High Performance Computing

13M37098

Yuki Takasaki

Review Paper

"On Distributed File Tree Walk of Parallel File System"

[SC'12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis]

Jharrod LaFon*†, Satyajayant Misra*,and Jon Bringhurst†

*New Mexico State University, †Los Alamos National Laboratory

Review Paper

"Mastiff: A MapReduce-based System for Time-Based Big Data Analytics"

[Cluster Computing (CLUSTER), 2012 IEEE International Conference on]

Sijie Guo*, Jin Xiong, Weiping Wang*, Rubao Lee†

*State Key Laboratory of Computer Architecture Institute of Computing Technology, CAS Beijing, China

†Dept. of Computer Science and Engineering Ohio State University Columbus, USA

Review Paper

“Comparative Performance Analysis of a big Data NORA Problem on a Variety of Architectures”

[Collaboration Technologies and Systems (CTS), 2013 International Conference on]

Peter M. Kogge*, David A. Bayliss†

*Dept. of Computer Science and Engineering Univ. of Notre Dame Notre Dame, IN, USA

†Lexis Nexis Risk Solutions, Inc. Boca Raton, FL, USA

Outline

1. Abstract
2. Introduction
3. Related work
4. Building blocks of our framework
5. A framework for distributed parallel file system traversal
6. Experimentation and empirical results
7. Conclution

- Comment

1.Abstract

- Research goal is proposing three algorithm
  - Improve Centralized Parallel File Tree Walk
- DRQS : Distributed Random Queue Splitting
  - All processes are logically equivalent
- PA-DRQS : Proximity Aware Distributed Random Queue splitting
  - Proximate aware version of DRQS

2.Introduction

- The amount of scientific data produced today has been increasing and scientists often use sophisticated tools to write application.
- However, the tools and algorithms used to traverse file systems are often serial, making data archiving or searching time consuming.
- The few tools that exist for parallel processing and archiving use centralized parallel algorithms.
  - For load balancing and work distribution
  - Leading to unnecessarily high communication overhead

Problem Motivation

- Parallel tree traversal problem
  - centralized parallel algorithms have communication overhead
    - Example : MapReduce uses master and slave strategy.
    - The master process need to keep track of which slave processes are busy
    - Each new task requires two messages of the dispatch of work unit from the master to slave and the reply from the slave to the master
    - The master process must maintain a global list of tasks to be performed

Propose of this study

- We propose a framework and three efficient algorithms.
  - the improvement in running time and message complexity
    - By dispensing with the synchronization requirement
    - By avoiding a centralized control process altogether

3.Related Work

- Centralized Parallel File Tree Walk Algorithm
  - The first centralized parallel (CP) file tree traversal algorithm was developed in house at LANL (2007)
  - This algorithm is used a dynamic centralized load balanceing technique.

Algorithm 1-1

Algorithm 1 Centralized Parallel File Tree Walk

```
S = ∅ for slave processes, root for the master
if processor rank == 0 then
    i = 0
    while |S| > 0 do
        Receive Message from Processor j
        if Message is a work request then
            p = S.dequeue()
            Send p to j
        else
            S.queue(Message) {Work to be processed}
        end if
    end while
```

Algorithm 1-2

```
else
    repeat
        if |S| = 0 then
            Send work request to Processor 0
            Receive Message from Processor 0 into path
        end if
        if path is termination sentinel then
            exit
        end if
        if path is a file then
            process(file)
        else
            S = ∅
            for all child in path.children() do
                S.queue(child)
            end for
            Send S to Processor 0
        end if
    until path == ∅
end if
```

Problem of CP algorithm

- Until the queue is empty, the master process sends a portion of work to each slave process, and then waits for a response from each one
  - Requires process synchronization

Communication cost

- Experiment environment
  - Supercomputer at LANL using a 471TB Panasas file system consisting of approximately 6.5 million files.
- Observe that communication strictly occurs between the master prosess and slaves, but never between two slaves

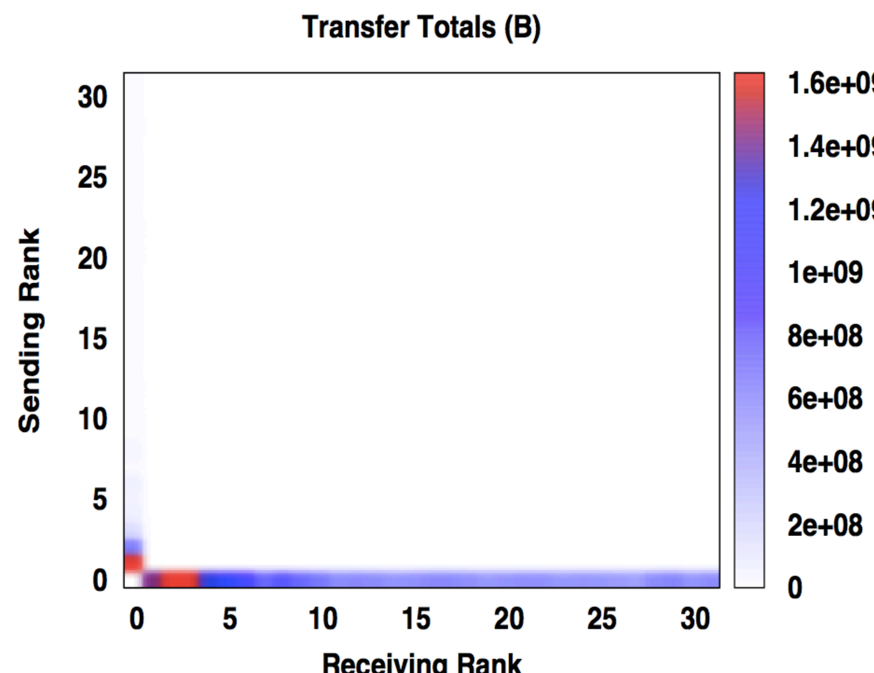


Fig. 1. Centralized Parallel Tree Walk: Communication Cost

4.Building blocks of our framework

- A. Parallel Tree Traversal
  - Our goal is to design a parallel algorithm for parallel file systems tree exploration.
  - We seek an ideal load balance, with equitable load distribution
    - All the parallel processes performs the same amount of work

B.Inter-Process Communication without Global Synchronization

- We seek to visit all nodes within a tree in parallel, as quickly as possible.
- One way to achieve this efficiently is by avoiding global process synchronization.
  - Synchronization between all processes in a parallel job must be coordinated by way of communication, and this is known to be costly.

Pair-wise communication

- Pair-wise communication refers to a message transfer that occurs between two processes.
- Collective communication is a message exchange which is meant for all process.
  - Collective communications are a form of synchronization
- We use pair-wise communication which is non-blocking

5. A framework for distributed parallel file system traversal

- A.Design Principles for the Framework (1)
  - Parallelism via the Message Passing Interface:
    - We implement our algorithms using the MPI
  - Anyone-Asks-Anyone:
    - There is no master process
    - All processes in the system are equal
    - Any process can ask any other process for work
  - Light Weight Process v/s Single Process :
    - Use multiple threads/processes on each compute node
    - One of threads in node seek work from remote processes, after which all co-located threads/processes can share the work

A.Design Principles for the Framework (2)

- Random Splitting v/s Equal Splitting :
  - Use random splitting which may be better technique than equal splitting in balancing amortized load
- Termination Detection:
  - Use Dijkstra's Token Algorithm
    - 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 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 the token black, colors itself white, and then forwards the token
    - 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
    - termination is detected when the root process receives back a white token.

B.Distributed Random Queue Splitting

- 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.
- Each process maintains its own local work queue
  - Rank 0 contais the root of the parallel file system

Algorithm 2 Distributed Random Queue Splitting

```
S = root for the Rank 0 process, and S = ∅ for processes of higher rank.
Terminated = False.
while not Terminated do
    checkForRequests() and satisfy. {Checks for work requests from peers}
    if |S| == 0 then
        sendWorkRequest(). {Sends work request to random peer}
    else
        process(S.dequeue()).
    end if
    if |S| == 0 then
        checkForTermination(). {Checks for termination conditions}
    end if
end while
```

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

- 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.

Co-located process

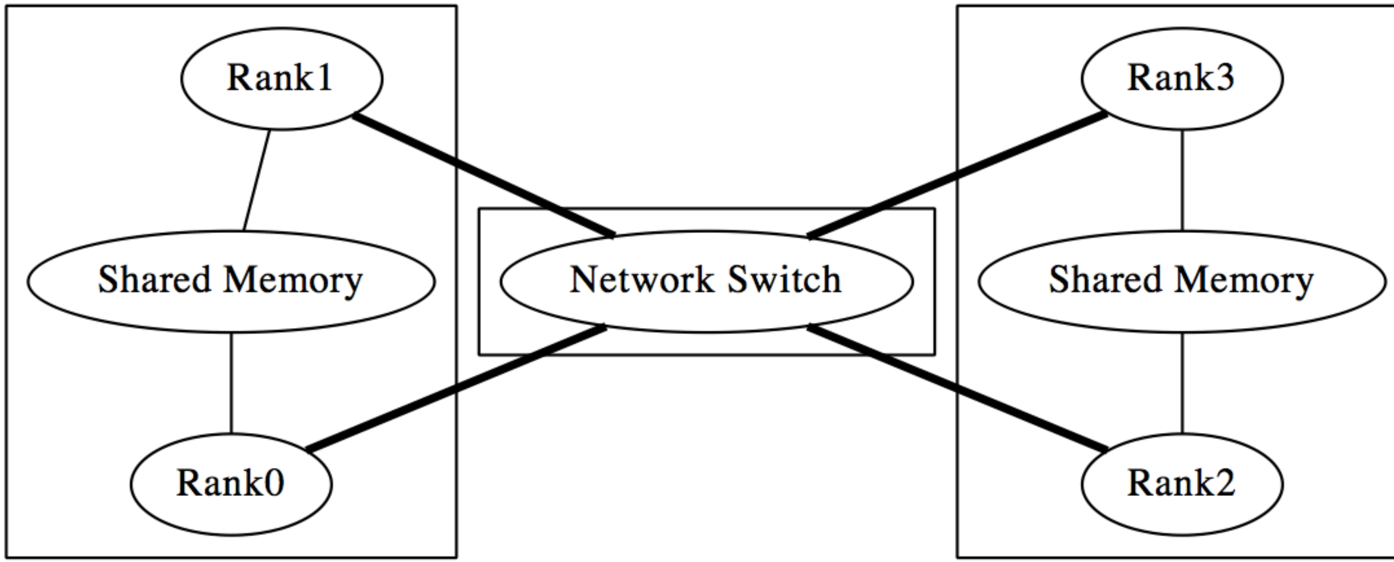


Fig. 2. Co-located processes have lower communication cost in comparison with non co-located processes

Work Request Ordering

- It is preferable for a process to request work from a co-located process before asking a remote process.
- We have designed and implemented PA-DRQS
  - A proximity aware version of DRQS
- We impose an order to the request.
  - In PA-DRQS, a process asks other processes for work in increasing order of their distance from it.
- We must determine which ranks are co-located.

Way to determine which ranks are co-located (1)

- Each process obtains it network number, as defined by RFC 1166.
- An MPI_All_gather 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, sort them
  - We use OuickSort in our implementation

Way to determine which ranks are co-located (2)

- Each process then determines its location in the list, and then determines its group number, which is refereed to as its color.
  - 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 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.

Way to determine which ranks are co-located (3)

- From that information, a list of processes is created
  - co-located ranks are at the beginning (starting with local Rank 0) and non local ranks comprise the remainder of the list
- Each process has an additional rank.
  - Global rank : a unique identifier within the entire job
  - Local rank : a unique identifier among co-located processes

Algorithm 3

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

$S = root$ for the $Rank$ 0 process, and $S = \emptyset$ for processes of higher rank.
$Terminated$ = $False$.
$requestVector$ = createRequestVector().
while not $Terminated$ **do**
    checkForRequests() and satisfy. {Checks for work requests from peers}
    if $|S| == 0$ **then**
        sendWorkRequest(). {Sends work request to the next peer from the request vector}
    else
        process(S.dequeue()).
    end if
    if $|S| == 0$ **then**
        checkForTermination(). {Checks for termination conditions}
    end if
end while

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

- 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)
  - Each compute node spawn an arbitary number of LWPs(threads)
  - Only original master thread is allowed to participate in MPI communication.
  - All threads in compute node share work queue in this node
- We prevent race conditions
  - Ensure that the enqueue/dequeue operations are guarded by using a mutual exclusion lock (mutex).
  - Ensure that the queue is not modified by any threads during a queue split by using counting semaphores.

Algorithm 4

- All LWPs share one logical address space
- The cost for exchanging data/messages between threads is minimal

---

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

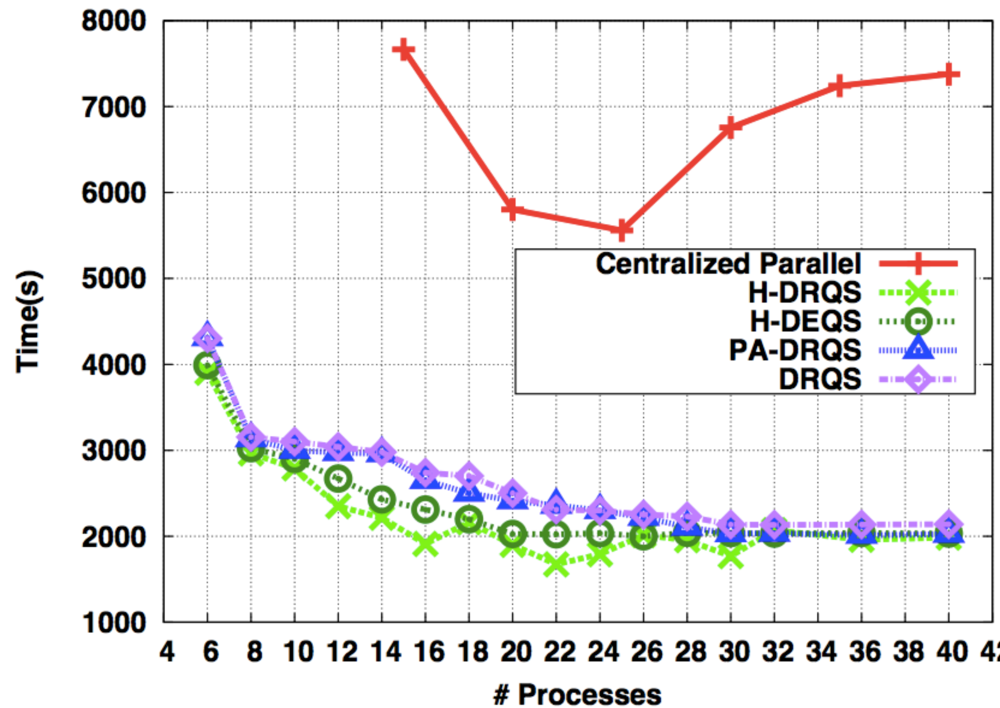
$S = root$ for the $Rank$ 0 process, and $S = \emptyset$ for processes of higher rank.
$Terminated = False$.
thread_guard = semaphore_init(threads).
master_guard = semaphore_init(master).
startThreads().
while not $Terminated$ **do**
    checkForRequests() and satisfy. {Checks for work requests from peers}
    if $|S| == 0$ **then**
        sendWorkRequest(). {Sends work request to random peer}
    else
        $count$ = min(threads.count(),queue.count()).
        semaphore_increment(thread_guard,count).
        {Threads process work queue elements}
        semaphore_decrement(master_guard,count).
    end if
    if $|S| == 0$ **then**
        checkForTermination(). {Checks for termination conditions}
    end if
end while

6.Experimentation and empirical results

- Experiment enviroment
  - File system : Panasas file system
    1. A 6.5 million files, of size 471 TB
    2. A 12 million files, of size 2 PB
    3. A 100 million files, of size 7 PB
  - Machine : Cielo
    - 8944 compute nodes and 16 cores per compute node
    - Network : torus

Centralized Parallel vs. Hybrid Distributed

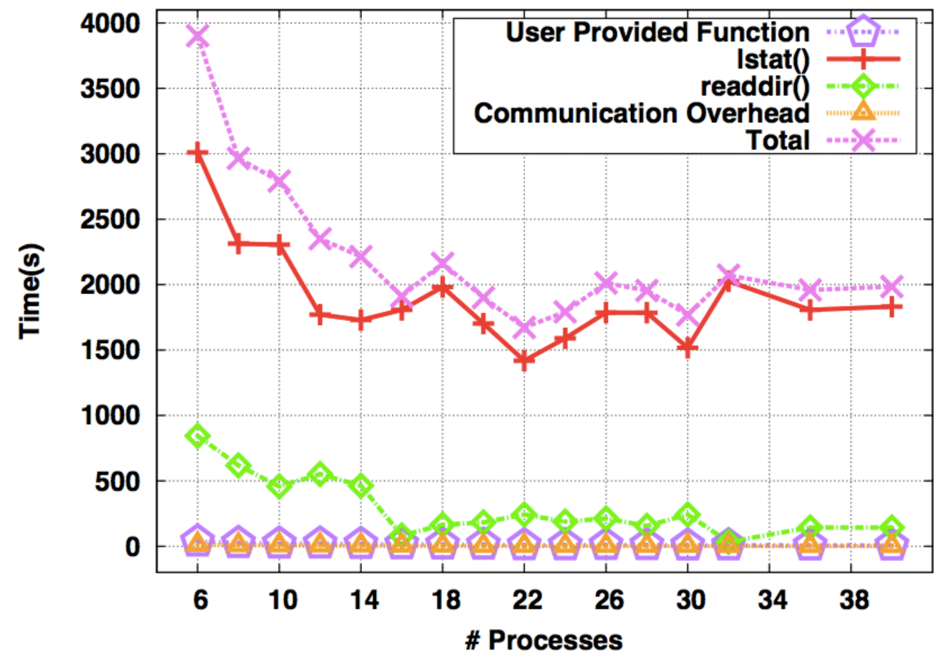
- The DRQS/DEQS variants outperformed the existing CP algorithm by more than 300% percent
- The H-DRQS algorithm performs the best among all the DRQS/DEQS algorithm



(a) Running Time Comparisons

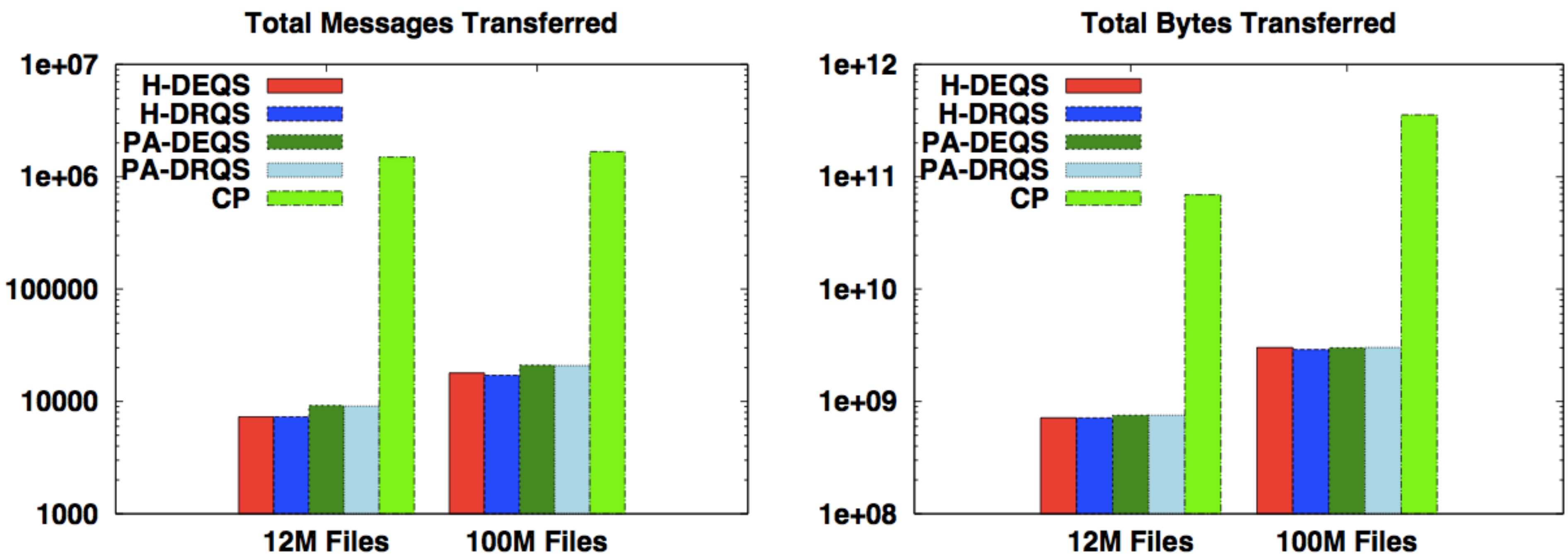
Hybrid DRQS Profile

- Lstat() and readdir() dominate the running time of our algorithm
- With increase in the number of processes the commnication cost does not increase



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

Message and Data Transfers (1)



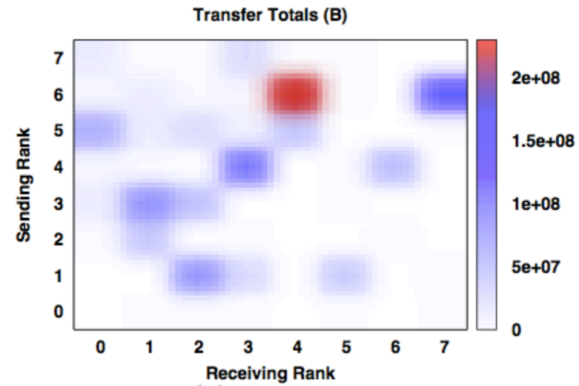
(a) Total Number of Messages

(b) Total Number of Bytes

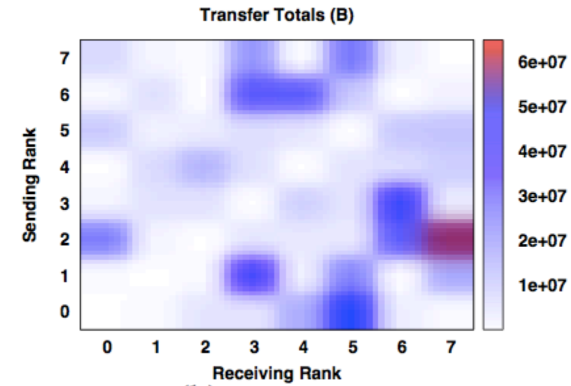
Fig. 4. Messages and Bytes Transferred Statistics

Messages and Data Transfers: In addition, to demo

Message and Data Transfers (2)



(a) DRQS Heatmap



(b) PA-DRQS Heatmap

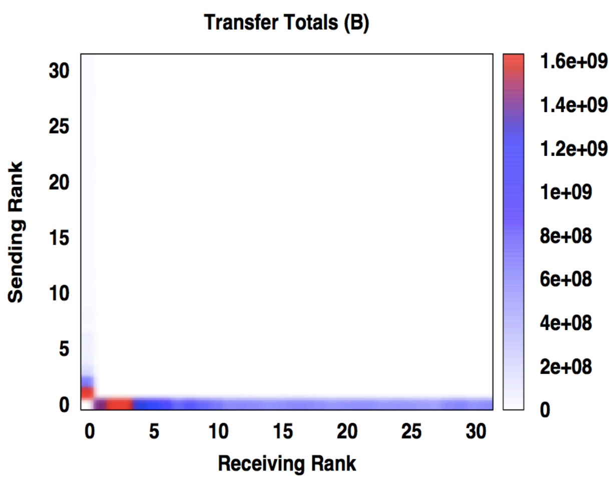
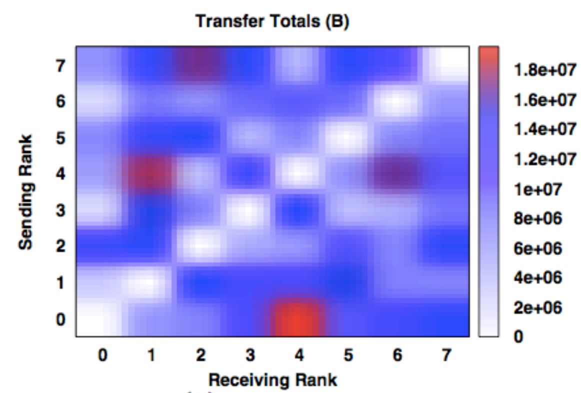


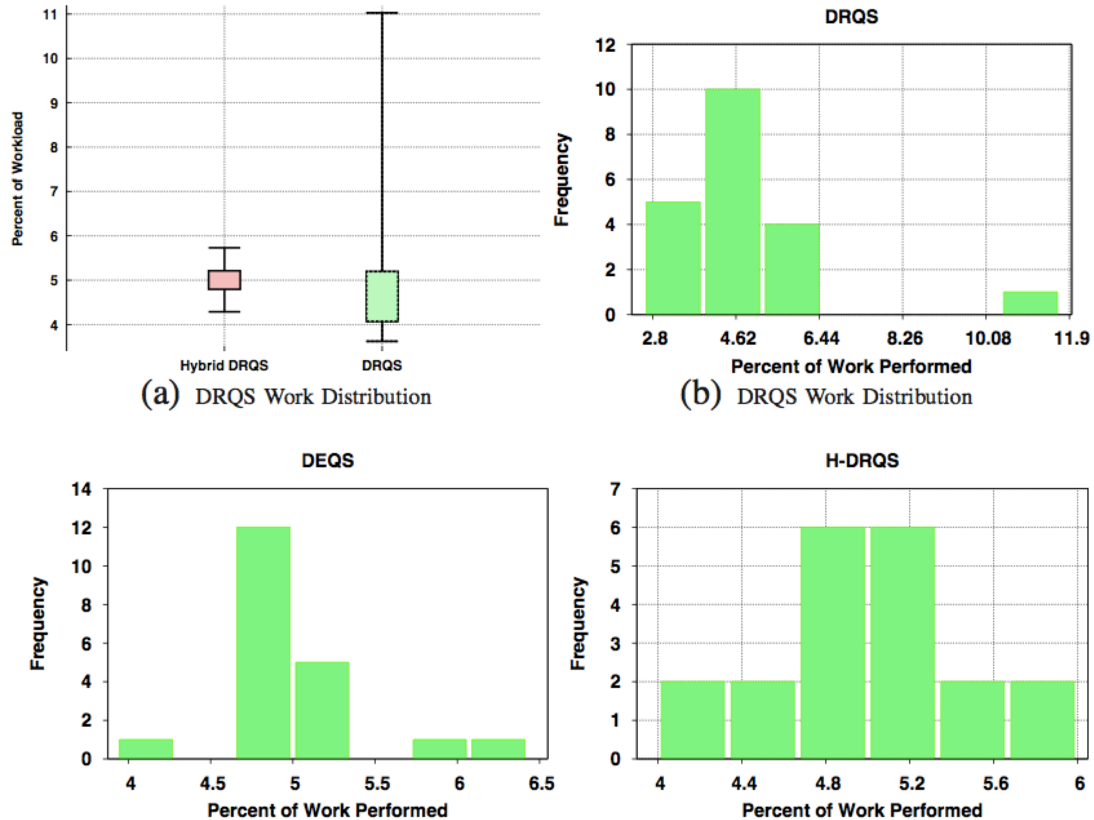
Fig. 1. Centralized Parallel Tree Walk: Communication Cost



(c) H-DRQS Heatmap

Fig. 5. Heat Maps Showing Message Exchanges

Work Distribution



(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.

7.Conclusion

- We propose a novel framework and three novel parallel algorithms
  - Facilitate distributed file system operations with low message complexity
  - Balance file system work loads uniformly in real-world experiments and with low communication cost without global process synchronization

Comment

- Strong point
  - Experiment environment is suitable
    - Supercomputer in LANL
  - How to improve the algorithm is systematic
- Weak point
  - Don't compare empirical result of existing algorithm

Thank you