Boise State University
ScholarWorks

Computer Science Graduate Projects and Theses | Department of Computer Science

3-1-2013

Parallel Copying Tools for Distributed File Systems

Kevin Matthew Nuss
Boise State University

PARALLEL COPYING TOOLS
FOR DISTRIBUTED FILE SYSTEMS

by

Kevin Matthew Nuss

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

March 2013

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the project submitted by

Kevin Matthew Nuss

Project Title: Parallel Copying Tools For Distributed File Systems

Date of Final Oral Examination: 01 March 2013

The following individuals read and discussed the project submitted by student Kevin Matthew Nuss, and they evaluated their 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
Murali Medidi, Ph.D.	Member, Supervisory Committee
Tim Andersen, Ph.D.	Member, Supervisory Committee

The final reading approval of the project was granted by Amit Jain, Ph.D., Chair, Supervisory Committee. The project was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ABSTRACT

Parallel distributed files systems are increasingly being used on clusters to allow greater throughput of data to the many compute nodes. They are also an effective way to store massive amounts of data. However, using the standard core utility `cp` does not make good use of the potential parallelism of the file systems. Using multiple `cp` commands has inherent problems too.

Two utilities were created to help recursively copy directories containing large amounts of data on parallel distributed file systems. One of the test data sets contains very many files, and the other contains large files. One utility is a C program that submits a single job on a user specified number of nodes. The work of copying the files is dynamically distributed among those nodes using MPI communications. Multiple threads are used to traverse the directories. Speedups of 9.57 and 7.36 were attained for the many files set and the large files set, respectively. A second utility is written in Java. It also uses multiple threads to traverse the directories, but it performs the copying by creating Bash scripts and submitting them to the job scheduler. The work is balanced among those scripts and the number of jobs is specified by the user. It reached speedups of 3.67 and 7.32 for the same two data sets. Both utilities can also be used to track the progress of the jobs they have submitted.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF TABLES vi

LIST OF FIGURES vii

LIST OF ABBREVIATIONS viii

1 Introduction 1

1.1 Similar Tools 2

1.2 Multithreaded Directory Traversal 5

1.3 Problems That Needed Resolution 7

2 Implementation Strategies 9

2.1 Unimplemented Strategies 9

2.2 Dynamically Distributed Work: `pct` 10

2.2.1 Overview 10

2.2.2 Command Line Parameters 13

2.2.3 Multithreaded Directory Traversal 17

2.2.4 Work Distribution and Management 19

2.2.5 MPI Messaging Strategies 21

2.3 Batched Work Distribution: `Bpct` 25

2.3.1 Overview 25

2.3.2 Command Line Parameters 25

2.3.3 Object Oriented, Java Implementation 28

2.3.4 Work Distribution 30

3 Testing and Results 32

3.1 Clusters 32

3.1.1 BSU GeneSIS Cluster 32

3.1.2 INL's Fission and Icestorm Clusters 34

3.2 Characteristics of the Test Data 36

3.3 Performance Tests 37

3.3.1 Tests Using `pct` 38

3.3.2 Tests Using `Bpct` 51

4 Conclusions 62

4.1 General Conclusions 62

4.2 Future Directions 63

REFERENCES 66

LIST OF TABLES

3.1 Distribution of File Sizes in the "Many Small Files" Set 37

3.2 Distribution of File Sizes in the "Few Large Files" Set 37

3.3 Distribution of File Sizes in the "Partial Large Files" Set 38

3.4 Example Timings for `pct` Copying "Many Small Files" Set 39

3.5 Example Timings for `pct` Copying "Few Large Files" Set 39

3.6 Time/Speedup for `pct` on GeneSIS Copying "Partial Large Files" Set . 41

3.7 Time/Speedup for `pct` on GeneSIS Copying "Many Small Files" Set . . 43

3.8 Time/Speedup for `pct` on Fission Copying "Many Small Files" Set 47

3.9 Time/Speedup for `pct` on Fission Copying "Few Large Files" Set 48

3.10 Time/Speedup for `pct` on Icestorm Copying "Many Small Files" Set . . 50

3.11 Using `Bpct` on GeneSIS to Copy "Many Small Files" Set 52

3.12 Using `Bpct` on GeneSIS to Copy "Partial Large Files" Set 54

3.13 Using `Bpct` on Icestorm to Copy "Many Small Files" Set 56

3.14 Using `Bpct` on Icestorm to Copy "Few Large Files" Set 57

3.15 Using `Bpct` on Fission to Copy "Many Small Files" Set 59

LIST OF FIGURES

2.1 `pct` Generated Job Script 15

2.2 `pct` Help Text 15

2.3 Displaying Progress of `pct` Jobs 16

2.4 `Bpct` Help Text 26

2.5 Displaying Progress of `Bpct` Batches 27

3.1 `pct` on GeneSIS Copying “Partial Large Files” Set: Graphed by Nodes 41

3.2 `pct` on GeneSIS Copying “Partial Large Files” Set: Cores Per Node . . . 42

3.3 `pct` on GeneSIS Copying “Many Small Files” Set: Graphed by Nodes . 43

3.4 `pct` on GeneSIS Copying “Many Small Files” Set: Cores Per Node 44

3.5 `pct` on Icestorm and Fission Clusters Copying “Few Large Files” and “Many Small Files” Sets Using a Single Node 45

3.6 `pct` on Fission Copying “Many Small Files” Set 47

3.7 `pct` on Fission Copying “Few Large Files” Set 49

3.8 `pct` on Icestorm Copying “Many Small Files” Set 51

3.9 Best and Worst `Bpct` Jobs on GeneSIS Copying “Many Small Files” Set 53

3.10 Best and Worst `Bpct` Jobs on GeneSIS Copying “Partial Large Files” . . 55

3.11 Best and Worst `Bpct` Jobs on Icestorm Copying “Many Small Files” Set 56

3.12 Best and Worst `Bpct` Jobs on Icestorm Copying “Few Large Files” Set . 58

3.13 `Bpct` Jobs on Fission Copying “Many Small Files” Set 59

LIST OF ABBREVIATIONS

BSU – Boise State University

GeneSIS – Gene Sequence Information System

INL – Idaho National Laboratory

MPI – Message Passing Interface

PBS – Portable Batch System

CHAPTER 1

INTRODUCTION

The perpetual improvements in computer hardware, files systems, and network connectivity continue to march forward, providing users increasing parallelism, compute speeds, and disk storage options. The tools and utilities available to users and system administrators also change and grow as new or increased capabilities are created. However, one gap in the cluster management tool set has been identified. The Linux core utility `cp` can perform recursive copies of directory structures, but since it copies a single file at a time, it can be very time consuming to make copies of large data sets containing many files. Occasionally, users want to copy a portion of their data to preserve a snapshot of an existing set of output. Cluster or network administrators may want to take a quick snapshot of large portions of the file system so that the backup process makes copies of a self-consistent set of files. In both cases, users need to not be using those files or else risk that the copies are of files that do not necessarily match with each other. The multiple network connections used for parallel distributed file systems plus the multiple nodes of a cluster allow the possibility of decreasing the time required to make extensive copies and thus reduce the time those files are unavailable to the user.

With parallel distributed file systems, multiple `cp` commands can be executed concurrently on subdirectories or files within the source directory to take advantage of

the capability of handling multiple files simultaneously, but the user is then forced to mentally estimate divisions of the source directory and to also monitor each command for successful completion. This creates the possibility of errors by the user both in potentially leaving out subdirectories or in poor estimation of time to complete the copies and thus creating an unbalanced load and poor use of the resources. The need for a command line parallel copy command was identified both by practical use of the local Boise State University (BSU) cluster and by others [3].

This project investigates the issues and implements two new command line utilities to perform recursive directory copies in parallel. By having a program read and analyze the contents of a directory, the user is spared the error prone tasks of dividing up the contents and creating the copy commands for those contents that are to run in parallel. By using parallelism in the copying of a directory, the operation can be expected to run much faster than a simple `cp` command that runs serially [13]. That will greatly help in the data management tasks involving large data sets, especially since those files often need to be unavailable to the user during the copy. Additionally, by programmatically managing the parallel copy operation, less errors are expected because the programs 1) divide the work load, 2) monitor the progress, and 3) report any problems. Each of those three tasks are subject to error if separately performed by a user.

1.1 Similar Tools

One tool with similar goals is SPDCP, which was developed by the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory [9]. On the multi-cluster Lustre implementation at OLCF, SPDCP demonstrated a speedup of 93

times the performance of `cp` [13]. One of the big advantages of SPDCP is its awareness of the LUSTRE system. That is also a major disadvantage. SPDCP only works for copying from one Lustre system to another Lustre system. The informational instructions on that tool's website states: "SPDCP should not be used to transfer data to/from non-Lustre filesystems. Transfers between non-Lustre filesystem will fail."

Another file system dependent copying tool is DistCp [1]. It is used with Hadoop Distributed File System (HDFS) to perform the inter/intra-cluster copying. But because of the MapReduce aspect of the implementation, the source must be HDFS. The documentation does not specify whether the destination must also be HDFS, but all the given examples apparently are. Others have proposed developing a parallel copy from non-HDFS systems to HDFS, but have not yet implemented that functionality [12].

There are several tools that use parallelism to enhance the copying of data between computers. There is a parallel version of `scp` called `pscp` [14], but it is used for moving one file to multiple locations. There are multiple parallel versions of `ftp`, but all those investigated seem to merely use multiple tcp connections, each connection copying a different piece of the same file.

Another tool, `newcp` [15], indicates parallelism, but actually increases throughput of a set of backgrounded `cp` commands by enforcing serialism to reduce disk head contention. This is accomplished by intercepting the requests within the `cp` command code. This tool is apparently not designed for distributed parallel file systems.

A set of tools named Parallel Storage Interface (PSI), developed internally at Los Alamos National Laboratory for their High Performance Storage System, includes local and remote parallel copy functionality [11]. However, no indication of general

release of the tool was found, though there is some indication that it can be licensed. One description indicates that only multithreading is used rather than multiple node parallelism [10]. Without multiple nodes, the bottleneck becomes the single network connection to the node running the multiple threads. Additionally, file aggregation is used to increase performance but makes the resulting data unreadable by standard file reading operations. The main parallelized operation is the simultaneous writing to multiple archival tapes.

An application named PFTOOL at Los Alamos National Laboratory has similar functionality to the proposed new utilities, but like PSI, it is oriented toward archival copies onto multiple tapes using multiple scripts, an SQL database, and 7000 lines of code [4] [5]. The tool was built for IBM's proprietary General Parallel File System (GPFS), which is a parallel distributed file system. PFTOOL seems to use the GPFS API as part of its implementation. However, mention is made for plans to accommodate other parallel distributed file systems too.

And a reference was found for a parallel copy script on a website for A Large Ion Collider Experiment (ALICE), a collaboration at CERN, the European Laboratory for Nuclear Research [7]. The script creates multiple processes to copy a large set of files in parallel, but there is no indication that the processes are on different cluster nodes. If that is the case, the limiting factor will be the network since all copying must pass through a single node and a single network connection. But apparently the script is aware enough that if the same command is simultaneously run on several nodes, each file will only be copied once. However, there seems to be additional awareness that makes the script only useful for certain data formats that are zipped.

1.2 Multithreaded Directory Traversal

Both proprietary and open source, distributed parallel files systems such as Lustre, Gluster, GPFS, and HDFS have file contents spread over multiple disks and computer nodes. The metadata is frequently on a single metadata server and sometimes replicated on multiple metadata servers. The separation of data from its metadata can potentially cause additional latency in finding and using needed information, especially if the servers used for metadata perform other file serving or compute functions. The metadata may not currently be cached if other demands on the server have used up available memory. Informal tests on the BSU GeneSIS cluster have shown that even on a single disk traditional file system, much of the directory traversal time is spent accessing the disk rather than processing the retrieved metadata. And the traversal time can be an order of magnitude faster on a second traversal because the pertinent metadata is then in cache.

There was an expectation of improved results if multiple threads were used to traverse the source directory. Each thread can be investigating a different subdirectory's contents. This will not guarantee that all the metadata nodes servicing the parallel file system are equally busy, but if metadata is available on more than one file server, the metadata requests may be spread out to more than one of them and thus have multiple requests being serviced at one time. Additionally, the independent threads can simultaneously process the retrieved data, generating the commands that perform the actual copying of file contents.

There are several aspects of the directory traversal part of the operation that were considered with the results potentially affecting the implementation of these new parallel copy commands. The first aspect relates to an optimum number of

threads to use. It was expected that the maximum number of cores available to the command line program would provide the best performance. However, overloading the metadata requests with even more threads than are available may prevent metadata servers from being idle when multiple threads happen to need data from the same server. If the subdirectories names are fed into a stack or queue, idle threads can pop off a name and make their requests relating to that subdirectory. While risking that threads interfere with each other, having many of them reduces the likelihood that an available metadata server is idle.

A second aspect of multithreaded directory traversal that was considered was the availability of metadata about which data server holds which physical files. If this metadata can be retrieved during the traversal process, additional algorithms can be developed to better load balance the file copy operations. By knowing the server affected by a particular file copy request, the load on each file server could be balanced better by splitting up the work evenly. Without that information, a random distribution of content has to be assumed and load balancing will be haphazard. There is a strong likelihood that such data is not readily available to a program. It may simply be part of the file systems internals. The scope of this project did not extend into making program changes to the parallel file systems themselves to additionally allow that information to be externally visible. But if the needed information is available, multiple queues or stacks could be created, one for each server, so as to balance the copy requests.

During implementation of the new directory copy commands, both design and coding allowed for alternate implementations that can take advantage of known file systems. The type of file system of both the source and the destination directories can be determined by reading the `/proc/mounts` pseudofile. The program design of

these new commands was written so as to allow for easy insertion of any discernible file system optimizations into the program flow.

But in the end, the multithreaded directory traversal that was implemented in both programs became a lesser issue in the overall project. Using multiple threads did speed up the gathering of information about the files needing to be copied, but overall, it was a relatively small amount of the overall time since both implementations are meant to copy very large amounts of data. Nonetheless, the user is given the ability to control the number of threads allocated to this part of the task. Presumably, the allocation will fit the particular architecture of the system on which the programs are deployed, mainly in the choice of threads matching the cores per node. If the utilities were to determine how many threads to use, it may not correctly decide when to maximize use of the command line staging node or when to leave capacity for other users.

1.3 Problems That Needed Resolution

An initial implementation used by Kameryn Williams for exploratory testing of the concept, used the `swarm` command to create PBS (Portable Batch System) jobs for the copying. This initial version used one job for each file copy. The downside of this approach becomes apparent when the directory copy involves millions of files. Even if many copies are combined, the multitude of jobs 1) creates overhead that may slow down the task and 2) fills the queue, preventing other users from access until completion. Additionally, the many jobs would need to be checked and monitored by the submitting program for successful completion of all copies. This large number of jobs would also be problematic on clusters that limit a user's number of concurrent

jobs. As described later, this restriction hampered the `Bpct` testing even though thousands of files were contained in each job.

Another problem that needed to be resolved is the checking for successful completion of each file copy. As a job finishes execution, the shell used to execute the job commands generally returns a status that is captured by the job scheduler. And the status can be used to make one job depend on the successful completion of another. However, shells like `csh` and `bash` both allow for the possibility of an auxiliary script being run at the close of the shell. For `bash`, the existence of a file named `.bash_logout` is checked. For `csh`, the check is for `.logout`. If the relevant file exists in the pertinent directory, the file is run as a script and can potentially replace or lose the status of the primary job script. Existing log out scripts might already contain the needed commands to preserve and propagate the status or be changed to do so, but making the assumption that a log out script does not exist or already has the needed code, may hamper the parallel copy command from properly managing the task. And if an existing logout script file is modified or deleted, the original wishes of the user concerning shell executions may be lost. To avoid this point of uncertainty, the error status of each copy is captured, accumulated, and displayed in the progress reports. To get individual error messages, users can view the output of the PBS jobs.

Although the job scheduler has some potential to help manage and load balance the parallel copying of files, there are some limitations that simple job scheduling is not able to flawlessly address. Some investigation was required to insert enough sophistication so as to produce a robust tool that can be made available to others.

CHAPTER 2

IMPLEMENTATION STRATEGIES

During the project proposal process, four possible implementation methods were identified. The two strategies that seemed most practical for copying multiple terabytes of data in a production environment were chosen for implementation. The two implemented methods are described in detail in Section 2.2 and in Section 2.3. Both have several 'convenience' features to improve their usability and aid in monitoring their progress.

2.1 Unimplemented Strategies

Of the two strategies that were not implemented, the first was that described earlier as a `swarm` implementation, with possibly more than one copy command per `swarm` job. This strategy was not chosen because of its lack of ability to easily control the number of jobs submitted and to monitor their progress. Likewise, there would be additional difficulties in monitoring and reporting the success of all the jobs or failures in copying individual files.

The second unimplemented strategy was to leave a program running on the command line that communicated with the allocated nodes in a job. The communication could have been through RMI for Java programs or UDP/TCP for C programs. Alternatively, files known to both the master program on the command line and those

in the job could be used for communication. This strategy would use dynamically distributed work like that implemented in `pct`. The persistent command line presence would have made this more like the `cp` command with extra progress reporting, but that was also a downside; maintaining the shell presence is required over very long copies. And there is no guarantee that the job containing the copying programs would be instantly run, so that may have created an even longer time needed to maintain the shell containing the job submission. Also the complexity of communication creates a weak point susceptible to failure.

2.2 Dynamically Distributed Work: `pct`

2.2.1 Overview

In this implemented method of copying files, a C program named `pct` submits a cluster job using the command line arguments provided by the user. The program was tested using PBS cluster schedulers, but the `qsub` command and its parameters work similarly for several other schedulers as well. The job script consists mostly of an `mpirun` command running the same `pct` program and uses several of the original command line options given by the user, plus `-x` to indicate that this is meant to actually perform the copying rather than submit a job. Below, the command line options are described more fully in Subsection 2.2.2. A single program handles 1) the gathering of command line information and submitting the job, 2) the copying of the files within a job, and 3) reporting the progress of a copying job. This multiple functionality was meant as a convenience in installing the program and minimizing information needed by the user to run the program and monitor progress.

Within the job, the master node, which is MPI rank zero, performs the directory traversal and generates the list of files to be copied. The master may also be described as the producer node since it generates the work needing to be performed. The user can control the number of threads created to perform the directory traversal when entering the parameters on the command line. If not specified, only one thread is created. For each file found during directory traversal, the source and destination directories, along with the file name, are sent out to the various consumer nodes, which perform the bulk of the copying work. A unique identifier for the destination directory and the size of the file are included in the outgoing work message so that the consumer can send that information back to the producer. The producer uses that returned information to preserve the attributes of the destination directories, if needed, and to report on the overall progress of the copying.

As the producer receives replies from consumers that files have been copied, it writes completion statistics to a file so the user can be informed of the progress. Potential update of the progress file is at the receipt of every consumer reply, but is currently limited to no more frequently than every 60 seconds so as to not incur much overhead from the progress updates. If the producer node has sent out work messages to all of the consumer nodes and has processed all currently available incoming replies from them, it will pick the next available file from the small files list and copy it rather than sit idle. If the small files list is exhausted, the smallest one from the large files list is extracted and copied. The large and small files lists are described below in Subsection 2.2.4, titled "Work Distribution and Management."

The situation could arise where several other nodes are waiting for more work while the producer is copying a file. This is most likely to occur when there are many other members in the MPI cohort. It is also more likely to occur as the messages get

shorter and the consumers are able to complete the work and reply quickly. Each idle consumer reduces the benefit of letting the producer perform copying. But if there are only a few other nodes, leaving the producer idle is a relatively significant waste of resources. So an algorithm was developed to determine the usefulness of letting the producer continue copying files. It compares the number of files copied during each lull between message handling bursts to the number of consumers waiting at the end. If the trend in that evaluation indicates so, the producer merely waits during such lulls rather than doing work. Once switched off, the producer does not resume its copying ability. The trend is tracked using an exponential moving average (EMA), which is frequently used to smooth noise in stock prices. EMAs give greatest weight to the most recent values. The values used within the trend analysis are calculated by dividing the number of files copied during an idle period by half the number of consumers waiting at the end of that period.

$$V_n = FilesCopied/(WaitingConsumers/2) \geq 1$$

The number of waiting consumers is halved assuming that each waiting consumer was waiting about half of the time of the last file copy. If the result is not less than one, then it was worthwhile to have the producer use idle time to copy files; the producer performed as much or more work as the waiting consumer could have during that idle period. The number of values used is the greater of ten or half the number of consumers: $P = max(10, (Consumers/2))$. The formula for calculating EMA is:

$$\begin{aligned} EMA_0 &= 1.0 \\ EMA_{n+1} &= EMA_n + ((V_{n+1} - EMA_n)(2/(P+1))) \end{aligned}$$

After all work has been sent out to the consumer nodes, the master node sends out an "end of work" message. Each consumer node acknowledges with an MPI reply message, waits at an `MPI_Barrier()` call, finalizes its MPI connectivity, and terminates. The master node acts similarly after receiving all of the end of work acknowledgments from the consumers.

2.2.2 Command Line Parameters

The GNU command line parser, called `argp`, was used within `pct`. `Argp` is a set of procedures and an interface used in many Unix and Linux utility programs. It also provides help information for the program when requested. Figure 2.2 shows the help that is displayed by `pct` when `--help` is included on the command line.

The `--threads` command line option indicates how many threads should be used by the master node to traverse the source directories, producing work to be performed. This information is only used by the master node and defaults to one if not provided.

The `--verbose` option can be used to tell the program to display additional information as it performs its work. When used during a job submission, the additional output mostly consists of a display of the parameters that were determined from parsing the command line. When used within a job, those parameters are also displayed, by the master node only, but other information indicating creation and termination of the various threads is also output.

Submission parameters for the cluster job are controlled with the `--queue` option. These would be the normal command line parameters for the `qsub` command. As an example, "`--queue=-j oe -l nodes=8:node,walltime=5:00:00`" might be used for a long copy using 8 nodes. Because of the spaces and embedded options within it, the option will probably need to be enclosed in quotes. The `--queue` options are used

in the job submission rather than in the job script file. For practical and security reasons, the option can not contain an ampersand or semicolon. The usually available `qsub` options `-z` and `-I`, which prevent return of the job name and cause interactive jobs, respectively, are removed if present.

The `--mpirun` option gathers user supplied arguments to be passed to the `mpirun` command within a submitted job. Typically, this would be something similar to "`--mpirun=-np 8`" but could also specify other `mpirun` arguments. This option also can not contain an ampersand or semicolon.

The `--cp` option controls details concerning the copying of individual files. When a file is copied by a work consumer node, the core utility command `cp` is used in a call to `execvp()`. However, some of its options i.e. `--recursive` are not relevant for copying a single file and will be removed if the user specifies them. Others such as `--parents` and `--target-directory` are relevant at some points of the program but not at the actual individual file copy, so they are noted and removed. And several other options are left for the file copy but also need to be captured so as to affect other aspects of `pct`. One such example is the `--preserve` option of `cp`. It is needed during the individual file copying but is also needed in the other parts of the program that manage target directory creation. Other useful options are those that control archiving of files or that allow updating only those files that have been modified since the last copy. This option also can not contain an ampersand or semicolon.

The `--progress` command line option shows the progress of all `pct` jobs submitted by the current user, both running and complete. Alternatively, a single job name can be specified so that only that job is displayed. Figure 2.3 shows an example of the output. If ongoing updates are desired, the Linux command `watch` can be used in combination with `pct --progress` to get periodic updates.

```
$ pct "--queue=-l walltime=4:00:00 -l select=8:ncpus=8:mpiprocs=2 -j oe"
        "--mpirun=-np 16" -t 4 "--pre=date;source startscript.sh"
        "--post=date;source stopscript.sh" srcdir1 srcdir2 destdir
```

generates the following script file:

```
#!/bin/sh
date;source startscript.sh
mpirun -np 16 pct -x -t 4 -f "/home/user/.pct/stat.4FE47D8D"
        "srcdir1/" "srcdir2/" "destdir/"
date;source stopscript.sh
```

Figure 2.1: This shows an example `pct` command and the job script that it generates and submits to the cluster's scheduler.

```
$ pct --help
Usage: pct [OPTION...] SOURCE... DEST
pct is a program that uses multiple cluster nodes to copy one or more
directory trees from SOURCE... to DEST

 -c, --cp="CPOPTS"          Options to be used for 'cp' commands
 -d, --delete[=JOBNAME | all]   Delete one or all past job summaries
 -m, --mpirun=MOPTS         Options passed to mpirun command
     --post="POSTCMD"       Cleanup commands after execution e.g. mpdallexit
     --pre="PRECMD"         Setup commands before execution, e.g. mpdboot
 -p, --progress[=JOBNAME | all]   Display progress of one or all pct jobs
 -q, --queue="QOPTS"        Options for qsub command, e.g. "-l nodes=8:node"
 -s, --save                 Save current set of options as the defaults
 -t, --threads=NUMTHREADS  Number of threads to use for directory traversal
 -v, --verbose              Display additional information in the output
 -?, --help                 Give this help list
     --usage                Give a short usage message
 -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or
optional for any corresponding short options.
```

Figure 2.2: Help text generated by `pct`.

```
$ pct --progress

Submitted 2012-10-12 09:52:00 job: 406482.fpbs
Updated   2012-10-12 10:22:36
  2460 files found(1.047 TB) Search Complete.
  2460 files copied(1.047 TB) Copy Complete.
  0 cp errors

Submitted 2012-10-18 15:18:41 job: 408393.fpbs
Updated   2012-10-18 15:25:32
  203819 files found(181.721 GB) Search Complete.
  2641 files copied(115.032 GB) Still copying.
  0 cp errors
```

Figure 2.3: Example showing the progress of two `pct` jobs. The first is already complete. The second is still running.

The `--delete` option is needed so that every `pct` job's status does not persist forever. If a specific job is specified, only that job's status is removed; otherwise all job statuses are deleted.

When a `pct` job is submitted, by default a file is created in which the progress statistics of that job are written. A unique file name is generated by the submitting `pct` program and is specified to the `pct` program in the PBS job using the `--statfile` option. This file is created in a directory named `.pct` within the user's home directory, as determined by the `$HOME` environmental variable. The `--progress` and `--delete` options look in this directory. If the user has a special reason for having the progress statistics written to a different file, it can be specified during the job submission and that file will be carried over into the running of the job and used for generating progress updates. However, `--progress` and `--delete` will only find such files if they are in the `.pct` directory. This option is not currently visible in the help descriptions but could be if alternative tools are desired for tracking progress.

The `--pre` option indicates a line to be added before the `mpirun` command in the PBS job script. A typical example may be "`--pre=mpiboot`" to start the MPI daemons on all the nodes in the job or "`--pre=source jobbegin.sh`" to run a more complex user script.

Similarly, `--post` specifies a command after the `mpirun pct` command in the job script. Unlike the `--cp`, `--mpirun` and `--queue` options, both `--pre` and `--post` can have semicolons so that several commands can be added to the script for execution. Alternatively, for a complex set of commands, a custom script could be created and specified instead.

A convenience option, `--save`, will cause several current command line options to be saved to a file in the same directory as the job progress files. These saved options are subsequently read each time `pct` is run and used as defaults. The options that are saved are: `--cp`, `--queue`, `--mpirun`, `--pre`, `--post`, and `--threads`. These options may have complexities that the user prefers but does not want to try to remember from one `pct` run to the next.

The `--help`, `--usage`, and `--version` options merely display information about the program in a format customary to Linux style core utilities.

2.2.3 Multithreaded Directory Traversal

When a directory is chosen during the traversal process, all of its entries are identified before moving on to another directory. Files and links are put into the linked list work stack for copying, and subdirectories are put into an array of directories yet to be traversed. The effect is most similar to a breadth first traversal since that directory array is used as a queue. If a pure breadth first traversal were desired for some

reason, the threads would have to coordinate more closely rather than simply add subdirectories to the queue as they are found.

However, the multithreadedness of the program can cause a less easily identifiable pattern of traversal. Each thread works independently, pushing subdirectories onto the same queue. Although the overall pattern of traversal will tend towards breadth first, there will be partial depth first access on many of the directory tree levels. Trying to direct the traversal into a depth or breadth first order was not a significant factor in program design. It was considered, but the randomness of the chosen method seemed slightly beneficial. Users may have organized their directories into a system that groups files in ways that keep large or small files together. Also, there may be an order to the file system metadata that preserves some of that order. Since simultaneous access to grouped files may overload a particular area of the file system, either actual storage or metadata, there may be some benefit to a randomness in the order of the access. The opposite could also be true, but since it is an unknown component of the files systems, the choice was for ease of programming as long as it was not suspected of being detrimental.

The target directories into which files are to be copied are created as the source directories are found and added to the directory queue. This ensures that they exist before any attempts are made to copy files to them. By creating them early in the process, the consumer nodes do not need to check for the directories' existence; queries of metadata are fast, but steps and thus contention can be reduced by preventing the need for a query before every file copy.

Since multiple threads generate work and add files to be copied to the work list and directories to be traversed to the directory array, mutexes (`pthread_mutex_t`) are used to control access and prevent race conditions.

2.2.4 Work Distribution and Management

Even files containing zero bytes require work to create a directory entry for the file and to update the metadata in the distributed file system. Strictly using only the accumulated sizes of the files to be copied would not properly represent the work to be done. A thousand one byte files is not the same amount of work as one file containing a thousand bytes. Therefore, the target size for an MPI message containing work for a consumer node is based on either number of files or bytes to be copied. And scenarios can be imagined where the last file needing to be copied is quite large, and all the other nodes are already done and waiting for that one last, large file. In such a scenario, the entirety of the job could be extended because of poor work balancing. To help prevent this, a second list of files is kept that only contains large files. Whenever a work message is being generated, one or more files is taken from this list of large files so that none are left for the very end of the job.

While `pct` is traversing source directories, found files are divided into large and small, with the distinguishing size currently set at 5MiB using the compiler command `#define LARGE_FILE_SIZE 5L*1024*1024`. This value could be changed or converted to a command line option. The small files are merely added to a LIFO linked list of files to be copied. But the large files are added to a large file linked list. This list is kept in sorted order, large to small. This allows the largest files to be sent out first to the consumer nodes, partially avoiding potentially unbalanced workloads if they otherwise would have happened to be sent out last. The algorithm to gather files to be sent to consumer nodes can be described thus: a total number of remaining files and total number of remaining bytes needing to be copied are calculated, divided by the number of nodes in the MPI team, and divided by four. These two numbers are targets

for the outgoing MPI messages. The targets decrease as work is completed, thereby helping reduce unbalanced work completion by the consumer nodes; the smaller and smaller messages containing work are less likely to end at greatly different times. To create the messages, the largest files are removed from the large file linked list first to quickly reach the target number of bytes to copy. If the target for total bytes is not reached because the large file list is exhausted or only contains files that greatly exceed the total bytes target, additional files are taken from the list of small files to help reach either of the two targets. And as files are added to or removed from the linked lists, the accumulated count and file size is adjusted so that list traversal is not needed to recalculate those values. There are values in `#define` commands that partially override the targets, helping reduce the total MPI communication required but perhaps causing partial unbalancing of work. These overrides could also be adjusted or made into command line options, but every cluster and perhaps every directory might have a different optimum set of parameters.

Maintaining a sorted linked list requires considerably more computation than an unsorted list. However, tests were run using a definition of a 'large' file ranging from 5MiB to 8GiB; the differences in time were smaller than the testing noise. A small number was chosen so that the producer node would mostly be choosing small files when it performs copying during its idle time. This reduces the chance that any consumers are sitting idle, waiting for work, while the producer is copying a file.

A minor downside was identified with using the large messages with large files that are sent out in the early part of the job: they take a long time to complete so the only reported progress at the beginning of a job is that made by the producer node using its idle time to copy small files. A user may be discouraged by the lack of reported progress. Remedies could be added to the work distribution algorithm to create a mix

of sizes in the early part of a job so that reported progress better represents actual progress. And as mentioned earlier, the code was designed to allow easy additions of other algorithms. But the program is meant for system administrators rather than general users that need extra comfort from feedback while doing their work.

2.2.5 MPI Messaging Strategies

Initial implementations used the standard `MPI_Send()` and `MPI_Recv()` calls in simple ways to pass messages containing work and the corresponding replies. After adding code to time the amount of waiting for MPI calls, more efficient methods were explored and tested. Some methods caused deadlocks, perhaps because of the MPI resources being fully used without enough additional available memory to move messages in the pipelines. This seemed to happen even though there were no program logic problems to cause deadlocks. It was also discovered that the standard `MPI_Init()` procedure did not initialize for thread safe MPI calls on the test platform. Segmentation faults within MPI code were encountered until the `MPI_Init_thread(0,0,MPI_THREAD_MULTIPLE,&var)` procedure was specifically used.

In attempting to streamline MPI communications, at the consumer nodes the non-blocking `MPI_Irecv()` was used to request a message before the files in the previous message are processed. This allows MPI to request the next message and move its data to the user work space while file copying is being performed. Obviously, two message buffers are needed, one for the message currently being processed and one for the incoming message. This reduced the idle time during tests on the BSU GeneSIS cluster.

Some complexity was also used to reduce idle time in the sending thread on the producer node. The non-blocking procedure `MPI_Issend()` was used to try to keep

two messages to each consumer node in the sent state. The idea is to always have messages built and on their way to the consumer nodes. Preventing any consumer node from idling while waiting for work was a high priority. To manage this process, two outgoing buffers are needed for every consumer node, including the master node which eventually also takes on a consumer role when it has completed all directory traversals. The `MPI_Testany()` and `MPI_Waitany()` procedures were used to monitor which messages are no longer in the send state. Mutex managed arrays were used to track and hold send buffers and replies that were not yet processed.

Unfortunately, the added complexity which worked on BSU's GeneSIS cluster did not work successfully on INL's clusters when access to it was eventually obtained. It sometimes acted as if messages were sent but never received. A few rewrites of the code were performed to add sophistication to thread handling and to separate send and receive threads, but all seemed to experience the same phenomenon of lost messages. Valgrind was used extensively to look for mismanaged memory. Different rewrites used different options for the `MPI_Init_thread()` calls. Various blocking and non-blocking sends and receives were tested. Various versions of Intel's MPI implementation with various Intel C compiler versions and MVAPICH2 implementations with GCC compiler versions were tested in case the flaw was within one of them. None of the combinations were consistently reliable. It is possible that similar thread implementation mistakes were made in all of the code rewrites, but the problems may also be related to INL's allocation of single cores causing thread related problems. The job scheduler allows individual cores on a node to be allocated independently, which pegs a process to a particular core. Perhaps the methods needed to do that affect multithreaded MPI operations. Personal experience with a widely distributed weather modeling program being run on an INL cluster also suffered from multiple threading

issues. That MPI program could not be compiled and run with multiple threads enabled. Support personnel at INL confirmed this problem with that program.

Other speculation about the problem relates to the creation of a separate process to perform the actual file copying. Perhaps the duplication of memory by the `fork()` procedure for the new process caused it to unknowingly intercept some MPI communications. Or perhaps the separate process caused inopportune interruptions to MPI communications. This speculation was driven by comments in the descriptions of many MPI procedures on a website [2] that state:

> "This routine is thread-safe. This means that this routine may be safely used by multiple threads without the need for any user-provided thread locks. However, the routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as malloc or other non-MPICH runtime routines that are themselves not interrupt-safe."

The documentation for Intel MPI did not contain such warnings, but the attempted multithreaded versions did not work with Intel MPI libraries either.

Eventually, a version of `pct` was written that still used multiple threading for the directory traversal, but all MPI messaging calls are postponed until directory traversal is complete, and they are performed by a single thread. The delay caused by this trial and error approach of code rewrites, debugging, and testing was substantial, especially since there were no indications in initial tests that a threading problem was likely. But even the simplified version required changes because optimal MPI procedures such as `MPI_Irecv()` (non-blocking receive) and `MPI_Rsend()` (ready send) seem to be implemented efficiently in the MPICH2 used on the GeneSIS cluster but not in the MVAPICH2 or Intel MPI on the INL clusters. This was surprising since

MVAPICH2 is based on MPICH2 [8]. These routines should allow the consumer to post a request to receive a message but perform copying while it waits for it. And the producer could send a message, knowing that the consumer has already posted a receive. The standard `MPI_Send`, `MPI_Recv`, and `MPI_Sendrecv` were eventually used because code used to accumulate times spent in MPI calls were consistently best with these procedures. The code in `pct` was correspondingly adapted to make best use of these procedures. The `MPI_Recv` gets called often within the producer node so that MPI resources are not locked by any waiting messages and new work is sent with `MPI_Send` before the received messages are processed. The `MPI_Sendrecv` used by consumers does not allow for overlapping messages but was optimized well enough on all three platforms to perform better than the MPI calls that do.

The possibility still exists that some solution or workaround could be found to allow more multithreading. Such a solution may slightly improve performance by allowing files to be sent out to the consumer nodes and file copying to start before directory traversal is complete. However, directory traversal using several threads is often a fairly minor portion of the run time, so the small speed benefits would not otherwise outweigh the current benefit of having a robust utility that works well on production clusters with varying MPI libraries. Timing checks that were added throughout the code identified no areas other than MPI communications as needing optimization. Some example timings for consumer processes using the final MPI strategy are included and discussed within Subsection 3.3.1 under the title "Evaluating MPI Implementation and Work Load Balancing in `pct`."

2.3 Batched Work Distribution: `Bpct`

2.3.1 Overview

In this method of copying files, a Java program traverses the source directories, finding all the files needing to be copied. The number of threads performing the traversal can be specified on the command line. Then the files are distributed among a user specified number of PBS jobs that perform the actual copying. This is a 'batch' of jobs. The jobs are independent of each other so they may or may not run concurrently. The jobs that are created make use of a Bash function, defined within each job, to help reduce the size of the job script and to manage updates of the progress of the copying. The function receives the source and destination file paths and the percentage of completion of that job that this file represents. If the time since the last update written to the job's progress file is more than or equal to 60 seconds, an update is written. `Bpct` requires Java version 1.5. Originally it used a little version 1.6 code to make the job scripts have the executable attribute, but it was not necessary and easily removed.

Each batch of jobs gets its own subdirectory in the `$HOME/.bpct/` directory. The subdirectory's name is encoded in hexadecimal with the time it was created so as to allow easy sorting. The individual job scripts, the individual files for the progress of each job, and a file containing the PBS job name all reside within a batch's subdirectory.

2.3.2 Command Line Parameters

A custom command line parser was created to accept the arguments specified by the user. The options are very similar to those allowed by `pct`. A notable exception is

```
$ java Bpct --help
Usage: java Bpct [OPTION]... SOURCE... DESTINATION
      --del BATCH        Delete info for batch # BATCH; "all" for all info.
      --cp "opts"        Options to use for cp command, in quotes.
  -j, --jobs             total # jobs created to perform copying.
  -p, --progress         Print progress of copying jobs.
      --pre "commands"   Commands at start of each job.
      --post "commands"  Commands at end of each job.
  -q, --queue "opts"     Options to use for submitting jobs, in quotes.
      --save             Save current options as default for current user.
  -t, --threads          # threads used for identifying files to be copied.
  -v, --verbose          Display additional information about activities.
      --version          Display version information and exit.
  -h, --help             Display this help and exit.
```

Figure 2.4: Help text generated by `Bpct`.

that MPI is not used within this implementation, so no `mpirun` options are needed. However, there is an addition of a `--jobs` option to specify the number of jobs in a batch into which the copying is divided. As with `pct`, it also provides help and usage information for the program when requested. Figure 2.4 shows the help that is displayed by `Bpct` when `--help` is included on the command line. The same restriction against ampersands and semicolons apply to the `--queue` and `--cp` options as were mentioned for `pct`.

Similar to `pct`, `Bpct` uses a file `$HOME/.bpct/default.txt` to store user specified defaults for the `--threads`, `--jobs`, `--queue`, and `--cp` options.

Since each job within a batch is independent of the others, the progress of each is tracked and reported separately. Figure 2.5 shows an example of such reporting.

```
java Bpct --progress
Progress of bpct jobs:

Batch #1: Sun Oct 14 11:22:51 MDT 2012
Source:  /home/nusskevi/panfs/n975
Destination: /home/nusskevi/panfs/d975
406874.fpbs
COMPLETE! job 1; Errors: 0; Files: 617; Bytes: 261627194155
        Sun Oct 14 11:59:59 MDT 2012
406875.fpbs
COMPLETE! job 2; Errors: 0; Files: 617; Bytes: 261611826055
        Sun Oct 14 12:00:00 MDT 2012
406876.fpbs
COMPLETE! job 3; Errors: 0; Files: 617; Bytes: 261284014127
        Sun Oct 14 11:59:53 MDT 2012
406877.fpbs
COMPLETE! job 4; Errors: 0; Files: 609; Bytes: 262038143619
        Sun Oct 14 12:00:11 MDT 2012

Batch #2: Thu Oct 18 18:23:24 MDT 2012
Source:  /home/nusskevi/panfs/nschmidt/nullomer_prediction
Destination: /home/nusskevi/panfs/delThis
408465.fpbs
job 1; Errors: 0; Files: 610 of 617 99%; Bytes: 207261470171
        of 261627194155 79% Thu Oct 18 18:52:04 MDT 2012
408466.fpbs
job 2; Errors: 0; Files: 85 of 613 14%; Bytes: 220693656488
        of 261653394823 84% Thu Oct 18 18:52:34 MDT 2012
```

Figure 2.5: Example showing the progress of two Bpct batches. The first had four jobs and is already complete. The second is in progress running two jobs.

2.3.3 Object Oriented, Java Implementation

`Bpct` is implemented in a way to take advantage of the rich class library provided within the Java Development Kit. Rather than implement custom thread handling code, the `ThreadPoolExecutor` class is used to manage the threads used for directory traversal. Each source directory from the command line and each found subdirectory is simply added to the `ThreadPoolExecutor` as a `Runnable` task. As files are found, they are added to a custom class, `BatchSizeBalanced`, which stores the file information until traversal is complete. It too is a `Runnable` task that gets added to the `ThreadPoolExecutor` as a task that generates a balanced set of file lists for the PBS jobs. The overall design allows other strategies of generating jobs, either for PBS or another job scheduler. To do so, design patterns are used for future flexibility. As one example, the `BatchSizeBalanced` class is an example of the "Strategy" design pattern. If other strategies are developed, command line options could be used to choose them. To aid in that choosing, the class that handles parsing of the command line options has two "Factory Methods," another design pattern, so that the objects used in the program reflect what the user specified. One factory method currently provides only a `BatchSizeBalanced` object and the other currently only provides a custom `PbsJobs` object for generating and submitting jobs to the PBS scheduler. So the strategy for distributing the files among those jobs and the strategy containing details for creating the jobs could be differently implemented as new subclasses of `JobScheduler` and of `BatchTask`, which are the superclasses of `PbsJobs` and `BatchSizeBalanced`, respectively. Having the factory methods be part of the options parser removes the need for the options to be queried and then the proper object being instantiated. The option parser class, `BpctOptions`, already

contains the the needed information and therefore makes the decision. This is a form of "dependency injection" and "inversion of control."

A `Vector`, which is inherently synchronized, is used to accumulate information about files needing to be copied. The only explicitly synchronized methods needed within the program relate to creating sequential job numbers. And this would only be needed if a new strategy were implemented that created jobs before the directory traversal is complete. The program design allows that a replacement for the class `BatchSizeBalanced` could decide to submit jobs before all the files are identified.

Since the found directories are added to the `ThreadPoolExecutor`, which uses a queue to manage its tasks, the traversal most closely resembles a breadth first pattern, similar to that described for `pct`.

While files are being added to job scripts, a `HashMap` object is used to record which directories have already been created for that job. This partially reduces the number of unneeded `mkdir` commands required within a job script. The implementation is simplistic and could be modified to further reduce the needed `mkdir` commands by tracking created directories by portions of their paths rather than the entire path. For example, if `/home/user/x/y` has already been created, `/home/user/x` does not need to be created before copying files into it. The current implementation may try to unnecessarily create `/home/user/x`, depending on the order of files in a particular job. Additionally, a sophisticated work distribution class could divide up files based on their containing directories. However, the small performance gain of reduced directory creation would probably be lost in the preprocessing and division of the file information.

2.3.4 Work Distribution

The `BatchSizeBalanced` class, which is currently the only implemented strategy for dividing up the work, tries to balance both the number of files and the number of bytes copied by each job in a batch. To do this, all the files are first sorted by size, large to small, using the `Arrays.sort()` method, and the file sizes are summed. For each job to be created, the sorted array is traversed. A file is assigned to the job's list of files if it has not already been assigned and number of files and bytes assigned to that job is less than the relative proportion of all the files and bytes for that position in the sorted list of remaining files. In other words, if there are to be eight more jobs created and the iterator is part way through the list, the current job should have 1/8th of the bytes summed so far and 1/8th of the files counted so far. If not, the file is added to the current job. The last job gets all remaining files.

Initially, the files that were assigned to a job were added to the job script in small to large order. The thinking behind this is that processing all the small files first would have the slight advantage of their metadata still being in cache from the traversal portion of the program if the time until they are copied is short. However, that strategy was changed because that caused all the heavy metadata access to happen simultaneously across all the jobs, potentially creating a bottleneck. Instead, when moving files from a job's list of files, sorted by size, to the batch file, the starting point in that list is evenly staggered. For example, if there are 8 jobs in a batch, the first job starts 1/8th of the way into its list, the second 2/8ths, etc. This helps mix the sizes being simultaneously copied across the jobs in a batch.

Two other options may be worth consideration for distributing files among jobs. One would create a new job whenever directory traversal found a predetermined

number of files to be copied. When the limit is found, a job would be immediately created and submitted. The second option would do the same but use the accumulated size of the files rather than the count.

CHAPTER 3

TESTING AND RESULTS

3.1 Clusters

3.1.1 BSU GeneSIS Cluster

The GeneSIS cluster at BSU has 17 nodes, each with a quad core, 2.67GHz, Intel core i7-920 processor. One node serves as a staging node for logging in, compiling, submitting and monitoring jobs, and for postprocessing data. Eight nodes serve as general purpose compute nodes. Eight others service a Lustre parallel distributed file system. These each have 8 1TB drives. Using a combination of RAID 6 and RAID 10, this provides 36TB of storage. This system was initially used for concept development and testing. Final performance testing was also done on this cluster to see whether the implemented software had different characteristics on Lustre than the highly used INL clusters that use the PanFS file system. GeneSIS's small and limited number of users was quite useful in running controlled tests. Also, because of that limited use, special techniques were developed to make comparison runs start from similar conditions. However, the file system is cross mounted on another cluster, so there is still an element of unknown disk usage between and during performance tests.

Since the GeneSIS cluster often has periods without use, especially in the short

periods between timing runs, there is the potential for file system data and metadata to remain in the IO servers' cached memory. This is not representative of a production environment where heavy use by many users compete for cache space. Such a heavily used production environment is expected to be the most likely audience for the tools developed in this project. To help remove data from cache, a utility program was written to simply allocate all the memory on a file server. It was named `usemem`. This C program merely queries the operating system for the size of the memory pages and the number of physical pages on that node. Then memory allocation is requested for the number and sizes of those pages. Each allocated page is also written to so as to mark the pages as used. Using the physical memory should cause the file system caches to release its data. For convenience, a small script was written to `ssh` to each IO server and run `usemem`.

On the GeneSIS cluster, a set of files in a directory was identified that seemed like a reasonable example for performance tests. A utility was written so the files could be recreated if they were deleted and also so they could be created on the INL cluster without the overhead of an extended copy. First, the directory traversal program was modified to output the directory, file name, and size of all files it found. Another Java program named `Dupfiles` was written to read that output and recreate the directory structure and files. The data in the recreated files is merely zeros, but the copying utilities do not concern themselves with details of what is in a file while it copies it.

The relevant, standard software products on GeneSIS used for this project are MPICH2 version 1.1.1, GCC version 4.1.2, and Oracle Java version 1.6.0.

3.1.2 INL's Fission and Icestorm Clusters

The Fission cluster at INL consists of 391 compute nodes and 2 service nodes. The compute nodes have 64 GB of RAM and four processors. Each 2.4 GHz AMD Opteron processor has 8 cores, thus 32 cores per node. Overall, there are 12,512 cores available for jobs. The staging nodes are similar but only have half the cpus and half the memory. The service nodes are used for compiling, data manipulation, and other staging activities related to submitting jobs on the compute nodes. The interconnect is QDR InfiniBand. PBS Pro is used as the job scheduler. There are several queues used to establish job priorities and quotas, but they were generally not a factor in this project.

The Icestorm cluster at INL also uses 2 service nodes for staging. The service and compute nodes each have 2 quad core 2.66 GHz Intel Xeon processors, thus 8 cores per node. There are 256 compute nodes providing a total of 2048 cores for jobs. There is 16 GB of RAM per node and the interconnect is DDR 4X InfiniBand.

The PanFS parallel distributed file system, a product of Panasas, Inc., is mounted on both clusters. The system consists of 98TB of raw storage, with two 1TB SATA drives on each of the 49 storage blades, which reside on 5 shelves. Each shelf has a 10GB/s Ethernet connection. The system also contains 6 director blades that are used for mounting NFS, distribution of the metadata, and overall management of the Panasas realm [6]. Users are encouraged to use the PanFS file system for temporary storage of large data sets that result from jobs, but there are other traditional file systems for users' home directories and for special projects. These contain 500TB of raw storage and consist of two servers, each consisting of a shelf of 24 600GB SAS drives and 4 shelves containing 96 1TB SATA drives. Additionally, a third server

containing 16 shelves holding 384 1TB SATA drives serves as backup storage of the other two servers' data. There is a 10GB/s Ethernet connection to each of the three servers.

Access to software on INL's clusters is managed through modules, which expedite the setting of establish environmental variables controlling paths to various software tools and their libraries. Here are the pertinent available compiler and MPI modules:

- Compilers
  - Intel versions 9.1, 10.0, 10.1, 11.0.069, 11.0.074, 11.1.046, 11.1.073, 12.1.0, 12.1.1, 13.0
  - PGI versions 7.1, 7.1-3, 7.1-6, 7.2, 7.2-1, 8.0-1, 10.2, 11.9, 12.4
  - GCC versions 4.6.0, 4.6.1
- MPI Implementations
  - Intel versions 3.1.026, 3.1.038, 3.2.0.011, 4.0.0.028
  - MVAPICH2 (compiler/version) GCC/1.0.2, GCC/1.5.1p1, GCC/1.6, Intel/1.0.2, Intel-1.4, Intel/1.5.1p1, Intel/1.6, PGI-1.4
  - OpenMPI (compiler/version) GCC/1.2.6, GCC/1.4.3, GCC-4.6.0/1.4.3, Intel/1.2.6, Intel/1.4.3

The Icestorm cluster has an IBM Java compiler and runtime, version 1.5.0. The Fission cluster has an Oracle Java compiler and runtime, version 1.6.0.

INL does maintain other clusters too, but they were not used or even considered for testing in this project.

3.2 Characteristics of the Test Data

As mentioned, test data was taken from production directories on BSU's GeneSIS cluster. A single directory provided the data and was chosen for both its overall quantity of data and its large number of files. Three subsets of that directory were used for timing tests. The first subset is called "Many Small Files" because it has the majority of the files but a relatively small portion of the overall content. This subset, whose file size distribution is given in Table 3.1, is meant to represent a heavy burden on the metadata and directory entry creation portion of the file system. The "Many Small Files" set contains 203,819 files in 14,869 directories and represents 181.7GB of data. For exercising the throughput of data, a second subset of files, called "Few Large Files," contains the bulk of the data but has a relatively small number of files. It too was taken from the original directory, but it was doubled to create an even larger set for copying and to increase the total number of files. When using many nodes for the copying, a large enough task needs to be created, thus the doubling of size. Additionally, when distributing the work over many cores and nodes, doubling the number of files and creating a larger task helps keep the load balanced. Otherwise, the few biggest files get assigned to a few nodes and there is simply not enough work for the others. The files size distribution for the "Few Large Files" set is given in Table 3.2. It contains a total of 1.0TB in 2460 files, in 161 directories. The "Few Large Files" set is too large for testing on the GeneSIS cluster, so a subset of the "Few Large Files" set was made and named "Partial Large Files." It contains 432 files in 22 directories and contains 174.9GB of data. The distribution of its file sizes is given in Table 3.3. Only one group of tests was performed with this set.

Table 3.1: Distribution of File Sizes in the “Many Small Files” Set

File Sizes	0 - 1KB	1KB - 10KB	10KB - 100KB	100KB - 1MB	1MB - 10MB	10MB - 100MB	100MB - 140MB
Count	144,703	22,985	15,109	7,224	10,502	2,814	482

Table 3.2: Distribution of File Sizes in the “Few Large Files” Set

File Sizes	0 - 1KB	1KB - 540KB	540KB - 25MB	25MB - 100MB	100MB - 1GB	1GB - 8GB	10GB - 30GB
Count	1076	674	0	46	294	364	6

3.3 Performance Tests

Many timing tests were performed to gauge the performance and usefulness of the implemented copying tools. The tests often show a duplication of the general pattern of results but are included as a way to bolster conclusions based on tests containing unquantifiable components. Since the production clusters on which the tests were performed are variably busy with unknown other applications, the effect of those other applications on access to the PanFS parallel distributed file system is unknown. Even with the Lustre file system on the GeneSIS cluster, which is often otherwise unused for lengths of time, the file system is cross mounted on another, busier cluster, so some unknowns remain. By including results from multiple performance tests on different clusters and file systems, the general pattern of performance improvement can be shown.

Although the basic algorithms for distributing work among participating processes in a test was well thought out and remains essentially intact, tests made using first draft implementations of those algorithms helped establish patterns used to make tests on final versions more efficient. Early tests indicated that using all the cores on a node is often counterproductive. And some times of day should be avoided to prevent anomalous values in the data. The tests presented here still show some

Table 3.3: Distribution of File Sizes in the "Partial Large Files" Set

File Sizes	0 - 1KB	1KB - 540KB	540KB - 29MB	29MB - 60MB	390MB - 1GB	1GB - 2.25GB
Count	235	97	0	10	10	80

testing noise from variably busy clusters, but were done in cohesive groups of tests, avoiding times when cluster activity varies most.

The `Bpct` utility performs its directory traversal before submitting jobs and does so on the command line node, which is always a staging node. To avoid using all the cores on a shared resource, only four threads were used to perform directory traversal. And simply for consistency, the same number of threads were chosen for the `pct` tests, even though they occur on exclusively assigned compute nodes.

3.3.1 Tests Using `pct`

On the INL clusters, resources can be allocated one core at a time. On the BSU GeneSIS cluster, an entire node of four cores is allocated but not all cores need to be used. This flexibility allowed for testing both the number of nodes used and the number of cores per node used. The hope was to possibly distinguish between the capacity of the file system being exhausted versus the capacity of individual nodes. To help alleviate additional unknowns, when cores were allocated on any cluster, an entire node was requested but not all the cores were always used. As evident in the following data, this sometimes wastes resources, but it removes the possibility that a node is shared with some unknown application using the memory, processors, network, and file system in unknown ways. Also, allocating half the cores for a node for each of multiple `pct` processes, creates the possibility that they are assigned to the same nodes, or worse, only sometimes being assigned to the same nodes.

Table 3.4: Example Timings for `pct` Copying "Many Small Files" Set

Consumer Process	Fission: 20 nodes 6 cores per node Traversal time=54.0 sec Run time=814 sec			Icestorm: 20 nodes 8 cores per node Traversal time=91.2 sec Run time=653 sec		
	Min	Max	Mean	Min	Max	Mean
Wait for work	54.8	66.1	58.1	91.6	96.4	92.6
Wait for work w/o traversal	0.8	12.1	4.2	0.3	5.2	1.3
Wait at barrier	0.0	5.7	2.4	0.1	10.1	4.6
Files copied	1351	2027	1700	1098	1527	1277
MB copied	1151	2188	1526	608	1264	1143

Table 3.5: Example Timings for `pct` Copying "Few Large Files" Set

Consumer Process	Fission: 4 nodes 8 cores per node Traversal time=1.7 sec Run time=2175 sec			Icestorm: 4 nodes 8 cores per node Traversal time=9.4 sec Run time=2354 sec		
	Min	Max	Mean	Min	Max	Mean
Wait for work	3.8	117.2	77.3	26.8	102.3	78.5
Wait for work w/o traversal	2.1	115.5	75.6	17.4	92.9	69.1
Wait at barrier	0.0	72.6	47.0	0.0	14.1	11.5
Files copied	3	70	36	4	70	34
MB copied	30,035	35,419	32,827	31,493	36,054	32,770

Evaluating MPI Implementation and Work Load Balancing in `pct`

Code was added to `pct` to aid in identifying performance problems. This was used to evaluate the chosen set of MPI communication options and to quantify the quality of the even distribution of work. Representative examples of those outputs for consumer processes are given in Tables 3.4 and 3.5. The first table shows some statistics for copying the "Many Small Files" data set on the two INL clusters. The "Wait for work" row indicates the time consumers spent sending a "waiting for work" message to the producer and waiting for a reply using the `MPI_Sendrecv()` procedure. That time was 7% on Fission and 14% on Icestorm. But most of that time was the first

request for work during the directory traversal phase. If traversal time is subtracted, as given in the "Wait for work w/o traversal" row, those number are reduced to less than 1%. This indicates that the final choice for MPI implementation resulted in good performance, but also shows how a solution, such as multithreading combined with MPI, could improve overall performance by allowing the commencement of copying before directory traversal is complete.

The first table, Table 3.4, also shows data related to work load balancing. The time waiting for the last consumer to finish is quite small as given in the row labeled "Wait at barrier." This time is also less than 1% for both clusters. The variability in the number of files and number of bytes copied is merely an artifact of the distribution of file sizes in the data set; the implemented work load algorithm seems to compensate well by giving different consumers different numbers and sizes of files.

In the second table, Table 3.5, example timing values are given for copying the "Few Large Files" data set on both INL clusters, using 32 cores on each. Again, there is a large variability in the number of files copied by each core. Since the total number of bytes is the predominant factor for this data set, balancing the work caused less variability in this number. However, the work load balancing was less effective for this data set. The "Wait for work" percentages are smaller, 3.5% and 3.3%, but are only reduced marginally after the directory traversal time is subtracted. The "Wait at barrier" time is also a little larger. With the fewer and larger files, the work load balancing was a little less effective because the producer is copying larger files during its idle time rather than sending work to consumers. And larger files also make it more difficult to get consumers to end at the same time. But overall, the numbers are good except for the need to start distributing work before traversal of very large directories is complete.

Table 3.6: Time/Speedup for pct on GeneSIS Copying "Partial Large Files" Set

Cores/Node	1 Node	2 Nodes	4 Nodes	6 Nodes	8 nodes
1	7150 / 0.98	3839 / 1.83	3447 / 2.04	2335 / 3.01	1735 / 4.06
2	4321 / 1.63	2890 / 2.43	2018 / 3.49	1521 / 4.63	1912 / 3.68
3	3981 / 1.77	2424 / 2.90	1609 / 4.37	1304 / 5.40	1630 / 4.32
4	3737 / 1.88	2714 / 2.59	1522 / 4.62	956 / 7.36	1302 / 5.40

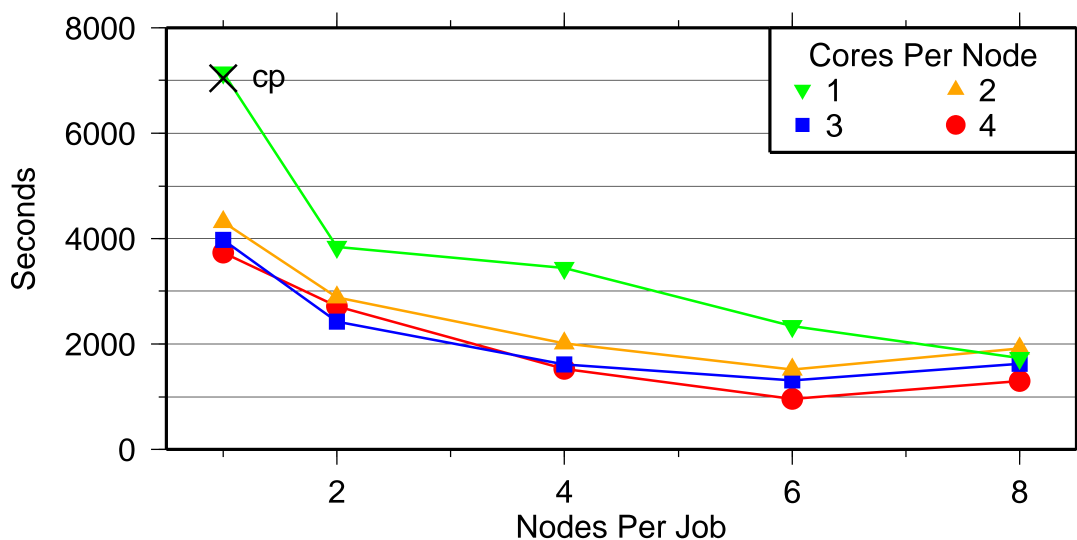

Figure 3.1: pct on GeneSIS Copying "Partial Large Files" Set: Graphed by Nodes

Copying "Partial Large Files" Set on GeneSIS with pct

Table 3.6 lists the times for a group of tests run on the GeneSIS cluster using the previously described set of data called "Partial Large Files." Many of these files are large and are meant to test the data throughput capabilities. Figure 3.1 graphs those results showing different number of cores per node on a varying number of nodes. The time for cp to copy the same set of files was 7037 seconds. Using a single core on a single node, pct is able to attain a similar time. And by using additional cores and nodes, it attains a 7.36 speedup. Interestingly, this does not occur when all the nodes are used. Instead, when multiple cores are used on 8 nodes, the run times increase

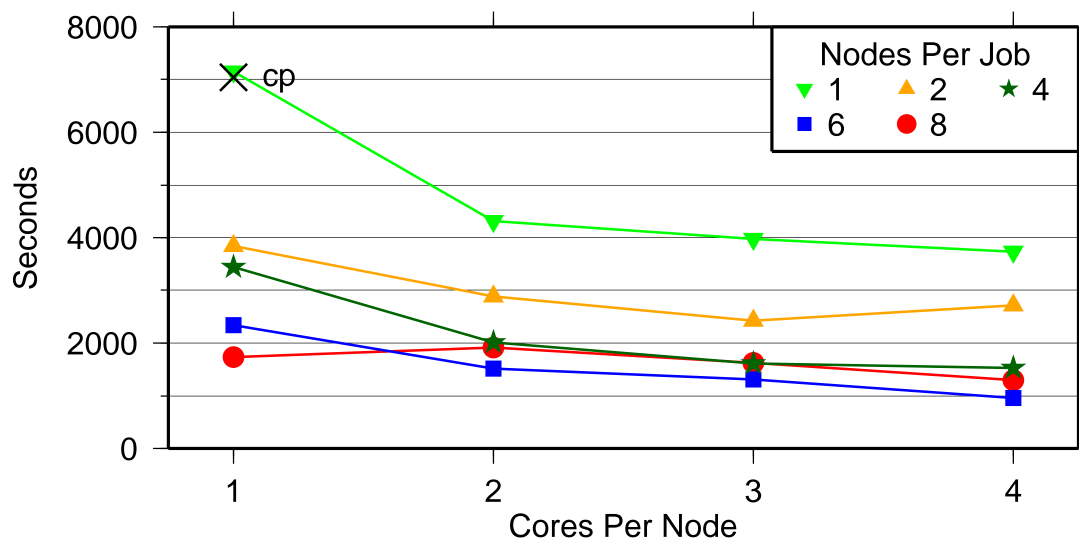

Figure 3.2: `pct` on GeneSIS Copying "Partial Large Files" Set: Cores Per Node

compared to those on 6 nodes. This may indicate that the throughput of the Lustre file system was maximized with multiple cores on 6 nodes and that the extra nodes are creating competition that slows down the copying. But if only one core per node is used, the 8th node is a benefit.

Figure 3.2 graphs the same data but with cores per node on the abscissa. It too indicates that the 8th node is generally not an improvement over the 6th, but also shows that using the 3rd and 4th cores per node are of less benefit than adding the 2nd core. But since even the 4th core is generally an improvement, the data throughput per node does not seem to have been reached.

Copying "Many Small Files" Set on GeneSIS with `pct`

This group of tests shows `pct` being used to copy the "Many Small Files" set of data on GeneSIS. This set of files, which contains many files but a relatively small amount of data, is meant as a test of the file system's ability to create many file entries in

Table 3.7: Time/Speedup for `pct` on GeneSIS Copying "Many Small Files" Set

Cores/Node	1 Node	2 Nodes	4 Nodes	6 Nodes	8 nodes
1	7647 / 1.13	5306 / 1.63	3304 / 2.61	2725 / 3.17	2423 / 3.56
2	6341 / 1.36	3366 / 2.56	2072 / 4.16	1964 / 4.39	1819 / 4.74
3	4163 / 2.07	2813 / 3.07	1776 / 4.86	1542 / 5.60	1640 / 5.26
4	3303 / 2.61	2202 / 3.92	2014 / 4.28	1445 / 5.97	1418 / 6.08

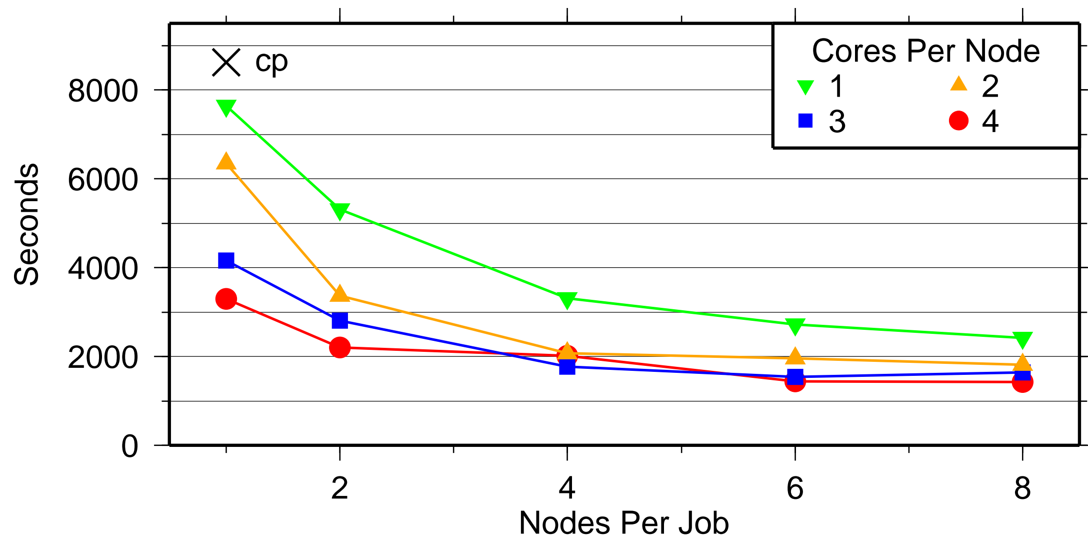

Figure 3.3: `pct` on GeneSIS Copying "Many Small Files" Set: Graphed by Nodes

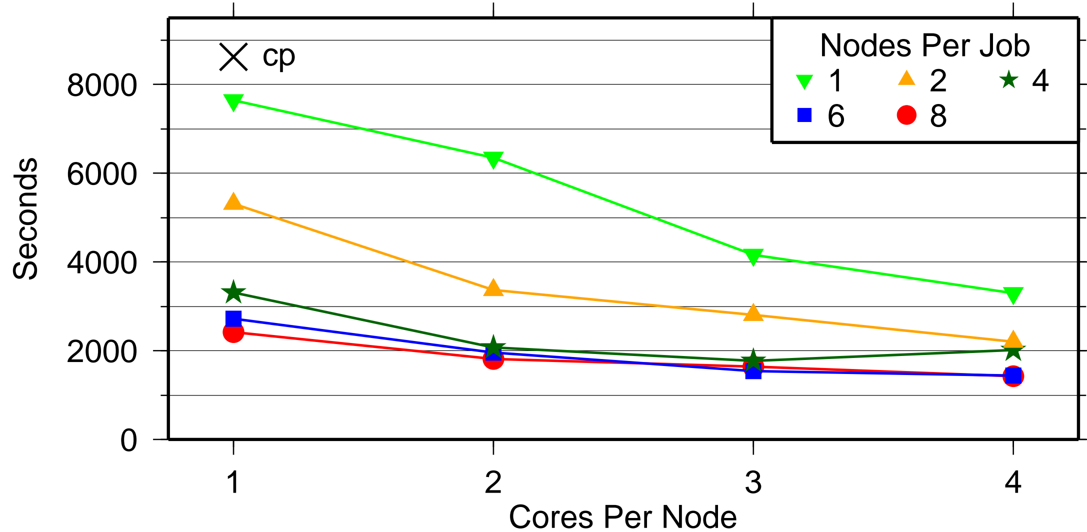

Figure 3.4: `pct` on GeneSIS Copying "Many Small Files" Set: Cores Per Node

its metadata tables. The time values and speedups are listed in Table 3.7. The graph in Figure 3.3 shows the substantial speedups of up to 6.08, with the biggest improvements occurring with the addition of the first few cores and/or nodes. The flatness of the improvements with the near maximum cores and nodes may be an indication that the metadata throughput may be close to reaching its maximum.

Surprisingly, the time of 8628 seconds for using `cp` on a single node and core, is greater than any of the `pct` times. This may be due to the nearly full state of the file system. When preliminary tests were run to find the maximum amount of space available to include in the "Partial Large Files" set, a run of `pct` on a single core and a single node could sometimes complete the copy, but `cp` could not and gave reports of insufficient file space. The difference in that case may have simply been due to a different ordering of the files to be copied. But that situation may be an indication that the near fullness of the file system affects the copying. As seen later in other test groups, `cp` generally performed better than `pct` and `Bpct` when many files were

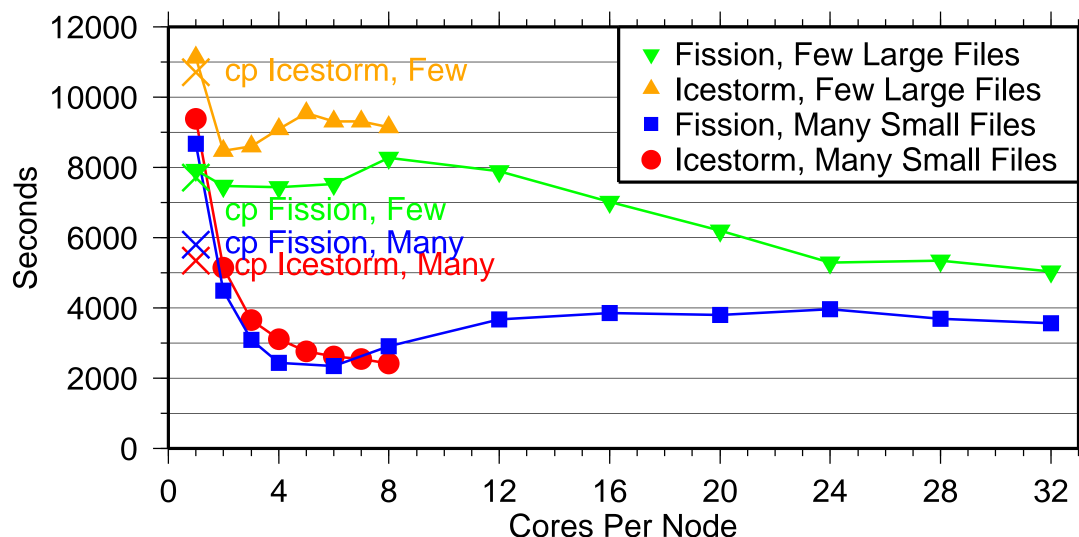

Figure 3.5: `pct` on Icestorm and Fission Clusters Copying "Few Large Files" and "Many Small Files" Sets Using a Single Node

involved and only a single core was used.

The same data from Table 3.7 is graphed differently in Figure 3.4 but merely supports that there is a general speedup gain when using more cores and nodes, but at a decreasing rate of improvement.

Single Node Tests on INL Clusters with `pct`

The GeneSIS cluster has relatively few nodes and cores per node. The tests covering the full range of combinations of nodes and cores per node becomes impractical on INL's clusters. Asides from simply allocating sufficient time for the tests, the length of time needed for such tests would certainly include significant changes in use by other users. That would have created additional uncertainties about the validity of those tests; unreasonable testing noise would get included.

Instead, the tests are split up to find the capacity of an individual node copying

a set of files and using those results to inform tests that investigate the cluster's capacity to copy those sets of files. Figure 3.5 graphs the results of single nodes on the Icestorm and Fission clusters when copying the "Many Small Files" and "Few Large Files" sets. On Icestorm, performance improves as more and more cores are used to copy the "Many Small Files" set, up through the maximum of 8 cores per node. And peak performance is found with only 2 cores when copying the "Few Large Files" set. However on Fission, the best performance is seen with about 6 cores per node for copying the "Many Small Files" set. Using more cores only seems to create a detrimental competition between them. This effect was also seen in other groups of tests before the final work distribution algorithms were implemented, so it is not likely to be caused by variability in cluster use. The shape of the single node performance for Fission copying the "Few Large Files" set is odd in that it has a local minimum of 4 cores per node but goes on to increase throughput through 32 cores per node. unless this is related to allocating and pegging individual processes to individual cores, no other explanation of the curve's shape is apparent.

Because of the overhead of handling each of the 203,819 files in the "Many Small Files" set individually, the `cp` command outperformed `pct` running on a single core on a single node. However, simply adding a second core made up that difference. With the "Few Large Files" tests, `cp` had a much smaller advantage because of the few number of files.

Copying "Many Small Files" Set on Fission with `pct`

Table 3.8 shows the results of copying the "Many Small Files" set using `pct` on the Fission cluster. Since the single node tests in Figure 3.5 indicate that 6 cores per node is optimum for single node throughput, that is used as the primary configuration. To

Table 3.8: Time/Speedup for `pct` on Fission Copying "Many Small Files" Set

Nodes	1 Core Per Node	6 Cores Per Node	32 Cores Per Node
1	8519 / 0.67	2341 / 2.45	3530 / 1.63
2	4520 / 1.27	1289 / 4.46	1888 / 3.04
4	2082 / 2.76	851 / 6.75	980 / 5.86
6	1464 / 3.93	1098 / 5.23	659 / 8.72
8	1325 / 4.34	799 / 7.19	1079 / 5.33
12	1014 / 5.67	1112 / 5.17	1323 / 4.34
16	971 / 5.92	708 / 8.12	1002 / 5.74
20	921 / 6.24	814 / 7.06	811 / 7.09

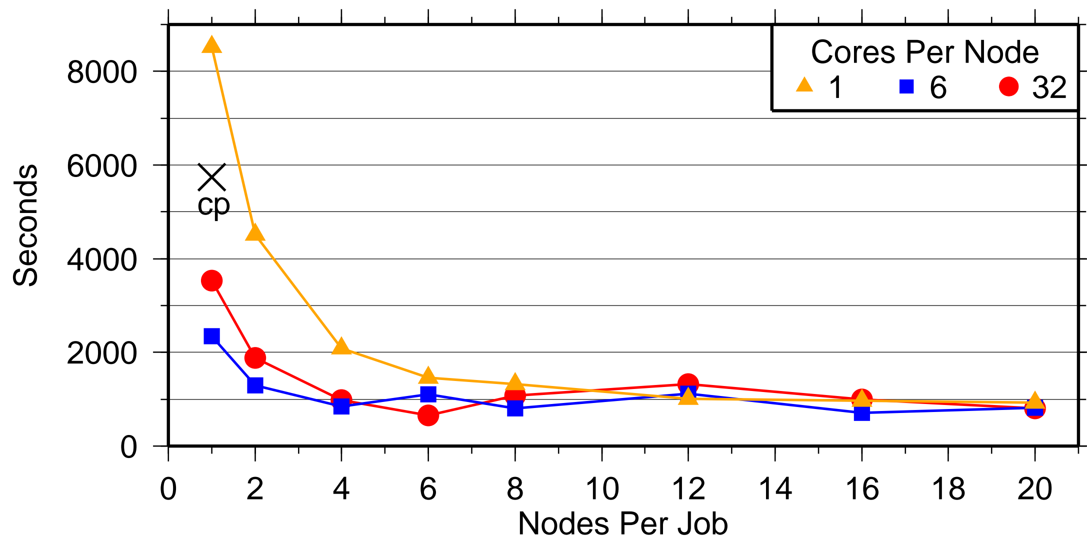

Figure 3.6: `pct` on Fission Copying "Many Small Files" Set

Table 3.9: Time/Speedup for `pct` on Fission Copying "Few Large Files" Set

Nodes	1 Core Per Node	8 Cores Per Node	32 Cores Per Node
1	7919 / 0.97	7017 / 1.10	5036 / 1.53
2	4683 / 1.64	4076 / 1.89	2805 / 2.75
4	2611 / 2.95	2175 / 3.54	1684 / 4.57
6	1852 / 4.16	1496 / 5.15	2034 / 3.79
8	2106 / 3.66	1731 / 4.45	2107 / 3.65
12	1430 / 5.39	1698 / 4.54	1948 / 3.95
16	1412 / 5.45	1971 / 3.91	1989 / 3.87
20	1291 / 5.97	1912 / 4.03	1945 / 3.96

show the range and to check that the per core relationships continue when additional nodes are used, both 1 core per node and 32 cores per node are included in this group of tests. For comparison, `cp` took 5747 seconds, which is significantly better than the single node, single core test. The results in Table 3.8 are graphed in Figure 3.6. Speedups over 8 are achieved. But as with other groups of tests, the most benefit is seen when adding the first several nodes.

For this group of tests, the times for the directory traversal portion of the program ranged from 19 to 368 seconds, with an average of 113 and a median of 104 seconds. Obviously, there is a lot of variability in this program task even though every test used 4 traversal threads.

The producer program actively participated in copying files except when there were at least 96 MPI tasks specified. In those cases, the producer stopped copying after it determined that it was more efficient to remain idle between servicing requests from the consumer nodes.

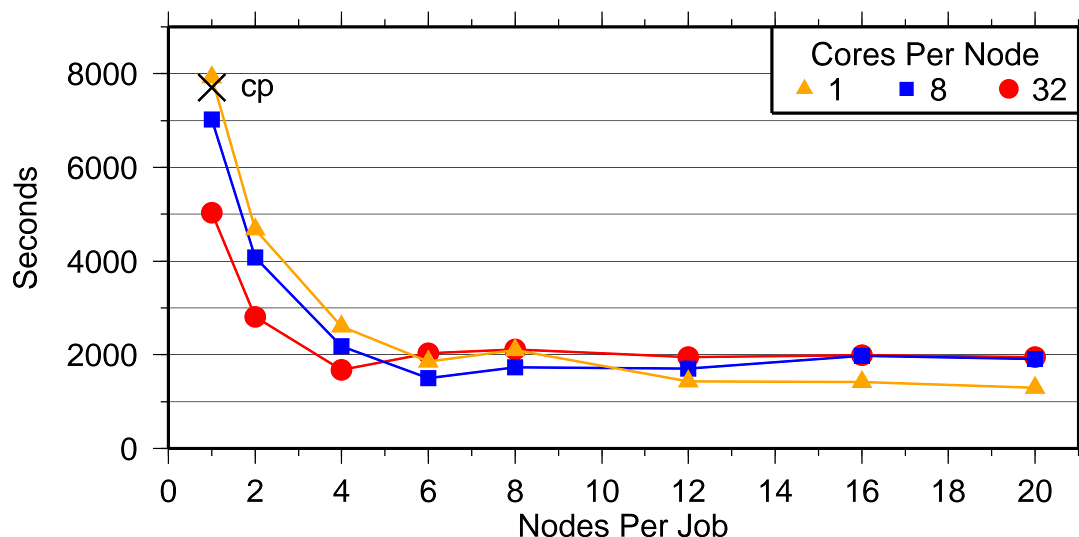

Figure 3.7: `pct` on Fission Copying "Few Large Files" Set

Copying "Few Large Files" Set on Fission with `pct`

When using `pct` to copy the "Few Large Files" set on the Fission cluster, the same general pattern of great improvement with the first additional cores and nodes that is seen with other test groups is also seen here. The times and speedups are given in Table 3.9. For the speedup calculations, the time for `cp` was 7701 seconds. A speedup of almost 6 is attained. For this group of tests, directory traversal times ranged from 1 to 8 seconds.

As seen in the graph of the data in Figure 3.7, using 32 cores per node has a minimum time at 4 nodes. For 8 cores per node, the minimum is at 8 nodes. Using more nodes yields a poorer time for both, but interestingly, that worse time also has a maximum. This seems to be caused by the paucity of files; there are simply not enough files to create more competition with each other. Adding more cores is simply irrelevant.

Unless it was part of some testing noise, using 1 core per node with 20 nodes

Table 3.10: Time/Speedup for `pct` on Icestorm Copying "Many Small Files" Set

Nodes	1 Core Per Node	4 Cores Per Node	8 Cores Per Node
1	9728 / 0.59	3112 / 1.83	2408 / 2.37
2	4334 / 1.32	1629 / 3.50	1343 / 4.25
4	2224 / 2.56	1377 / 4.14	899 / 6.34
6	1551 / 3.68	701 / 8.13	630 / 9.05
8	1233 / 4.62	640 / 8.91	653 / 8.73
12	1007 / 5.66	673 / 8.47	596 / 9.57
16	807 / 7.07	628 / 9.08	659 / 8.65
20	855 / 6.67	638 / 8.94	653 / 8.73

did not reach a minimum. Since this test, which had a speedup of 5.97, required allocation of 640 cores (20 nodes at 32 cores each), additional tests were not done past this point.

The algorithm that decides whether the producer node stops copying files also has an interesting pattern in this group of tests. As with the "Many Small Files" tests, it continues copying when there are not a lot of other processes in its MPI cohort to keep it busy servicing requests for work. This is true with 1, 2, 4, and 6 nodes using 1 core. It also occurs with miscellaneous cores on 1 node. With more processes, it makes the decision to stop copying. However, when the number rises to 256 cores doing the copying, there are not enough messages sent out with idle time between them to establish a trend to make a decision about stopping. This occurs because of the relatively few files for the number of cores performing the copying. The producer runs out of files that it can copy before it can decide to stop doing copy itself.

Copying "Many Small Files" Set on Icestorm with `pct`

Another group of tests were run using the "Many Small Files" set but were done on the Icestorm cluster. The speedups and times are given in Table 3.10 and are

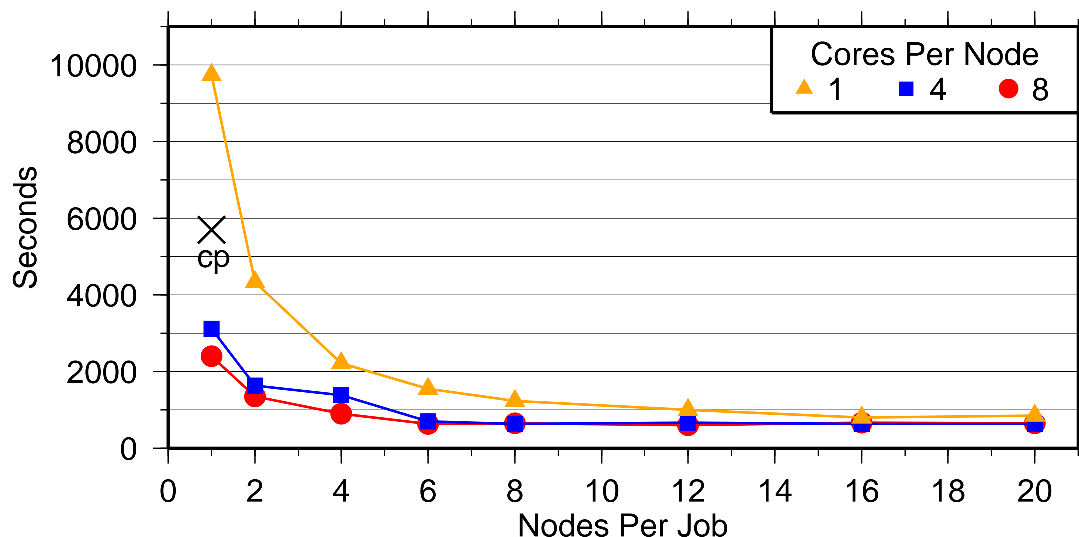

Figure 3.8: `pct` on Icestorm Copying "Many Small Files" Set

graphed in Figure 3.8. The pattern is quite similar to that on Fission but the times and speedups are a little better. A speedup of 9.57 was achieved when compared to `cp` running on a single core, which took 5702 seconds.

For this group of tests, the directory traversal times range from 28 to 156 seconds with a mean of 84 and a median of 76 seconds. This is less variable than those for Fission. Figure 3.8 also seems to indicate less testing noise in this group of tests. The core count at which the producer node decides to stop participating in the copying of files was 64, which is less than Fission's 96. On Icestorm, the producer continues to copy with 4 cores on each of 16 nodes, which is 64 cores, but not on 8 cores on each of 8 nodes, also 64 cores.

3.3.2 Tests Using `Bpct`

Each job created by the Java program `Bpct` only uses one processor since it is merely a Bash script. As was done with `pct`, rather than risk adding another unknown of

Table 3.11: Using `Bpct` on GeneSIS to Copy "Many Small Files" Set

# Nodes	Traversal Time	Submission Time	Best Time	Worst Time	Speedup
1	37	9	n/a	8556	0.88
2	32	9	5247	5672	1.32
3	31	8	4301	4587	1.63
4	32	7	3636	3799	1.97
5	32	13	3237	3348	2.24
6	32	9	2987	3163	2.37
7	32	10	2918	3117	2.41
8	41	11	2701	2771	2.71

having `Bpct` jobs share nodes with each other or other users' applications, each job is allocated an entire node for itself. The unused cores are idle. For all `Bpct` tests, 4 threads were used to traverse the source directories.

Since the jobs within a batch are independent of each other, the run times for both the first and last jobs to finish are reported. For any run time comparisons and for speedup calculations, only the slowest job's time is used since it represents the actual completion of the task.

Copying "Many Small Files" Set on GeneSIS with `Bpct`

The program `Bpct` was tested on the GeneSIS cluster using varying numbers of nodes to copy the "Many Small Files" set. The results are shown in Table 3.11 and graphed in Figure 3.9. The "Best Job Time" is the time from the starting of `Bpct` to the end of the first job to finish. The "Worst Job Time" is to the end of the last job. Both times include the directory traversal and job submission times.

Because of the additional overhead of creating scripts and processing copy commands individually rather than as a single `cp` command, using `Bpct` on a single node performed worse. As with `pct`, the small overhead accumulates over the 203,819 files.

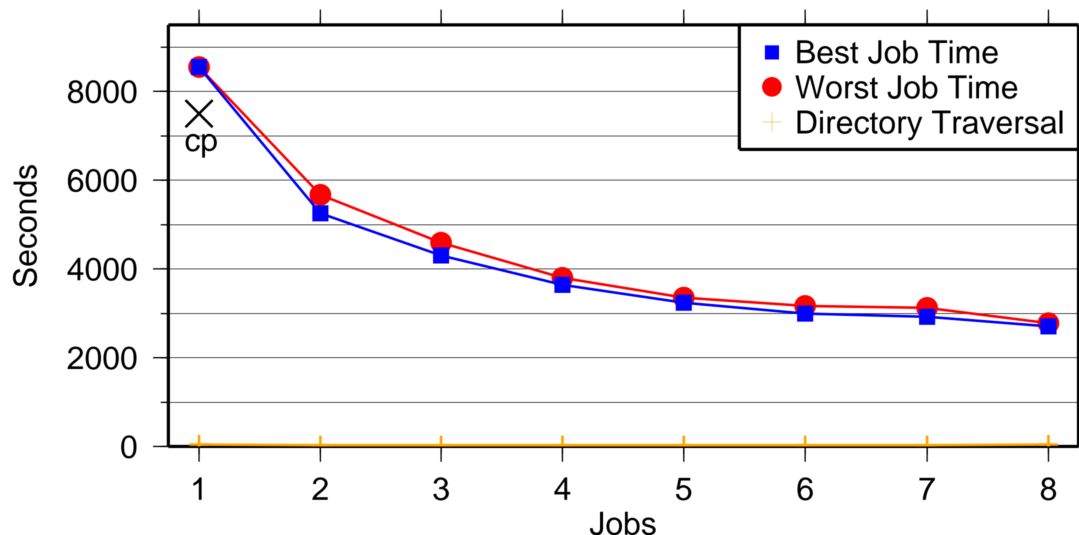

Figure 3.9: Best and Worst `Bpct` Jobs on GeneSIS Copying "Many Small Files" Set

But again, that penalty is quickly overcome by using additional nodes. Using `Bpct` gave speedups of up to 2.71 compared to the 7499 seconds needed for `cp`, with almost half of the benefit gained by adding a single additional node.

The closeness in times between the best and worst jobs in each run indicates a fair degree of success in balancing the work between them. Visual comparisons of the number of files and total bytes assigned to each job showed an even better distribution, so some variation comes from randomness in performance. But with a large number of files, as in this group of tests, some of the randomness is smoothed.

The times for directory traversal and to submit the jobs, listed in Table 3.11, show that neither were a large part of the total time. The `usemem` program was used before each test, but the traversal performance for such a large number of files and subdirectories was quite good. That may indicate some other form of caching that was not released. But the relative consistency between tests is still useful for these comparisons. If there is unreleased caching, the same amount would presumably

Table 3.12: Using Bpct on GeneSIS to Copy “Partial Large Files” Set

# Nodes	Best Time	Worst Time	Speedup
1	n/a	6425	1.01
2	3030	3441	1.89
3	2363	2726	2.38
4	1864	2329	2.79
5	1632	2155	3.02
6	1119	2140	3.04
7	1239	1532	4.24
8	897	1186	5.48

apply to all tests in the group, including the `cp` run.

Copying “Partial Large Files” Set on GeneSIS with `Bpct`

This group of tests uses `Bpct` to copy large files on the GeneSIS cluster, testing throughput. The reduction of the number of files is apparent in that the single node time is very similar to the `cp` command. And without the multitude of files to smooth out variations in copying, the spread between best and worst jobs increases. In one test, one of the jobs only had a single file to copy. The directory traversal times were all under one second, so they were not included in the table.

The speedup times compared to `cp`’s time of 6499 seconds are quite substantial, increasing up to 5.48, but are again most dramatic with the first additional nodes. The ability to have greater speedups with copying fewer but larger files, may indicate that the previous group of tests were partially limited by the file system’s ability to create the many new files rather than there being a throughput problem to and from the nodes. The throughput with this group of tests is still increasing almost linearly when the last node is included.

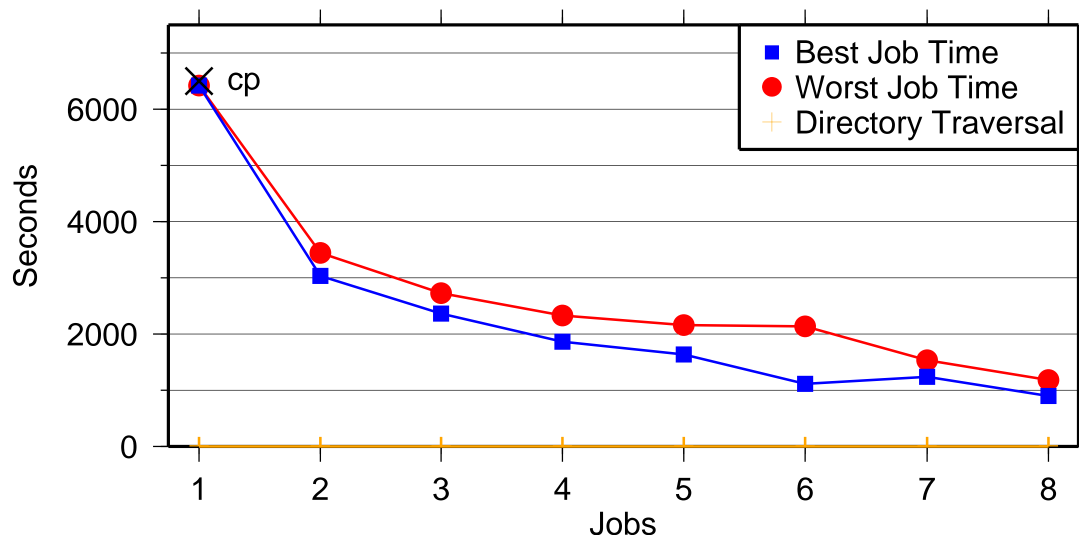

Figure 3.10: Best and Worst `Bpct` Jobs on GeneSIS Copying "Partial Large Files"

Copying "Many Small Files" Set on Icestorm with `Bpct`

The run times for this group of `Bpct` tests on INL's Icestorm cluster is somewhat similar to the times for the GeneSIS cluster except for Icestorm's significantly larger directory traversal times. The times are listed in Table 3.13 and graphed in Figure 3.11. However, the time of 4951 seconds for `cp` is shorter. This leads to poorer results in speedups for the same number of nodes, but the overall pattern is similar, having the most benefit with the first additional nodes. But still, a speedup of 3.67 is achieved with 28 jobs.

The relatively close times between best and worst jobs again supports the observations that the work balancing algorithm is fairly effective, especially with a large number of files to smooth the results.

The time needed to traverse the directories with this data set becomes noticable relative to the job runtimes. Much of this traversal time is related to recognizing symbolic links. By default, the `isFile()` and `isDirectory()` methods of Java's

Table 3.13: Using `Bpct` on Icestorm to Copy "Many Small Files" Set

# Nodes	Traversal Time	Submission Time	Best Time	Worst Time	Speedup
1	443	10	n/a	8981	0.55
2	455	12	5882	6377	0.78
4	658	13	4253	4301	1.15
6	493	15	3407	3613	1.37
8	671	15	2563	2681	1.85
12	670	18	2029	2098	2.36
16	637	21	1780	1955	2.53
20	457	19	1306	1488	3.33
24	674	24	1493	1641	3.02
28	680	22	1244	1350	3.67

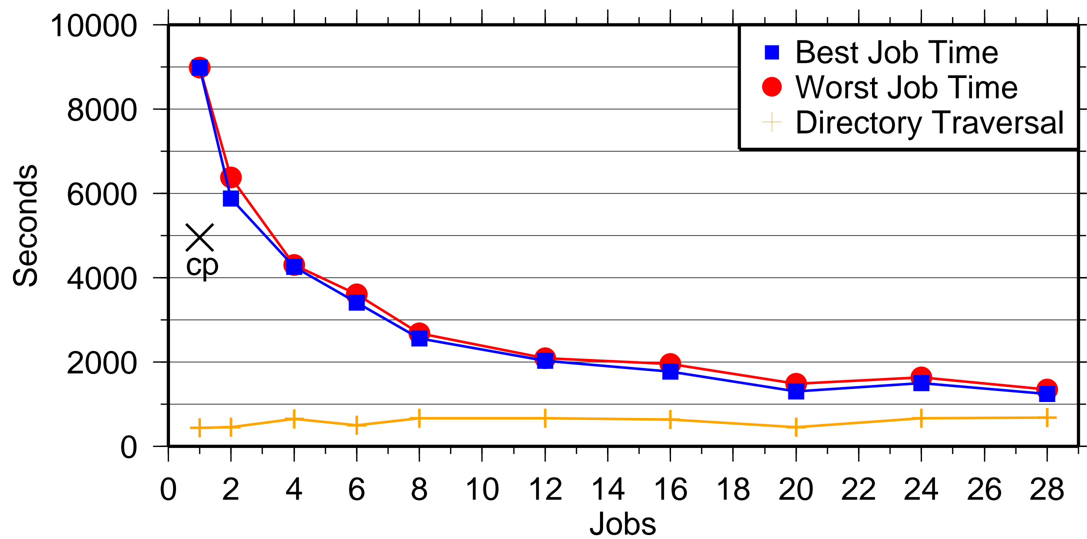

Figure 3.11: Best and Worst `Bpct` Jobs on Icestorm Copying "Many Small Files" Set

Table 3.14: Using `Bpct` on Icestorm to Copy "Few Large Files" Set

# Nodes	Traversal Time	Submission Time	Best Time	Worst Time	Speedup
1	5	0	9676	9676	1.01
2	3	1	5233	5241	1.87
4	5	0	2813	2882	3.39
6	6	0	2054	2110	4.63
8	4	0	1786	1851	5.28
12	14	2	1436	1512	6.47
16	7	2	1511	1607	6.08
20	5	1	1232	1335	7.32
24	5	1	1270	1340	7.30
28	6	1	1299	1351	7.24

`File` class dereference links to provide information about the target. However, by default `cp` does not dereference links during a recursive copy, thus `pct` and `Bpct` do not either unless the `--no-dereference` option is used within `--cp`. So to detect symbolic links in the available versions of Java, multiple `File` objects need to be instantiated to compare paths and parent paths. Twenty command line runs of `Bpct` on Icestorm with the `--no-dereference` were interleaved with twenty without and timed. The result is a fourfold increase in directory traversal times when detecting the symbolic links. The same runs were done on Fission with the same results. Java 1.7 which is not available on the test clusters, has new classes and methods to detect file, link, and directory attributes; this contains some potential for increased performance.

Copying "Few Large Files" Set on Icestorm with `Bpct`

On Icestorm, `Bpct` is able to achieve a speedup of 5.28 on 8 nodes, similar to the 5.48 on GeneSIS for 8 nodes. Even though the data sets are of different sizes, both are meant as tests of data throughput. Since both have relatively few files, their directory traversal times are small. And with 28 nodes, this group of tests reaches a speedup

Figure 3.12: Best and Worst `Bpct` Jobs on Icestorm Copying "Few Large Files" Set

of 7.24. The `cp` time on one node for this data set is 9776 seconds. The results are listed in Table 3.14 and graphed in Figure 3.12.

Copying "Many Small Files" Set on Fission

Some early tests on the Fission cluster copying the "Many Small Files" set revealed both a downside and a benefit of `Bpct` compared to `pct`. The downside is related to the PBS Pro job scheduler. With large scripts, there can be a delay of 2 minutes or more between a job transitioning from 'queued' to 'running.' In this test set, shown in Table 3.15 and graphed in Figure 3.13, the large script files took a long time to become active, but as the scripts became smaller as more jobs were added to the batch, the transitions became shorter. But even so, the delay was substantial enough that the first jobs completed before the final jobs became active. This was not a matter of the restrictions on the number of concurrent jobs per user, which is 25; even with 36 jobs created, less than 25 were concurrently active. It seemed to

Table 3.15: Using `Bpct` on Fission to Copy "Many Small Files" Set

# Nodes	Traverse & Submit	Best Script	Worst Script	Speedup	Best Time	Worst Time	Speedup
1	161	n/a	6755	0.66	n/a	7339	0.60
4	239	2326	2525	1.75	2498	3139	1.41
8	291	1537	1845	2.40	1641	2442	1.81
12	223	995	1178	3.76	1077	1756	2.52
16	239	905	1068	4.15	1100	1664	2.66
20	273	808	1116	3.97	871	1702	2.60
24	336	768	1050	4.22	823	1672	2.65
28	357	770	929	4.77	880	1555	2.85
32	369	757	896	4.94	853	1517	2.92
36	230	538	850	5.21	587	1448	3.06

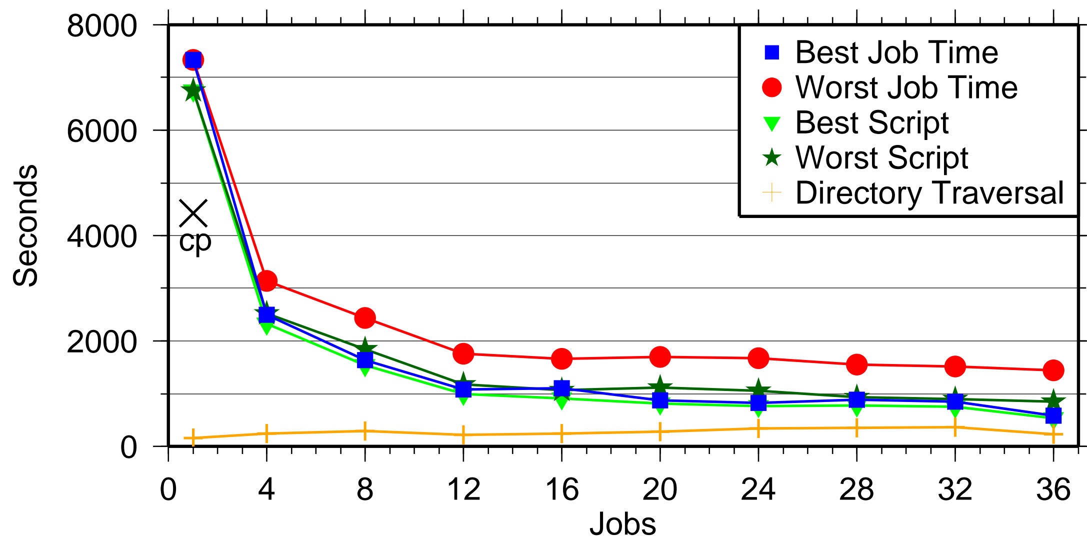

Figure 3.13: `Bpct` Jobs on Fission Copying "Many Small Files" Set

depend primarily on the length of the job script. The single job script was 70MB and 218,691 lines long. This phenomenon occurred even though the cluster had only a quarter of its cores allocated and little job submission activity other than these tests was noticed. So apparently the scheduler performs preprocessing of the script and its slowness affects the jobs themselves. Potentially, Bash itself also preprocesses the scripts, but since the Icestorm cluster, which also uses Bash as its shell, showed very little sensitivity to script size, PBS Pro on this cluster is the more likely cause.

Table 3.15 has additional columns, corresponding to additional lines in Figure 3.13, which may need some explanation. The "Worst Script" and "Best Script" columns, both of which include the directory traversal and job submission time, are the times needed to run the scripts, without the delay caused by waiting in the queue to become active. The speedup column partially represents the potential improvement if the slow queue transitions were not present. The "Best Job" and "Worst Job" are the same as previous tables; they are the times from the beginning of the command line `Bpct` program to the end of the first and last jobs to finish. The speedup listed with them is based on the "Worst Job" time and indicates the actual observed improvement. However, neither speedup column is truly representative of improvements if the queue delays did not exist. Since some jobs finished before others even started, they did not suffer the effects of competition with each other for file system resources.

In this group of runs, the long time needed to traverse the directory and identify files becomes a significant factor when many jobs are specified. This was seen on Icestorm too but much less on GeneSIS. This may represent the general busyness of the PanFS system mounted on both Icestorm and Fission, or it may be a characteristic of the file system itself. Additional threads may help reduce this time, but since using most of the processor time on the service nodes, which are used by all users, would

generally be both impolite and unpredictable, it was not investigated. The variability seen in this group of tests, even though the same number of threads are used, show the difficulties in trying to quantify the differences in threads used. The consistency of using 4 threads for all tests was maintained.

The final improvements in `Bpct`'s algorithm for distributing work were not implemented for this group of tests. Since quotas on resources were used up for the Fission cluster, these tests were not redone with the improved algorithm. However, those final improvements were fairly minor and likely would only cause slight narrowing of the values for the "Best" and "Worst" values.

The benefit of `Bpct` over `pct` that these tests highlight is that `Bpct` does not need to have all the resources ready for it to begin its copying. If a cluster is busy, any available nodes can be allocated to some of the jobs and as those jobs finish or other users' jobs finish, the remaining jobs can then run. The user can even purposefully designate many more jobs than available nodes, and then the job scheduler becomes part of the work distribution.

CHAPTER 4

CONCLUSIONS

4.1 General Conclusions

Both implementations of a parallel copying tool have demonstrated large potential for improving the chore of making quick, accurate, monitorable copies of large directories. Each has advantages. Each makes better use of the architecture of parallel distributed file systems than the native `cp` command is able to do. The timing runs show that there are scaling limitations that may possibly result from the file servers, the network and subnetworks, or the throughput to individual nodes. This is actually a positive in regards to this project because the software that was developed is not the limiting factor. Both of the implemented strategies are able to maximize the physical architecture of the clusters. The implication is that these tools will not quickly become outdated as the physical capabilities of cluster computing expand.

All of the current scaling issues that were encountered by these tools are outside of the scope of both this project and of a single generic copying tool. Software intelligence plus knowledge of the cluster architecture would be needed to fully optimize parallel directory copying. Because flexibility is built into the software of both `pct` and `Bpct`, some improvements and more intelligent plugins to the tools are future possibilities.

To investigate the details of where the bottlenecks occur, better inside information

would be needed regarding the configuration of the parallel distributed file systems. Some systems allow grouping a user's files on one server. Others allow distributing a user's files across multiple servers. Both strategies have advantages. So does a strategy of letting files reside on servers based on when they are created. But knowing how files are allocated and distributed would be needed to fully understand what improvements would optimize the performance of the two copying programs.

4.2 Future Directions

There are a couple features still missing in the `Bpct` program, two small and one substantial. (The program `pct` has these implemented.) The first small missing feature is the detection and implementation of the `cp` option `--strip-trailing-slashes`. This option affects the handling of a link to a directory on the command line. If the slash trails the link name, by default the link is resolved to the directory to which it points. The `--strip-trailing-slashes` overrides that behavior and treats it as if it does not have the trailing slashes and is merely a link to be copied. The second small missing feature is the `--parents` option which causes extra directory creation below the destination directory that mimics the directory tree from the source directories.

The larger missing feature is the preservation of directory attributes. Preserving the attributes of the files is automatically handled by the `--cp` options selected by the user and carried out by the `cp` command. But a destination directory's attributes should not be reset to the source directory's values until all the files are present, otherwise the modified times will change. Since the multiple jobs in a batch copy process are independent and since files are balanced between jobs rather than kept in their original directory grouping, there is not a simple way to detect when a directory's

attributes should be set. One or a combination of two approaches is recommended. The first is to create a subclass of the file batching class that determines how files are distributed across jobs for copying. By making it aware of the directory structure and the files contained by each directory, some of the directories' attributes could be adjusted by commands within the job script when the files for that directory and its subdirectories have been copied. Another possibility is to have one job do all or the remaining attribute adjustments when all the other jobs are complete. The signaling for completeness could be the simple creation of empty files in the directory used to monitor the progress of the jobs. The last job waits for the presence of all files signaling the completion of the other jobs and then makes the changes to the attributes of directories. Of course, this is only needed if the user's selection of `cp` options would require preservation of directory attributes too. The command line parser already detects that situation.

Apparently, the current versions of the program are able to make full use of the architecture of the clusters on which they were tested. If other or future clusters are able to provide more capacities that are beyond these utilities, a more decentralized method of distributing work may be needed. Using a single node to traverse directories and send work to consumer nodes may become a limiting factor if much larger file systems become available. With a suitable termination detection algorithm, such as a tree or a dual pass ring [16], subdirectories could be distributed as work rather than only files being distributed. Either multiple producer nodes could be used or any participating process could be allowed to traverse subdirectories.

During the testing of `Bpct` on the Fission cluster, when job scripts were large, long delays occurred while jobs transitioned from the queued state to running. Perhaps a scheme could be devised that puts the file information into a separate file so that the

script itself remains small.

The `--save` option allows the user to create defaults for complicated options for `mpirun` and `qsub`. However, there are values that cross between those options when choosing the number of cores and nodes. Perhaps, a template of sorts could be devised that calculates the proper changes to the `mpirun` and `qsub` options when the user picks the number of nodes to use.

Mention was made of the possibility that the use of `fork()` and `execvp()` may be the cause of failure when multiple threads are used to optimize the copying of files while waiting for MPI calls to complete. Rather than using the existing `cp` command, the code from that core utility could be incorporated into `pct` so that a separate process with its accompanying interrupts does not need to be created. This may also help reduce the overhead that accumulates when very many files are involved.

And possibilities exist for improving or replacing existing algorithms for distributing work. For instance, if `cp` is continued to be used for the actual copying, some overhead could be saved if it was given subdirectories rather than individual files. And in `pct`, the current policy of copying large files first causes an overemphasis at the beginning on data throughput. This may be a less efficient way of doing things. Perhaps a better mix of file sizes could be used throughout the process so that no part of the file system is unnecessarily overwhelmed. The current code is designed to allow new strategies to be added without difficulty. Major divisions within the programs are logically separate so that they may be replaced. So such changes could be easily added as replacements or as additional options.

REFERENCES

[1] Apache Software Foundation. *Hadoop 0.20 Documentation*. http://hadoop.apache.org/docs/stable/distcp.html

[2] Argonne National Laboratory. *MPI_Irecv*. http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Recv.html

[3] Shane Canon, Don Maxwell, Josh Lothian, Kenneth Matney, Makia Minich, H. Sarp Oral, Jeffrey Becklehimer, and Cathy Willis. "Integrating and Operating a Conjoined XT3+XT4 System," *Cray User Group Conference*, 2007. https://cug.org/5-publications/proceedings_attendee_lists/2007CD/S07_Proceedings/pages/Authors/Canon/Canon_paper.pdf

[4] Hsing-bung (HB) Chen, Gary Grider, Cody Scott, Milton Turley Aaron Torres, Kathy Sanchez, John Bremer. "Integration Experiences and Performance Studies of A COTS Parallel Archive System - A New Parallel Archive Storage System Concept and Implementation" *IEEE International Conference on Cluster Computing 2010,* Heraklion,Crete, Greece. 2010. http://www.cluster2010.org/presentations/session 6/HBChen.ppt

[5] Hsing-bung Chen, Gary Grider, Cody Scott, Milton Turley, Aaron Torres, Kathy Sanchez, and John Bremer, "Integration Experiences and Performance Studies of A COTS Parallel Archive System, cluster," *2010 IEEE International Conference on Cluster Computing, 2010* pp.166-177. http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-10-01765, 2010.

[6] Idaho National Laboratory. *HPC Data Storage Plan. Document ID:PLN-4369, revision 0*. 2012.

[7] A Large Ion Collider Experiment. *Manual Data Transfer*. http://alice-wiki.gsi.de/cgi-bin/view/Data/ManualDataTransfer, 2011.

[8] Network-Based Computing Laboratory, Ohio State University. *MVAPICH2 1.9a2 Features and Supported Platforms*. http://mvapich.cse.ohio-state.edu/overview/mvapich2/features.shtml

[9] OLCF User Assistance Center, Oak Ridge Leadership Computing Facility. *Transferring Data with SDPCP[sic].* https://www.olcf.ornl.gov/kb_articles/transferring-data-with-sdpcp.

[10] Mark A. Roschke, Bart J. Parliman, Danny P. Cook, and C. David Sherrill, "Parallel Processing of Data, Metadata, and Aggregates within an Archival Storage System User Interface (Toward Archiving a Million Files and a Million Megabytes per Minute)", *2008 Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, pp.36-43, 2008. http://www.osti.gov/bridge/servlets/purl/960622-IpyhEq/960622.pdf

[11] Mark A. Roschke and C. David Sherrill. *PSI - A High Performance File Transfer User Interface.* http://outreach.scidac.gov/fsbp/PositionPapers/Usability_LANL_Roschke.pdf

[12] Saba Sehrish. *Improving Performance and Programmer Productivity for I/O-Intensive High Performance Computing Applications.* Ph.D. Dissertation: University of Central Florida, Orlando, FL, 2010. http://purl.fcla.edu/fcla/etd/CFE0003236

[13] Galen M. Shipman. *Lustre at the OLCF: Experiences and Path Forward.* http://wiki.lustre.org/images/d/d5/Gshipman_LUG_2010.pdf, 2010.

[14] Sourceforge.net. Xcat utilities website. *pscp - parallel remote copy.* http://xcat.sourceforge.net/man1/pscp.1.html.

[15] Albert Tannous, Christina Patrick, and Jesse Scott. *Sequential I/O Streams versus Parallel I/O Streams: A Case Study.* http://newcp.googlecode.com/files/NewCopy.pdf.

[16] Barry Wilkinson and Michael Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, New Jersey, 1999.