Parallel Graph Reduce Algorithm for Scalable Filesystem Structure Determination

Matthew Andrews*, Jason Hick*, John Emmons†, Kirby Powell‡
*National Energy Research Scientific Computing Center
†Drake University ‡St. Edwards University

***Abstract*—High performance computing is essential to continued advancement in almost all disciplines of science. Computing facilities that process scientific data have increased their data storage and processing capabilities to meet the growing demand; however, preventing data loss across large arrays of error-prone storage devices limits growth when naïve approaches are employed for filesystem back ups. This article describes a scalable, parallel algorithm implemented to determine the structure of filesystems, a necessary component of the back up procedure, which uses Apache's Hadoop MapReduce framework.**

Introduction

Researchers in academia and industry are increasingly utilizing High Performance Computing (HPC) centers for the analysis of large datasets in major scientific breakthroughs. Fields such as artificial intelligence, astronomy, bioinformatics, and particle physics all rely on the ability of modern supercomputers to analyze and store large quantities of data. File systems at HPC facilities have adapted to meet the greater performance demands, but are struggling to find efficient means of backing up all the newly created data.

Furthermore, the need for efficient archival technology has become substantially more important for the continued growth of HPC. The rising number of drives used in contemporary file systems exacerbates the existing problem of high failure rates of current data storage technology. In order to maintain the current rate of scientific advancement, new file systems must be backed up both efficiently to minimize file system downtime in order to ensure a short recovery time in the event of data loss.

In this project, the target file system used IBM's high-performance, clustered General Parallel File System (GPFS) for data storage. GPFS was designed to achieve very high bandwidth for concurrent access to a single file; it achieves this by spliting data across multiple drives [2]. As a result, GPFS is able to operate with the combined bandwidth of all drives, delivering significantly lower read and write times to its files than in other file system designs.

However, with a parallel scheme, the failure of one drive means the loss of data on many more files that with simpler file systems. In order to combat this danger, most GPFS systems typically used a Redundant Array of Independent Drives (RAID) setup and perform frequent back ups to a more stable archival system. The first countermeasure, RAID, is easy to implement, maintain, and does not tax the file system's resources; however, conducting frequent back ups does interfere with the performance of the system.

The challenge associated with frequent back ups is a consequent of the poor facilities available for determining the filesystem structure, the full path to all files or directories. Obtaining the path to all files and directories in file system is vital to performing system back ups; however, GPFS keeps track of files and directory locations using numeric, unsigned-integer values, rather than the path. This gives ways to lower latency and increased data rates in filesystem reads and writes; however, it makes back ups of the system using naïve approaches computationally expensive.

This paper describes the implementation of a highly parallalizable algorithm for the back up of the multi-petabyte GPFS archive at the National Energy Research Scientific Computing Center's (NERSC) HPC faclity. This algorithm reduces the time and resources needed to determine a filesystem's structure; consequently diminishing the overall time and resources used during back up.

Background

In order to prevent the loss of any data between back ups it must be possible to recreate a filesystem with only the information stored during a back up. A simple approach might store a complete copy of a filesystem during each back up; however, this method would heavily disrupt the productivity of a system and

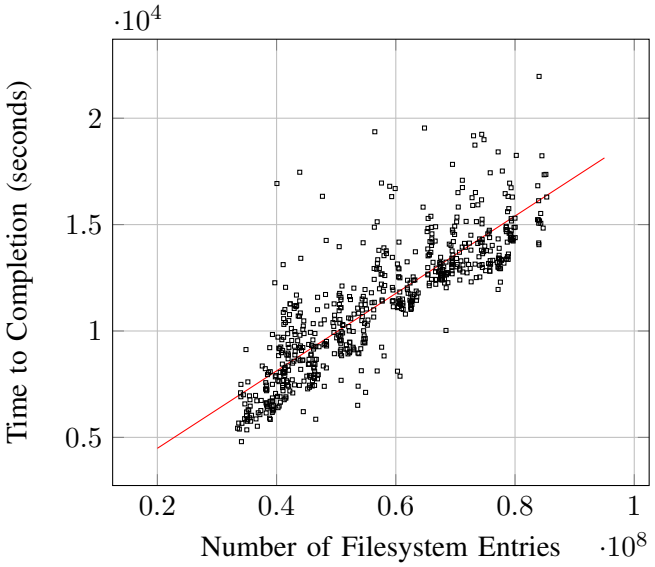

Fig. 1: Time complexity of tree-walk algorithm acting on NERSC Project U1 filesystem.

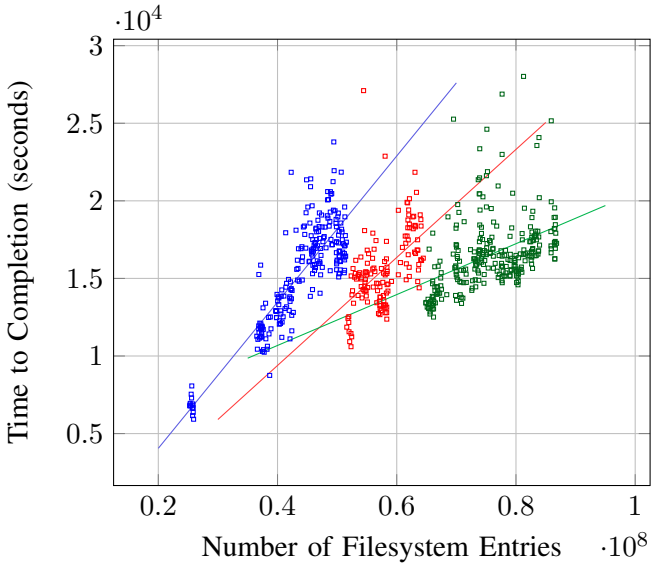

Fig. 2: Tree-walk algorithm acting on NERSC Project U2 filesystem with upgrades.

waste valuable storage space by creating copies of files that have not changed between back ups. Most modern filesystem back up algorithms take complete back ups infrequently and perform incremental back ups on a regular basis. During an incremental back up, only the changes made a to filesystem since the previous back up will be stored. With this method, a filesystem can be restored to a previous state by applying the changes made to a system since the last complete back up. This method ensures the security of a filesystem's data, but uses a fraction of the memory and compute resources of a complete back up.

Despite the use of more efficient back up algorithms, the large filesystems of today's HPC facilities can still take several hours to days to back up. Researchers have focused their attention to a known bottleneck of the back up procedure, the determination of the filesystem structure. Determining the path to all files and directories, the filesystem structure, is key since to retrieve a file or directory's data the path must be known. In order to address this bottleneck, NERSC implemented a parallel, tree-walk algorithm that determines the path with linear time complexity.

Fig. 1 illustrates the time complexity of the NERSC tree-walk algorithm. The graph shows a strong linear correspondence between time to completion and filesystem entries, the number of files and directories in the filesystem, which implies the time to determine the structure of a filesystem is directly proportional the size of the filesytem.

The slope and y-intercept of the curve are both dependent on the speed of random access to the filesystem. Since the speed of filesystem access from a GPFS system is directly dependent to the number of storage devices, the algorithm can be scaled to accomodate large amounts of data by simply increasing the number of the storage devices. Fig. 2 illustrates the time complexity of the tree-walk algorithm on a single filesystem that undergoes two upgrades, where the filesystem recieves additional storage devices in each upgrade.

The three curves plotted represent different states of the same filesystem. The leftmost curve shows the performance of the tree-walk algorithm on the initial implementation of the Project U2 GPFS system hosted by NERSC. The center and rightmost curves show the performance of the algorithm as the number of storage devices were increased during upgrades to the same system. As expected, the slopes decreased as the filesystem access time descreased.

Despite obtaining a linear time complexity, the tree-walk algorithm contains major inefficiencies that are inherit to its operation: it is limited by random access time to the filesystem, and it cannot scale beyond a single compute node. Also, since the algorithm does not utilize information about previously determined structures of the filesystem, it must perform a read operation for every entry in the filesystem (inode) during each back up.

The algorithm described in the remainder of the paper takes advantge of under-utilized resources and implements more efficient methods for filesystem struc-

ture determination. Unlike the tree-walk algorithm, the graph reduce algorithm uses old filesystem scans to reduce the amount of filesystem access. The number of changed filesystem entries is strongly correlated with the number of entries that must be accessed during a filesystem back up, typically orders of magnitude less than the total number of filesystem entries.

Also, its graph reduce nature improves the overall time-complexity of the algorithm. Rather than determing the filesystem structure with linear dependence on depth, the graph reduce algorithm can determine filesystem structure with logrithmic dependence on filesystem depth when sufficiently scaled.

Finally, the graph reduce algorithm can incorporate additional compute nodes to speed up the rate of filesystem structure determination. This is not possible with the tree-walk algorithm which is a simple multi-threaded applicaiton that can only run on a single compute node.

Algorithm Design

Hadoop, developed by the Apache Software Foundation, was the programming framework used for the implementation of the graph reduce algorithm, it was choosen due to its abstraction of parallelization. Hadoop is an implementation of the MapReduce programming paradigm first used by Google in its page rank algorithm [1].

Hadoop MapReduce

The MapReduce model consists of two phases, accordingly known *Map* and *Reduce* which are carried out by user defined mapper and reducer functions. The Apache Hadoop framework was designed such that users need only define mappers and reducers while the framework determines how to parallelize the job across available compute nodes.

For example, a MapReduce job might identify all unique strings in the collected works of Shakespeare. In this example, the *Map* phase would parse the text of these works according to a set of user-defined parameters, then *map* it into key-value pairs, where each unique word is set to the key with the value set to integral one. Once all mappers have completed, the data is then sorted and handed off to the *Reduce* phase according to more user-defined parameters.

In the *Reduce* phase the ordered data is parsed again, then *reduced* into the desired format. Using the same example, the *Reduce* phase consists of summing all of the values associted with each key. Once complete, the output would be a list of each unique string that occurs in the collected works of Shakespeare, with its total number of occurrences. Of course the MapReduce paradigm can be adapted to many problems.

The ability of MapReduce to be scaled across many compute nodes results from the rules imposed on map and reduce functions. Map function must be able to accept any subset of the data as an input and reduce function must be able to accept data associated with a subset of the map generated keys. In this way, each compute node is only ever working on a subset of the dataset, which provides greater potential for parallelism without intervention by the framework user.

Hadoop's ability to abstract parallelization simplified the graph reduce algorithm design. As shown in fig. 3, the system implementation requires three inputs, although in the degenerate case of a filesystem without previous back up data (e.g. a new filesystem), a *First-Time Scan* case can be invoked. The inputs are:

- Previous back up's relative path data (data necessary to generate filesystem paths)
- GPFS filesystem mount point
- Previous back up's inode data (filesystem entry number plus associated filesystem metadata)

In the first time case, only the GPFS filesystem mount point is nessecary to generate full filesytem entry paths

Graph Reduce Algorithm

Firstly, the algorithm runs a full *Inode Scan* of the filesystem using function provided by GPFS to scan the inodes. IBM's implementation of this process is efficient, allowing a multi-petabyte filesystem scan to complete on the order of minutes. The results of this scan are dated and saved to disk for use both in the current and future runs.

Then a *Difference Analyzer* function compares the current and previous inode scan data. This phase indentifies directory inodes that have been added, modified, or deleted since the previous scan. In the first time run case, the algorithm will consider each entry in the current scan to be added/modified. This list of added, deleted, and modified directory indoes then becomes input to the next phase of the algorithm.

A *Filter* phase takes the list of changed directories and removes them (and all their child files or directories) from the relative path data of the previous scan. The previous scan's relative path data consists of the data nessescary to resolve each entry's complete filepath. The Filter removes this data from the previous scan so that once the relative paths of all changed

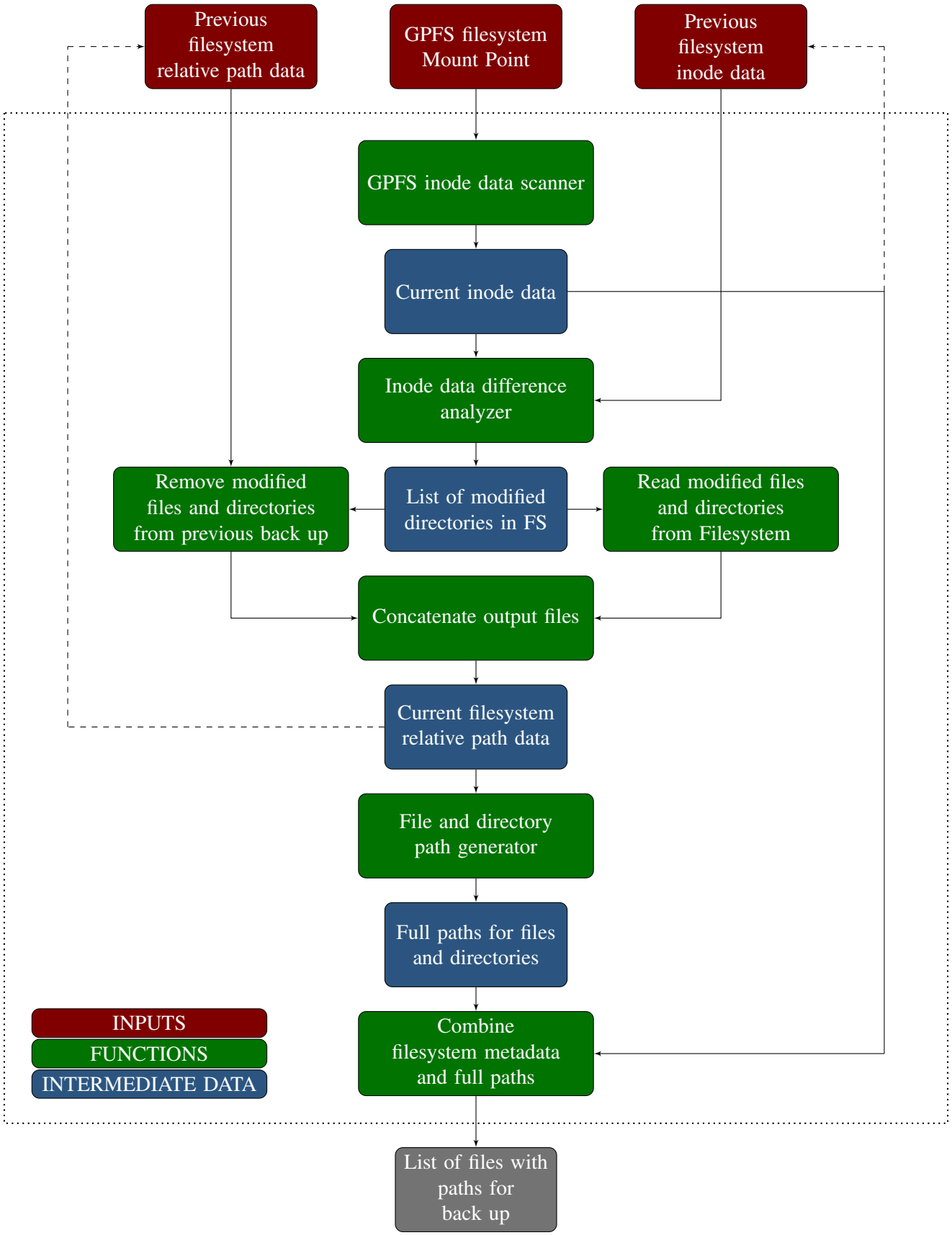

Fig. 3: Overview of filesystem structure determination algorithm. Note functions are internally parallelized with Apache Hadoop for increased scalability. Dashed lines represent data storeage for future structure determination.

Algorithm 1 Path Generator Mapper

```
input: curr_filesystem_data
  for inode in curr_filesystem_data do
    if inode is a directory then
      inode_flag←false, sort_key←inode_parent
      write inode
      inode_flag←true
      inode_parent←sort_key←inode_self
      write inode
    else if inode is a file then
      inode_flag←false, sort_key←inode_parent
      write inode
    end if
  end for
```

Algorithm 2 Path Generator Reducer

```
input: partitioned_mapper_output, filesystem_mount
  children_parent←null
  for inode in partitioned_mapper_output do
    if inode_flag is true then
      children_parent←inode_parent
    else
      inode_parent←children_parent
      write inode
      if children_parent is not filesystem_mount then
        report iterated_algorithm
      end if
    end if
  end for
```

files and directories have been determined their updated relative paths can then be merged back into to the previous list. Once again, in the first time case all data will be treated as new.

A *Read Directories* phase also takes the list of changed directory inode data and reads each of the modified directories along with all child files and directories and adds the relative path name to the respective bit of inode data. Therefore once the Filter and Read Directories phases have both completed there exists a list of all relative path data that is unchanged, and a list of all relative path data that has changed. The outputs from both the Filter and the Reader are then taken and combined to form an updated list of all filesystem data including relative paths.

This list is then passed to the *File Path Generator*, which generates complete file path data for the entire file system. In contrast to the prior tree-walk algorithm, the graph reduce algorithm makes use implements a solution where paths are expanded outwards from each individual entry, consolidating paths as they resolve. This process is preformed recursively until all inodes have the filesystem mount point as their immediate parent. Using the graph reduce method knowledge of the tree grows greatly with each pass through the filesystem, which leads to a marked speed increase compared to the previous algorithm. Futhermore, the graph reduce algorithm only reconstructs the paths that could have possibly changed since the prior scan, rather than reconstructing the entire filesystem. Of course during the first time run case, the entire filesystem will need to be reconstructed.

As shown in Algorithm 1 the mapper of the File Path Generator simply serves to prep the given data for the reducers. During the map phase, each record of data is analyzed to see if it is a directory or a file. If the record in question is a directory then the mapper will output two lines for that record with one being an exact replica with no changes, and the other being an exact replica with the record's own inode listed as its parent. If the record in question represents a file then the mapper will simply output a single line with no changes before moving on. Additionally, a secondary (sort) key is added to each record that allows hadoop to sort all data into the order the reducers need to function properly.

The reducer of the File Path Generator then takes the sorted output of the mapper extends each inode's current filepath to include that of its parent. As shown in Algorithm 2 the reducer looks to see if the each inode is fully resolved. The reducers will then go through all all unresolved input and prepend the relative path of each inode's immediate parent to each inode's current relative path, and update each inode's immediate parent to math the new path. If unresolved paths still exist the reducers will increment a counter. If that counter is non-zero once all reducers complete, the File Path Generator phase will recurse, with the collected and sorted reducer output being input for a new set of mappers.

It is important to note an exception to the input recieved by mappers following File Path Generator recursion. The reducers assume that they will recieve all inodes for a given key. Therefore if one reducer detects a single fully resolved path, it must contain all fully resolved paths. As the algorithm recurses more, the number of fully resolved paths grows logrithmically, leading to a nontrivial imbalance in the workload distributed to the reducers. This is mitigated by saving all fully resolved paths that do not represent directories in a separate location (as these paths will never be needed

in future recursive steps) and only giving the new bath of mappers those fully resolved paths that do represent directories.

The final phase of the algorithm, the *combiner*, takes the the list of all fully resolved filepaths and combines them with the current inode data to form a list that contains the fully resolved file paths of the entire filesystem along with all filesystem metadata. It is this complete representation of the filesystem that is then passed to the back up system.

Algorithm Performance

To analyze the performance of the graph reduce algorithm it was necessary to identify the area which constrain performance at the extrema, in this case filesystem sizes approaching infinity. Much information about the scalibility of an algorithm can by obtained by examining only the internal functions that dominant at the extremes.

In the graph reduce algorithm, the time to completion of the generate paths function dominants all other factors as the size of the filesystem becomes arbitrarily large. Therefore, information about the scaling of the graph reduce algorithm was investigated by testing only the generate paths function.

Metrics

The performance of the graph reduce algorithm was measured using the time to complete a filesystem scan as the primary metric. Time to completion provides a useful measure of scalibility and a common variable for comparisons between the tree-walk and graph reduce algorithms. It also maintains underlying mathematical relationships between various parameters despite being measured using machines with differing components.

However, time to completion does have some limitations. It does not provide sufficient information to make accurate predictions of algorithm performance on an arbitrary machine. Further metrics would need to be tested to determine the performance on machines with varying components.

Also, while some comparisons can be made between the tree-walk and graph reduce algorithm, the inherit differences in their designs provided largely misleading information when using time to completion. For example, both the tree-walk and graph reduce algorithms benefit from decreased filesystem access times; however, since the graph reduce algorithm performs substantially less filesystem reads, it does not benefit to the same degree. Nonetheless, time to completion serves as a reliable metric such that care is taken to avoid making unwarranted assumptions.

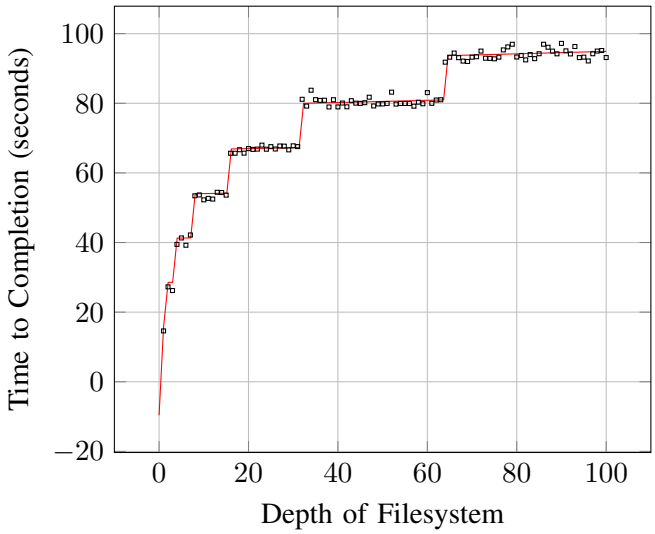

Fig. 4: The time to completion data for analysis of a uniform filesystem tree with branching factor of one.

Time Complexity

Fig. 4 illustrates the logarithmic dependence of the alogirthm on the maximum depth of a filesystem. The filesystem investigated was completely artificial; it contained no files and had a simple linear structure, only a single branching directory at each level of the filesytem tree. For example, in a linear filesystem containing only three directories, A, B, and C, directory C would be contained within directory B, which would be contained within direrctory A. In this way, the algorithm spent a negligible amount of time processing data with most of the time to completion attributed to the algorithm setup and cleanup.

The logarithmic step relationship between time to completion and depth of the filesystem was a consequent of the *graph reduce* nature of the algorthim. Since file paths are built by prepending the relative path of parent inodes, the number of cycles necessary to fully resolve all paths is $\lfloor \log_2(\text{depth}) \rfloor$. For example, filesystems of depth 2 and 3 required two iterations of the generate paths function; however, once the filesystem depth increased to 4, 5, 6, 7, or 8, the algorithm required three iterations. The discontinuous jumps are attributed to the additional setup cost of performing an increased number of generate path cycles.

The logarithmic step relationship were also observed when the generate paths function acted upon more complex filesystems. Fig. 5 plots time to completion data taken on an artificial filesytem with uniform branching

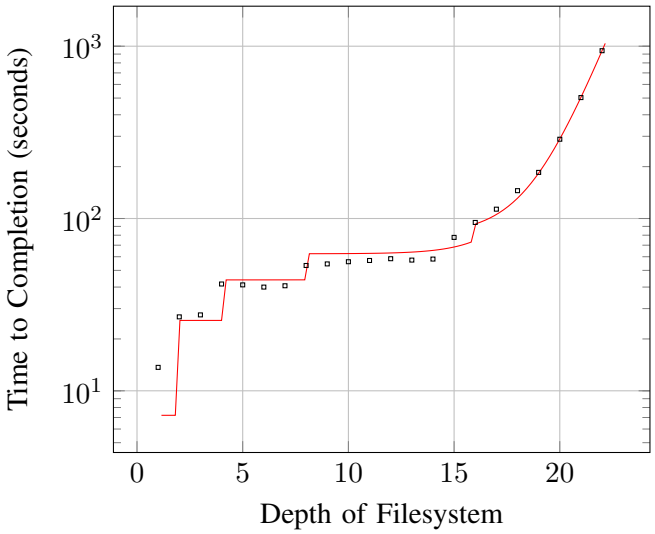

Fig. 5: Time to completion for analysis of a uniform filesystem tree with uniform branching factor two plotted with depth.

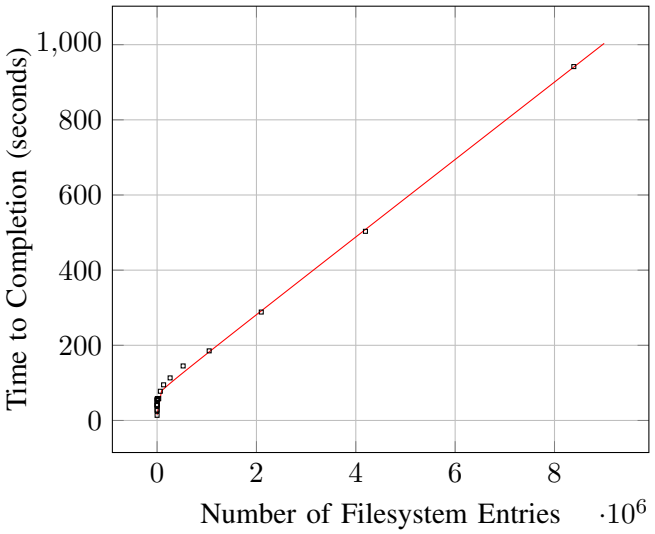

Fig. 6: Time to completion for analysis of a uniform filesystem with uniform branching factor two plotted with entry counts.

factor two, meaning each directory contained two other directories as children.

When isolated, the first part of the data had the same logarithmic step relationship between filesystem depth and time to completion was observed; however, this relationship broke down as the depth increased further. This can be explained by the exponential increase in the number of filesystem entries as the depth increased.

As a result, the time spent on the function setup and cleanup became negiligible compared to the time spent processing the data during each iteration. Since this plot has a log y-scale, this is seen as a straight line relationship.

The same relationship can be seen in fig. 6, which follows a similar setup to fig. 5 except with the number of filesystem entries plotted against the time to completion. Note the region of step logarithmic growth is much smaller than in fig. 5; this shows how quickly the logrithmic behavior was dominanted when limited compute resources are employed. Also, note the linear behavior as the number of filesystem entries increased. Like the tree-walk algorithm, the graph reduce algorithm also had a linear time complexity.

Scaling

Filesystems at most large HPC centers contain millions of entries; thus, the extent to which the graph reduce algorithm can be scaled is vital to its value as a back up algorithm. However, before the scaling potential of the algorithm across many compute nodes could be accurately determined, the optimal parameters had to be found for single node. This is underscored by the great effect input parameters had on the performance of the algorithm while operating on a single compute node.

The key input parameters of the graph reduce algorithm were the number of map and reduce tasks that run concurrently on a single compute node. Fig. 7 shows how the time to completion varies with the two parameters.

The plot shows the time to completion was minimized by using the maximum number of map and reduce tasks allowed by Hadoop, six maps and two reducers. This was a logical conclusion as that set of

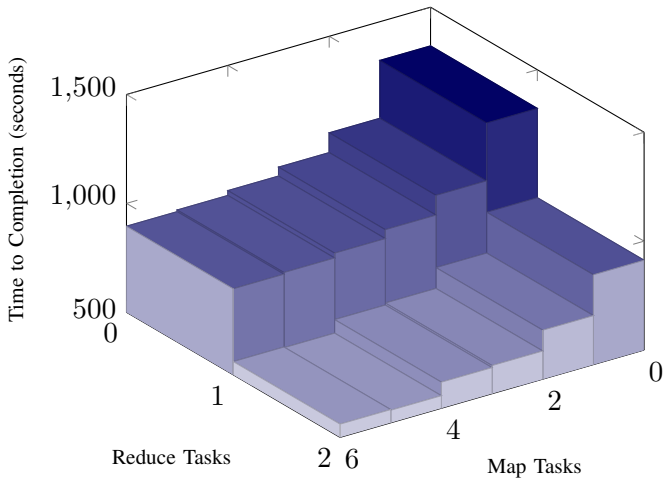

Fig. 7: Analysis of a constant size filesystem with varying map and reduce task on a single compute node.

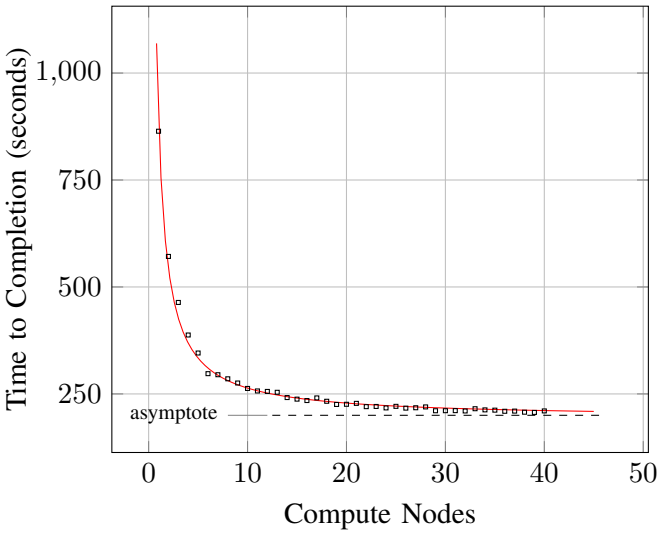

Fig. 8: The time to completion data for analysis of a uniform filesystem tree with branching factor of one

parameters takes advantage of all eight processors that make up a single compute node in the testing computer.

Once the optimal number of map and reduce tasks was found, the number of compute nodes used to the process the data was increased. Fig. 8 shows how the time to completion decreased as the number of nodes was increased. The filesystem that was analyzed was of constant size (depth 23 and uniform branching factor 2) which contained a large enough number of entries such that the setup and cleanup time of the algorithm was neligible.

The inverse relationship between time to completion and the number of compute nodes is consistent with the time complexity data provided by fig. 6. Since the filesystem size was constant, the data was partitioned into smaller and smaller pieces as more compute nodes are added to the job. Interestly, as the number of compute nodes become arbitrarily large, the time to completion approaches a nonzero asymptote. This results from the inescapable setup, cleanup, filesystem reads, and intermediate data processing required as part of Hadoop and the graph reduce algorithm

Conclusions

The graph reduce algorithm yields a significant performance improvement over the tree-walk algorithm for generating fully resolved paths in filesystem scans. This improvement is a result of several differences between the two algorithms, chiefly in the design.

As shown in fig. 1, the prior algorithm can scaled; however, this scalability is dependent on the latency and data access rate of the filesytem, lower latency and higher data rates increases perforamce. When a parallel filesytem scheme like GPFS is employed the latency and access rate of fileysystem is dependent on the number of independent drives that compose the filesystem. As the number and speed of drives increase, so too does the relative preformance of the algorithm. However, adding a vast amount of drives to a complex system without causing disruption in service or loss of data to meet increased performance demands is nontrivial.

Conversely, as shown in fig. 4, the graph reduce algorithm scales primarily by the number of compute nodes while also benefting from the same filesystem upgrades used to scale the tree-walk algorithm. The compute node scaling is a consequent of the Hadoop parallel MapReduce framework employed in the project.

HPC centers, such as NERSC, typically manage several hundred thousand compute nodes, many of which are idle during off-peak hours. While no thorough cost-benefit analysis of scaling the graph reduce and tree-walk algorithms was done, by using these underutilized compute resources the overall cost of scaling the graph reduce algorithm is small. Additionally, even systems without excess computing capabilities may still benefit from the graph reduce algorithm. The compute resources used by the graph reduce algorithm will only be unavailable for general purpose tasks during back ups, they can be absorbed into the larger system during normal operation, recovering their cost.

Differences in implementation and scaling between the graph reduce and tree-walk algorithms limits the value and increased the difficulty of obtaining direct comparisons between them. The key performance differences between the two algorithms is best compared using Big O notation and an argument on overall scaling and necessary user interference.

Paradoxically, using Big O notation the graph reduce algorithm has a greater time complexity than the tree-walk algorithm it is designed to replace. Nonetheless, due to the outlined differences in scalability, a user of the graph reduce algorithm is able to more cost-effectively back up their filesystem. To reiterate, the tree-walk algorithm scales only with the number of storage drives; while improving filesystem and back up performance, adding more drives may not be the best overall investment. Unlike the graph reduce algorithm, the resources used by the tree-walk algorithm cannot be yielded back to the HPC center during normal opertaion. This greatly increases the opportunity cost of scaling the tree-walk algorithm.

Furthermore, there major costs associated with

filesystem upgrades. The upgrades present in fig. 2 took months of planning and weeks of effort to complete. The graph reduce algorithm scales simply by adjusting compute node parameters.

Additionally, the tree-walk algorithm must make many more filesystem access requests during a back up than the graph reduce algorithm. As a result, during a back up the overall performance of the filesystem decreases substantially using the the tree-walk algorithm. This hinders the ability of high demand HPC centers, like NERSC, from reaching their full operating potential.

In conclusion, the graph reduce algorithm has many advantages over the tree-walk algorithm. It capitilaizes on historically under-utilized resoures, yields significant performance improvements, and can be scaled much more easily and with less human intervention.

Future Work

Given that the current codebase is written primarily in Python, rather than in native Hadoop Java code, there exists the possibility that statistically significant performance gains could be had simply by rewriting the current codebase in native Java. It would be interesting to see if a Java implementation of the algorithm would be worth the extra development time.

The *Filter* and *Read Directories* Hadoop jobs currently execute sequentially, but no need exists for this behavior. As shown in Fig. 3, these portions of the algorithm share no dependancies, and nontrivial performance gains potentially exist if these phases were parallelized.

During the process of implementing the current algorithm an addition to the Hadoop framework named "Hama" was brought to the attention of the team. As discussed in the Algorithm Design section, the implemented solution for the *File Path Generator* is a graph reduce problem. While Hadoop is obviously more than capable of solving graph reduce problems efficiently, it does not appear to be the most ideal solution. Hama was built to apply the strengths of Hadoop to graph problems using a paradigm better suited for solving graph solutions [3]. This focus on graph problems should intuitively give a Hama implementation of the File Path Generator a nontrivial perforamnce increase over Hadoop.

Acknowledgements

The team would like to thank Matt Andrews and Jason Hick for their limitless aide and advice over the course of the summer. The team would also like to thank TRUST Program Director Aimée Tabor for her extraordinary support. Finally, the team cannot thank our respective families enough. Thank you all for putting up with our extended absence and limited availability this summer. This work was supported in part by TRUST (Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422).

References

[1] "Apache hadoop." http://hadoop.apache.org. Accessed July 2013.

[2] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters.," in *FAST*, vol. 2, p. 19, 2002.

[3] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 721–726, IEEE, 2010.