On Distributed File Tree Walk of Parallel File Systems

Jharrod LaFon[*†], Satyajayant Misra[*], and Jon Bringhurst[†]
[*]New Mexico State University , [†]Los Alamos National Laboratory

***Abstract*—Supercomputers generate vast amounts of data, typically organized into large directory hierarchies on parallel file systems. While the supercomputing applications are parallel, the tools used to process them requiring complete directory traversals, are typically serial. We present an algorithm framework and three fully distributed algorithms for traversing large parallel file systems, and performing file operations in parallel. The first algorithm introduces a randomized work-stealing scheduler; the second improves the first with proximity-awareness; and the third improves upon the second by using a hybrid approach. We have tested our implementation on Cielo, a 1.37 petaflop supercomputer at the Los Alamos National Laboratory and its 7 petabyte file system. Test results show that our algorithms execute orders of magnitude faster than state-of-the-art algorithms while achieving ideal load balancing and low communication cost. We present performance insights from the use of our algorithms in production systems at LANL, performing daily file system operations.**

***Index Terms*—File Systems, Metadata, Parallel Algorithms**

LANL unrestricted release number: LA-UR-12-20996.

I. Introduction

The amount of scientific data produced today has been keeping pace with the increases in disk/memory capacities and densities. On large compute clusters consisting of hundreds of thousands of processors, applications easily generate millions of files per job. Scientists often use sophisticated tools to write applications that read, compute, and store these data, in parallel, on large distributed storage systems. However, the tools and algorithms used to traverse file systems to archive the data to long term reliable storage, or post-process the data for visualization or statistical analysis, are often serial, making data archiving or searching time consuming. The few tools that exist for parallel processing [1] and archiving [2] use centralized parallel algorithms for load balancing and work distribution, leading to unnecessarily high communication overhead. Traversal of large file trees in parallel is a common problem encountered in parallel storage systems, but has received limited attention.

Problem Motivation: There are numerous motivational examples for the parallel tree traversal/walk problem. One such example is the problem of copying a large file tree to long term archival storage. In a naive implementation, a single client could serially copy each file, under-utilizing the parallelism available. A simple parallel implementation may use a pool of worker processes and a master process, where the master process dispatches tasks to the worker processes. The MapReduce framework [3] designed by Google uses this master and slave strategy. In the MapReduce architecture, the file tree operations tasks given to slave processes could be copying a single file, or a segment of a file, allowing large files to be copied by multiple workers.

While parallel, this centralized implementation suffers from communication overheads. The master process needs to keep track of which worker processes are busy, and what they are working on. Each new task requires a minimum of two messages – the dispatch of work unit from the master to a slave, and the reply from the slave to the master. Also, the master process must maintain a global list of tasks to be performed. For very large file trees, such a list can outgrow the memory available to the master process.

Another example is the checkpoint and restart facility in large parallel applications. Parallel applications run across hundreds or thousands of distributed processors in a parallel system. With the number of components in such a system rapidly increasing, the probability of a hardware component failure is also increasing rapidly [4]. Checkpointing and restart facilities are being used widely to help parallel applications improve fault tolerance in the event of hardware/software failures. In addition to allowing an application to restart from a checkpoint, this data can also serve as application output. Searching, indexing, and processing this output from very large compute jobs can be prohibitively expensive for serial/naive parallel implementations. Our algorithms are especially geared towards performing these actions with little overhead.

Another example is file tree traversal (walk). A serial file tree traversal mechanism is implemented as part of the Linux kernel in accordance to a POSIX specification [5]. A user provides a function to be called back for each node in the tree, and then calls the function $ftw(path)$ [5] to begin a traversal on a given path. The file system tree rooted at $path$ is then explored in a pre-order traversal. In addition to the call back, metadata about each node is supplied to the user provided call back function to avoid performing an additional metadata lookup for the same file. This solution is adequate for normal sized directories where serial performance is satisfactory, but the serial nature of the Linux file tree walk severely limits the speed of directory traversal. Moreover, there is a serious problem traversing deep file trees.

An obvious improvement to the serial file tree walk is a parallelized version. At Los Alamos National Laboratory (LANL), a parallel file tree walk algorithm has been developed in-house and is used for gathering metadata on very large

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

parallel file systems [6]. This algorithm was a first attempt at large file tree exploration in parallel, and showed a significant speedup over the serial algorithm. It uses a centralized task distribution paradigm with a master process for control and slave processes to perform individual node explorations. It also allows multiple directories to be explored simultaneously by multiple worker processes. However, the centralized master/slave algorithm requires communication between the slave processes and the master for each task assigned by the master to a given slave. This occurs first when the task is assigned, and then again when the work is returned to the master. As we will show in Section II, this overhead is significant. **This served as our motivation to study the problem of parallel file tree walk and design/develop algorithm(s) to improve not only the operating time, but also reduce the message complexity.**

In this paper, we propose an algorithm framework and develop three efficient algorithms, each of which can be used in different scenarios. In our framework, we obtain the improvement in running time and message complexity by dispensing with the synchronization requirement and by avoiding a centralized control process altogether. Our framework is a fully distributed framework for workload distribution, applied specifically to the problem of file tree exploration. We use a *randomized* work stealing scheduler [7] to efficiently balance workload between the worker processes. This mechanism is popularly termed as work stealing because idle workers 'steal' work from other processes which have pending work in their queue [7]. Our algorithms show a large speedup over the centralized parallel algorithm, with drastically reduced communication overhead. This reduced communication overhead (and complete lack of collective communication) allow our algorithms to scale much better than the previous solution. For instance, when run on a file tree consisting of 100 million files, our algorithms *exchange two orders of magnitude less* bytes than a centralized parallel algorithm and take less than a fourth of the time to run on an average with the *same number of processes*. We also perform comparisons among our algorithms and identify their suitability for different situations.

To our best knowledge, our proposed framework and the algorithms are a first, and a novel, attempt in the literature to perform truly distributed operations in large parallel file systems without the need for explicit synchronization. Our solution can find application in many large scale data storage systems and provide significant speed-ups at reduced communication costs. *In fact, it is currently under use in production systems at LANL.* The results presented in this paper are obtained from the runs of our algorithms in these production settings. As our solution has potentially widespread usage and draws on several areas of High Performance Computing for concepts. We begin by describing file system metadata storage techniques, the serial file traversal algorithm, works in parallel file tree traversal at LANL, and relevant graph exploration algorithms in the related works section, Section II. Then, we describe the parallel file tree traversal, and other relevant concepts that will help understand our framework better, in the Background section, Section III. We present our framework and the three related algorithms, and improvements in Section IV. We present the empirical results of our experiments on large scale systems at LANL in Section V. Finally, we conclude our paper in Section VI.

II. Related Work

File systems are organized into trees, which can be traversed serially using well-known algorithms, such as Breadth First Search (BFS) or Depth First Search (DFS) [8]. The supercomputing research community has expended significant research efforts in designing efficient parallel graph exploration techniques, a few notable ones being [9]–[11]. Many existing graph algorithms have been studied and modified with success to exploit parallelism [12]. For instance, a typical parallel graph search and exploration algorithm used widely is the Depth-First Branch and Bound (DFBB) [13] algorithm. In the DFBB algorithm, parts of the search space (subtrees) that do not contain an optimal solution are eliminated by using heuristical branch and bound [14]. Breadth First Search (BFS) is parallelized by maintaining a global frontier of unvisited vertices [15]. During each iteration, every process atomically acquires one vertex from the frontier. It then explores the vertex to discover the vertex's neighbors. The process then adds the unexplored neighbors of that vertex (if any) to the frontier of unexplored vertices. Between explorations, one or more processes need to take the list of unexplored vertices and reduce it to a *set* to eliminate repeats. However, all these parallel implementations require process synchronizations, which becomes very costly as the number of parallel processes is increased [16]. In normal BFS, there is a possibility of having multiple paths to a given vertex. For a tree with a branching factor b, and a depth d the asymptotic time complexity for BFS is $O(b^d)$ if multiple paths are allowed. These are some of the challenges in parallel graph exploration.

However, parallel file tree exploration is different from parallel graph exploration on both fronts. The fact that in a parallel file tree traversal every node in the tree must be visited makes the problem unique as we cannot exclude or ignore any subtree within the file tree. This is in contrast to other parallel tree or graph algorithms. Moreover, there is no express need for synchronization of the slave processes, as we will show in this paper. Thus there is scope for major run time speed-ups. In essence, the algorithms for parallel graph exploration do not apply to parallel file tree exploration, because of these major differences, among others. The problem of exploring a file tree as a graph differs from conventional graph exploration in an important way – it's representation. A graph can be represented efficiently, however the biggest impediment for file tree exploration is the that in order to construct the graph representation, complete graph information must be known *a priori*. Moreover, the cost of determining the neighbors of a vertex in a standard network graph in a serial setting is generally attributable to the cost of the data structures used to represent the graph. In contrast, for a parallel file tree this cost is affected by external factors, such as the network hop count, the type of parallel file system, network bandwidth, metadata storage paradigm, file system attribute caching and other variables. This makes the task of optimizing file tree traversal more complex.

However, parallel file tree exploration has not received much attention in the community. *Our work is a novel attempt to fill this void.*

A. Filesystem Metadata

Smaller, serial file systems can reduce the cost of attribute gathering (a requirement for any other file operation) by indexing directories for fast traversal. In Linux ext3 file systems, HTrees are used for directory indexing [17]. HTrees are based on B-Trees, except that the maximum depth is limited. The cost for searching a B-Tree is the same as that of searching a binary search tree, $O(\log n)$, but under the assumption that the directory structure is indexed and in RAM. This requirement makes serial file systems unsuitable for large supercomputing applications. The obvious answer is to use large parallel file systems.

One popular open source file system is Lustre [18]. Lustre stores all file system metadata in a single high performance data store. This effectively converges the data path for gathering attributes to a single server. However, many Lustre metadata optimizations have been implemented to maintain high throughput from the metadata server, so that parallel processes can make use of it. Despite using an optimized single metadata server, the problem of traversing file trees in parallel is still applicable to Lustre. This is because, once a process has credentials for reading or writing a file, the metadata server is no longer in the data path. A credentialed Lustre client communicates directly with storage devices to speed-up the parallel file operations (until new credentials are needed).

Another state-of-the-art parallel file system, Ceph [19], uses a novel approach to metadata. Ceph employs a scheme known as Dynamic Subtree Partitioning to distribute metadata. Ceph dynamically stores file system subtree attributes on individual metadata servers, unless certain criteria are met. The notion of a subtree's popularity is measured and stored for a short amount of time. Using this information, Ceph can remap a subtree's attributes to multiple metadata servers if the workload warrants it. Ceph also allows for proximity optimizations of the file subtree's attributes, stored across multiple metadata servers.

The current file system in production at LANL is a state-of-the-art proprietary parallel file system called Panasas [20]. Panasas distributes metadata for files as file components across the storage system for performance and redundancy. These metadata storing components can be accessed in parallel, and the Panasas file systems has implemented many optimizations for fast metadata access.

We note that our framework for distributed parallel file operations can be plugged into any existing parallel file systems and improve their performance in indexing searches, by virtue of the enhanced parallelism provided. As we will show in the empirical results section, our algorithms require 25% of the time on an average to perform indexing in comparison to existing techniques.

B. File Tree Walk

We know that serial file systems are not useful in the supercomputing environment. So let's look at parallel file tree walk.

Algorithm 1 Centralized Parallel File Tree Walk

```
 1: $S = \emptyset$ for slave processes, $root$ for the master
 2: if $processor\ rank == 0$ then
 3:   $i = 0$
 4:   while $|S| > 0$ do
 5:     Receive Message from Processor $j$
 6:     if Message is a work request then
 7:       $p = S$.dequeue()
 8:       Send $p$ to $j$
 9:     else
10:       $S$.queue($Message$) {Work to be processed}
11:     end if
12:   end while
13: else
14:   repeat
15:     if $|S| = 0$ then
16:       Send work request to Processor $0$
17:       Receive Message from Processor $0$ into $path$
18:     end if
19:     if $path$ is termination sentinel then
20:       exit
21:     end if
22:     if $path$ is a file then
23:       process($file$)
24:     else
25:       $S = \emptyset$
26:       for all $child$ in $path.children()$ do
27:         $S$.queue($child$)
28:       end for
29:       Send $S$ to Processor $0$
30:     end if
31:   until $path == \emptyset$
32: end if
```

Centralized Parallel File Tree Walk: The first centralized parallel (CP) file tree traversal algorithm was developed in-house at LANL (2007) and used a dynamic centralized load balancing technique [8]. For the benefit of the reader Algorithm 1 presents the CP algorithm. Line 1 initializes the work queue S to be empty for all slave processes, and $root$ as the master. The master process (Process 0) executes Lines 2-12, while the slaves execute Lines 14-31. The master enters a while loop in Line 4, and stays in the loop until S is empty. The basic idea of CP is the following: When the master process receives a message from a slave process j (Line 5). If j is requesting work, then a work element is removed from the queue and sent back to j (Lines 6-8). Otherwise, j has

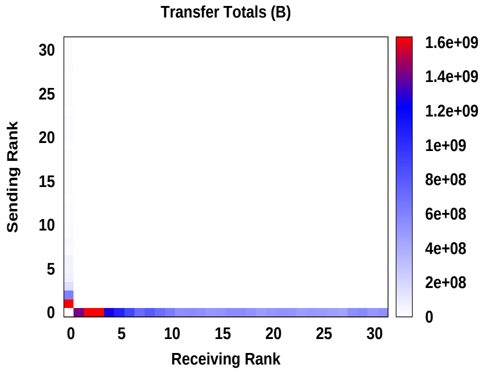


Fig. 1. Centralized Parallel Tree Walk: Communication Cost

returned work results (directories and files) that need to be to be added to the queue (Line 10). The slave processes begin their execution in Line 14, entering a repeat until loop. A slave checks (Line 15) to see if there is work to do, if not, the slave requests work from the master (Lines 16-17). After receiving a response, the slave checks for a termination sentinel from the master (Lines 19-20). Otherwise, the master has sent a path to explore. The user provided callback function is called in Line 23 to perform any necessary work on the path if it is a file. Lines 25 through 29 determine the descendants of $path$ (if any), and send that information back to the master process.

In CP, the master process distributes work units to each of the worker processes, one at a time. Each worker performs the work assigned and sends the results back to the master process (which may contain new work to be added to the work queue). Until the queue is empty the master process meters out a portion of work to each slave process, and then waits for a response from each one, which requires process synchronization. Additionally, a minimum of two network transfers is required for every item in the work queue, which could prove to be very time consuming for large diameter networks. The actual implementation of this algorithm assigns certain tasks only to certain slaves. For example, in the LANL implementation only one work process from the slave pool is used to read the contents of directories. This causes a disproportionate amount of work to be done by one slave process in the event that large directories are present in the input.

Fig. 1 shows the total number of bytes exchanged between different process ranks. The results are output from an instrumented file tree walk using CP. This test was performed on a supercomputer at the Los Alamos National Laboratory using a 471 TB Panasas [20] file system consisting of approximately 6.5 million files. The graph is a heat map showing the total volume of bytes exchanged between each process pair. The color spectrum goes from white, which represents zero bytes exchange, to red, which represents 1.6×10^9 bytes exchange. The vertical and horizontal axes represent the MPI rank numbers of the processes (total of 30 ranks). A square, for example, the square at $(1, 0)$ represents the number of bytes sent from Rank 1 to Rank 0 (master) for the entire job. Observe that communication strictly occurs between the master process and slaves, but never between two slaves. The heat map clearly demonstrates inefficient use of the network interconnects, which in turn creates a performance bottleneck at the master process.

For every leaf node in the file tree, one communication occurs between the master and a slave process, namely transmission of the results from the file operation. However, for every internal node (directory) in the file tree, two communications occur – the work from the master to the slave process and the results returned by the slave. As we will show later, this leads to an asymptotic network cost of $O(n^2)$, where n is the message length. The first component is the network latency incurred by every transfer. The second component is the worst case message size for a process which operates on a directory containing $n - 1$ entries in it. This would occur in the case where the file system contained one directory with $n-1$ files in it. The quantity is multiplied by two because each operation requires two transfers, from master to slave and then back to the master. Since the file system is random access, all operations done by slave processes are equivalent in cost.

III. Building Blocks of Our Framework

A. Parallel Tree Traversal

Our goal in this paper is to design a parallel algorithm for parallel file systems tree exploration, in which each node in the tree is visited exactly once. Additionally, we seek an ideal load balance, with equitable load-distribution, where all the parallel processes performs the same amount of work. In the case of a file tree, the work is visiting nodes in the tree, and performing any legal file operations on each node. Operations on one node are mutually independent from those on other nodes – a fact we exploit in the design of our algorithms. There is a provision in most file systems to use links (symlinks, hardlinks, etc.). Links can cause cycles in the file tree, which would result in a traversal algorithm going into an infinite loop. To prevent this from happening, we ignore links in the file tree during traversal. We note that the algorithms we propose in the paper will duplicate effort proportional to the number of hardlinks. However, in real world production systems, such as in LANL (and others), for simplicity, the parallel filesystems are generally not POSIX compliant, that is, they do not use hard links, inodes, and symlinks. So, our assumption holds.

We note that this problem is embarrassingly parallel. That is to say, there are no dependencies between the units of work (subtrees) to be processed. This allows us to ignore the order in which items are processed. Now we describe in detail some of the concepts that form the bare-bones of our framework. This detail description here would facilitate better understanding of our algorithms and the empirical results.

B. Inter-Process Communication without Global Synchronization

We seek to visit all nodes within a graph (in this case a tree) in parallel, as quickly as possible. One way to achieve this efficiently is by avoiding global process synchronization. Any rendezvous between all processes in a parallel job must be coordinated by way of communication, and this is known to be costly [16]. We make the distinction between pair-wise communication, and collective communication.

Pair-wise communication refers to a message transfer that occurs between two processes. Note that one transfer may be referred to as one message, but if a reliable network protocol is used that single message may require more than one packet, which can cause even a single message to incur network latency cost more than once. Communication may be blocking (synchronous), or non-blocking (asynchronous). We use non-blocking communication semantics, thus we avoid processes being blocked. One process may initiate a communication request and continue doing useful work, periodically checking to see if that request has been answered. Likewise, while a process is processing work, it can periodically check for outstanding communications from other processes. To accomplish this, we use the asynchronous messaging mechanism provided by Message Passing Interface (MPI) [21].

We define a collective communication as a message exchange which is meant for all processes. This type of communication is normally used due to an implicit order dependency in the computation, such as a need to find a minimum or maximum value of a variable across all processes synchronously. In other words, collective communications are a form of synchronization. Our approach avoids collective communication, using pair-wise communication exclusively. Additionally, we benefit from the pair-wise communication being non-blocking.

C. Work Depth Model

To compare the complexity of our parallel algorithm with existing work, we use the Work Depth (WD) model's [22] notion of complexity for parallel computation, which is typically used for Parallel Random Access Machines (PRAM) [23]. A PRAM model is used as an analogous way to describe parallel machines without the need for communication or synchronization so long as the memory access performed by the processes obeys the semantics described in the PRAM model. In the WD model, the work $W(n)$, where the input size is n, is defined as the number of operations to be executed, and the depth $D(n)$ is the greatest number of sequential dependencies occurring in the computation. Brent *et al.* [24] have shown that if we are given $W(n)$ and $D(n)$, we can place reasonable bounds on the running time for p processors [24]. They also showed that the running time T can be bound as $\frac{W}{P} \leq T < \frac{W}{P} + D$. The parallelism of an algorithm is then defined as $P = \frac{W}{D}$. Additionally, we use the simple model for network cost: $C(n) = \alpha + n * \beta$ where α is the network latency incurred for every network transfer, β is the average transmission cost for one network unit, and n is the message length (in units). We note that the work is fixed as $W(n) = O(n)$ for all of our parallel algorithms, since we must visit every node in the tree exactly once. The areas for improvement therefore are the network cost $C(n)$ and the depth $D(n)$ of the algorithms. Note that the depth $D(n)$ of the CP algorithm is $\frac{n}{p}$ for p processes operating on n tree nodes, since the master process distributes the work evenly among the slaves.

We also make the assumption that the file system tree is random access, that is, the cost for querying the neighbors of any given vertices are equivalent. Modern parallel file systems do perform caching, but we are interested in very large file trees that do not fit in any file system cache. In fact, we are interested in file trees that do not even fit in the memory of a single machine.

IV. A Framework for Distributed Parallel File System Traversal

We first present the framework on which the three algorithms to be presented in this section are based. The framework is essentially a set of design principles that we follow.

A. Design Principles for the Framework

Parallelism via the Message Passing Interface: We implement our algorithms using the Message Passing Interface (MPI) [21] standard for parallelism. MPI allows parallel processes to communicate within an MPI Communicator, which can be created dynamically and have multiple processes associated. Each process has a unique integer identifier, its MPI *rank*, within a given communicator. A rank can then be used to uniquely identify processes for communication [21].

Anyone-Asks-Anyone: We create a distributed parallel algorithm framework by using the principle that there is no master process, all processes in the system are equal, and any process can ask any other process for work. The file tree exploration does start at a single process, in our case it is process $Rank\ 0$ termed as the $root$ process. But once the work gets distributed, a process can ask any other process for work. The only centralized operation is termination detection, which is centralized for the sake of efficiency. The root process is responsible for initiating the termination procedure (via token generation) and detection. More on this follows. Of course, in a large system it is inherently wasteful for a process to ask another process, which is several hops away, for work. Hence, it makes better sense in terms of latency and communication cost for a process to execute an expanding search for requests – ask direct neighbors, ask neighbors neighbors, and so on.

Light Weight Processes v/s Single Processes: Given the I/O bound nature of the tree walk, it makes sense for having multiple threads/processes on each compute node. However, if all co-located threads/processes ask each other for work, don't find any, and all send work requests over the network, it is inherently wasteful. Instead, one of them can be designated to seek work from remote processes, after which all co-located threads/processes can share the work.

Random Splitting v/s Equal Splitting: When a process P_i receives a request for work from another process P_j, if P_i has a queue of pending tasks to complete (directories/files to explore), it can provide P_j a part of its pending tasks, this is termed *queue splitting*. Process P_i may split the queue in equal halves or into two unequal parts randomly and assign one of the parts to P_j. Unless a task in the queue is the exploration of a particular file, that is, it is a directory instead, then process P_i has no idea how much time it would take to explore the task. This is because it is not possible to know the nature of the descendants of a directory (whether they are files only, directories and files, or directories) and the depth of the subtree rooted at the directory without exploring it. Due to the unbalanced nature of file trees, an even split may lead to suboptimal load balancing. For example, if P_i always splits the queue equally (non-cognizant of the next level), it may lead to P_i giving away a large proportion of leaf nodes, while keeping its portion of the queue filled with nodes representing large subtrees – an unbalanced split. There is a possibility that such a split happens often, and will result in P_i exploring a larger portion of the internal nodes in comparison to its requesting processes. This not only causes inequitable load balancing, but also causes more network traffic as the requesting processes end up sending out more work requests. It is easy to see that the network cost will be $O(n^2)$.

For even work distribution, queue splits may occur many times; the challenge is to ensure an even balance with the least number of splits. In our algorithms, splits are not globally synchronized, only the work exchange between a pair of processes is locally synchronized. Asymptotically, the queue can be split at most $O(n)$ times for n elements. This is because for every split, the splitting process will always keep at least one element to process. No matter how the work is distributed amongst the processes, the pending work of a process decreases by at least one unit after a queue split. This is true for all processes, thus ensuring termination.

If we assume that there are p processes, and each process consumes one work unit (initially n units), between two consecutive splits of its queue, then it can be proved that the work to be done by a process decreases by at least one for each queue split. This in turn implies a reduction by p units over p processes after their corresponding queue splits. Note that the splits do not have to synchronized. Given p processes consuming n elements (at least p at a time), the depth is then by definition: $D(n) = \frac{n}{p}$. The communication complexity is dependent on the queue splitting – every queue split involved a request and a reply. In the worst case, where a queue splits $n-1$ times, we show now that the communication cost is $C(n) = O(n^2)$. As we have noted, the network cost for a message of size n units is $C(n) = \alpha + n * \beta$ (Section III-C). Since the queue decreases by at least one element for every split, the messages becomes smaller by at least one element. Each exchange consists of a work request of size one unit, and a response of size from $\{n-1, \ldots, 1\}$ units, with a total cost of $C_a = \{C(1) + C(n-1)\} + \{C(1) + C(n-2)\} + \cdots + \{C(1) + C(1)\}$. Expanding the formula and combining all the $C(1)$ terms, we have $C_a = n \cdot (\alpha + \beta) + \{\alpha + (n-1) \cdot \beta\} + \cdots + \{\alpha + 1 \cdot \beta\}$, that is, $C_a = n \cdot (\alpha + \beta) + n \cdot \alpha + \frac{\beta}{2} \cdot (n-1) \cdot (n-2)$. It is easy to see that $C_a = O(n^2)$.

Also note that if the queue splits $n-1$ times, each process could send a maximum of $p-2$ non-fruitful work requests to other processes. This increases the total network cost by adding one request and one response to p other processes for each of the $n-1$ queue splits $C_b = 2 \cdot (n-1) \cdot p \cdot (p-2) \cdot C(1) = O(n \cdot p^2)$, as $p << n$. Thus, the total network cost is $C_{total} = C_a + C_b = O(n^2)$. However, this worst case is rare, occurring if the file system is a directory containing only files, and during each split all but one file is sent to a requesting process.

Given the above analysis, and in the absence of specific information about fan-outs and depths of filesystem tree, we believe that random splitting may be a better technique than equal splitting in balancing amortized load [25]. We follow this line of thought and use random splitting; empirical studies (Section V) validate this line of reasoning as well.

Termination Detection: Termination detection in a parallel distributed algorithm is an important aspect. In our algorithms, termination detection is also distributed, and is achieved using Dijkstra's Token Algorithm [26]. We note that in order for the Dijkstra's algorithm to terminate successfully, links (back edges, such as symlinks and hardlinks) must not be explored – they would cause infinite work loops. We utilize this Dijkstra's algorithm for all three of the algorithms presented in this work. Dijkstra's algorithm can be implemented very simply, using the following rules [26]: all processes are logically ordered (numerical order is used for convenience); each process can be colored black or white, every process starts as white; a token can be passed between processes, and the token is also colored black or white; when the root process (Rank 0) is idle, it generates a white token and sends it to the next process (Rank 1); any time a process sends work to a process with a lesser rank it colors itself black; if a black process receives a token then it colors the token black, colors itself white, and then forwards the token; if a white process receives a token then it forwards the token unchanged, tokens are only forwarded by a process when it is idle; and termination is detected when the root process receives back a white token.

We present our three algorithms now. The processes in the presented algorithms perform random splitting. We do not present the equal splitting variants, but note that the splitting procedure can be conceptually thought of as a plug-and-play module, with random replaced by equal. The three algorithms we present in this section, combine one or more of the functionalities mentioned above in their design. Now we present each of them in order.

B. Distributed Random Queue Splitting

Our Distributed Random Queue Splitting (DRQS) algorithm is presented in Algorithm 2. Except for the purposes of termination initialization and detection, all processes are logically equivalent. There is no centralized master process, and no centralized work queue. Instead, each process maintains its own local work queue. One of the processes (Rank 0) contains the root of the parallel file system, that is where we start. In Line 1, the queue is initialized empty for all processes, except for the root process (Rank 0), which has only one item in it initially – the root path to be explored.

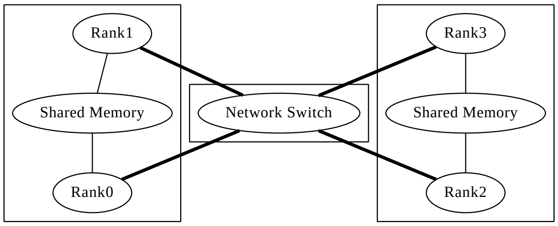


Fig. 2. Co-located processes have lower communication cost in comparison with non co-located processes

After initialization, every process executes a work loop in Lines 3-13. The execution of this work loop is not synchronized between processes. For each iteration of the loop, a process begins by checking for outstanding work requests from other processes in Line 4. If there are outstanding requests, it performs one of two things. If the process's queue is empty, it sends a negative response to each requester. Alternatively, if the process's queue is not empty it then splits its queue randomly and sends a portion of it to each requesting process. The next step in each iteration is also dependent on the local work queue. If it is non empty (condition checked in Line 5), the process dequeues one item from the queue and processes it (Line 8). Note that processing the item may add more items to the work queue if that work item is a directory. If the queue is empty (condition checked in Line 5), a work request is sent to another process chosen at random on Line 6. Lastly, Line 11 checks for termination by running Dijkstra's termination algorithm at the end of each work loop iteration. Algorithm 2 achieves good load balancing, but incurs overheads due to the number of messages being transmitted for requesting and receiving work. This may not be an impediment in small compute clusters, but with in large clusters with high network diameter, the penalty of network transmission rises rapidly. To address this impediment, we propose our next algorithm.

Algorithm 2 Distributed Random Queue Splitting

```
1:  $S = root$ for the $Rank$ 0 process, and $S = \emptyset$ for processes of higher rank.
2:  $Terminated$ = $False$.
3:  while not $Terminated$ do
4:      checkForRequests() and satisfy. {Checks for work requests from peers}
5:      if $|S| == 0$ then
6:          sendWorkRequest(). {Sends work request to random peer}
7:      else
8:          process($S$.dequeue()).
9:      end if
10:     if $|S| == 0$ then
11:         checkForTermination(). {Checks for termination conditions}
12:     end if
13: end while
```

C. Proximity Aware Distributed Random Queue Splitting (PA-DRQS) Algorithm

Typically, multiple processes run concurrently on the same compute node within a compute cluster. The cost for two co-located processes (same compute node) to participate in pair-wise communication is generally much lower than two processes running on separate compute nodes due to the absence of the latency that is introduced in each hop of network communication. The cost difference is also enhanced by MPI's choice of shared memory segments for communication between co-located processes. This idea is shown in Fig. 2, where thicker edges represent a higher cost of communication. Each rank is an MPI process, and the left and right rectangle represent two different compute nodes in a supercomputer.

Work Request Ordering: Because of the latency cost, it is preferable for a process to request work from a co-located process before asking a *remote* process. To that end, we have designed and implemented a proximity aware version of DRQS, known as Proximity Aware Distributed Random Queue Splitting (PA-DRQS). We note that in the literature, proximity awareness is popularly termed topology awareness. However, we use "proximity aware" as we believe it is more apt given that a process asks other processes for work in increasing order of their distance from it. The PA-DRQS algorithm is structured similarly to DRQS, but rather than choosing a random process to request work from, we impose an order to the requests. Before we can do that, we must determine which ranks are co-located. This can be achieved in the following way:

- Each process obtains its network number, as defined by RFC 1166 [27].
- An MPI_All_gather (from the MPI 2.0 specification) operation is performed so that every process has the complete list of all networks numbers. This is a synchronous step. After the MPI_All_gather, further operations are compute node local.
- Each process, having the entire array of network numbers, sorts them (we use QuickSort in our implementation).
- The resulting lists contains all network numbers, where equal network numbers are adjacent in the list. Each group of identical network numbers within the list is then assigned a group number. Each process then determines its location in the list, and then determines its group number, which is referred to as its *color*.
- Each process uses its color as a parameter to MPI_Comm_split, which creates an MPI Communicator containing co-located (same color) processes on each compute node within the compute cluster.
- From that information, a list of processes is created where co-located ranks are at the beginning (starting with local Rank 0) and non local ranks comprise the remainder of the list.
- Once the sub-communicators have been established, each process has an additional rank. The first is its global rank, which a unique identifier within the entire job. The second is its local rank, a unique identifier among co-located

processes.

The PA-DRQS algorithm begins in Line 1 by initializing the work queues for all processes to be empty, except for the master process which contains $root$. A function to populate the work request vector for each process is called in Line 3, and is implemented as described above in work request ordering. For requests to co-located remote processes of the same rank in a compute node, the requesting process makes the request in a sequential fashion based on their order in an ordered list of global MPI ranks. Thus processes occurring earlier get asked more often, which may result in more network communication if the requests return empty. A possible solution to this is to perform global ranking of co-located processes in real-time based on decreasing order of work availability, thus ensuring that a process with more work is asked first. However, maintaining such a real-time list at each compute node incurs many more message exchanges than sending sequential requests to co-located remote processes – a higher message overhead.

Lines 4-14 constitute the work loop performed by all processes. Each process checks for outstanding work requests from other processes in Line 5 and services them by sending each requester a portion of the work queue if it has pending work. If the work queue is empty (determined on Line 6), then a work request will be sent to another process. Whereas in Algorithm 2 a process was chosen at random, in PA-DRQS, the process to request work from is chosen by the ordering in the request vector list created in Line 3. When a process needs to request work from another, it iterates over this computed list from top to bottom.

We further modify the algorithm, so that not all ranks favor asking local ranks. This is because in our empirical observations this leads to a large number of non-fruitful local work requests preceding non-local work requests when all local ranks have empty work queues. We mitigate this by changing the order of the work request so that the local root rank (the process within a group of co-located processes with a local rank of 0) will first ask non-local ranks for work. If the work queue is non-empty, the process will call the user provided callback function on one element of the work queue. Finally, Lines 11-13 check for a termination condition.

The PA-DRQS algorithm has much fewer network communications than the DRQS algorithm, which is desirable. However, the design of the algorithm allows even a non-Rank 0 process (Rank x) to seek work from a process outside it's compute node after receiving negative responses from it's co-located ranks (the process goes through it's request vector list). In our empirical observation, often this creates unnecessary network communication, because in the time that Rank x is requesting from its neighboring ranks, Rank 0, which had no work, has already received work requests from some other compute node. Technically, Rank x can get the work now from Rank 0, but does not.

We address this concern in our implementation by requiring non-Rank 0 processes to only request their local ranks. However, this results in an increase in the number of local request messages in the compute node. As noted, another concern with this algorithm is that it suffers from extra messages originating from the Rank 0 process in the compute node even when there are pending work in other co-located ranks. This is because Rank 0 is required to ask a remote process for work. To address these issues, we propose an extension to the PA-DRQS scheme by using light-weight processes and propose the hybrid algorithm. We note that in the absence of the ability to use LWPs, PA-DRQS is still the algorithm of choice.

Algorithm 3 PA-DRQS: Proximity Aware Distributed Random Queue Splitting

```
 1: $S = root$ for the $Rank$ 0 process, and $S = \emptyset$ for processes
    of higher rank.
 2: $Terminated$ = $False$.
 3: $requestVector$ = createRequestVector().
 4: while not $Terminated$ do
 5:    checkForRequests() and satisfy. {Checks for work requests from peers}
 6:    if $|S| == 0$ then
 7:       sendWorkRequest(). {Sends work request to the next
          peer from the request vector}
 8:    else
 9:       process($S$.dequeue()).
10:    end if
11:    if $|S| == 0$ then
12:       checkForTermination(). {Checks for termination conditions}
13:    end if
14: end while
```

D. H-DRQS: Hybrid Distributed Random Queue Splitting Algorithm

In order to improve our proximity aware algorithm, we design a hybrid parallel approach, which we term the Hybrid Distributed Random Queue Splitting (H-DRQS) algorithm. Whereas the previous algorithms presented utilize multiple MPI ranks per compute node in the compute cluster, our hybrid approach is able to leverage parallelism with only one MPI process per compute node. We achieve this by utilizing light-weight processes (LWP). In H-DRQS, each compute node may spawn an arbitrary number of LWPs (threads) corresponding to each MPI process. Only the original master thread is allowed to participate in MPI communication. This is not a problem as all processes share a common memory, and hence share one work queue. The master thread is able to populate the work queue which is accessible to all other threads on the same compute node.

We prevent race conditions by ensuring the enqueue/dequeue operations are guarded by a mutual exclusion lock (mutex). We also ensure that the queue is not modified by any threads during a queue split, which we do using counting semaphores. The master thread of each process executes Algorithm 4. It is initialized as in Algorithm 3 in Lines 1-2, and to begin it creates and initializes a counting semaphore for the threads to zero in Line 3. An additional semaphore is created in Line 4 for use by the master and is also initialized to zero. The master thread executes the algorithm shown in Algorithm 4, all other threads execute a simple work loop. When a thread is created (Line 5) it immediately enters its work loop. In its loop, a thread first blocks on the *threads* semaphore (initially threads is zero, master has to increment it) until it can decrement it to zero. Once a thread has successfully decremented the semaphore, it dequeues one work queue item and processes it. The queue modifying operations are still protected by the mutex.

After processing the item, the thread *increments* the master semaphore $master\_guard$. Meanwhile, the master thread enters its work loop and operates in the same fashion as Algorithm 3 by checking for and servicing work requests, and requesting work from other processes if necessary (Lines 6-10). If there is work to be processed in the queue, the master increments the thread semaphore by the value $\min\{$the number of threads, number of items in the work queue$\}$ (Lines 11-12). After incrementing the semaphore, all threads (master included) process one work queue item (Line 13). Next, the master thread attempts to decrement the master semaphore, the same number of times that it incremented the thread semaphore on Line 12. Recall that the master semaphore was initialized to zero. This means that the master thread will block until all other threads finish their work and increment the master semaphore. Finally, the master checks for a termination condition just as in Algorithm 3.

As all LWPs share one logical address space, the cost for exchanging data/messages between threads is minimal, which improves communication efficiency. Let's see how we can improve communication efficiency. For example, consider the case where 16 MPI processes are launched on each compute node using Algorithm 3. Suppose that Rank 0 has no work in it's queue, but one of the other processes (local Rank 15) has work. When Rank 2 requests Rank 0 for work, Rank 0 sends a negative reply and sends a request over the network even though there is work available in the locality that can be shared. At the same time, Rank 2, is clogging up the shared memory sending out requests to processes in in turn. Both aspects are inherently wasteful.

Now suppose that instead of the 16 MPI processes being co-located, we use one MPI process with 16 LWPs (all sharing one work queue). Only when this shared queue is empty (no process has work), the master thread sends a work request to a non-local rank (there are no co-located ranks now). Moreover, since the queue is shared a slave thread will block on the queue when it is empty, but never block if there is work in the queue. This is done because the threads must not modify the work queue during a potential queue split, as previously described.

In the next section, we present empirical results from our experiments in large parallel files systems at LANL and study the efficiency of our algorithms.

Algorithm 4 Hybrid Distributed Random Queue Splitting (H-DRQS)

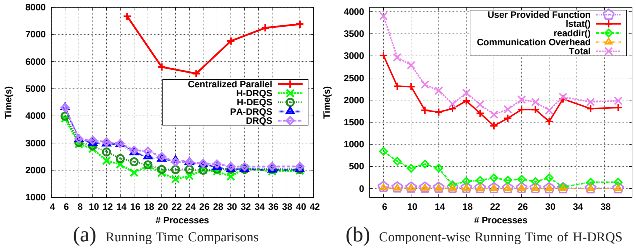
```
 1: $S = root$ for the $Rank$ 0 process, and $S = \emptyset$ for processes of higher rank.
 2: $Terminated$ = $False$.
 3: thread_guard = semaphore_init(threads).
 4: master_guard = semaphore_init(master).
 5: startThreads().
 6: while not $Terminated$ do
 7:     checkForRequests() and satisfy. {Checks for work requests from peers}
 8:     if $|S| == 0$ then
 9:         sendWorkRequest(). {Sends work request to random peer}
10:     else
11:         $count$ = min(threads.count(),queue.count()).
12:         semaphore_increment(thread_guard,count).
13:         {Threads process work queue elements}
14:         semaphore_decrement(master_guard,count).
15:     end if
16:     if $|S| == 0$ then
17:         checkForTermination(). {Checks for termination conditions}
18:     end if
19: end while
```

V. Experimentation and Empirical Results

To evaluate our algorithms, we designed a simple API based on application call-backs, which is similar to the POSIX specification for FTW. For the purposes, of testing we used multiple Panasas [20] file systems attached to multiple compute clusters at LANL, through a specialized storage network known as PaScal [28]. Specifically, we studied three file systems, namely a) a 6.5 million files, of size 471 TB; b) a 12 million files, of size 2 PB; and c) a 110 million files, of size 7 PB. The implementation of our algorithms was done using Open MPI [29], combined with the GNU implementation of the POSIX threading library. We ran our tests on Cielo – 8944 compute nodes and 16 cores per compute node. We chose to run 2 processes per compute node 40 processes on 20 nodes to (i) have more memory available per process, and (ii) guarantee that we use all 12 available network routes (depending on node placement in the torus). Our aim is to choose the least number of cores for traversal so that more cores are available for computation.

Our experiments are run on production systems where the file systems change significantly within hours, so results averaged over several runs, taking several hours, will not be indicative of algorithm effectiveness. For this reason, we ran on three different file systems to ensure diversity of results, but only 2-3 times, to ensure consistency.

Centralized Parallel vs. Hybrid Distributed: Figure 3(a) shows a comparison of the running time of the previous Centralized Parallel algorithm (Algorithm 1) versus our DRQS



(a) Running Time Comparisons (b) Component-wise Running Time of H-DRQS

Fig. 3. Running Time Illustrations

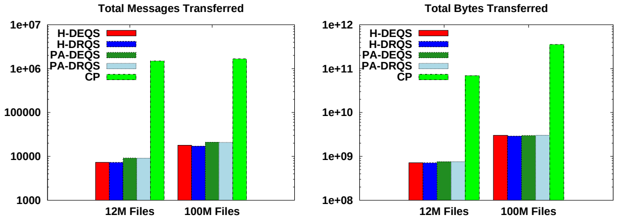
variants equal splitting) for gathering file system metadata on the file system containing 6.5 million files. The horizontal axis shows the number of processes used for each experiment. The vertical axis shows the running time of the algorithms in seconds. Note that DEQS (Distributed Equal Queue Splitting) refers to to the case where a requested process splits its work queue into two equal halves, giving one half to the requesting process. The DRQS/DEQS variants outperformed the existing Centralized Parallel (CP) algorithm by more than 300% percent, in terms of time. In fact, with 15 or less processes, the CP algorithm took long enough that we cannot represent it in the same graph.

We also implemented an enhancement of the CP algorithm where a slave process keeps one of the work elements from its exploration and returns the rest to the master. This variation ensures that the slave does not have to make a new request for work again, thus reducing network communication. However, the running time of this enhancement was close to the original CP algorithm because network communication was still the major bottleneck.

As is noticeable, the H-DRQS algorithm performs the best among all the DRQS/DEQS algorithms, hence we use it more often for other comparisons in the interest of brevity and space.

Hybrid DRQS Profile: Figure 3(b) shows the profile of time required for each component of the algorithm. As the H-DRQS is the best algorithm we have, we use its profile. For the purposes of profiling we grouped the output into: *(i)* user provided function (a standard user call-back function); the lstat() system call; the readdir() system call; and the network communication overhead.

The X-axis represents the number of processes used in the experiment and the Y-axis shows the amount of process time consumed by each component. Observe that the components which dominate the running time of our algorithm are lstat() and readdir() (both operating system calls), which implies that the components of the algorithm have very low footprint. Another positive of our algorithm is: with increase in the number of processes the communication cost does not increase.

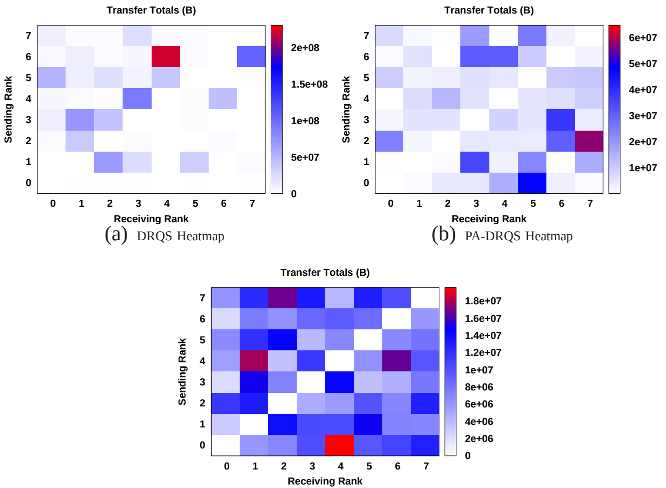


(a) Total Number of Messages (b) Total Number of Bytes

Fig. 4. Messages and Bytes Transferred Statistics

Messages and Data Transfers: In addition, to demonstrating the communication efficiency of our algorithms, we also performed experiments to measure the number of messages and bytes transferred in the network. It is desirable to have low total messages/bytes transferred and also a balance in communication between the nodes. The results for total number of messages and bytes are shown in Fig. 4. The X-axis represents the number of files in the file system and the Y-axis represents the number of (messages or bytes) and use the logarithmic scale. The comparison is between CP, H-DRQS, and PA-DRQS, where the tests were run on two parallel file systems. As expected, H-DRQS and PA-DRQS perform much better than the CP algorithm in terms of both messages (an order of magnitude less) and in terms of bytes (more than two orders of magnitude less). The performance of both H-DRQS and PA-DRQS are similar, which is expected.

Fig. 5 shows the heatmaps for bytes exchanged between processes when the algorithms were run on the 6.5 million files Panasas file system. The heatmaps underscore the efficiency of our proposed algorithms versus the CP algorithm. The bytes exchanged in DRQS are an order of magnitude better than CP, while the H-DRQS has the best result, being almost two orders of magnitude better than the CP.



(a) DRQS Heatmap (b) PA-DRQS Heatmap

(c) H-DRQS Heatmap

Fig. 5. Heat Maps Showing Message Exchanges

Work Distribution: An important goal of work distribution is to achieve uniform load distribution to prevent resources underutilization. Fig. 6 shows the results of a large scale test run on the 110 million files performed using 20 processes. Fig. 6(a) shows the span of the work distribution. The figure is a histogram showing the distribution of the workloads percentages among processes. H-DRQS has a much smaller span than DRQS. For the rest of the figures (Figs. 6(b), 6(c), 6(d)), the the horizontal axis is composed of ranges of work load percentages, and the vertical axis shows how many processes performed a part of the work within the given range. We note that using DRQS without proximity information resulted in one process performing a disproportionate amount of the work, as can be seen on the far right hand side of the graph. This is due to a propensity for unfruitful work requests being sent by the other processes. If a process is required to sent out several non fruitful work requests before a fruitful request is fulfilled, then a process with work to do will proceed uninterrupted while others are idle.

The results of our proximity aware algorithm in Fig. 6(d) demonstrate the improvement in workload distribution as a by product of fewer non fruitful requests being sent, and multiple co-located processes sharing a work queue. Fig. 6(d) shows the results of a similar test to that of Fig.6(b), but with proximity awareness enabled. The remaining three subfigures show a comparison between DRQS, DEQS, and H-DRQS, the X-axis is the workload percentage and the Y-axis is the number of processes. It is easy to see that H-DRQS has a much smaller span and better balancing of work among the processes.

VI. Conclusion

In this paper, we propose a novel framework and three three novel parallel algorithms to facilitate distributed file system operations with low message complexity. Our techniques not only balanced file system work loads uniformly in real-world experiments, but did so with low communication costs, and without global process synchronization. Our algorithms have been tested on state-of-the-art parallel file systems at LANL, and are now used in productions systems on a daily basis for metadata management.

Acknowledgments

The authors thank the anonymous reviewers for their insightful comments to improve this paper.

References

[1] NASA open source MCP. [Online]. Available: http://ti.arc.nasa.gov/opensource/projects/mcp/

[2] LANL PSI. [Online]. Available: http://www.lanl.gov/orgs/tt/license/software/09.shtml

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] E. Heien, D. LaPine, D. Kondo, B. Kramer, A. Gainaru, and F. Cappello, "Modeling and tolerating heterogeneous failures in large parallel systems," in *ACM/IEEE conference on Supercomputing*. IEEE, 2011, pp. 1–11.

[5] *The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2008*, IEEE, 2008.

[6] LANL file tree walk. [Online]. Available: http://www.pdsi-scidac.org/events/FAST08BOF/resources/FAST08_PDSI_LANLD%ataRelease.pdf

[7] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," in *IEEE Foundations of Computer Science*, 1994, pp. 356–368.

[8] B. Awerbuch and R. Gallager, "A new distributed algorithm to find breadth first search trees," *IEEE Transactions on Information Theory*, vol. 33, no. 3, pp. 315–322, 1987.

[9] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of ACM symposium on Principles and practice of parallel programming*, 2011, pp. 267–276.

[10] J. Freeman, "Parallel algorithms for depth-first search," *Technical Report, University of Pennsylvania*, 1991.

[11] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *ACM/IEEE conference on Supercomputing*. IEEE, 2005, pp. 25–25.

[12] U. Vishkin, "Thinking in parallel: Some basic data-parallel algorithms and techniques," *College Park, MD*, vol. 1993, 2007.

[13] V. Rao and V. Kumar, "Parallel depth first search. part i. implementation," *International Journal of Parallel Programming*, vol. 16, no. 6, pp. 479–499, 1987.

[14] A. Reinefeld and V. Schnecke, "Work-load balancing in highly parallel depth-first search," in *IEEE Scalable High-Performance Computing Conference*, 1994, pp. 773–780.

[15] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," *Arxiv preprint arXiv:1104.4518*, 2011.

[16] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[17] Directory indexing. [Online]. Available: http://ext2.sourceforge.net/2005-ols/paper-html/node3.html

[18] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, 2003.

[19] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *USENIX Symposium on Operating Systems Ddesign and Implementation (OSDI)*, 2006, pp. 307–320.

[20] D. Nagle, D. Serenyi, and A. Matthews, "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *ACM/IEEE conference on Supercomputing*, 2004, p. 53.

[21] D. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing*, vol. 20, no. 4, pp. 657–673, 1994.

[22] Y. Shiloach and U. Vishkin, "An o(n2 logn) parallel max-flow algorithm," *Journal of Algorithms*, vol. 3, no. 2, pp. 128–146, 1982.

[23] S. Cook, C. Dwork, and R. Reischuk, "Upper and lower time bounds for parallel random access machines without simultaneous writes," *SIAM Journal on Computing*, vol. 15, p. 87, 1986.

[24] R. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 201–206, 1974.

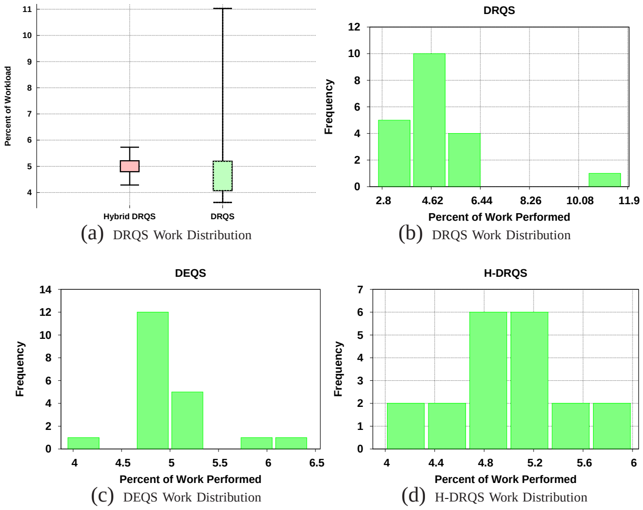
[25] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001.

[26] E. Dijkstra, W. Feijen, and A. Gasteren, "Derivation of a termination detection algorithm for distributed computations," *Information Processing Letters*, vol. 16, no. 5, pp. 217–219, 1983.

[27] S. Kirkpatrick, M. Stahl, and M. Recker, "Rfc 1166: Internet numbers," *Internet Engineering Task Force*, 1990.

[28] G. Grider, H. Chen, J. Nunez *et al.*, "Pascal-a new parallel and scalable server io networking infrastructure for supporting global storage/file systems in large-size linux clusters," in *IEEE International Performance, Computing, and Communications Conference*, 2006, pp. 331–340.

[29] R. Graham, G. Shipman, B. Barrett, R. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A high-performance, heterogeneous MPI," in *IEEE International Conference on Cluster Computing*, 2006, pp. 1–9.



(a) DRQS Work Distribution

(b) DRQS Work Distribution

(c) DEQS Work Distribution

(d) H-DRQS Work Distribution

Fig. 6. Load Balancing: H-DRQS and PA-DRQS perform much better load-balancing than standard DRQS/DEQS.