# Overview

This document details a plan for validating the Trinity PFS (parallel file system) requirements. For each requirement, a test will be described and its execution documented so that the responsible parties may validate that all requirements are met.

# Section 3.4.1: Global PFS

## The Requirement

The system shall include a closely coupled parallel file system (PFS) that presents a global, consistent name space to the platform.

## The Test Plan

The System Administrators shall verify that the PFS is successfully mounted on all FE (front-end) and compute nodes when the system is booted.

The PFS team shall create a file on the PFS, and then launch a script to be run by every CPU core in the system that verifies that the file exists.

## Execution Instructions

1. Login to a FE.
2. Create a file on the PFS.  
   % rm -f $HOME/file\_existence\_test.tst  
   % touch $HOME/file\_existence\_test.tst  
   % ls -l $HOME/file\_existence\_test.tst
3. Login to the other FEs and verify that the file exists.  
   % ls -l $HOME/file\_existence\_test.tst
4. Run a job on every processor that will check to see if the file exists.  
   % msub run\_file\_exists.sh
5. Inspect the job’s log to ensure that all nodes verified that the file exists. If this command results in a 0 (zero) being output, then all processors verified the file’s existence. Any non-zero output indicates a failure.  
   % grep “ERROR” run\_file\_exists.out.<jobid> | wc -l

# Section 3.4.2: Data Corruption & Loss

## The Requirement

The parallel file system shall be designed to minimize the risk of data corruption and data loss. Reliability of the PFS will be assessed as part of the overall system reliability metrics.

## The Test Plan

The PFS team shall work with the ART (Applications Readiness Team) to continuously run a variety of jobs that fill XX% (defined by Project Management) of the CPU cores for YY days (defined by Project Management). During that time no jobs shall fail due to I/O and all data written shall be verified to be correct.

Cause file system components to fail. The PFS shall either complete the I/O in process during a component failure or result in an error that clearly indicates that the I/O did not complete successfully.

## Execution Instructions

The ART shall use its tools for loading the system with jobs that do regular I/O. The ART shall monitor the output and ensure that no applications fail due to I/O errors to the PFS.

Run a N-N POSIX I/O test that produces the maximum write (and let it do a read too) bandwidth. While this test is running, cause the component failures listed below. The N-N POSIX I/O test shall run at a degraded rate compared to the maximum bandwidth expected or fail with an error message that clearly indicates an I/O failure. After each test, confirm that the system has been returned to a normal functional state. This includes completion of all failbacks and verification that all components are operational.

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_posixio.py

The failures we will create are:

1. Normal shutdown of an MGS
2. Normal shutdown of an MDS
3. Normal shutdown of an OSS
4. Normal shutdown of an LNet Router.
5. Abnormal shutdown via power loss of an MGS
6. Abnormal shutdown via power loss of an MDS
7. Abnormal shutdown via power loss of an OSS
8. Forced shutdown of an LNet Router
9. Disconnect a network connection (assumes redundancy) between:
10. MGS and switch
    1. Ethernet
    2. Infiniband
11. MDS and switch.
    1. Ethernet
    2. Infiniband
12. OSS and switch.
13. LNet Router and switch.
14. Shutdown one of the two IB switches.
15. Remove 2 HDDs
16. Remove 1 SSD
17. mgmt00 to mgmt01 failover and failback
18. MDS Failure via unmount of all OSTs across all OSS nodes

## Special Notes

Loss of one IB switch should result in no data loss.

# Section 3.4.3: I/O Performance

## The Requirement

The PFS shall achieve the target bandwidth, as specified in Performance Requirements using any arbitrary collection of compute nodes starting at 10% of compute nodes up to the full scale of the system when the PFS is up to 70% full.

### Performance Requirements

Phase 2 - Aggregate main memory is 2 PB

Disk Capacity is 39 PB per file system with two file systems.

Phase 2 - Maximum PFS dump time for 80% of main memory shall be 20 minutes, thus 1.33 TB/s (1.21 GiB/s).

Phase 1 - 114 LNET routers out of the total 222 currently installed, therefore Phase 1 requirement is ½ of the Phase 2 expected bandwidth – 665 GB/s (605 GiB/s).

## The Test Plan

As stated above, the file system is actually comprised of two, equally sized file systems that share the same LNET routers. Phase one has a reduced amount of LNET routers (roughly ½ of total for phase two) and thus reduced number of links to the Sonexion File System.

Conduct a N-N POSIX I/O experiment varying the number writers and readers per OST. Use a large transfer size, 64 MiB. Let each experiment write for 3 minutes. Find the writer and reader count per OST that yields the maximum bandwidth. Do this on each file system independently to verify that maximum bandwidth is achieved using the same parameters on each file system.

Run a N-N POSIX I/O on each file system concurrently using the setup that yields maximum bandwidth that writes 0.8 PB (40% of memory). Verify that the test finishes in less than 20 minutes and consequently achieves a total bandwidth no less than the required 665 GB/s (605 GiB/s).

## Execution Instructions

The first order of business is to find the N-N POSIX I/O configuration that yields the maximum write and read bandwidths on each file system. To do this, use the Experiment Management framework to execute Python scripts. One script sweeps from one to sixteen writers/readers per OST and one writer/reader per node. The other does the same sweep, but with two writers/readers per node.

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_wr\_sweep\_1ppn\_fs\_1.py

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_wr\_sweep\_1ppn\_fs\_2.py

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_rd\_sweep\_1ppn\_fs\_1.py

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_rd\_sweep\_1ppn\_fs\_2.py

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 2 -c experiments/ior\_cray\_optimal\_wr\_sweep\_2ppn\_fs\_1.py

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 2 -c experiments/ior\_cray\_optimal\_wr\_sweep\_2ppn\_fs\_2.py

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 2 -c experiments/ior\_cray\_optimal\_rd\_sweep\_2ppn\_fs\_1.py

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 2 -c experiments/ior\_cray\_optimal\_rd\_sweep\_2ppn\_fs\_2.py

Using the parameters from the maximum write and read bandwidths, run a job that targets both file systems concurrently and writes a total of 0.8 PB. Verify it finishes in less than 20 minutes.

# Section 3.4.4: Metadata & Interactive Response

## The Requirement

The PFS shall provide a robust, interactive environment for users. The time required to perform an insert, delete, and enumerate file system object status within a single directory on login or file transfer service nodes shall be prompt and not be substantially impacted by unrelated applications running on the compute partition. Given a single directory with 1 million files (1,000,000), the Trinity file system shall:

* Insert one million objects in 6 seconds.
* Delete one million objects in 5 seconds.
* Enumerate one million objects in 4 seconds.

This requirement matches the phase two requirement because all LNET metadata capability is present in phase one.

## The Test Plan

This requirement has two parts: metadata and interactive response time. These two parts will be tested separately. The file system is actually comprised of two, equally sized file systems that share the same LNET routers. For this reason, the plan will state running tests on each file systems concurrently.

### Metadata Tests

Run a scaling study of file creation and deletion tests where each CPU core creates and deletes its share of 1,000,000 files and each CPU core creates and deleted files in its own directories. These directories are evenly spread amongst the MDTs (DNE Phase 1 configuration). Target both file systems simultaneously.

Run a scaling study of file creation and deletion tests where each CPU core creates and deletes its share of 1,000,000 files and each CPU core creates and deletes files in a single directory. Target both file systems simultaneously.

## Execution Instructions

The metadata performance will be assessed over the scale of 1 (one) to maximum number of processors.

When running mdtest one specifies the number of files each process will create and not the total number of files to create.

The exact invocation of the code depends on one’s MPI implementation. To test create/stat/remove of 1,000,000 files where each process does its share of the files in its own directory (DNE Phase 1 configuration), run one of the following commands concurrently on each file system. Note that as of this writing, lfs commands are not available on the compute nodes so mdtest can not create DNE directories within the program. For this reason, use one of two methods described below to run mdtest.

**LFS Available on Compute Node**

aprun –n <#pes> -N <#pes-per-node> ./mdtest -n <1000000/#pes> -d <path-to-pfs>/<nn\_unique-dir> -F -C -T -r -N <#pes-per-node> -M –u

**LFS Not Available on Compute Node**

Create directories per DNE as follows on each file system:

lfs mkdir –i 0 path-to-pfs/nn\_unique-dir/mdt\_0

lfs mkdir –i 1 path-to-pfs/nn\_unique-dir/mdt\_1

lfs mkdir –i 2 path-to-pfs/nn\_unique-dir/mdt\_2

lfs mkdir –i 3 path-to-pfs/nn\_unique-dir/mdt\_3

lfs mkdir –i 4 path-to-pfs/nn\_unique-dir/mdt\_4

aprun –n <#pes> -N <#pes-per-node> ./mdtest -n <1000000/#pes> -d <dne\_path> -F -C -T -r –u

where dne\_path is defined as follows:

path-to-pfs/nn\_unique-dir/mdt\_0@ path-to-pfs/nn\_unique-dir/mdt\_1@ path-to-pfs/nn\_unique-dir/mdt\_2@ path-to-pfs/nn\_unique-dir/mdt\_3@ path-to-pfs/nn\_unique-dir/mdt\_4

Verify that the performance meets the 1,000,000 (1 million) file creates in 6 seconds, enumeration in 4 seconds, and file deletes in 5 seconds. If results of this test do not meet the requirements, the directory structure may need to be changed based on balancing loads on metadata servers.

The exact invocation of the code depends on one’s MPI implementation. To test create/stat/remove 1,000,000 files where each process does its share of the files in a single directory, run this command concurrently on each file system.

aprun –n <#pes> -N <#pes-per-node> ./mdtest -n <1000000/#pes> -d <path-to-pfs>/<nn\_shared-dir> -F -C -T -r -N

Record the performance of both tests. We do not expect this test to meet the 1000000 (1 million) file creates in 6 seconds, in 4 seconds, and file deletes in 5 seconds. This is a baseline test to show 1 MDS performance capability.

### Contention Tests

Run two N-N POSIX I/O tests that produce the maximum write (and let it do a read too) bandwidth at the same time. The sum of bandwidth achieved by the competing file system clients should approximately the same as when a single file system client has the file system dedicated to it.

Create a compressed tar file (.tgz, created with tar cvfz) that contains ~10,000 files totaling ~100 MB to be used in these tests.

Time how long it takes to untar this file into a directory using tar xvfz.

Time how long it takes to copy the file tree that was untarred, using cp -r.

Time how long it takes to delete both file trees, using rm -rf.

Run a N-N POSIX I/O test that produces the maximum write (and let it do a read too) bandwidth. While this test is running, time how long it takes to untar the file (tar xvfz), copy the file tree (cp -r), and delete both file trees (rm -rf). The N-N POSIX I/O test shall run at near the maximum bandwidth expected. The interactive operations on the compressed tar file shall not take longer than 10x what they took when no N-N POSIX I/O test was running.

## Execution Instructions

1. Run two N-N POSIX I/O tests that produce the maximum write (and let it do a read too) bandwidth at the same time. The sum of bandwidth achieved by the competing file system clients should approximately the same as when a single file system client has the file system dedicated to it.   
     
   % ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_possixio.py  
   % ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_posixio.py
2. Set your current working directory to the parallel file system in a directory on the same volume where the IOR program will write its output and then read it back in.  
     
   % cd <path-to-pfs>/interactive
3. Gather data on the performance of interactive file system operations before anything else is running. The tar file used should be ~300 MB in size and have ~15,000 files. The sequence is:  
     
   % time tar xvfz <tarball>.tgz  
   % time cp –r <tarball-dir> <tarball-dir>\_dup  
   % time rm –rf <tarball-dir> <tarball-dir>\_dup
4. Run one N-N POSIX I/O test that produces the maximum write (and let it do a read too) bandwidth.  
     
   % ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal.py
5. Wait for IOR to indicate that it is writing the files. Repeat the sequence outlined in #3.
6. Wait for IOR to indicate that it is reading the files. Repeat the sequence outlined in #3.

# Section 3.4.5: POSIX/IO Versus MPI/IO Performance

## The Requirement

The system shall provide POSIX I/O and MPI-2 I/O functionality that is tightly integrated with file system software to provide high performance small and large block I/O and for single and shared files. MPI I/O shared file performance shall achieve 80% of POSIX I/O performance using a single file per processor at full system bandwidth.

## The Test Plan

Run a N-1 MPI/IO striped over all OSTs. Show that the MPI/IO performance was at least 80% of the N-N POSIX I/O performance.

## Execution Instructions

Run with the number of processes for optimal N-N POSIX/IO, but create a shared file (N-1) striped over all OSTs using MPI/IO.

% ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_n1\_optimal\_mpiio.py

# Section 3.4.6: External Versus Internal Clients

## The Requirement

The PFS system shall support access to external clients with the same functionality as internal clients.

Doug Doerfler clarified this requirement with this statement, “The intent of requirement 3.4.6 is to ensure that non-compute nodes, e.g., the FTAs, have the same functionality when accessing the PFS as the main platform. The FTAs and potentially external login nodes are the only ‘external’ clients I anticipate having on Trinity. Table 3 specifies 100 GB/s ‘External client bandwidth to/from local PFS’.”

## The Test Plan

The FTAs will not be deployed fully for phase one so this test will be executed in phase two only.

# Section 3.4.7: File System Maintenance Operations

## The Requirement

Define the expected PFS maintenance procedures and their impacts on the PFS performance under normal load and other routine operations including purging, file system health monitoring, performance statistics, problem alerts, diagnosis and repair, and reconstruction after a drive replacement, including reconstruction time.

## The Test Plan

Run a N-N I/O test at the CPU core count and write size that produces the maximum bandwidth. While this test is running, use the health monitoring system to investigate alerts, performance statistics, etc. Measure any reduction in expected performance.

Run a N-N I/O test at the CPU core count and write size that produces the maximum bandwidth. While this test is running, run the PFS purger utility. Measure any reduction in expected performance.

Run a N-N I/O test at the CPU core count and write size that produces the maximum bandwidth. While this test is running, use the archive utility on the FTAs to move files from the PFS to the archive. Measure any reduction in expected performance.

Run a N-N I/O test at the CPU core count and write size that produces the maximum bandwidth. While this test is running, take a disk offline and replace it. Allow reconstruction to take place. Measure any reduction in expected performance.

## Execution Instructions

1. Run the optimal performance test.  
     
   % ./run\_expr.py -w 3600 -d list -M '-j oe -V' -p 1 -c experiments/ior\_cray\_optimal\_posixio.py
2. At the same time, use the health monitoring system to investigate alerts, performance statistics, etc. Measure any reduction in expected performance. Determine if was more than a 10% reduction in the optimal performance.
3. Repeat #1.
4. At the same time, run the PFS purger utility. Measure any reduction in expected performance. Determine if was more than a 10% reduction in the optimal performance.
5. Repeat #1.
6. At the same time, use the archive utility on the FTAs to move files from the PFS to the archive. Measure any reduction in expected performance. Determine if was more than a 10% reduction in the optimal performance.