bloom filter, i.e., an element that is actually in the set is always returned as it is. However, false positive errors could happen with a probability, meaning that an element is not in the set might be reported as it is, since the k bits in the bloom filter might be set to 1 by other elements due to hash collisions. The false positive error probability p relates to the size (length) of the bloom filter m, the number of elements inserted n, and the number of hash functions k, according to

$$p = e^{-\frac{m}{n}(\ln 2)^2}. \tag{1}$$

In other words, to reach a certain false positive error probability, we should set the bloom filter size to

$$m = -n\frac{\ln p}{(\ln 2)^2}. \tag{2}$$

Therefore, given p, the bloom filter size is linearly related to the number of elements that are intended to be inserted. In our application, n is the number of chunks.

3) Chunk-level checksums aggregation algorithm: From a different point of view, the bloom filter can be considered as a bitmap signature of a set after all elements have been inserted, as long as the bloom filter parameters are set properly. Due to the design and bit array nature of the bloom filter, we notice it has two features as a signature of a set:

- The bloom filter is independent of insertion orders. For example, the set of $\{A, B, C, D, E\}$ and the set of $\{E, D, C, B, A\}$ lead to the same bloom filter.
- We can perform OR operation of bit arrays to represent the union of multiple datasets. For example, if the bloom filter b_1 is calculated based on the set of $\{A, B, C\}$ and b_2 is based on the set of $\{D, E\}$, the result of $b_1 \vee b_2$ represents the set of $\{A, B, C, D, E\}$, requiring that b_1 and b_2 have the same parameters.

Due to these two features, we utilize the bloom filter as an intermediate data structure to aggregate the chunk-level checksums. More specifically, first at each process, we insert all local chunk-level checksums into a bloom filter, of which all processes has the same parameters, and then send it to the root process, where we gather all the bloom filters using OR operation. Finally we calculate SHA1 hash function based on the final bloom filter and we obtain the dataset-level signature. The bloom filter based dataset-level signature generation algorithm is shown in Algorithm 1.

Algorithm 1 Chunk-level checksums aggregation algorithm

```
1:  procedure CHECKSUM AGGREGATION((C, np))
2:      for each c_i in C do
3:          for j = 1 → k do                    ▷ Insert c_i to BF
4:              idx ← h_j(c_i)
5:              BF(idx) = 1
6:          end for
7:      end for
8:      if rank > 0 then                        ▷ Send BF to root process
9:          MPI.send(BF, dst = 0)
10:     else                                    ▷ Gather BFs from all processes
11:         for i = 1 → np do
12:             BF = BF ∨ MPI.recv(src = i)
13:         end for
14:     end if
15:     MPI.comm.barrier()
16:     if rank == 0 then                       ▷ Generate the signature
17:         sig = SHA1(BF)
18:         return sig
19:     end if
20: end procedure
```

4) Memory and computation overhead: The bloom filter is highly compact in terms of memory overhead. Since the bloom filter size m relates to the error probability, we discuss how to set p and m in Section II-C5.

For the computation overhead, the insertion of each element calculates k hash functions, which takes constant time, hence the overall aggregation process takes $\mathcal{O}(n)$ time. On the other hand, a typical sorting algorithm takes $\mathcal{O}(n \log n)$, which is less efficient than bloom filter based approach when n is at extreme scale.

5) Error probability: [1]

Due to probabilistic nature of the bloom filter, `fsum` might end up in error: two different datasets have the same signature, i.e., the calculated bloom filter of two different chunk-level checksum sets are the same.

Consider two sets, $C_1 = \{a_0, a_1, ..., a_{n_1-1}\}$ and $C_2 = \{b_0, b_1, ..., b_{n_2-1}\}$, after inserting all elements in each set into two bloom filters, B_1 and B_2, accordingly, if $B_1 = B_2$, we have the following two facts:

- Since we calculate the bloom filter size using the same equation 2 and $B_1 = B_2$, n_1 is equal to n_2.

[1]To focus on errors caused only by the bloom filter, we assume that the signatures of two different chunks, or two different bloom filters, are always different.

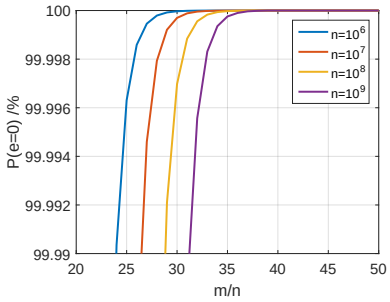

Fig. 2. Variation of $P(e = 0)$ while m/n increases.

- Suppose there are $r(r \geq 1)$ elements that are in C_1 and C_2 are different, i.e., $C_1 = \{x_0, x_1, ..., x_{r-1}, c_r, c_{r+1}, ..., c_{n-1}\}$, $C_2 = \{y_0, y_1, ..., y_{r-1}, c_r, c_{r+1}, ..., c_{n-1}\}$ $(x_i \neq y_j (i, j = 0, 1, ..., r-1))$. When performing query operation on B_2 using x_i, since x_i has been inserted to B_1 (no false negative error) and $B_1 = B_2$, we have false positive error, and the probability is p. Similarly, when performing query operation on B_1 using y_j, we also have false positive error with probability of p.

Conversely, if all x_i and $y_j (i, j = 0, 1, ..., r-1)$ have false positive errors, the two bloom filters are the same, in which case `fsum` has lost the ability to detect the signature difference of these r chunks, which results in an error. Therefore, given r different chunks, the error probability of `fsum` is p^{2r}.

We consider the worst case: there is only one different chunk-level checksum, i.e., $r = 1$. Since there are a total of n chunks that might be different, the number of errors of `fsum` follows a binomial distribution $e \sim B(n, p^2)$. Furthermore, we calculate the probability $P(e = 0) = (1 - p^2)^n$. To make sure $P(e = 0) > 99.99\%$, we set p to

$$p < \sqrt{1 - \sqrt[n]{0.9999}}. \tag{3}$$

Since p relates to the ratio $\frac{m}{n}$ according to equation 1, we set m to

$$m > -n \times \frac{\ln \sqrt{1 - \sqrt[n]{0.9999}}}{(\ln 2)^2}. \tag{4}$$

We plot the variation of $P(e = 0)$ when $\frac{m}{n}$ increases in Figure 2, with different values of n. For example, given 100 million chunks, suppose $r = 1$ as the worst case, to make sure we are 99.99% confident that there is no errors from BF-based signature: $P(e = 0) > 99.99\%$, i.e., $(1 - p^2)^n > 99.99\%$, we set $p \leq 0.000001$, hence $m \geq -n \times \ln 0.0000001/(\ln 2)^2 \approx 29n$, which is about 2.7GiB memory to store the bloom filter at each process. While for the sorting approach, since each chunk-level checksum is a 160-bit SHA1 hash value, all of them take 14.9GiB memory. If 8 processes are used, at each process, the memory usage is only 1.86GiB. However, in the aggregation of sorting, all checksums have to be stored in the root process, which becomes the memory bottleneck.

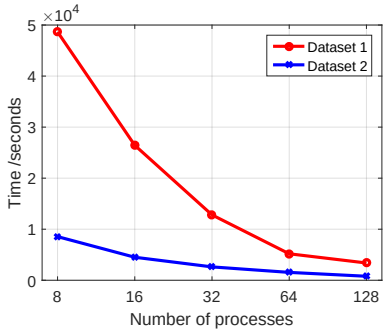

Fig. 3. Runtime with different number of processes.

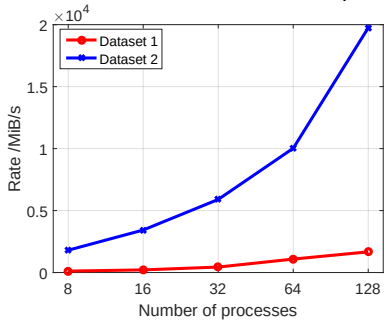

Fig. 4. Checksum rate with different number of processes.

III. Experiment Results

In this section, we evaluate the scalability performance of `fsum`, and compare `fsum` with other related approaches.

A. Evaluation configurations

All the experiments are conducted on Rhea, a 512-node Linux cluster at OLCF. Each Rhea node has two 8-core 2.0 GHz Intel Xeon processors with 128GB of memory. The underlying file system is Spider 2 Lustre file system. Due to page limits, we don't consider any effect of stripe count, which is set to the default value 4 in all experiments.

We choose two datasets that scientific users created on Spider 2, D_1 and D_2, and their parameters are shown in Table I. They are representative in the sense that, in D_1, there are more than 28 millions files, and 93% of them are less than 4KiB, while the average file size in D_2 is 1.19GiB and the largest file is 514.41GiB.

TABLE I
Parameters of two datasets

	D_1	D_2
Total size	5.39TiB	14.74TiB
Number of files	28,114,281	15,590
Average file size	205.83KiB	1.19GiB
Chunk size	16MiB	64MiB
Number of chunks	28,343,725	251,629

B. Evaluation results

First, we evaluate the scalability performance of `fsum`, using 8 Rhea nodes with different number of process to generate

signatures of the two datasets, and we plot the runtime and processing rate in Figure 3 and Figure 4. As expected, there are speedups when using more processes. We notice that the speedup increases almost linearly as the number of processes increases, until the performance is bounded by I/O bandwidth. In addition, comparing results of the two datasets, we observe that generating signature for D_1 is much less efficient than for D_2, since handling small files is bounded by metadata retrieval in Lustre file systems.

The signatures generated using different number of processes for the same dataset are always the same, which verifies that the signature generated by `fsum` is only dataset content dependent, hence the bloom filter based dataset-level signature generation approach is feasible and correct.

Next, we conduct experiments to validate that `fsum` can detect data corruption. Specifically, we compare the signatures of the original dataset and the one that has been corrupted. For each dataset, we conduct 100 experiments, in each of which we randomly choose a file and change a single byte at a random position. We notice that the signatures of the corrupted datasets are always different from the one of the original dataset. Therefore, `fsum` can detect data corruption and can be a tool for data integrity check.

Finally, we compare `fsum` with related approaches in terms of of memory usage and time efficiency. First, we compare the memory usage of the bloom filter in `fsum` and the maximum usage of sorting at the root process, as shown in Figure 5. For bloom filter parameters, we set $P(e = 0) > 99.99\%$, and m is calculated according to equation 4. We observe that the bloom filter approach uses about 5.8 and 7.1 times less memory than the sorting for D_1 and D_2 accordingly.

Second, we compare the runtime of `sha1sum` and `fsum` on a single node, using a 514.41TiB file. For `fsum`, we utilize 4 processes with different chunk sizes, and the results are shown in Figure 6. We notice that `fsum` is around 2.6 to 4.4 times more efficient than the single-threaded `sha1sum`. Furthermore, for large files, a larger chunksize results in better performance.

IV. Related Works

Ensuring data integrity is essential in data storage and transfer [4]. In the cases of millions of small files or a single terabye-file, checksumming in parallel is truly preferred. In general, there are two types of parallelism, and the first of which is parallel hash functions [9], [10]. These approaches, however, are hardly deployed in clustered systems due to lack of ability to fully exploit the available computation resources. The alternative is parallel workload [11], i.e., break files into chunks and use multiple processes to calculate a checksum for each chunk, then aggregate all the checksums to a single dataset-level signature. The master-slave architecture in [11] is one possible mechanism to distribute workloads. And Merkle tree [12], which allows different hash subtrees to be computed without synchronization at the root, is utilized to aggregate all the checksums. However, the master-slave architecture suffers from tremendous communication overhead between master and slave workers. On the other hand, the work stealing pattern is a decentralized and self-organized approach [6], [7], and due to its randomness, Merkle tree is no longer feasible without a universal sorting, which is less efficient than `fsum` in terms of both memory and computation overhead.

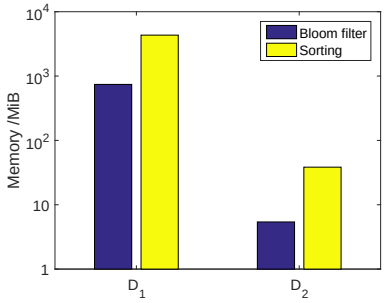

Fig. 5. Memory comparison of the bloom filter and the sorting of two datasets.

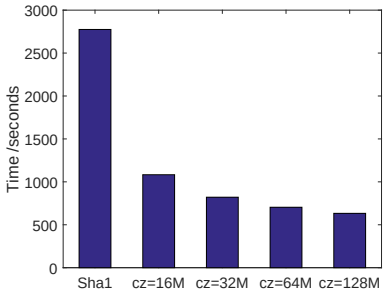

Fig. 6. Comparison between `sha1sum` and `fsum` with different chunk sizes.

V. Conclusions

In this paper, we present the design, implementation and evaluation of a scalable parallel checksumming tool, `fsum`, for extreme-scale dataset integrity check. It is built upon the principle of parallel tree walk and work stealing pattern to maximize parallelism and to overcome the limitation of traditional serial and file-based checksumming tool. It can generate a single and consistent dataset-level signature by aggregating chunk-level checksums, which addressed the unwieldiness of generating and maintaining large number file-based signatures, particularly when the number of files keep growing in large scientific dataset. We also came up with a novel bloom filter-based algorithm to reduce memory footprint and to increase scalability. Using representative and real production datasets, we demonstrated that `fsum` exhibits near-linear scalability, and is able to detect data corruption while it's both memory and computation efficient than others approaches.

Acknowledgment

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] S. Sagiroglu and D. Sinanc, "Big data: A review," in *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, May 2013, pp. 42–47.

[2] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim, S. Gupta, D. Tiwari, S. S. Vazhkudai, J. H. Rogers, D. Dillow, G. M. Shipman, and A. S. Bland, "Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 217–228. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.23

[3] S. Narayan, J. A. Chandy, S. Lang, P. Carns, and R. Ross, "Uncovering errors: The cost of detecting silent data corruption," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09. New York, NY, USA: ACM, 2009, pp. 37–41. [Online]. Available: http://doi.acm.org/10.1145/1713072.1713083

[4] G. Sivathanu, C. P. Wright, and E. Zadok, "Ensuring data integrity in storage: Techniques and applications," in *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, ser. StorageSS '05. New York, NY, USA: ACM, 2005, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/1103780.1103784

[5] F. Wang, "A parallel file system profiler," https://rawgit.com/olcf/pcircle/master/man/fprof.8.html, 2016.

[6] 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.

[7] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "Load balancing using work-stealing for pipeline parallelism in emerging applications," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 517–518. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542358

[8] M. E. Hellman, B. W. Diffie, and R. C. Merkle, "Cryptographic apparatus and method," Apr. 29 1980, uS Patent 4,200,770.

[9] K. Atighehchi, A. Enache, T. Muntean, and G. Risterucci, "An efficient parallel algorithm for skein hash functions." *IACR Cryptology ePrint Archive*, vol. 2010, p. 432, 2010.

[10] P. Sarkar and P. J. Schellenberg, "A parallel algorithm for extending cryptographic hash functions," in *International Conference on Cryptology in India*. Springer, 2001, pp. 40–49.

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

[12] R. C. Merkle, "Protocols for public key cryptosystems." in *IEEE Symposium on Security and privacy*, vol. 122, 1980.