PERFORMANCE, SCALABILITY, AND ROBUSTNESS IN DISTRIBUTED FILE TREE COPY

by

Christopher Robert Sutton

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2018

© 2018
Christopher Robert Sutton
ALL RIGHTS RESERVED

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Christopher Robert Sutton

Thesis Title: Performance, Scalability, and Robustness in Distributed File Tree Copy

Date of Final Oral Examination: 19th June 2018

The following individuals read and discussed the thesis submitted by student Christopher Robert Sutton, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Amit Jain, Ph.D.	Chair, Supervisory Committee
Tim Andersen, Ph.D.	Member, Supervisory Committee
Steven Cutchin, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Amit Jain, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Amit Jain, for his incredible patience and for all the help and support he has given throughout this project.

I would like to thank Ben Peterson for all the help he has given, setting up experiments on the GeneSIS cluster and addressing all the technical issues that arose.

I would also like to thank Eric Whiting and Peter Cebull for their help and support getting access to the High Performance Computing cluster at Idaho National Laboratories, and getting started running our experiments there.

This research made use of the resources of the High Performance Computing Center at Idaho National Laboratory, which is supported by the Office of Nuclear Energy of the U.S. Department of Energy and the Nuclear Science User Facilities under Contract No. DE-AC07-05ID14517.

ABSTRACT

As storage needs continually increase, and network file systems become more common, the need arises for tools that efficiently copy to and from these types of file systems. Traditional copy tools like the Linux `cp` utility were originally created for traditional storage systems, where storage is managed by a single host machine. cp uses a single-threaded approach to copying files. Using a multi-threaded approach would likely not provide an advantage in this system since the disk accesses are the bottleneck for this type of operation. In a distributed file system the disk accesses are spread across multiple hosts, and many accesses can be served simultaneously. Volumes in a distributed file system still look like a single storage device to the operating system of a client machine, so traditional tools like cp can still be used, but they cannot take advantage of the performance increase offered by a distributed, multi-node approach. While some research has been done in this area and some tools have been created there are still shortcomings in the area, particularly when it comes to scalability and robustness to process failure. The research presented here attempts to adress these shortcomings.

The software created in this project was tested in a variety of enviornments and compared to other tools that have attempted to make improvements in these areas. In the area of scalability, this software performed well from a small cluster to large high-performance-computing cluster. In comparisons to other tools, the performance was comparable to or better than the other tools measured.

The unique contribution of this project is in the area of robustness, which other

tools have not attempted to address. The software allows the user to specify a minimum number of processes that can fail without losing progress or requiring a restart of the copy job. This means process death can be tolerated, or resources can be diverted after the start of the copy job. This is especially helpful in very large, long-running jobs. Many clusters are made with low-cost, consumer-grade hardware which is susceptible to failure. A study entitled "Failure Trends in a Large Disk Drive Population" [2], published in the proceedings of the USENIX FAST'07 conference, looked at the failure rate of disk drives across Google's various services. It found that a large number of drives fail within a few years. Drive failure rate was somewhat high within the first few months, weeding out the lower quality drives, but at the two and three year mark as many as eight percent of drives failed per year. This high rate shows the importance of software like this that can tolerate these types of failures.

TABLE OF CONTENTS

ABSTRACT v

LIST OF TABLES x

LIST OF FIGURES xi

1 Introduction 1

1.1 Rationale 1

1.2 Problem Statement 2

2 Literature Review 4

2.1 Introduction 4

2.2 Kevin Nuss - PCT 4

2.3 Los Alamos National Laboratory - MPI-FTW 5

2.4 LaFon, Moody, and Bringhurst - DCP 5

2.5 Bisson, Patel, and Pasupathy 6

2.6 Kolano and Ciotti - MCP 6

3 Methods 8

3.1 Introduction 8

3.2 Traversal 9

3.3 Producer 13

3.4 Consumer 14

4 Experimental Setup and Results 16
4.1 Introduction 16
4.2 Experimental Setup 16
4.2.1 Linux VM 16
4.2.2 BSU GeneSIS Cluster 17
4.2.3 INL Falcon Cluster 18
4.2.4 Amazon AWS 19
4.3 Data Sets 20
4.4 Timing Measurements 21
4.4.1 BSU GeneSIS Cluster - Copy, Zero Backups 21
4.4.2 INL Falcon Cluster 22
4.4.3 Amazon AWS 23
4.4.4 AWS Cluster - Tree Traversal, Varying Backups 23
4.5 Comparison to Existing Tools 24
4.5.1 dcp - Setup 25
4.5.2 mcp - Setup 25
4.5.3 Results 26
5 Conclusions 29
5.1 What have we done so far? 29
5.2 Future directions 30
REFERENCES 33
A Idaho National Laboratories - Falcon Cluster 34
B Boise State University - GeneSIS Cluster 37

C Amazon - AWS Cluster 39

D Test Tools 41

LIST OF TABLES

4.1 AWS VM Instances 19

4.2 Many Small Files 20

4.3 Fewer Large Files 21

4.4 AWS Files 21

4.5 BSU GeneSIS Cluster Results - Many Small Files 22

4.6 BSU GeneSIS Cluster Results - Linux 'cp' command - Many Small Files 22

4.7 BSU GeneSIS Cluster Results - Fewer Large Files 22

4.8 INL Falcon Cluster Results - Many Small Files 23

4.9 INL Falcon Cluster Results - Fewer Large Files 23

4.10 AWS VM Cluster Results 23

4.11 AWS Cluster Results - Tree Traversal, Varying Backups 24

4.12 AWS VM Tool Comparison - "AWS Files" Set (Table 4.4) 27

LIST OF FIGURES

1.1 Structure of a Parallel Filesystem [10] 2

3.1 PCT parts - 4 producers, 4 consumers 9

4.1 DCP parts - 4 MPI processes 26

4.2 MCP parts - 4 MPI processes 27

4.3 Tool Comparison - AWS Cluster with Lustre 28

CHAPTER 1

INTRODUCTION

1.1 Rationale

While storage size and availability has increased, the tools for manipulating that storage have lagged behind. Parallel file systems increase the capability to store large amounts of data and to access it at high speed. Current tools are not able to take full advantage of these capabilities. A Linux machine connected to a parallel file system may use the standard 'cp' tool to copy a large directory. In this case a single-threaded process will walk the file tree and copy each of the files, one by one. This makes sense for a traditional file system on a single disk; multiple threads or multiple concurrent copy requests could not increase the speed since it is bound by the speed of the disk. But with a parallel file system, especially one with many nodes servicing metadata and IO requests, this type of parallelism can dramatically increase the speed of the operation. See Figure 1.1 for an example of a parallel filesystem.

Some progress has been made in this area, see Literature Review (Chapter 2). This project seeks to develop a tool that provides expanded capabilities in this area and compare it to existing tools.

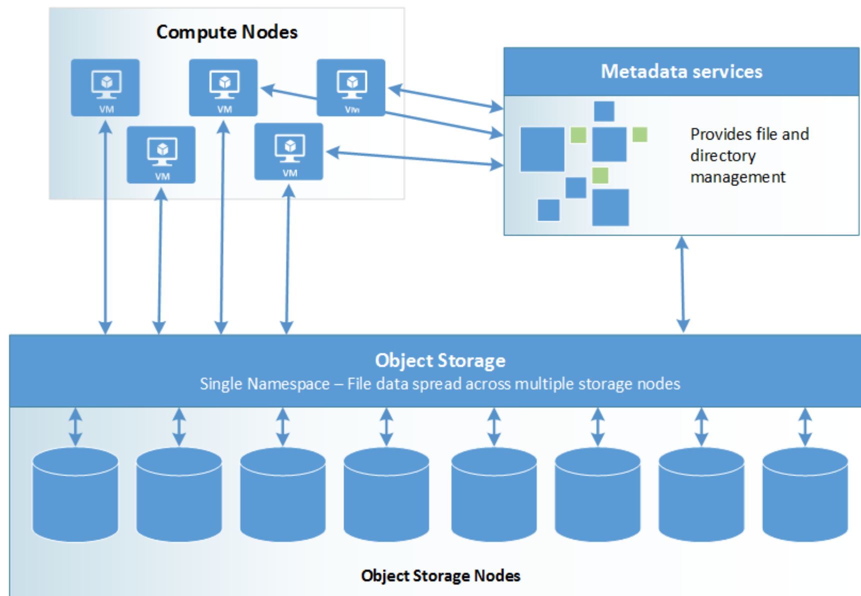

Figure 1.1: Structure of a Parallel Filesystem [10]

1.2 Problem Statement

These are the three principal areas of improvement:

Performance - The tool should utilize the characteristics of a parallel file system to improve upon the performance of earlier tools. It should use multiple threads and multiple machines to take advantage of the parallelism offered by this kind of file system.

Scalability - The tool should be able to run on a smaller network with only a few low power machines, all the way up to a large high performance cluster with many high power machines. It should effectively take advantage of the increased capabilities in the larger, higher powered clusters.

Robustness - Even with improvements in the areas of performance and scalability it still may take hours or even days to completely copy a very large set of files, millions of files in millions of subdirectories. To restart this type of job requires a

lot of time. This tool should gracefully handle most failures, such as the death of one of the processes, without losing progress and without the need to restart the job. The robustness should be tunable, the user should be able to decide what level of protection is needed weighed against the performance impact of maintaining increasing levels of redundancy.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Some work has been done by other groups, working to address the limitations of current copy tools and take advantage of the opportunities presented by parallel filesystems. This includes the work that this project built on (Section 2.2) as well as work by groups from other universities and laboratories. Some of this work is summarized below.

2.2 Kevin Nuss - PCT

In 2013 Kevin Nuss, a graduate student at Boise State University (BSU), created a Message Passing Interface (MPI) based tool called parallel copy tool (`pct`) [4]. This tool divides up the job of copying files among multiple nodes, each with multiple worker threads. The process of walking the directory tree and dividing the files into jobs is performed by a single node with multiple threads. This multithreaded approach was found to speed up the directory traversal over a single-threaded program, but it was still found to be the bottleneck in the process of copying a directory that has many small files in many subdirectories. The need for a tool that uses a distributed

algorithm to perform this tree walk was identified. The work presented in this thesis builds on Nuss's `pct` tool.

2.3 Los Alamos National Laboratory - MPI-FTW

A tool called MPI File Tree Walker (`MPI-FTW`) was developed at Los Alamos National Laboratory (LANL), and was open sourced in 2007. `MPI-FTW` performs a file tree walk with multiple nodes communicating via MPI. It uses a centralized algorithm; the first two process ranks are reserved for management and collection. A single management process tracks the directories yet to be traversed, tracks the available worker processes, and load balances the jobs between them. The worker threads perform the directory traversal and perform a user-specified command on each file. The program collects and reports statistics on the files found. Although this tool makes use of multiple nodes to perform the traversal, its centralized approach to job management limits its scalability. The robustness of this algorithm is limited by the single management process; a loss of this node means failure of the copy job.

Some details about this can be found at the referenced url [5]. The link at LANL has since become unavailable, and has said for months that it is currently undergoing maintenance.

2.4 LaFon, Moody, and Bringhurst - DCP

A team from New Mexico State University (NMSU) and LANL described an approach that overcomes the limitation of a single-node tree traversal, and the limitations of a centralized distributed approach. With this tool, all nodes participate in the file tree walk. If a node has work in its queue, it can continue to process those subdirectories.

If a node runs out of work to do, it can "steal" jobs from a random node. This team found a significant speedup in this decentralized approach. The main drawback in this implementation seems to be robustness, of which no mention was found in the paper. If any node is lost, the jobs queued on that node are also lost. Since no centralized management process is tracking jobs for the threads, the other nodes cannot determine what jobs were dropped by the node when it went down. This would require any node failure to trigger a second scan to fill in the holes. In a file system in which files change frequently, this increases the window in which a file may change.

This research was published in Supercomputing Conference 2012.

This software is now part of the `mpifileutils` suite [8].

2.5 Bisson, Patel, and Pasupathy

In tackling the problem of creating a fast file tree crawler, Bisson, Patel, and Pasupathy described a multi-threaded approach to the file tree traversal [6]. In addition to discussing optimizations for CPU and memory utilization, they compared the performance of a multithreaded file tree crawler to a single threaded approach. They found a significant speedup, but this implementation is still limited by the same factors as the traversal algorithm used by `pct`; it is unable to take advantage of the speedup offered by a multi-node, distributed approach.

2.6 Kolano and Ciotti - MCP

Another open source tool in this area comes from the NASA Advanced Supercomputing Division [7]. `MCP` provides the same functionality as the Linux `cp` utility, but

uses parallel copying to achieve speedup over the standard tool. This parallelism is implemented in both a single node multithreaded approach, and a multi-node distributed approach. Although the actual file copies are performed in parallel this tool has the same potential bottleneck as `pct`; the manager node is the only node that runs a traversal thread and stat thread. This management design limits the scalability and robustness of this tool. As demonstrated by the experiments of Kevin Nuss with `pct`, this approach is best suited for a file tree structure that contains a few large files in a small number of subdirectories. With many small files in many subdirectories, this single manager node becomes the bottleneck and the algorithm cannot scale to keep many worker nodes busy with file copies.

CHAPTER 3

METHODS

3.1 Introduction

This project started by building on the work done by Kevin Nuss with his `pct` program. The `pct` tool is an MPI based program. It uses a single process to perform the directory tree traversal and builds up a list of all the files that need to be copied as well as their size. It then batches these together into jobs based on the number and size of files, and sends these jobs to the consumer processes. The consumer processes then perform the actual copies.

To increase the performance and scalability of this program we needed to move away from the single producer model. In the new implementation the directory traversal is performed in parallel, and all of those traversal processes then act as producers. So now instead of a single producer, many consumer model we have a many producer, many consumer model. This allows better utilization of the capabilities of a network filesystem.

Each phase of this must be decentralized in order to be able to handle the failure of an arbitrary process, and to be scalable. The following sections describe how this was accomplished for each of the phases of `pct`. See Figure 3.1, showing the parts of `pct`.

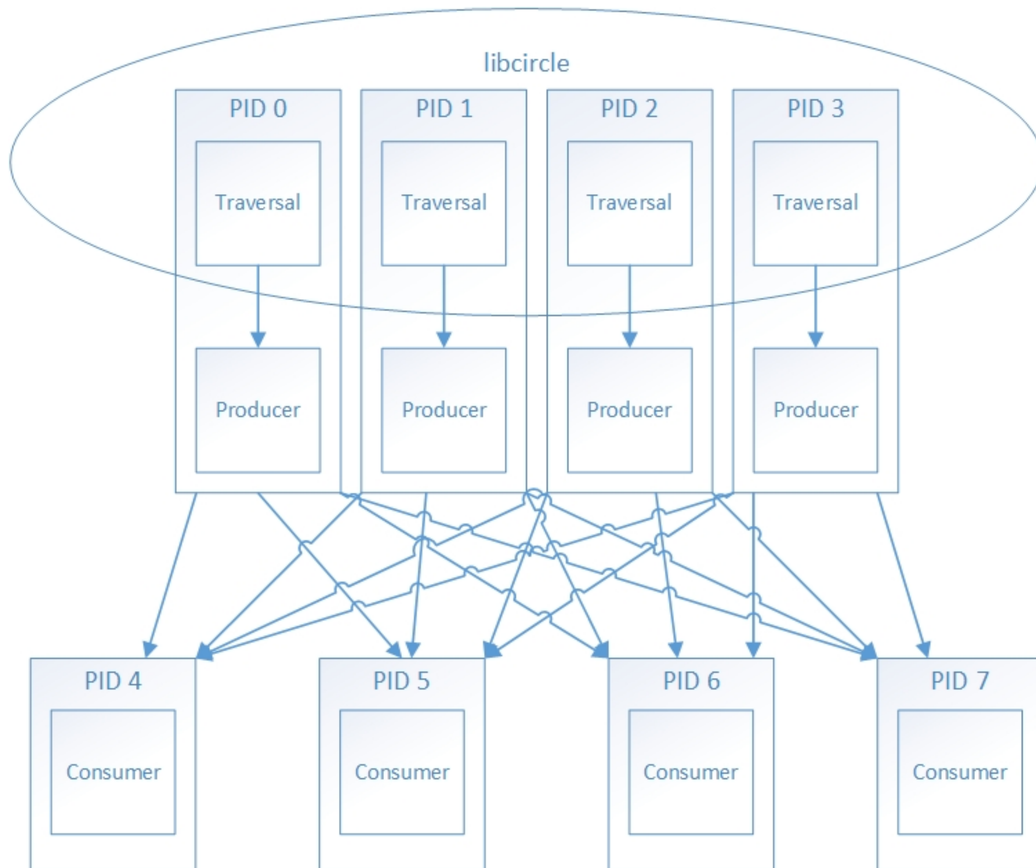

Figure 3.1: PCT parts - 4 producers, 4 consumers

3.2 Traversal

The traversal phase was modified to use the `libcircle` library. This is an open-source library that can be obtained through GitHub [9].

The `libcircle` library provides an API for scalable operations. Each process maintains its queue of work items and is able to exchange work items with random processes without a central coordinator. To keep the processes balanced, `libcircle` makes work requests to random nodes when work is needed. If the node requested has work in its queue it will randomly split that queue with the requestor.

The user of the API defines a function to produce a work item and a function to process a work item. The `libcircle` work item is an array of characters, describing the job to be performed. This makes `libcircle` very versatile, since the interpretation of the items is left to the user-defined functions. It can introduce some inefficiency however since the work item is a fixed-sized array of characters that must be large enough to contain any potential item descriptor.

In the `pct` implementation, the function to produce the initial work item takes the parameters passed in by the user and enqueues a string describing the source and destination directories. The function to process a work item walks a single level within its directory. It adds the files to its producer queue to be combined with other files into a copy job. Each of the subdirectories are added to the `libcircle` work queue to be processed independently.

```
create_work
  for item in directory
    if item is directory
      enqueue to libcircle queue
    else if item is file
      enqueue to local producer queue
```

Although the `libcircle` provides much of the decentralization necessary for fault tolerance it does not handle the failure of a process. This project modified the `libcircle` code to be able to handle this. It was modified to use intercommunicators, and to tolerate a failure to communicate with a process. The idea to use intercommunicators to facilitate fault tolerance came from the "Fault Tolerance in MPI Programs" paper by William Gropp and Ewing Lusk [1].

Using standard communicators in MPI does not allow for the failure of a process. By instead using an array of intercommunicators, a process still has a means to communicate with other processes when an error is returned. This also requires using

a version of MPI that supports returning an error to the program when a send or receive error is encountered. When an error is returned on a send or receive, the program notes that that rank is no longer available, and continues communicating with remaining ranks via their respective intercommunicators.

When processes fail in their random work requests they pick a new random node to request work from. Barrier operations are now handled by a two-pass token ring. This token includes a mask of the nodes that are currently alive. All processes must agree that they are done and agree on the mask of alive nodes before the operation can complete. The first pass begins when processes reach the barrier. The process that starts this operation sets an identifier for the barrier it has reached, and the mask of ranks it believes are still alive. The token is then passed to the next rank. If that process finds that it disagrees on the nodes that are still alive, it will change the mask to a matching subset of nodes between its mask and the mask it received in the token. It will also set its current barrier number; since the barriers in this program are sequential it will set the barrier to a lower number if it has not yet reached the latest barrier. This indicates to the other processes that it is not yet time to move on. The token continues around until a sender receives a token identical to the one it started around the loop. At this point the token is modified to indicate to all processes that it is time to move on from the barrier, and the token is sent once more around the loop.

```
token barrier
  receive token with token_mask and token_barrier_number and token_originator

  if token_barrier_number == end barrier number
    done with this barrier
    if token_originator != me
      send token to next process
    continue
```

```
if token_originator == me and token_mask == my_mask and
        token_barrier_number == my_barrier_number
  increment my_barrier_number
  send token with updated barrier number to next process
  continue

if my_barrier_number < token_barrier_number
  set token_barrier_number to my_barrier_number

if my_mask != token_mask
  set my_mask to (my_mask & received_mask)
  set token_mask to my_mask

send token to next process
continue
```

The token ring is also used to implement a heartbeat to remove nodes from the mask when they stop responding. This heartbeat token works just like the barrier token, but its operation is a bit more simple. It just provides updates to all alive processes on which ranks are still available.

Within the API's user-defined function for consuming work, in this case the function to walk a particular directory, we added the capability to send backup copies to a specified number of backup nodes. This is how robustness to process failure is enforced in this phase. Each node has a queue of the jobs it owns, as well as queues of jobs for which it is the first backup, second backup, etc. The number of backups is determined by the user when the program is launched, based on the number of process failures the job should tolerate. If, for example, the number of backups is two then process 0 will send a copy of the work items it creates to the first backup queue on process 1, and the second backup queue on process 2. If process 0 fails then process 1 will add the work items from its first back queue to its owner queue. If processes 0 and 1 both fail, then process 2 will pick up the items from its first backup

queue (previously owned by process 1) and its second backup queue (previously owned by process 0). In this example each process is sending to the next two processes in numerical order, looping back to process 0 at the end. Because of this model a run where two backups are used may actually tolerate far more than two process failures. It can tolerate at most the failure of two adjacent processes before data is lost.

```
replicate directory to backups
   for i in 1 to backup_count
      set backup_process to (my_rank + i) % number_processes
      send to queue i on process backup_process
```

Once a directory has been traversed and its files have been added to the producer queue, and its subdirectories added and replicated, then the directory work item can be released along with its backup copies. This means the queue of work items and their backups consist of only the lowest unexamined level of each branch of the directory tree.

When a traversal process randomly splits its queue to pass work to the another process, we must take steps to retain reduncancy. Jobs are first copied to the new backups, then the new owner, then removed from the old backups. This ensures that if a process fails while splitting its queue, the job will still be picked up by a backup.

When all the traversal processes have run out of work items the file tree has been walked in its entirety, and all the file descriptors have been added to the queue for the copy job producer.

3.3 Producer

The producer phase also implements this same type of redundancy. This phase is also using intercommunicators, and using the token ring heartbeat to keep track of

the processes that are alive.

When a file has been added to the producer's list, and before its parent directory can be forgotten in the traversal process, one or more backup copies of the file's description is added to backup queues in other proccesses. Then if one of the producer processes fails its backup will add the list of files into its owner queue. The producer phase leverages the code from the `pct` project to batch files into jobs based on the number of files and their size. The producer sends these copy jobs to the consumer processes. The producer does not remove the files from its list until the consumer acknowledges that the corresponding copy job has completed successfully.

```
create copy job
   combine files into a job with a ticket number
   send job to ready consumer

release copy job
   receive completion acknowledgement with ticket number from consumer
   look up job in local queue
   for file in job
      release file in backup queues of remote processes
      release file in local queue
```

3.4 Consumer

Since the producer maintains its list of files, protected by the specified level of redundancy, until the copy job is completed there is no need for another level of replication in the consumers. If a consumer process dies while a producer has outstanding jobs to that process, then the producer will simply assign those files to a new consumer. For this reason, regardless of the level of redundancy specified at the start of the job any number of consumer processes may be lost until there is only one remaining without the program failing.

Again the token ring is used to maintain a heartbeat and keep track of a mask of the nodes that are still communicating. The token ring is also used for barrier operations.

CHAPTER 4

EXPERIMENTAL SETUP AND RESULTS

4.1 Introduction

This software was tested in a number of different enviornments, for correctness as well as to look at the scalability and portability. This ranged from a VM running on a standard PC, to a small cluster at BSU, to a large high performance computing cluster at INL. Tests were also performed in a cloud-computing environment through Amazon AWS. See the sections below for details on the setup in each of these enviroments as well as the results.

4.2 Experimental Setup

4.2.1 Linux VM

This environment was used for development testing. All MPI processes run on the same machine. This environment was useful for initial development and for testing of process failure and recovery. Since this is a Linux VM running on a standard desktop PC, with one hard drive and no parallel filesystem, this setup was not used for testing the performance and scalability of this implementation.

```
Linux VM details:
  VM running in Oracle VirtualBox
  Guest OS - Fedora 20
```

```
CPUs - 1
Memory - 2 GB
```

4.2.2 BSU GeneSIS Cluster

The GeneSIS cluster at BSU has the Lustre parallel filesystem. This environment was used to test the correctness of this product in a real-world cluster.

See Appendix B for more information about this cluster.

Debug code was added to `pct` to test recovery from process failure in this environment. The program reads a configuration file when it starts up. Each line in the file specifies a process number, a specific point in the code, and a count. Once that process has executed that code the specified number of times it will trigger an abnormal termination. This forces the surviving processes to go through their recovery mechanisms and process the backup work items for that process.

This code uses a simple mechanism to accomplish this. Calls to the function `code_point_count` are sprinkled throughout the code. Each place that calls this function calls it with a unique identifier. If the configuration file had an error set up for this process and point, and the count is reached, it triggers a seg fault.

```
void code_point_count(int point)
{
    int *x = 0;
    int y;
    code_point_counters[point].cur_count++;
    if (code_point_counters[point].cur_count == code_point_counters[point].limit) {
        // Segfault
        printf("*** Injected error ***\n");
        y = *x;
    }
}
```

Some examples of places where this code was used:

- Traversal - After a process has added a directory to its work queue. When this process panics, ensure that the backup picks up this work and the item is not lost.

- Producer - Panic a producer with a number of items in its queue. Make sure the backup picks up this work and sends it to the consumers.

- Consumer - Panic a consumer when it has an outstanding job from a producer. Ensure the producer reassigns the job to another consumer.

In addition to this programmatic method of testing, this was also tested by randomly killing process via the command line while the program was running.

The GeneSIS cluster was also one of the environments used to test the performance of `pct`. Results are recorded below.

4.2.3 INL Falcon Cluster

The Falcon cluster at INL was used to test the scalability of this product. This is a much larger cluster, allowing for experiments with 16 and 32 MPI processes, with each process on its own node. Processes were spread out one per node to reduce contention for network interfaces. Nodes were not reserved exclusively for this experiment, jobs from other users were allowed to run on other CPUs on the same node. This, along with contention for the network filesystem, created some variability in the amount of time taken to complete each job.

See Appendix A for more information about this cluster.

4.2.4 Amazon AWS

The pct tool was also tested in a cloud computing environment. Amazon AWS provides a template to launch a cluster of VMs using the Lustre parallel filesystem [3].

A user fills out the template, choosing the VM type for each of the components required to set up the Lustre filesystem, as well as the clients that will have access to that system. Once setup completes, each of the clients has the Lustre filesystem mounted. After installing MPICH and other necessary packages the cluster has everything necessary to test pct.

The following table shows the VM types that were used in this experiment. See Table 4.1. A full description of the types can be found through AWS, a brief description including number of virtual CPUs and memory size has been included here.

Table 4.1: AWS VM Instances

Quantity	Purpose	VM Type	vCPU	Mem(GiB)
16	Clients	c4.xlarge	4	7.5
1	MGS	c4.xlarge	4	7.5
1	NAT	m3.medium	1	3.75
5	OSS and MDS	c4.2xlarge	8	15

This setup was also used to measure performance against the Linux ‘cp’ command and ‘mcp’ from Nasa Supercomputing Division. Section 4.5 has details on the results of these experiments.

4.3 Data Sets

This was tested on the GeneSIS cluster at BSU and the Falcon cluster at INL with two different data sets. The first set is the "Many Small Files" set. This set is a directory tree with 17,596 total directories. It contains 1,000,000 files totaling 904.1 GB. The files sizes range from 0.25 MB to 2.50 MB. See Table 4.2. The second set is the "Fewer Large Files" set. This set is a directory tree with 185 directories. It contains 2,460 files totaling 989 GB. The file sizes range from 1 MB to 3480 MB. See Table 4.3. In both cases the distribution used to determine the file sizes is a "zipf" distribution. The files are uniformly distributed across the directories in the tree. Files were filled with random data generated by `/dev/urandom`

The data set used on the AWS cluster was a randomly generated directory tree of 1,469 directories. It contained 75,000 files, rangining from 1 MB to 5 MB. A zipf distribution was used to determine the file sizes. The files were uniformly distributed across the directories. Files were filled with random data generated by `/dev/urandom`. See Table 4.4.

Table 4.2: Many Small Files

File Size	Number of Files
0.25 MB	289,244
0.50 MB	168,465
0.75 MB	119,901
1.00 MB	93,002
1.25 MB	76,031
1.50 MB	64,516
1.75 MB	56,017
2.00 MB	49,037
2.25 MB	44,204
2.50 MB	39,583

Table 4.3: Fewer Large Files

File Size	Number of Files
1 MB to 1023 MB	2,097
1024 MB to 2047 MB	205
2048 MB to 3071 MB	122
3072 MB to 3480 MB	36

Table 4.4: AWS Files

File Size	Number of Files
1 MB	29,116
2 MB	16,974
3 MB	11,967
4 MB	9,376
5 MB	7,567

4.4 Timing Measurements

4.4.1 BSU GeneSIS Cluster - Copy, Zero Backups

The tests on the GeneSIS cluster at BSU used the "Many Small Files" (see Tables 4.5 and 4.6) and "Fewer Large Files" (see Table 4.7) sets described above. The results of these tests are shown below. Measurements were taken for the amount of time taken to traverse the directory tree and discover the files to be copied, and the total time taken for both traversal and copying the files.

Table 4.5: BSU GeneSIS Cluster Results - Many Small Files

	Avg Traversal Time	Avg Total Time
2 producers, 2 consumers	183.4 s	27,682.9 s
4 producers, 4 consumers	146.6 s	14,832.6 s

Table 4.6: BSU GeneSIS Cluster Results - Linux ’cp’ command - Many Small Files

Avg Total Time
46,202.687 s

Table 4.7: BSU GeneSIS Cluster Results - Fewer Large Files

	Avg Traversal Time	Avg Total Time
2 producers, 2 consumers	1.3 s	6,728.5 s
4 producers, 4 consumers	1.8 s	7,943.0 s

4.4.2 INL Falcon Cluster

Tests on the Falcon cluster at INL used the same “Many Small Files” and “Fewer Large Files” sets as the tests on the GeneSIS cluster at BSU. Only the data in the files was different, since the data set was rebuilt on this cluster and files are filled from `/dev/urandom`. The results of these tests are shown below. Measurements were taken for the amount of time taken to traverse the directory tree and discover the files to be copied, and the total time taken for both traversal and copying the files. See Tables 4.8 and 4.9.

Table 4.8: INL Falcon Cluster Results - Many Small Files

	Avg Traversal Time	Avg Total Time
8 producers, 8 consumers	125.7 s	19,474.0 s
16 producers, 16 consumers	150.3 s	10,918.7 s

Table 4.9: INL Falcon Cluster Results - Fewer Large Files

	Avg Traversal Time	Avg Total Time
8 producers, 8 consumers	14.2 s	1,478.7 s
16 producers, 16 consumers	16.2 s	885.7 s

4.4.3 Amazon AWS

The tests on the AWS VM cluster used the “AWS Files” set detailed above. The results of these tests are shown below, see Table 4.10.

Table 4.10: AWS VM Cluster Results

	Avg Total Time
2 producers, 2 consumers	1,077.8 s
4 producers, 4 consumers	566.8 s
8 producers, 8 consumers	381.8 s

4.4.4 AWS Cluster - Tree Traversal, Varying Backups

Tests were performed to measure the impact of the redundancy features that were added to `pct`. This involved creating a very large directory tree and performing only the traversal portion of the copy. Multiple tests were performed, varying the number of backups each traversal and producer process will use to replicate job data in case

of failure. The impact of increasing levels of redundancy is detailed below. The tree used contained 500,000 files in 181,593 directories. The results are in Table 4.11.

Table 4.11: AWS Cluster Results - Tree Traversal, Varying Backups

	Avg Traversal Time
4 producers, 0 backups	144.2 s
4 producers, 1 backups	154.7 s
4 producers, 2 backups	159.1 s
8 producers, 0 backups	76.4 s
8 producers, 1 backups	80.1 s
8 producers, 2 backups	82.7 s
8 producers, 3 backups	84.4 s

Even on this very large directory tree, replicating work to multiple backups did not have a large impact on the time needed to perform the directory traversal. The time jump from zero to one backups is a bit larger than the jump from one to two. This is because with zero backups there is no replication; with one there is replication to the single backup as well as re-replication to the new owner and the new backups when the libcircle queue is randomly split to share work. Moving from one to two adds just one more backup into the steps needed for one backup.

4.5 Comparison to Existing Tools

The performance and scalability of `pct` was compared to existing tools using an Amazon AWS cluster with the Lustre parallel filesystem. The experiments with other

tools used the same VM setup and the same file tree setup as the `pct` experiment detailed above.

cp was tested with the standard parameters to provide a baseline to compare the performance of `pct`, `dcp`, and `mcp`.

The results of tests with `pct`, `dcp`, and `mcp` are shown each using 4, 8, and 16 MPI processes.

4.5.1 dcp - Setup

`dcp` is part of the mpifileutils suite developed by LANL [8]. Like `pct`, `dcp` uses `libcircle` as part of its implementation. See Figure 4.1 for the `dcp` design. In addition to the requirements for `mcp`, the following packages were installed to compile `dcp`: automake, libtool, gcc-c++, libarchive-devel, git, openssl-devel. There are also additional packages required from github: libcircle [9], lwgrp [11], and dtcmp [12]. This left compilation errors for some of the utilities in mpifileutils but was sufficient to compile `dcp`.

4.5.2 mcp - Setup

`mcp` is part of the mutil suite developed by NASA [7]. See Figure 4.2 for the `mcp` design. This comparison was done with mutil-1.76.1. This is a patch for coreutils-7.6. In addition to coreutils and the mutil patch, the following packages were required to compile and use this software: kernel-headers, gcc, gperf, mpich-devel. The configure file was modified to work with mpich. Experimentation showed that with the default parameters, including –mpi, the program never exits. Instrumentation was added to track down the problem. MPI process 0 never exits because it is waiting for a flag set by the thread printing stats on copied files. This thread is not created unless the

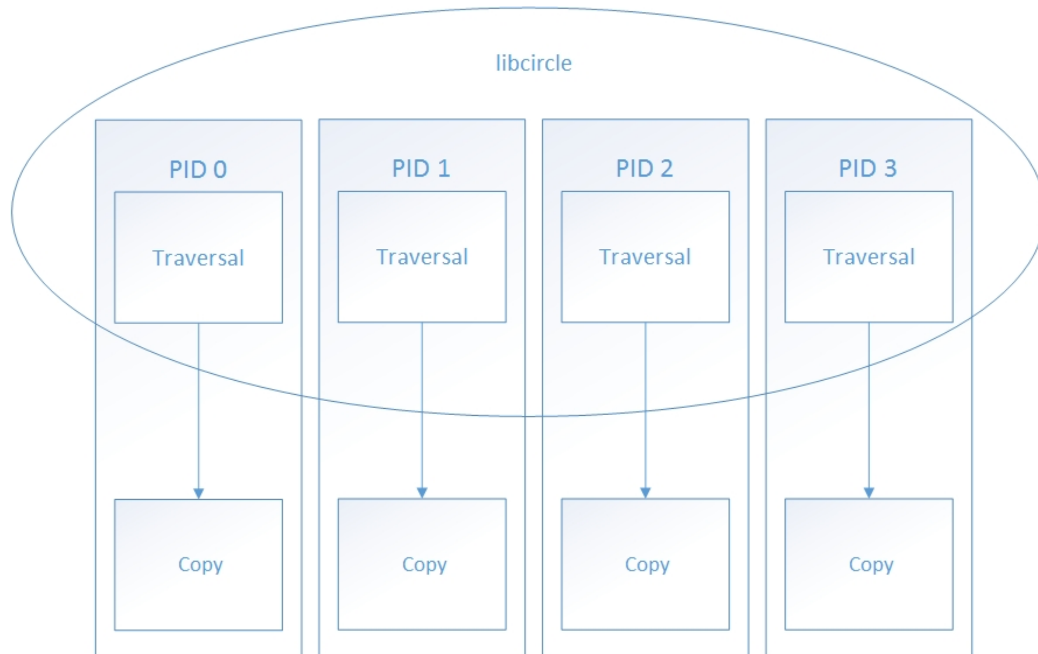

Figure 4.1: DCP parts - 4 MPI processes

–print-stats flag is used. For this reason –print-stats was included on the command line in the tests using `mcp`.

4.5.3 Results

`pct`, `dcp`, and `mcp` all showed a significant improvement over the standard `cp`. `pct` showed significant improvement with an increasing number of processes. `dcp` was faster than `pct` with 4 and 8 processes, although `pct` outperformed `dcp` at 16 processes. It is worth noting that `dcp` is using all MPI processes for the actual copying, whereas `pct` is using only half with the given parameters. `mcp` was comparable to `dcp` at 4 MPI processes, but showed no significant improvement increasing to 8 or 16 processes. Both `pct` and `dcp` outperformed `mcp` with the higher numbers of processes.

Full results are shown below, in Table 4.12 and Figure 4.3.

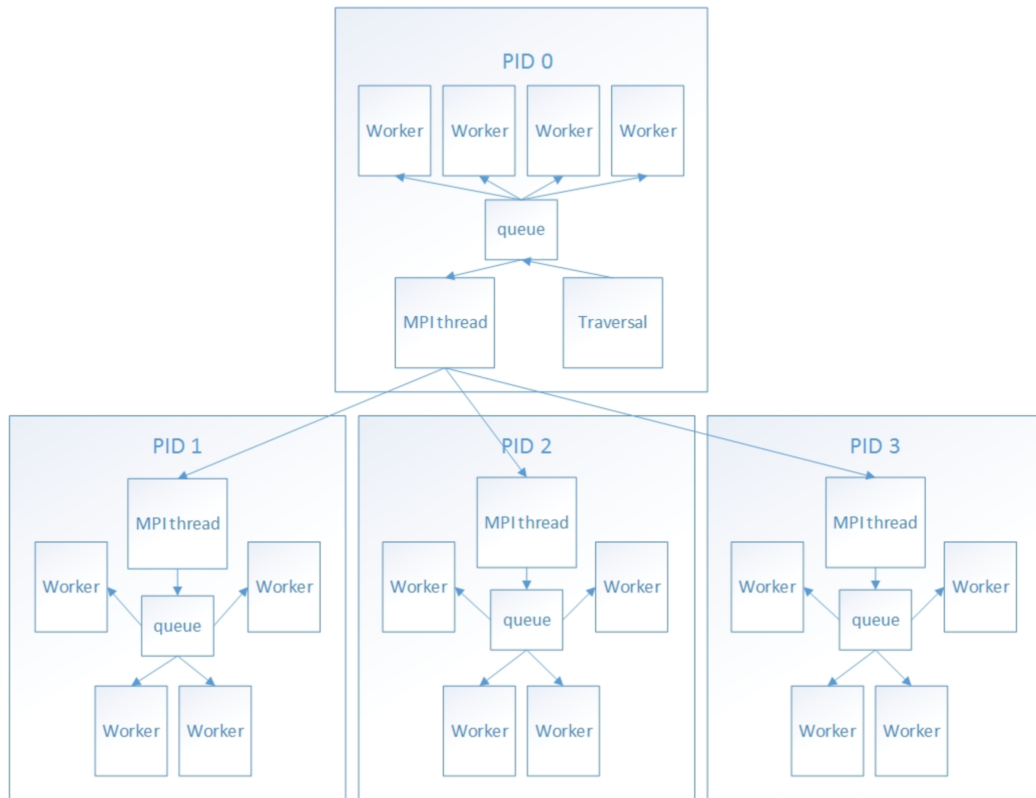

Figure 4.2: MCP parts - 4 MPI processes

Table 4.12: AWS VM Tool Comparison - "AWS Files" Set (Table 4.4)

Tool	MPI Procs	Params	Avg Total Time
cp	NA	default	1938.1 s
pct	4	2 prod, 2 cons	1077.8 s
pct	8	4 prod, 4 cons	566.8 s
pct	16	8 prod, 8 cons	381.8 s
dcp	4	default	608.6 s
dcp	8	default	403.1 s
dcp	16	default	393.6 s
mcp	4	–mpi –print-stats	627.1 s
mcp	8	–mpi –print-stats	604.7 s
mcp	16	–mpi –print-stats	607.3 s

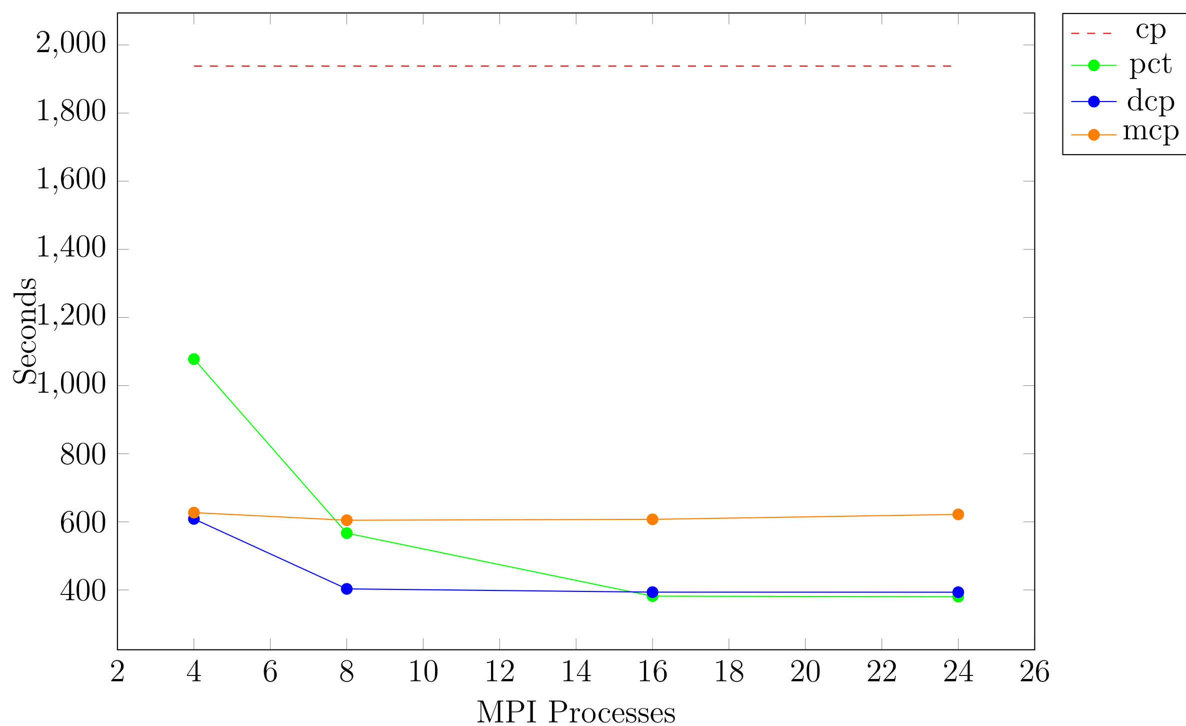

Figure 4.3: Tool Comparison - AWS Cluster with Lustre

CHAPTER 5

CONCLUSIONS

5.1 What have we done so far?

The goal of this project was to create a solution to the problem of copying files in a network filesystem, building on the work of Kevin Nuss. The software tool should meet the three criteria stated earlier: Performance, Scalability, and Robustness.

The experiments detailed here have shown that this software is able to perform well in and scale to a variety of environments, from a single VM instance, to a large high performance computing cluster, to a VM cluster in a cloud computing environment. For a sufficiently large file tree, speedups were realized increasing to very large sets of processes working on the tree. This showed very large improvements over the standard Linux `cp` command.

`pct` performance was comparable to the tools it was compared against, `mcp` and `dcp`. At lower numbers of `MPI` processes these tools outperformed `pct`, though as noted above much of this can be explained by the fact that `pct` uses only half of its processes to do the actual copying. With increasing numbers of `MPI` processes `pct` actually outperformed both of these tools on the randomly generated file tree with which they were tested.

By adding options to tolerate process failure, this software becomes even more useful in environments where very large copy jobs are needed. Processes may be

taken out by failures outside this software, or even intentionally when resources need to be diverted after a copy job has begun. This is one of the larger contributions of this project, and is not addressed in other of the other software that was studied.

5.2 Future directions

There are some opportunities for enhancements and improvements in this project.

In the current implementation the traversal phase runs to completion before the producer/consumer phase begins. This means the source and destination descriptions for the entire file tree are queued up all at once. If the file tree is sufficiently large compared to the memory available for the `pct` processes, this could lead to issues with running out of memory. This issue is compounded when using a higher number of backups, since the memory required to queue up all these items is directly proportional to the number of backups (i.e. adding one backup doubles the amount of memory needed for this queue). One way to alleviate this memory pressure would be to allow the file copying to begin while the traversal is still ongoing. As files finish copying, the record of source and destination description is released, along with the backup copies. A configurable memory cap could be implemented, where the traversal processes would be throttled from adding new items to the producer queue until the producer/consumer processes have completed enough work to make room for the new items. This could potentially affect the speed of the copy, especially if that cap is set at a low number, but it would add greater flexibility to allow this software to work in a larger set of environments.

Another area of opportunity is in the `libcircle` library itself. The `libcircle` library defines the work items in its queues as fixed sized strings. Keeping this

fixed size means that all work items use the same amount of memory regardless of the actual string length needed to describe the source and destination paths. Files with small path descriptor lengths have a lot of unused space that is wasted. The maximum length must be set long enough to include the maximum path length for both the source and the destination paths, or `pct` will fail to copy files of a large enough path length. Significant modification would be required to introduce variable buffer sizes in the `libcircle` work items, as well as modification on the `pct` side. This modification could potentially make the software more flexible. As with the improvements described above, this would also increase the set of environments where this software could be effective.

Currently the MPI processes that perform the traversal are the same processes that act as producers in the copy phase of `pct`. This introduces a couple of limitations. The test results showed that as the number of processes performing the traversal gets sufficiently large, it actually takes more time to traverse the tree than it would with a smaller number of proceeses. This is because the overhead of communication between the processes, and exchanging jobs between processes in `libcircle`, becomes large compared to the time needed to actually walk the tree. The traversal and producer processes could be decoupled. This would allow more fine tuning, since a different number of processes could be specified for performing the traversal and for the number of processes acting as producers. This would require some additional work, but would also open the door to another area of improvement. Currently the process that traverses a directory keeps all files discovered in the first level of that directory in its queue, while queueing subdirectories to be walked by any process. Although the traversal phase allows directory walk jobs to be swapped between processes to more evenly spread out the work, this does not happen at the producer level. This means

that all files added to a producer queue by a process will be owned by that process. In general, in a large tree with files spread across the tree, the files will spread out fairly evenly. But it is possible to construct a file tree that in the worst case would highly skew the file list to one process. The producer processes could be enhanced to balance their load. It is questionable how much benefit this would provide in overall speed, since the consumers doing the actual copy are the bottleneck at this phase and can request jobs from any producer with available items.

REFERENCES

[1] Gropp and Lusk. *Fault Tolerance in MPI Programs*, Jul. 2016, http://www.mcs.anl.gov/publication/fault-tolerance-mpi-programs

[2] Pinheiro, Weber, Barroso. *Failure Trends in a Large Disk Drive Population*, 2007, https://www.usenix.org/legacy/events/fast07/tech/full_papers/pinheiro/pinheiro_old.pdf

[3] Bhakti, Micah. *Intel Cloud Edition for Lustre - Self Support* Feb. 2018, https://wiki.hpdd.intel.com/display/PUB/Intel+Cloud+Edition:for:Lustre*+-+Self+Support

[4] Knuss, Kevin. *Parallel Copy Tools for Distributed File Systems* Mar. 2013, http://scholarworks.boisestate.edu/cs_gradproj/4/

[5] Grider and Torrez. *MPI File Tree Walk* Apr. 2007, https://www.osti.gov/scitech/biblio/12311011-mpi-file-tree-walk

[6] Bisson, Patel and Pasupathy. *Fast File System Crawler* https://atg.netapp.com/wp-content/uploads/2012/12/FS_crawler_Bisson.pdf

[7] Kolano and Ciotti. *Multi-Threaded Multi-Node Utilities (Mutil)* https://pkolano.github.io/projects/mutil.html

[8] Bringhurt, et al. *Mpifileutils* https://github.com/hpc/mpifileutils

[9] Bringhurst and LaFon. *Libcircle* https://github.com/hpc/libcircle

[10] Wu, Tony and Hopkins, Tycen "Parallel file system architecture: compute nodes look up file information via metadata services, then perform I/O tasks against object storage" *Parallel File Systems for HPS Storage on Azure* Ed Price, Microsoft, 2017, https://blogs.msdn.microsoft.com/azurecat/2017/03/17/parallel-file-systems-for-hpc-storage-on-azure/

[11] Unknown *Lwgrp*, Lawrence Livermore National Laboratory, https://github.com/hpc/lwgrp

[12] Unknown *Dtcmp*, Lawrence Livermore National Laboratory, https://github.com/LLNL/dtcmp

APPENDIX A

IDAHO NATIONAL LABORATORIES - FALCON CLUSTER

These details of the Falcon cluster at Idaho National Laboratories were provided by Peter Cebull.

Storage: Our high-speed scratch storage resides on a Panasas ActiveStor 14 system, using a PanFS parallel file system. Each compute node on the Falcon compute cluster runs a Panasas DirectFlow client to optimize file I/O between the compute nodes and the storage server. There is 230 TB of scratch storage.

Falcon is an SGI ICE-X distributed memory system with 34,992 cores, 121 TB of memory and a LINPACK rating of 1,087.58 TFlops. Falcon came online in November 2014 and ranked 97th on the TOP500 list. After a series of upgrades to the current 34,992 cores, it is currently #143 on the TOP500.

Specifications:

- Overview
  - 34992-core SGI ICE X distributed memory cluster
  - 121 TB total memory
  - FDR InfiniBand Network (56 Gbit/s), Single-Plane Enhanced Hypercube Topology
  - SUSE Linux Enterprise Server 11 Service Pack 4 operating system
  - LINPACK: 1087.58 TFlops
- 972 Compute Nodes with:

- 2 Intel Xeon E5-2695 v4 CPUs
  - Broadwell chipset
  - 18 cores per CPU
  - 2.10 GHz
- 128GB of RAM
- FDR InfiniBand Interconnect

APPENDIX B

BOISE STATE UNIVERSITY - GENESIS CLUSTER

The details on the GeneSIS cluster at Boise State University were provided by Ben Peterson.

The GeneSIS storage cluster which is a 21 node Beowulf style cluster dedicated to bioinformatics research is:

- Connected via both dual channel bonded GigE and low latency infiniband SDR.
- Eight dedicated processing nodes, Intel i7 Quadcore processors, 12 GB system memory.
- Eight dedicated storage nodes supporting a parallel distributed storage system with Intel i7 Quadcore processors, 12 GB system memory each node, total 78 terabytes of raw storage space.

APPENDIX C

AMAZON - AWS CLUSTER

Amazon AWS provides a template to launch a cluster of VMs using the Lustre parallel filesystem [3]. This template does most of the work creating the cluster. It sets up all the Lustre filesystem components, creates the worker VMs of the user-specified type, then mounts the file system on the worker VMs. After that, it is up to the user to install software and all the necessary packages.

`pct` requires installing kernel-headers, gcc, gperf, and mpich-devel packages. It also requires git in order to retrieve the code from github.

Setting up the cluster to be able to run the mpi program requires setting up ssh keys along with an ssh config file to be able to log into each node without a password. It also requires adding the all nodes to the /etc/hosts files.

One of the sources of frustation trying to set this up is the default firewall settings. Adding VMs to the same security group made no difference in allowing them to communicate via MPI. The firewall has all relevant ports locked down. The easiest method found to get around this is to completely disable the firewall on all VMs.

APPENDIX D

TEST TOOLS

Some tools were created to aid in testing `pct`

- rand_dir - This tool is used to created a randomly generated directory. A configuration file is used to specify the maximum depth of the directory tree and the maximum number of children per directory. It also specifies a number of files to create, their size range, and the distribution to use when determining the size. Distributions currently supported are normal and zipf. Seeds are used to be able to create the same tree on multiple systems. Files are filled with random data from the /dev/urandom device.
- dist_dir - This tool provides some distributed operations to aid in testing. These operation include distributed checksum between two trees, and distributed deletion of a tree.