Evaluation and Characterization of Secure File Storage in Client Parallel I/O

Author

David F. Beck
The University of New Mexico
High Performance Computing, Education & Research Center
Disclaimer

The High Performance Computing, Education & Research Center (HPCERC) provides a focus for high performance computing and communication at the University of New Mexico (UNM). HPCERC is committed to innovative research in computational and computer science with emphasis on both algorithm development and application. As part of this commitment, HPCERC sponsors this technical report series. The technical reports are subject to internal review by the HPCERC. However, the material, as presented, does not necessarily reflect any position of the HPCERC. Further, neither UNM, nor the AHPCC, makes any warranty or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information contained in this report.

Frank L. Gilfeather, Executive Director, HPCERC
Brian T. Smith, Chief Scientist, HPCERC
John S. Sobolewski, Chief Technologist, HPCERC
Ernest D. Herrera, Associate Director, HPCERC
Susan Atlas, Associate Director, AHPCC Computer Systems Research
Bob Ballance, Associate Director, AHPCC Science & Engineering Research
Evaluation and Characterization of Secure File Storage in Client Parallel I/O

David F. Beck

Abstract

While POSIX has proven itself to be a robust and useful file system interface definition for serial programming, it does not meet the needs of parallel I/O. Rather, requirements such as grouping, collective buffering and strided access are being addressed by I/O extensions to the message passing interface (MPI-IO). However, in distributed computing environments where clusters of workstations are employed as parallel compute engines, the Network File System, NFS, is also generally used to provide the common file space; this can add a number of security concerns for some environments. This paper documents work that investigated the use and relative performance of the secure Distributed File Service, DFS, as an alternate to NFS.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>i</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ii</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 FILESYSTEM SECURITY</td>
<td>2</td>
</tr>
<tr>
<td>2.1 Network Communications</td>
<td>2</td>
</tr>
<tr>
<td>2.2 File Services</td>
<td>2</td>
</tr>
<tr>
<td>2.2.1 NFS security issues</td>
<td>2</td>
</tr>
<tr>
<td>2.2.2 Enter DCE</td>
<td>5</td>
</tr>
<tr>
<td>3 THE TEST SUITE</td>
<td>6</td>
</tr>
<tr>
<td>3.1 Test Descriptions</td>
<td>7</td>
</tr>
<tr>
<td>3.1.1 Test 1</td>
<td>7</td>
</tr>
<tr>
<td>3.1.2 Test 2</td>
<td>7</td>
</tr>
<tr>
<td>3.1.3 Test 3</td>
<td>7</td>
</tr>
<tr>
<td>3.1.4 Test 4</td>
<td>7</td>
</tr>
<tr>
<td>3.1.5 Test 5</td>
<td>7</td>
</tr>
<tr>
<td>3.1.6 Test 6</td>
<td>8</td>
</tr>
<tr>
<td>3.1.7 Test 7</td>
<td>8</td>
</tr>
<tr>
<td>3.1.8 Test 8</td>
<td>8</td>
</tr>
<tr>
<td>3.1.9 Test 9</td>
<td>8</td>
</tr>
<tr>
<td>3.2 General Observations</td>
<td>8</td>
</tr>
<tr>
<td>4 DFS-to-NFS COMPARISON TESTS</td>
<td>10</td>
</tr>
<tr>
<td>4.1 Test Details</td>
<td>10</td>
</tr>
<tr>
<td>4.1.1 Test 1</td>
<td>10</td>
</tr>
<tr>
<td>4.1.2 Test 2</td>
<td>10</td>
</tr>
<tr>
<td>4.1.3 Test 3</td>
<td>10</td>
</tr>
<tr>
<td>4.1.4 Test 4</td>
<td>11</td>
</tr>
<tr>
<td>4.1.5 Test 5</td>
<td>11</td>
</tr>
<tr>
<td>4.1.6 Test 6</td>
<td>11</td>
</tr>
<tr>
<td>4.1.7 Test 7</td>
<td>11</td>
</tr>
<tr>
<td>4.1.8 Test 8</td>
<td>11</td>
</tr>
<tr>
<td>4.1.9 Test 9</td>
<td>11</td>
</tr>
</tbody>
</table>
4.2 Test Results ................................................................. 12
4.3 Test Observations. ...................................................... 13

5 CODE DESCRIPTION .......................................................... 15
5.1 main() ................................................................. 15
5.2 Common Functions .................................................. 16
   5.2.1 cio.h and cio.c ................................................. 16
   5.2.2 fswrapper.h and fswrapper.c ............................. 17
   5.2.3 mpiosubr.h and mpiosubr.c ............................... 17
   5.2.4 vsscanf.h and vsscanf.c ................................. 17
5.3 Common Header Files .............................................. 18
   5.3.1 Config.h ...................................................... 18
   5.3.2 logger.h ........................................................ 18
   5.3.3 tests.h ........................................................ 18
5.4 Script Files .......................................................... 18
   5.4.1 mrun.sh ....................................................... 18
   5.4.2 runmaster.sh ................................................ 18
   5.4.3 runSummary.awk ........................................... 19
5.5 Miscellaneous Files ............................................... 19

ACKNOWLEDGEMENTS ....................................................... 20

REFERENCES .............................................................. 21

APPENDIX
LIST OF FIGURES

Figure 1  DFS-to-NFS comparison test results. .............................. 13

Figure 2  An illustration of the skewed data observed in tests 3, 5.I and 5.II. .......... 14
LIST OF TABLES

Table 1   DFS-to-NFS Comparison Test Results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
1 INTRODUCTION

While POSIX has proved itself as providing a robust and useful file system interface definition for the serial I/O required for many operating systems, it does not meet the needs of parallel I/O. These needs include, for example, grouping, collective buffering and strided access. In an effort to address such needs, I/O extensions to the message passing interface (MPI-IO)[1] were developed. Several implementations of MPI-IO are available[2] as follows:

**PMPIO**— Portable MPI I/O library developed by NASA Ames Research Center[3]. A full reference implementation that currently supports MPICH 1.1.2[4] or SGI's MPI 3.1.1.1.

**ROMIO**— A high-performance, portable MPI-IO implementation developed by Argonne National Laboratory[5]. Does not yet fully implement the standard (provides all features except support for file interoperability, I/O error handling, and I/O error classes.). However, it does provide support for more file system types than other implementations (including the Network File System, NFS). ROMIO is distributed with recent MPICH releases.

**MPI-IO/GPFS**— IBM partial implementation of the standard running on top of the IBM Generalized Parallel File System on IBM RS/6000 SP computers[6].

**HPSS[7] Implementation**— Developed by Lawrence Livermore National Laboratory as part of its Parallel I/O Project[8].

This paper documents a preliminary investigation into the functionality and performance of MPI-IO. The primary thrust of the project is to adapt standardized software tools used in evaluating NFS performance with a port that utilizes the MPI-IO libraries, and then to use these tools to estimate what levels of I/O performance could be expected from within a client program. The initial work was done on the HPSS test bed at Sandia National Laboratories. However, due to various library incompatibilities, development work was shifted to a heterogeneous collection of UNIX workstations running MPICH and using NFS as the common file system. (It is anticipated that future efforts will include a return to the HPSS implementation.) Once developed, the code was used to evaluate the relative performance of the Distributed File Service (DFS) as compared to NFS. The interest in DFS is driven by security concerns with
NFS, as outlined below. This paper serves both to document the software tools developed as well as to report on the results of the NFS-DFS comparison tests.

2 FILESYSTEM SECURITY

As can be expected for a national laboratory, security is an important concern. In using MPICH in a distributed environment, two issues over and above normal workstation security concerns come immediately to mind: (1) the security of process-to-process communications; and (2), the security of file I/O services.

2.1 Network Communications

The normal process start-up mechanism used by MPICH communicating via TCP/IP (p4) devices over networks is rsh[9]. The UNIX “r” commands (rsh, rlogin and rcp) were developed for use in a trusted environment. Entries in /etc/hosts.equiv or $HOME/.rhosts files allow “r” commands remote access without login, an obvious security concern where trust can not be assured[10]. Fortunately, MPICH has two other “p4” communication options: a “secure” server and secure shell (ssh)[11]. Due in part to the existing security environment at Sandia National Laboratories (SNL), the ssh option was selected for use in the tests presented herein. The ssh version used (1.2.20) was “kerberized” (modified and linked to kerberos[12] version 5 beta 2[13] libraries) rather than a version that used the standard public/private keys as discussed in the MPICH installation manual. Use of a kerberized ssh also enabled use of secure file services (see the DCE discussion below) because it supports forwarding of security credentials. Future work is planned that will evaluate and possibly modify the MPICH “secure” server to support credential authentication and forwarding as a way to improve job start-up speeds[14].

2.2 File Services

2.2.1 NFS security issues

In the 1980s Sun Microsystems developed the Network File System (NFS) as a way to create a file system on diskless clients. Because this protocol also provided a way for file system resources to be fully shared among network components in an almost user-transparent manner, its use quickly spread to
where it is now available for virtually all current UNIX operating systems (as well as many non-UNIX systems). User transparency arises because NFS is layered under the virtual file system (VFS) used by modern UNIX operating systems. User programs use the same I/O library function calls regardless of the file system underlying a particular node (virtual node, or vnode, in this case).

NFS requests are handled via a suite of client and server daemons and commands that utilize an implementation of remote procedure calls (RPC) also developed by Sun (Open Network Computing). Although some recent versions of RPC support the Transmission Control Protocol (TCP), NFS was initially designed as a stateless protocol implemented over the User Datagram Protocol (UDP). Most NFS/RPC implementations continue to use UDP.

NFS client requests (RPC calls) are nothing more than network messages to NFS daemons (nfsd) that include a file handle, the requested operation (e.g., read, write, etc.), and an identity. NFS in its ubiquitous form utilizes “AUTH_UNIX” authentication—the UNIX numeric user and group identities commonly called UID and GID, along with a machine name. Universal name space is not available, and no verifiers exist. The NFS server implicitly trusts the authentication identity passed by the client. Users never have to separately log into the server, nor supply a password, in order to use these RPC-based network services. A secure authentication option is defined for Sun RPC (RFC 1057) and thus NFS, but it is poorly integrated, cryptographically weak, and rarely used.

A file handle (to the “root” directory of the exported file system) is acquired when a client accesses a remote file system the first time (returned by a request to the mount daemon, rpc.mountd, via a portmapper daemon running on the server). Typically all file export restrictions are enforced at this point. UNIX access controls on files in the exported file space are enforced by the client’s own operating system.
From this brief description it is possible to identify a number of security issues concerning the use of NFS:

1. As for UNIX in general, anyone with administrative (“superuser”) privileges can masquerade as any user and thus gain access to their files.

2. Since UDP is not connection oriented, there is no implied authentication on the return path. As UDP packets can be easily forged, source addresses can not be trusted. (TCP packets can also be forged, just not as easily.)

3. Machine names and user UIDs/GIDs can be easily spoofed. A client using RPC can claim to be any user at all (except, perhaps, root).

4. RPC requests can be assembled using “sniffed” file handles. Since handles are generally based on the file system ID (disk device number), the file inode number, and the inode generation number, they can also be predicted. Thus all of the files on any NFS server (except possibly those owned by root) are potentially accessible over a network by anyone with the ability and inclination to search for valid file handles.

5. Once a client has a valid handle, removal of the client from the access list (typically /etc/exports) will not prevent its continued use (even after a server reboot!).

6. The portmap daemon is subject to spoofing. Attackers may be able to replace RPC services with Trojaned versions.

7. Data in transit is passed “in the clear,” and can be sniffed (copied) or patched (replaced or altered).

8. The daemons are subject to programming errors that may be exploit-able. Problems reported in the past include: buffer overflows (may cause NFS concerns because file systems that are intended to have export restrictions are really exported to the world, or system concerns because “superuser” access is gained); type conversion errors (UID altered, pro-
viding unintended file access), and request forwarding or proxy access (real source of request lost and unintended file access granted as a result).

While some of these concerns can be addressed by keeping up with system patches and updates, or reduced through system design (e.g., switched hubs to reduce the potential for sniffers), most are related to the fundamental assumptions underlying the design of NFS (i.e., a friendly network). Any decision to use NFS should be done with due consideration for these security weaknesses.

2.2.2 Enter DCE

In 1988 IBM, HP, DEC and other companies joined together to found the non-profit Open Software Foundation (OSF). The OSF, an organization now representing over 350 member companies, has a mission \[15\] of “bringing open, distributed, mixed-vendor computing from concept to reality.” One of the OSF products is the Distributed Computing Environment (DCE). At the heart of DCE is an implementation of RPC derived from the Network Computing System developed by Apollo Computers (now part of HP) in the mid 1980s. As released, the DCE RPC protocol has features such as:

1. Connection-oriented communication that can utilize either UDP or TCP.

2. Universal name space.

3. A rich suite of security services including:

   i. All users, programs and systems (called “principals”) are assigned a unique identity and secret key.

   ii. Kerberos authentication protocols are used to verify principal identification (“login”) for access to network (RPC-based) services.

   iii. When a client makes a RPC to a server, information is passed in the form of a unique “ticket” generated by a security
server for the specific client-server-event instance. This information includes the principal’s identity and a list of all authorization groups to which that principal belongs. Tickets are protected on the basis of secrets that principals only share with the security server, and that are never passed in the clear. Other features exist to prevent replay or spoofing attacks in case a ticket is sniffed off of the network.

iv. Access Control Lists (ACLs) are available to authorize all actions of principals on objects (e.g., read, write, execution). DCE ACLs are, in concept, very much like UNIX ACLs; however, DCE both defines more types of ACLs than UNIX and provides mechanisms to implement new types.

v. Several levels of data protection (e.g., DES encryption) are available for client-server communications.

Much like NFS that is built on Sun RPC, the OSF developed a Distributed File Service (DFS) based on their own RPCs. And also like NFS, the DFS client daemons interface with the VFS of the host operating system. While it also boasts of various enhancements like data replication, in the present context it is the inherent security of the underlying OSF RPCs that makes DFS attractive. Granted that it does not overcome the abuses that are possible by system administrators, nor escape possible programming errors, most of the security problems associated with NFS can be avoided by using DFS. The drawbacks of using the OSF products at the present time include limited platform coverage, waning vendor support, increased infrastructure costs (e.g., security servers), and licensing costs (for all server software, and for client software for some vendors).

3 THE TEST SUITE

Early in the project definition it was decided to adapt standardized software tools used in evaluating network file system (NFS) performance with a port that utilizes the MPI-IO
libraries. The “tools” identified to use as a template were the nine primary tests available in the NFS Revision 2 Testsuite [16]. The descriptions for each test follow below.

3.1 Test Descriptions

3.1.1 Test 1

Test 1 creates a tree structure a specified depth and with a specified number of file and subdirectories at each node. The main test loop makes extensive use of MPI_File_open(), MPI_File_set_size(), MPI_File_close(), umask(), mkdir(), and chdir().

3.1.2 Test 2

Test 2 removes a tree structure. The main test loop makes extensive use of unlink(), rmdir(), and chdir().

3.1.3 Test 3

Test 3 repeatedly looks up mount points. The main test loop makes extensive use of getcwd() and stat().

3.1.4 Test 4

Test 4 repeatedly looks up and sets attributes of a set of files. The main test loop makes extensive use of chmod() and stat().

3.1.5 Test 5

Test 5 evaluates file read and write performance. This test is further divided into three parts. Part I repeatedly creates a file of size zero and writes to it. After the last write the file contents are verified (not in timing data). Part I makes use of MPI_File_open(), MPI_File_set_size(), umask(), MPI_File_get_size(), MPI_File_write_at(), MPI_File_sync(), MPI_File_close(), stat(), and MPI_File_read_at(). Part II repeatedly opens a file, truncates it to a size of zero, and writes to it. Part II uses the same calls as Part I except for umask(), and also verifies the last write after the timing loop. Part III repeatedly
opens a file in a read only mode, reads in the contents, and then closes the file (no verify step). Part III makes extensive use of MPI_File_open(), MPI_File_read_at(), and MPI_File_close().

### 3.1.6 Test 6

Test 6 sets up a test directory with a specified number of files, and then opens and reads the directory. It then repeatedly rewinds and re-reads the directory a specified number of times. During each pass it: (1) looks to make sure it finds the expected files; and (2), deletes one of the files. The main test loop makes use of opendir(), rewinddir(), readdir(), unlink(), and stat().

### 3.1.7 Test 7

Test 7 sets up a test directory with a specified number of files. For a specified number of times the program then renames each file to a new name, links the file under its to name to the original name, and then unlinks the new name. The main test loop makes use of rename() [only root process], stat(), link(), and unlink().

### 3.1.8 Test 8

Test 8 first sets up an empty test directory. It then creates a set of symbolic links to nonexistent files, and checks the status of each link. The test then reads and verifies each link, and then removes it. This sequence is then repeated a specified number of times. The main test loop makes extensive use of symlink(), lstat(), readlink() and unlink().

### 3.1.9 Test 9

Test 9 makes repeated statvfs() calls against a test directory.

### 3.2 General Observations

As can be seen from the test descriptions in §3.1, MPI-IO (MPI_File type) calls are principally restricted to tests one (create a tree structure) and five (file read/write per-
formance). This is a result of the fact that the I/O portions of the MPI-2 standard do not define many types of operations including:

**directory operations** — mkdir(), rmdir(), chdir(), getcwd(), opendir(), rewinddir(), readdir(), rename()

**link operations** — link(), symlink(), lstat(), readlink()

**filesystem operations** — statvfs()

**file operations** — umask()

In addition, some operations that are defined are limited in scope or functionality (at least in some environments). Some examples include:

**MPI_File_get_size()** — provides a *very* limited stat() call (and did not function correctly prior to MPICH 1.2.0 in the development environment used for this work).

**MPI_File_delete()** — should serve for an unlink() call. However, in these tests it often did not work, returning an ambiguous “problems with MPI_File_delete Unknown error.”

**MPI_File_open()** — serves as open() and creat() calls without the ability to set file mode and without a truncate option.

It is not clear to what degree the lack of MPI-IO functionality impacts mainstream MPI coding projects. However, the functionality of the UNIX I/O calls was necessary for implementing the test suite. When used, all system calls, with the exception of rename(), were called by all processes. Since this would occasionally result in errors (e.g., since only one process would actually make a link, the other the processes would complain that the link already exists), return codes are checked, when necessary, to make sure the desired action was actually completed. This method also helped to make sure that kernel directory name lookup caches were kept up to date (or at least flagged as invalid).
4 DFS-to-NFS COMPARISON TESTS

The test environment consisted of a file server and two workstations. The file server was a Sun SPARCstation 5 workstation with an external SCSI hard drive partitioned into two separate file systems — a NFS partition and a DFS partition. An IBM F-50 PowerPC under AIX 4.3.2.0 was tried as a file server, but the test code would crash the machine during NFS accesses with multiple processors. No attempt was made to optimize the server performance (e.g., buffer sizes and kernel parameters; default values were used). A second SPARCstation 5 and a SGI O2 workstation served as the compute nodes. This arrangement kept the file server and network hardware from being an issue when NFS and DFS timing data were compared (as would happen if different file servers had been used). Ideally the network should also have been isolated so that any differences would only be due to the NFS and DFS client and server code; however, access to DCE security services was required. Since network traffic and service usage could thus affect the results, testing was sequenced to gather all of the test 1 data (i.e., DFS and NFS test 1 tests), then all of the test 2 data, etc. (rather than, say, run all of the DFS tests and then all of the NFS tests), in order to try to minimize this problem. In addition, each test was nominally run 30 times in order to generate statistical information. Details for each individual test are given below, followed by the resulting timing data.

4.1 Test Details

4.1.1 Test 1

Test 1 runs each created a total of 155 files and 62 directories in a tree structure five levels deep.

4.1.2 Test 2

Test 2 runs each removed a total of 155 files and 62 directories from a tree structure five levels deep.

4.1.3 Test 3

Test 3 runs each made a total of 500 getcwd() and stat() calls on a mount point.
4.1.4 Test 4

Test 4 runs each made a total of 1000 chmod() and stat() calls on 10 files.

4.1.5 Test 5

In test 5 Part I, each processor wrote a block of 8192 bytes to a new file 10 times. In Part II, each processor wrote a block of 8192 bytes to a file, after a truncation operation, 10 times. In Part III, each processor read a block of 8192 bytes from a file 10 times.

4.1.6 Test 6

Test 6 runs each read directory entries on 200 files a total of 20500 times.

4.1.7 Test 7

Test 7 runs each completed 2000 renames and links on 100 files.

4.1.8 Test 8

Test 8 runs each completed 2000 symlinks and readlinks on 100 files.

4.1.9 Test 9

Test 9 runs each made 1500 statvfs() calls against the test directory.
4.2 Test Results

The timing data, in the form of the DFS mean time normalized by the NFS mean time, are provided in tabular and graphical form below. Also included is information on relative timing data variability in the form of one “standard deviation *”.

Table 1: DFS-to-NFS Comparison Test Results

<table>
<thead>
<tr>
<th>test number</th>
<th>relative time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8 ± 0.1</td>
</tr>
<tr>
<td>2</td>
<td>1.1 ± 0.2</td>
</tr>
<tr>
<td>3</td>
<td>1.5 (± 2.2)</td>
</tr>
<tr>
<td>4</td>
<td>0.5 ± 0.2</td>
</tr>
<tr>
<td>5 Part I</td>
<td>1.8 (± 1.0)</td>
</tr>
<tr>
<td>5 Part II</td>
<td>1.8 (± 1.4)</td>
</tr>
<tr>
<td>5 Part III</td>
<td>0.3 (± 0.3)</td>
</tr>
<tr>
<td>6</td>
<td>1.2 (± 0.3)</td>
</tr>
<tr>
<td>7</td>
<td>1.2 ± 0.1</td>
</tr>
<tr>
<td>8</td>
<td>1.3 (± 0.0)</td>
</tr>
<tr>
<td>9</td>
<td>0.7 (± 0.2)</td>
</tr>
</tbody>
</table>

* relative time = \( \frac{\text{DFS mean} + 1\sigma}{\text{NFS mean} - 1\sigma} \) - \( \frac{\text{DFS mean} - 1\sigma}{\text{NFS mean} + 1\sigma} \)
Figure 1: DFS-to-NFS comparison test results

4.3 Test Observations

The first observations that should be made are: (1) general file I/O is possible under MPI even though MPI-IO is restricted in scope; and (2), the secure file services available under DFS can be used with MPICH. However, the first reaction to the data is probably to notice the large amount of scatter in tests 3, 5.I and 5.II. A closer look at the
data reveals that the timing measurements collected had a very skewed distribution (a long “tail”), as shown in Figure 2.

Figure 2: An illustration of the skewed data observed in tests 3, 5.I and 5.II

While the actual cause of this skew is unknown, it seems reasonable to assume that it was due to the network rather than some odd behavior in the server code. Truncating the tails gives new relative timing values of 1.3, 1.6 and 1.6 for tests 3, 5.I and 5.II respectively. Using all of the means (with the “corrected” values) would indicate that the DFS tests are 5% longer, on average, than the NFS tests; this would be consistent with the expected small performance penalty due to the overhead associated with DCE security services. It could be presumed that departures for individual tests from this average are due, at least in part, to different programming algorithms within the server code.

Another observation to make is the behavior of the file operations tests (4 and 5). With a mean relative time of 0.5, DFS performance in test 4 is significantly faster than NFS. Since the primary difference between test 4 and the other tests is its use of the chmod() call, this performance suggests that the access control list (ACL) mechanism in DFS is better than that used in NFS; this is somewhat of a surprise due to the more extensive ACL features available under DCE/DFS. Also somewhat odd is the difference
between the observed read and write performances. The write tests (5.I and 5.II) had relative timing values of 1.6, while the read test (5.III) had a relative timing value of only 0.3! The only obvious explanation would be differences in file block buffering and file locking mechanisms.

Finally it should be noted that this data does not capture several issues associated with the use of a secure shell between MPI processes and the use of DCE security services. First, if it was not clear in the test descriptions, the timing data only reflects the main test loop and not the setup time associated with each test. Use of the kerberized ssh typically added about 30 seconds to the length of each test. For long test runs this overhead is probably not significant; however, it is definitely noticable for short tests. Second, execution of the tests was planned to be done strictly by use of a set of shell scripts. However, timeouts and other seemingly random errors would occur for the DFS tests that required manual intervention. This was probably due in large part to the use of a test, rather than production, DFS server. Also, as DFS credentials have a limited lifetime, long running jobs require a credential renewal mechanism; while some progress was made in this area, the longer running tests had their parameters adjusted in order to reduce the test times to less than 2 hours each in order to overcome this issue.

5 CODE DESCRIPTION

The programs required to execute these tests consists of over 9000 lines of ‘C’ code, several Borne-shell and ‘awk’ scripts, plus various configuration and support files. Further details are provided below. The actual code listings are provided in the appendix.

5.1 main()

The main body of the code (6850 lines) includes both test setup and the main test loop. It is embodied in a series of files called mpiotestX.c, where ‘X’ is replaced with the test number of interest. Because of the large amount of similarity in setup and flow among all of these tests (approximately 570 lines of code in common), these files really represent roughly 2200 lines of unique coding. Each test can be described by the following pseudo-code:
declare local variables
initialize MPI
set up I/O communications channel
establish test configuration
set up environment
do “connectathon” tests
    final test set up
    start timing
    loop until count done
        do test as described in appropriate part of §3
    stop timing
    report test time
report overall test status
“clean up” memory and communications
terminate

5.2 Common Functions

All of the tests are supported by a set of common functions “I/O” functions collected into four ‘C’ program files and four header files that total to 2196 lines of code, as described below.

5.2.1 cio.h and cio.c

These files contain code that define a set of basic collective I/O operations. The primary functions are CIO_printf(), CIO_scanf(), CIO_error_check(), and CIO_getenv(). The names are close enough to common ‘C’ and UNIX system calls that their function should be apparent. The code for these functions was loosely adapted from Chapter 8 of Parallel Programming with MPI by Peter Pacheco[17].
5.2.2 fswrapper.h and fswrapper.c

These files provide wrappers for all of the directory, link, filesystem and file operation functions described under §3. The wrappers were initially required because of plans to run tests under HPSS. Since HPSS operations require use of API calls, rather than the system calls to vfs that are possible under NFS and DFS, use of wrappers was intended to allow the use of a common code base (just different compilations). However, a word of caution is in order. Since the problems associated with HPSS have yet to be resolved, none of the HPSS unique code found in these files has been completed, tested, or evaluated; that remains for future work.

5.2.3 mpiosubr.h and mpiosubr.c

These two files provide a parallel implementation for some of the functions found in the subr.h and subr.c files of the Lachman Test Suite. Specifically, the functions provided are:

- **mpiodirtree()** — Builds a directory tree a specified number of levels deep with a specified number of files and subdirectories at each node.
- **mpiormdirtree()** — Removes a directory tree structure. Requires information on the number of and base names used for file and directory entries. Only useful for cleaning up structures built using mpidirtree().
- **mpiotestdir()** — Sets up and moves to a test directory.
- **mpiomtestdir()** — Moves to a test directory.

5.2.4 vsscanf.h and vsscanf.c

While many operating systems provide some variable length argument list calls, the types of calls found are not consistent across different vendor platforms. In an effort to help minimize portability issues with the test code, it was decided early in the project that the code required for nonstandard variable list calls would be provided. As presented in this paper, the test code only used one such a function in order to copy the contents of a buffer into an argument list: vsscanf(). It should be noted that this function is not a complete implementation, but only provides support for string, integer, float and double types. (This func-
tion is an adaptation of some code that the author has had and used for quite some time. Its original provenance is unknown.)

5.3 Common Header Files

Common header files are used to define a number of macros and constants (includes 181 lines of code).

5.3.1 Config.h

This file provides definition of constants related to message logging, file operations, and TRUE/FALSE.

5.3.2 logger.h

This file provides macros for all of the logging routines.

5.3.3 tests.h

This file provides macro and constant definitions for test setup needs.

5.4 Script Files

In order to facilitate systematic execution of the desired set of tests, a set of scripts were developed (260 lines of code), as described below.

5.4.1 mrun.sh

A Bourne-shell compatible script, mrun.sh, was written to control repeated execution of a single test. The script tests for a proper calling sequence, checks for and updates a mpirun configuration file, tests for the required environment variables, sets up the data directory, loops over the test the specified number of times, and then invokes an ‘awk’ script (see below) to calculate test statistics.
5.4.2 runmaster.sh

A Bourne-shell compatible script, runmaster.sh, was used to control execution of the entire suite of tests. It sets the necessary environment variables that determine whether a test is accessing DFS or NFS, and then invokes mrun.sh (see above).

5.4.3 runSummary.awk

An ‘awk’ script used to calculate the mean and standard deviation statistics for any one test set. Called by mrun.sh.

5.5 Miscellaneous Files

Three other files are associated with this project: “Makefile,” “pgfile” (the mpirun machines file[18]), and a “README” file.
ACKNOWLEDGEMENTS

This work was supervised by Dr. David A. Bader of the UNM Department of Electrical and Computer Engineering to meet the requirements of EECE 551/651. This work was conducted to meet the project requirements of the UNM Post-Degree certificate in Scientific and Engineering Computation program and the Sandia National Laboratories Retraining Program in Computational Modeling and Simulation. Thanks go both to Dr. Bader, who was willing to sponsor this project, and to Dr. Richard Allen, who managed Sandia’s program and who gave his full support throughout the several years of my participation in it.
REFERENCES


[13] Local (SNL) modifications made to both kerberos and ssh by Glenn Machin (gmachin@sandia.gov). Ssh version 1.2.27 includes these changes. The as-modified kerberos code was publicly accessible for some time at ftp://lorien.ocf.llnl.gov/pub/mit/kerberos5B2.snla, but this server is evidently no longer up. The kerberos changes dealt with time stamps and pre-authentication from a DCE security server.


[16] Lachman Test Suite source dated 90/01/03.

