Parallel File System Testing for the Lunatic Fringe: the care and feeding of restless I/O Power Users*

Richard Hedges *richard-hedges@llnl.gov*
Bill Loewe *wel@llnl.gov*
Tyce McLarty *tmclarty@llnl.gov*
Chris Morrone *morrone@llnl.gov*

Scalable I/O Project
Lawrence Livermore National Laboratory

Abstract

Over the last several years there has been a major thrust at the Lawrence Livermore National Laboratory toward building extremely large scale computing clusters based on open source software and commodity hardware. On the storage front, our efforts have focused upon the development of the Lustre[1] file system and bringing it into production in our computer center. Given our customers' requirements, it is assured that we will be living on the bleeding edge with this file system software as we press it into production. A further reality is that our partners are not able to duplicate the scale of systems as required for these testing purposes. For these practical reasons, the onus for file system testing at scale has fallen largely upon us. As an integral part of our testing efforts, we have developed programs for stress and performance testing of parallel file systems. This paper focuses on these unique test programs and upon how we apply them to understand the usage and failure modes of such large-scale parallel file systems.

1. Introduction to the Lunatic Fringe

1.1. ASC Computing Requirements

The FY1993 Energy and Water Authorization Bill, signed in October 1992 by President Bush established a moratorium on nuclear testing. President Clinton extended that moratorium in July 1993. Furthermore, the U.S. decided to halt the production of new nuclear weapons.

To implement these policy decisions, the Stockpile Stewardship program was established by the Department of Energy and managed by the NNSA. The goal of the program is to provide scientists and engineers with the technical capabilities to maintain the nuclear weapon stockpile without the two key tools that had been applied for 50 years: (1) underground testing and (2) modernization through the development of new weapon systems.

This reality meant that a new and powerful role for modeling and simulation was required: The Advanced Simulation and Computing program (Formerly known as the Accelerated Strategic Computing Initiative, ASCI) was established in 1996 to develop this capability. ASC is creating these simulation capabilities using advanced codes and high-performance computing that incorporates more complete scientific models based on experimental results, past tests and theory.

To meet these needs in the year 2005 and beyond, ASC is solving progressively more difficult problems as we move further away from physical testing. Applications must achieve higher resolution, high fidelity, three-dimensional physics, and full-system modeling capabilities to reduce reliance on empirical judgments. This level of simulation requires computing at ever increasing levels of performance. Therefore, ASC collaborates with industry to accelerate development of more powerful computing systems and invests in creating the necessary software environment.

Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)
0-7695-2318-8/05 $20.00 © 2005 IEEE

Let's take a quick look at the high-level system specifications for the 2005 incarnation of ASC platform (i.e. ASC Purple):

- 100 teraflops
- 50 TB memory
- 2 PB RAID disk
- 100 GB/sec I/O delivered to single app

Combining these specifications with highlights of our local programming and usage model gives one a sense of life as a user on one of our present day systems.

- Many (~1000) shared memory nodes of modest local scale (2 – 16 processors)
- Individual jobs running on hundreds to a few thousand processors
- Primary programming model is distributed memory (MPI communications) with threads, openMP, shared memory communications secondary
- All nodes mount a shared parallel file system: codes can perform parallel I/O to a single shared file or a file per process in a single directory
- Codes written in C, C++, FORTRAN
- Writes dominate read by about a 5 to 1 ratio

Since this paper focuses upon the I/O component of the system, we need to be more precise about the meaning of a few terms. For purposes of the present discussion, we define "parallel file system" as a shared disk-based system, supporting a single namespace, mounted across many compute nodes, and capable of supporting concurrent transfers from all of the nodes to either a single shared file or to multiple (shared or private to each process) files. Our working definition of "parallel I/O" is the set of I/O operations executed by a single parallel program executing on a cluster mounting the parallel file system.

1.2. Livermore Linux Strategy

There are many research activities at the Livermore laboratory other than those involved directly in stockpile stewardship. The Laboratory's multiprogrammatic and institutional computing (M&IC) initiative [12] was conceived to bring cost-effective, high performance computing services to these LLNL programs and scientists. M&IC was created to enable all programs to benefit from the large investment being made by the DOE Advanced Simulation and Computing Program at LLNL by providing a mechanism to leverage resources and lessons learned. This led us to a technology development strategy that has been an adaptation of the ASC platform strategy:

- Replicate the ASCI programming and usage model
- Take on risks and responsibilities as a system integrator of components
- Base architecture on commodity hardware
- Utilize partners for all key components (compute nodes, cluster network, RAID storage)
- Use Open Source (i.e. Linux) software components as much as possible
- Develop Linux components and enhancements as required utilizing in-house teams and external partners

To replicate the ASC model, Livermore's Linux Project identified and addressed four areas of technical enhancement:

- High performance interconnect
- Resource manager
- Administrative tools

and

⇒ **Scalable, parallel file system**

In order to achieve the I/O requirements and the related components of the Livermore usage model, we are collaborating with other DOE National Laboratories at Los Alamos and Sandia. The three laboratories are frequently referred to as the Tri-Labs in a number of cooperative efforts. The Tri-Labs are working with industrial partners Cluster File Systems, Inc., Hewlett-Packard, and Intel to develop and bring the Lustre file system into production.

1.3. Broader Livermore I/O Strategy

In addition to the bandwidth requirements for ASC scale computing (estimated at 1 GB/sec per teraflop of computing performance), the Livermore Computing vision includes a usage model where an enterprise wide file system is mounted across all platforms of interest to our computational scientists (to minimize the need for multiple copies or transfers of files, for example). Implementation of this vision requires a high performance, site wide, global file system accessible to the compute clusters, visualization clusters, archival systems and potentially even each individual scientist's workstation.

2. Lustre

The name "Lustre" is a contraction of "Linux" and "Clusters". Lustre is a novel file system architecture and

implementation suitable for very large clusters typified by the assortment that we have in the Livermore Computing Center (See appendix I). Lustre is designed, developed and maintained by Cluster File Systems, Inc.

The central target in this project is the development of is a next-generation object-based cluster file system which can serve clusters with 10,000's of nodes, petabytes of storage, move 100's of GB/sec with state of the art security and management infrastructure.

Lustre is in use on many of the largest Linux clusters in the world and is included by partners as a core component of their cluster offering, such as in the HP SFS product.

The latest version of Lustre is available from Cluster File Systems, Inc. Public Open Source releases of Lustre are made under the GNU General Public License. The 1.0 release of Lustre is now available.

2.1. Object Based File Systems

A key advantage of Object Storage Devices (OSDs) in a high-performance environment is the ability to delegate low-level allocation and synchronization for a given segment of data to the device on which it is stored, leaving the file system to decide only on which OSD a given segment should be placed. Since this decision is quite simple and allows massive parallelism, each OSD need only manage concurrency locally, allowing a file system built from thousands of OSDs to achieve massively parallel data transfers [13].

Further aspects of this design include:

- Separates I/O and metadata functions
- Puts I/O functions in (OSD's) on a network
- Eliminates need for File Server nodes
- Reduces the size and complexity of the O/S dependent file system client

2.2. The Lustre Architecture

The Lustre architecture provides significant advantages over previous distributed file systems. Lustre runs on commodity hardware, using object based storage and separate metadata servers to isolate these functions and improve scalability.

Replicated metadata servers (**MDS**) with failover maintain a transactional record of high-level file and file system changes. The many Object Storage Targets (**OSTs**) are responsible for actual file system I/O and interfacing with storage devices. Lustre supports strong file and metadata locking semantics to maintain coherency even for situations of concurrent access. File locking is distributed, with each OST handling locks for the objects that it stores.

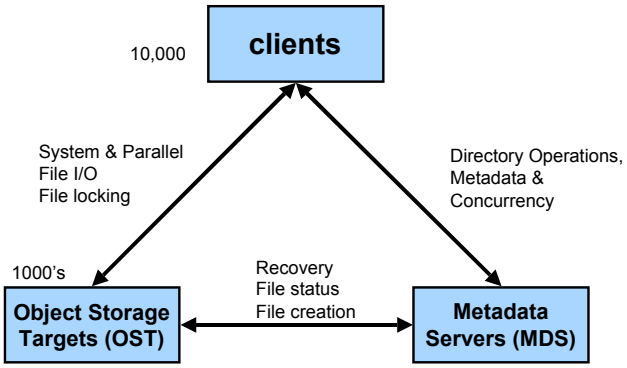

Figure 1. Lustre components and control

3. Lustre Testing Background

3.1. Testing efforts by CFS, HP and LLNL

There are several quasi-independent testing efforts of the Lustre file system under way. We are aware or involved to various degrees on efforts at CFS, Inc. and at HP in additional to the Tri-Lab efforts.

At CFS, daily Lustre testing as an integrated part of the development process is described in their project documentation on Lustre testing [14]. *ltest* is the suite of programs enabling automation of the routine CFS testing process. A complementary facility, *buffalo*, is a web-based interface allowing browsing both the recent and historical results of various tests.

Another formal testing process is directed by our partner, Hewlett-Packard, and is focused upon specific feature and performance deliverables of the Path Forward collaboration. Included among these testing efforts are performance milestones for metadata performance, security testing, and relevant portions of the POSIX test suites.

We mentioned in the abstract that our efforts are focused upon testing at scale on the many large clusters here at the Livermore Laboratory. While that is true, it is more accurate to say that our efforts are focused upon delivery of the high quality of file system service to our customers.

3.2. Related Work on File System Testing

At the inception of the Lustre efforts at LLNL, we surveyed existing file system and I/O test programs for their potential utility for our testing. We found a broad range of tools, for example: *Bonnie++* [3], *dbench* [4], *fsx* [5], and *IOzone* [6]. They are interesting and useful and are still used as minor elements of our present testing. The major shortcoming of these, and others, is that they

are not designed with much sense of coordinated or parallel I/O in mind and certainly not for the scale of systems of interest to us. *IOzone* is the lone exception since it can be run with multiple threads on a single processor.

Beyond these file system tests, there are other I/O tests and benchmarks, which like ours, intend to address issues peculiar to a specific workload or application. These include for example, The *SPEC SFS* (formerly *LADDIS*) benchmark, targeted for NFS servers, the *SPECweb99* benchmark for www servers, and the *PostMark* File system Benchmark, designed to probe the "ephemeral small-file regime" crucial to internet services such as Email, netnews, and general web interactions. The totally different types of workloads that these tests target are not like ours, nor do they probe the parallel file system characteristics.

Some closely related work by the GUPFS team at LBNL [18] includes efforts to evaluate parallel file system hardware and software and they have also developed throughput and metadata performance tests.

In the 1994 paper by Chen and Patterson [19], they suggest that the "Ideal" I/O benchmark would posses the following characteristics:

- Spends most of its time doing I/O
- Scales gracefully over a wide range of current and future machines
- Allows fair comparison across machines
- Relevant to a wide range of applications
- Helps designers to understand why the system behaves as it does

They go on to propose and explore a testing regimen, and apply it to some systems of the day. Our present testing is first for correctness and then for performance, yet their ideals still apply. Adapted in meaning to our particular purposes, these are excellent objectives for our present correctness testing. As the file system matures, we expect that our testing focus will shift to understanding the complex performance profiles.

4. Lunatic Fringe Parallel File System Tests

Before discussing in detail the codes that we have developed, it is a good point to summarize the details of our computing environment, our local programming and usage model, plus something about the nature of our application mix.

The bulk of the compute cycles in our center are consumed by scientific codes simulating various physical processes, often in three dimensions. These simulations may run on hundreds to thousands of processors. Simulations may consume a few hours to a few months of elapsed run time. This may represent many years of cpu time. Users do not have the luxury of continuous running time, so will periodically write a file or set of files which contain the state of the simulation in sufficient detail and form that they can be read to resume the simulation. These are referred to as "restart files". Files called "plot files" with a more concise set of state information may be written periodically with information for physical analysis and display.

The I/O patterns of these codes fall primarily into one of two basic models: 1) For many programs, each process writes independently to its own private set of files; and 2) For some programs, processes write concurrently to a file shared by all of the processes. We refer to these as "file per process" and "shared file" respectively.

Most codes are written in C, C++, or FORTRAN. The MPI (Message Passing Interface) programming model and variants of it are the basis for almost all code development of parallel programs in our center [7]. MPI provides the mechanisms to communicate between and synchronize processes executing on the processors of a computing cluster with distributed memory.

For our parallel I/O test programs, we have used the MPI programming model for several reasons. Primarily, MPI allows for synchronized I/O for large or small task counts. Second, there are many conveniences offered by the MPI parallel environment we use to simplify testing logistics. Finally we believe that the MPI coordinated tests can simulate workloads with greater instantaneous loading of the file system due to synchronization of the I/O operations. These synchronized tests more readily shake out locking issues and race conditions in the file system.

Finally, in keeping with our theme of Open Source development, all of the codes are Open Source and freely available.

4.1. IOR Test Code

IOR is a file system bandwidth testing code with a long history of use and development at LLNL [8]. It was initially developed to test GPFS [20] from IBM on the ASCI Blue Pacific and White machines and has been previously applied and discussed at this conference [21]. *IOR* was first designed for testing bandwidth performance of parallel file systems using various interfaces and access patterns. The supported interfaces and patterns attempt to represent the usage patterns of ASC applications. In our arsenal of testing tools, *IOR* is the most heavily used, in particular for situations of a high, sustained I/O load on a parallel file system.

IOR has the capability for either "shared file" or "file per process" operation. At present, most user codes employ the file per process I/O model (fig. 2). We have been encouraging the use of the shared file model to alleviate logistical and technical issues of dealing with

large sets of files, particularly as the node counts of our systems rise. The shared file model (figs. 3,4) presents challenges as well, in particular for the file system. Thus, we require that *IOR* offer both access modes as options to test file system throughput.

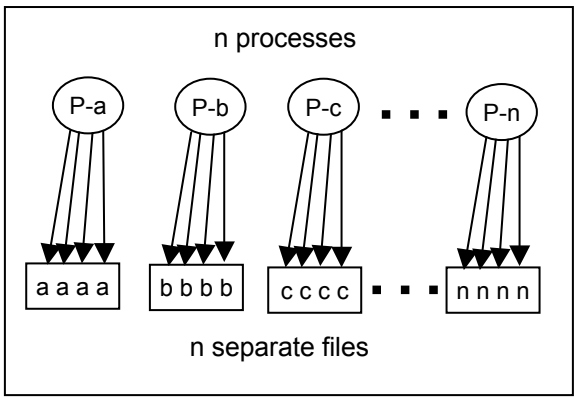

Figure 2. File Per Process I/O Model

IOR checks for data correctness and reports incidences, patterns, and locations of errors, should they occur. After writing or reading, a file check may be performed on the file to determine any error during the write or read phase. To prevent skewing of the performance results, the write and read data checks are not performed during the data transfer performance measurements. Further, to assure that stored rather than cached data is checked, there is an option that allows a different processor to perform the check than had performed the data transfer. The application of *IOR* utilizing these checks has been particularly valuable in detecting and debugging some rather obscure race conditions in the file system.

Initially named for the acronym from 'interleaved or random', *IOR* supports access patterns to a shared file which are periodic with respect to the set of MPI processes. At present, support for "random" patterns has been dropped, as they were found to be difficult to maintain in the test code while adding little practical coverage for the present file system test protocol.

IOR supports two general data layout patterns in the shared file I/O model: segmented and strided (or interleaved). The primary distinction between the two patterns is whether an MPI task's data is contiguous or noncontiguous in the file.

For the "segmented" pattern (fig. 3), each process (*P-a* and *P-b*, e.g.) stores its blocks of data in a contiguous region in the file. This pattern is generally less challenging to the parallel file systems that we are familiar with as potentially larger data buffers can be transferred and with fewer requests/revocations of locks for byte-ranges in the file.

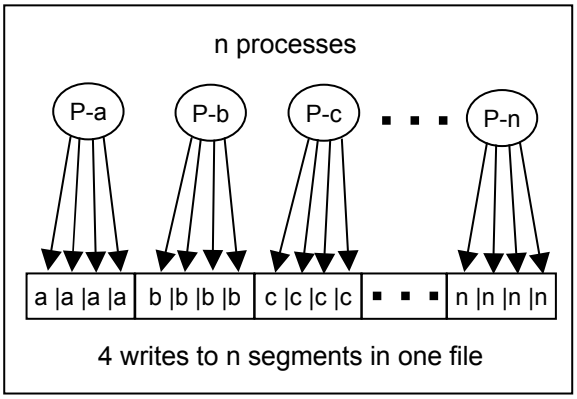

Figure 3. Segmented Access Pattern: Shared File I/O Model

With the "strided" access pattern (fig. 4), each task's data blocks are spread out through a file, interleaved by process rank. While this pattern may be more natural for storing a multi-dimensional mesh, it is usually inefficient from the viewpoint of the file system due to additional locking requirements and smaller data transfers.

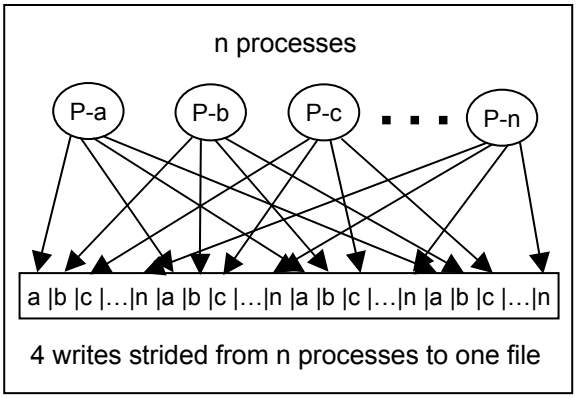

Figure 4. Strided Access Pattern: Shared File I/O Model

Transfer sizes as well as the number of transfers that a process writes into a contiguous portion of the file can be specified for test code operation. This combination of adjustable parameters allows for continuous tuning of the test from flat out bulk I/O transfers for large contiguous transfers in the segmented pattern to file system interaction being dominated by lock acquisition and revocations for short transfers, interleaved in the file

according to task rank. For further detail of runtime options for IOR, please refer to Appendix II.

4.2. simul Test Code

simul was developed to test simultaneous file system operations other than data transfers from many nodes and processes [10]. It is composed of simultaneous file system calls and calls to library functions to probe the correctness and coherence of parallel file system operation. The exact state of the file system is checked after each set of operations to determine if the commands were all executed correctly. A set of MPI tasks executes standard POSIX operations on either shared or private files.

There are forty individual tests that *simul* can perform to exercise simultaneous metadata operations [see appendix III]. For both the concurrent and independent file access modes, there are operations to both files and directories. For files, the operations of open, close, file stat, lseek, read, write, unlink, rename, creat, truncate, symlink, readlink, link to a single (shared) file, and link to file-per-process are performed. For directories, the chdir, directory stat, readdir, mkdir, and rmdir operations are tested. *simul* provides a high instantaneous load in metadata operations (depending upon the number of tasks), providing a rigorous test of a parallel file system functions other than data movement.

4.3. mdtest Test Code

mdtest allows multiple MPI tasks to create, stat, and remove shared or private files and directories[9]. The rate of these operations is measured, calculated and reported by *mdtest*. This information has been used to determine bottlenecks in the important area of metadata performance as a growing file system performance issue.

mdtest has many options [see appendix IV]. For example it has the option to perform its tests with tasks working in shared or private directories. The number of files or directories that each task accesses may be set, allowing one to simulate a range of behaviors for various application code situations. It has an option to prevent reading locally cached file or directory information when performing a stat on a newly-created file by using a 'read-your-neighbor' approach.

4.4. Other tests

Beyond the "general purpose" tests detailed above, we are sometimes called upon to create "one off" tests to reproduce specific user encountered problems in a more focused way. An example is as follows: A user was seeing intermittent errors in his restart files which were composed of periodic unformatted writes of six arrays from a single write() in FORTRAN90 code. Iterating back and forth with the user, a short test code (in FORTRAN90) consisting of the I/O kernel and some data checking was developed, and subsequently demonstrated the problem seen by the user. Tests such as this are integrated into the testing protocol as long as they are relevant, and become candidates for more general-purpose tests. This particular example will be further discussed later.

5. Case studies

This section will provide some more specific examples demonstrating how the test codes we have developed are used in practice for our quality assurance efforts, customer support, and operational planning.

5.1. Sanity check of new Lustre version

As each new version of Lustre is installed on one of out tests systems, it is immediately subjected to at least one round of *IOR* and one of *simul* at scale. We have found this to be a good initial check for the new file system software and for the process of installing and bringing it up.

5.2. File system performance profile

IOR was initially designed as a benchmark for file system throughput and is used here to develop an overview of file system performance. In Figure 5, we display a concise study of throughput on the Lustre file system of *Thunder*, presently our largest Linux cluster.

Thunder is comprised of 1024 quad 1.4 Ghz Itanium Madison Tiger4 nodes with 8.0 GB DDR266 SDRAM. There are 64 Object Storage targets fronting 8 couplets (racks) of DataDirect Network's S2A8500 Silicon Storage Systems totaling about 190 TB of storage with potential throughput approaching 8 GB/sec. In practice, performance is network limited for our present configuration, but we have sufficient GigaByte Ethernet connections to support approximately 6.4 GB/sec peak throughput to a single Lustre file system.

For the study represented by Figure 5, the transfer size of 64K bytes was used throughout and was selected because that is the Lustre stripe size. Each client writes 2.5 GB so that performance boosts due to caching are reasonably well eliminated. Our intent is to measure the throughput that can be sustained in a steady state, i.e. the rates for asymptotically large files.

For this study, the file per process model is expected to achieve the highest throughput rate because of the absence of overhead for locking. For the shared file model, the segmented case will have all transfers from an individual client to contiguous locations within the file.

In the strided case, segments of 16 contiguous transfers are interleaved among clients, leading to a segment size of 1MB, which was chosen to equal to the RPC size.

For the file per process mode, the files are striped across 4 OSTs, which is the default setting for this file system. For the shared file tests (segmented and strided) we use maximal striping across all 64 OSTs of the installation. While this may not be optimal for smaller client counts, it is for the large client counts.

I/O throughput rates are measured and the number of clients is increased from one to 128 in powers of 2. Three iterations of each individual test are performed, and the maximum measured throughput is reported.

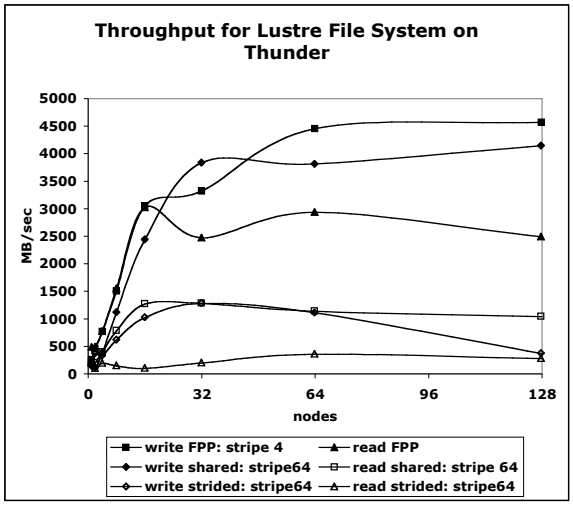

Figure 5. Performance Profile

Throughput rates for writes and reads using *IOR* for the three transfer patterns as number of clients is varied in powers of 2. File per process tests striped to 4 OSTs, Shared file tests striped to all 64 OSTs.

Our rational for selecting the best of a series of performance measurements has been the subject of debate within our team through the years. What we would like, in principle, is to present the mean and standard deviation for each set of test runs executed on a completely dedicated system: That is with no other workload of any sort taking place on the system. On these large systems however, we are only very rarely allowed the luxury of a machine entirely dedicated to our testing. Most often, we are running our tests as a component of the system's routine workload. It is possible for any individual test iteration to have other simultaneously running codes competing for the same I/O resources. In practice, this is apparent in terms of reduced performance observation for a test iteration. When we select the highest value, we assume that we achieved that great rate because there was no competing load during that run, and conclude that is a more accurate measure of the underlying capability of the system than an averaged value including some performance measurements suppressed by other workload competing for the file system resource.

Here are some observations:

- Performance is quite excellent for our dominant I/O situation, i.e. writing in the file per process mode.
- Achieve asymptotic rates with ~64 clients for system with 64 OSTs expected
- Write rates exceed read rate for every pattern.
- We were surprised how good the performance was for write shared-segmented.
- One might be initially disappointed with the share-strided performance. However, keep in mind that MPI-IO can be applied to convert this pattern at the application level into the shared-segmented pattern via a communication phase. It is likely that Lustre's read-ahead has hurt performance in this case, though it would help the other cases where there are large contiguous sections being read by the same client.

5.3. Daily automated testing

We currently test on a near continuous basis on 450 nodes (900 processors) of one of our large Linux clusters and at smaller scale on several other clusters. When testing at this scale on this portion of the cluster, we are usually running Lustre file system software which we believe to be the "latest stable" software: It is usually a known baseline release with fixes to problems that we have been focusing on, and has seen significant testing on the smaller test clusters mentioned.

The following is the output from recent testing period:

```
For the period beginning Wed, Dec 08, 2004, at 17:00 and ending Thu, Dec
09, 2004, at 08:00

kernel:  2.4.21-p4smp-76chaos
lustre:
1.2.8.3-19691231160000-PRISTINE-.usr.src.linux-2.4.21-p4smp-76chaos

General results:
----------------
atomicity   :  No tests run
bonnie      :  No tests run
cabot       :  Passed   1 of   2          50%
      test 39506: exited abnormally
dbench      :  Passed   5 of   5         100%
fsx         :  No tests run
ior         :  Passed  63 of  63          100%
iozone      :  No tests run
mdtest      :  Passed   1 of   1         100%
prodcon     :  No tests run
simul       :  Passed  13 of  13          100%
sppm        :  No tests run
```

IOR performance results:

File per process (locking on):

tasks (CPUs)	stripe ct	stripe size	xfer size	bytes/ task	rates (MB/s) write(%dev)(%opt)	rates (MB/s) read(%dev)(%opt)	sample count
450(1)	32	4M	32M	256M	2645(5)(44)	1259(15)(21)	2
450(1)	1	1M	1M	256M	1889(34)(32)	2301(2)(39)	2
450(1)	2	1M	2M	256M	1815(14)(30)	2167(0)(36)	2
450(1)	4	16M	32M	256M	2557(0)(43)	1756(0)(29)	1
450(1)	8	16M	32M	256M	2383(0)(40)	1350(0)(23)	1
450(1)	16	16M	32M	256M	2538(0)(43)	1689(0)(28)	1
450(1)	1	1M	2k	256M	2354(0)(40)	1488(0)(25)	1
450(1)	8	1M	8M	256M	2489(0)(42)	1360(0)(23)	1
450(1)	16	1M	16M	256M	1367(0)(23)	1228(0)(21)	1
450(1)	32	1M	32M	256M	2730(0)(46)	1231(0)(21)	1

Single file (locking on):

tasks (CPUs)	stripe ct	stripe size	xfer size	bytes/ task	rates (MB/s) write(%dev)(%opt)	rates (MB/s) read(%dev)(%opt)	sample count
450(1)	32	256k	8M	256M	2138(2)(36)	1092(1)(18)	2
450(1)	32	512k	16M	256M	1748(27)(29)	1177(1)(20)	4
450(1)	32	4M	32M	256M	1985(8)(33)	1309(3)(22)	4
900(2)	32	512k	16M	256M	2267(1)(38)	1023(1)(17)	2
900(2)	32	4M	32M	256M	1999(14)(34)	1119(2)(19)	3
450(1)	2	1M	2M	256M	209(2)(56)	78(9)(21)	2
900(2)	32	1M	32M	256M	2314(0)(39)	1061(0)(18)	1
900(2)	32	2M	32M	256M	1915(0)(32)	1093(0)(18)	1
900(2)	32	8M	32M	256M	961(0)(16)	939(0)(16)	1

In this example report, several different tests were run to check a variety of possible problem areas. The three codes that are the bulk of the present discussion (*IOR, simul, mdtest)* generally form the bulk of our daily testing, with IOR dominating. Parameters for *IOR* were selected for both the shared file and file per process I/O models to monitor changes in performance as Lustre patches may be applied. Five iterations of dbench [4] are included, testing for previously observed defects. The test entitled "cabot" is a one-off test code representing the problem discussed in detail in the next section.

This type of report in generated on a daily basis for each of the several systems being used for Lustre testing. Several problems, particularly those that are related to large scale have been discovered in this stage of testing. For the subtle types of race condition issues that can occur, repeated automated testing allows us to characterize the prevalence of such problems (i.e. the odds of being caught on the wrong end of the race) and assess the level of risk for our user community.

5.4. User reports data corruption on Thunder

Perhaps nothing gets the attention of our local Lustre team more quickly than the phrase "data corruption". From one of our most esteemed users:

> I was able to complete a couple of "identical" runs early this morning to test reproducibility. As you are supposed to anticipate by the quotes, the resulting restart dumps were not the same, but not in the way that Charles saw. Instead, I'm getting missing patches or miswritten words of data in the output files. I also saw this once last week.
>
> There are at least two patterns. In the first, some processes occasionally produce output with the first 511 8-byte words zeroed out (one word shy of 4K). This almost certainly occurs during output, since the corruption is pristine, i.e., not smeared out at all by the numerical solution. This type of corruption occurred 3 times in the 2nd run this morning, with evidence of another evolved corrupt patch that was in fact introduced 3 dumps prior. Values of the effected variable are quite small in that region, and this corruption would typically go unnoticed.
>
> The 1st run this morning was free of this, but three processes produced errors in the last 8-byte word of the output buffer, corresponding to values of a variable in a boundary plane in the flow. Typical values in this plane are small (~1e-12); one erroneous value was of similar size, but two were set to 1. It turns out that this plane of data is not used to advance the solution, so this corruption happens to be benign.
>
> Otherwise, aside from three bad patches and three bad points (1536 points altogether), the rest of the data was byte-for-byte the same. (On the positive side, that was 56,899,582,464 out of 56,899,584,000 right. Good enough for government work, eh?) Both runs used nodeset thunder[22-998,1000-1002].

Here is the chronology of the highlights of our debugging and testing efforts:

- User observes output files not identical for identical runs as a rare but disturbing occurrence. Parts of first and last block of I/O buffer zeroed in error.
- Code is FORTRAN F90, I/O model is file per process.
- A small run is "straced" to understanding the transfer pattern of the code.
- Lustre developer comments on "bizarre" I/O pattern, which turns out to be the insertion of FORTRAN control words at the beginning and end of each record.
- User instruments his code to check for this pattern of error while executing.
- Problem is linked to Fortran: the first attempt at a reproducer written in F77.
- Bounced the trial reproducer off of the user, he reworked it to be a precise kernel of the I/O for his code.
- We tried the test on the cluster where the problem had occurred as well as on our semi-dedicated test cluster.

- The problem was observed about once per day using several hundred nodes, and only on the Thunder (IA64) cluster.
- More detailed analysis by Lustre developer associated the error with a write of an incomplete page. (There were actually quite few writes of incomplete pages in the real case, which was why we were observing the problem only once per day).
- *IOR* parameters were selected so that every 1000 byte transfer would be an incomplete write. In particular, we realized that if we used the strided data pattern on a shared file, with transfers of 1000 bytes that: 1) few of the transfers would be aligned on page boundaries, and 2) buffering at the client could not "assemble" the transfers so as to produce complete pages.
- Problem was reproduced on a few (5-10) nodes after running for a few minutes (HUGE breakthrough for debugging - faster, smaller, simpler bug reproducer)
- CFS fixed Lustre problem.
- Problem as observed by user resolved.
- Problem with identical symptom diagnosed and fixed for shared file situation.

This example demonstrated of number of very important points about the particular environment at Livermore:

- It is crucial to the success of adopting a bleeding edge technology to have customer's commitment and good humor. This case in particular required sustained efforts from the developer, the test team and the end user.
- We have had excellent developer support from CFS, Inc.: also crucial to success.
- There can be some really obscure symptoms and behaviors of such complex systems.
- If the developer says "a user would never do X", the test team should create a test to do X
- The important differences of FORTRAN vs. C I/O were highlighted.
- *IOR* proved to be incredibly versatile in delivering a problematic data pattern once the bug was understood at a sufficiently low level.

5.5. Performance modeling with IOR

One of our important application code developers had some puzzling observations of his code's I/O performance and had been in touch with us to discuss and determine if we had any suggestions. We agreed to model his code's I/O using IOR to gain some insight. This Email below describes the I/O situation including some parameters detailing the pattern of transfers:

> Here is some more data about my code's I/O on thunder.
>
> The restart file to be read is on lustre:
>
> -rw-r--r-- 1 user1 user1 243771630993 Aug 9 18:16 h2o1024.xml
>
> Its size is about 244 GB.
>
> When the job starts, 3920 MPI tasks open that file. Each task reads a block of size "local_size".
>
> The local block size is determined as follows: (the total file size is "sz")

```
off_t block_size = sz / ntasks;
off_t local_size = block_size;
off_t max_local_size = local_size + sz % ntasks;
// adjust local_size on last task
if ( ctxt.mype()==ntasks-1 )
{
  local_size = max_local_size;
}
```

> reading proceeds then using the following statements:

```
off_t offset = ctxt.mype()*block_size;
fseeko(infile,offset,SEEK_SET);
fread(&buf[0],sizeof(char),local_size,infile);
```

> In the present case, each tasks reads approximately 62 MB. There are 4 tasks running on each node.

Our initial discussion suggested that this I/O situation could easily be modeled using *IOR*, and the appropriate selection of parameters. The description is of a large shared file, shared by 3920 tasks, being read in a segmented fashion, where each process reads its entire segment in a single transfer. For a read using the POSIX API, reference to the *IOR* usage and some trivial analysis yielded the following command line:

ior –r –a POSIX –s 1 –b 62186640 –t 62186640

A further discussion with the user led us to believe that he had used only the default Lustre striping (across 4 OSTs). In an effort to reconcile this user's observed performance, we ran the IOR test on the default striping of the file system (4-way) and maximum striping (64-way).

The object in this case was to present the user with some understanding of the performance that he observed and some useful guidance to achieve better performance. We were fairly confident in our theory that striping of the shared file would explain the problem, and chose to explore that issue at a smaller scale (128 nodes/512 processes).

The data in Figure 6 confirm our suspicion that poor performance would be observed should we default striping (across 4 OSTs) when maximum striping was

needed for a highly parallel I/O situation. Lustre striping other than the default (here across 4 OSTs) must be explicitly set either for a specific file or by directory, in which case subsequent files in that directory will inherit the stripe setting. Even for this sophisticated user, the Lustre interface to set striping is easy to overlook.

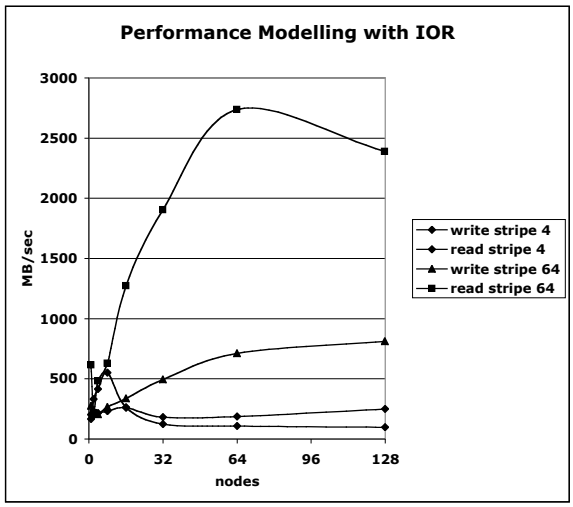

Figure 6. Striping and performance for I/O to a shared file

5.6. Performance evaluation of file system configurations with mdtest

To provide a consistent view of the file system from multiple compute clusters, it is preferable to configure the Lustre metadata servers to be networked via gigabit Ethernet to all clusters rather than have them directly attached to the high performance (Elan) switch fabric of one of the clusters, as is the present situation. *mdtest* can be applied to measure the impact of this configuration change upon metadata performance.

Here is the output resulting from a run of mdtest:

```
mdtest-1.7.1 was launched with 850 total task(s) on 850 nodes

Command line used:
mdtest -d/p/gt1/lustre-test/mdtest -i2 -f850 -l850 -s850 -n10 –tuv

Filesystem:             thunder4:/mds_p_gt1/client

1K-blocks       Used            Available       Use% Mounted on

162089444160 135716882140 26369179400  84%  /p/gt1

850 tasks, 8500 files/directories

  Operation       Duration          Rate
  ---------       --------          ----
 * iteration 1 *
Directory creation: 14.86 sec,    571.73 ops/sec
Directory stat   :    2.67 sec,  3180.60 ops/sec
Directory removal :  5.82 sec,  1459.37 ops/sec
File creation    :    5.62 sec,  1510.15 ops/sec
File stat        :    2.59 sec,  3281.94 ops/sec
File removal     :    6.89 sec,  1233.13 ops/sec

 * iteration 2 *
Directory creation: 22.02 sec,    385.87 ops/sec
Directory stat   :    2.67 sec,  3178.61 ops/sec
Directory removal :  5.83 sec,  1456.39 ops/sec
File creation    :    5.62 sec,  1510.90 ops/sec
File stat        :    2.60 sec,  3261.09 ops/sec
File removal     :    6.75 sec,  1259.24 ops/sec

SUMMARY: (of 2 iterations)
Operation            Max      Min      Mean  Std Dev
---------            ---      ---      ----  -------
Directory creation:   571.7   385.8   478.8   92.9
Directory stat   :  3180.6  3178.6  3179.6    0.9
Directory removal : 1459.3  1456.3  1457.8    1.4
File creation    :  1510.9  1510.1  1510.5    0.3
File stat        :  3281.9  3261.0  3271.5   10.4
File removal     :  1259.2  1233.1  1246.1   13.0
```

As yet, we have not reconfigured a system in order to perform the “after” evaluation. One can imagine however, that we would be quite comfortable in committing to a configuration change if minimal changes in these performance test measurements were observed.

5.7. Long Term Performance Regression Tests

Experience has shown us that even after a machine and file system is generally available for users, “aging” of the file system and/or upgrades to the hardware and software can cause significant changes to I/O and metadata performance.

For this reason we have set up monthly regression tests using *IOR* and *mdtest* to detect and report these long term changes. The regression tests are run when we know there are scheduled upgrades, but there have been cases when performance changes without update or with "trivial updates that could not affect the file system".

Not all of the results from these tests need to be retained, but at least one set a year needs to be kept and identified for long-term comparisons until the machine is retired.

6. Ideas for future work

Our efforts in developing such tests will continue to be driven by the needs of the center to support users and resolve problems that arise. At the present, we can think of the following four testing projects:

6.1. Fortran version of IOR

Many of our users' codes are written in some dialect of FORTRAN. By way of a user-discovered problem, we recently became aware of some of the subtlety of FORTRAN I/O (as opposed to C I/O). In FORTRAN, control words are inserted before and after each record. This resulted in an undesirable interaction with the Lustre file system. A FORTRAN version of *IOR* would be a valuable test tool that would represent this aspect of our center's workload.

6.2. Parameter space surveys with IOR

As our IBM systems running GPFS file systems matured, we had the opportunity to study throughput performance as a function of transfer sizes and data layout. *IOR* was designed with this type of study in mind. We would like to study our various configurations of Lustre file systems to get a more comprehensive understanding of the performance profile.

6.3. Future systems

We plan on running Lustre on future systems at the Laboratory, including on the BG/L [11] from IBM. This system has a different sort of architecture from the parallel system that we have had recently. Notably with regard to the I/O system, a set of 64 compute nodes will be serviced by one I/O node: for our recent systems each compute node could handle its own I/O. We anticipate a lot of activity in testing and test development in support of the Lustre efforts on this system.

6.4. Randomness returns to IOR: AMR codes

Several of our important application codes are now using adaptive mesh refinement. This means that the amount of data varies for each process depending on the physical behavior of the problem being analyzed. Because the imbalance in the amount of data can become quite large, it can potentially have a severe effect on the I/O performance for the code. To help understand the I/O behavior of such AMR codes, it would be useful to let IOR model the same type of access patterns. This would require introducing some kind of randomness in the amount of data and transfer sizes on each process somewhat similar to the original IOR.

7. Conclusions

The massive computing and corresponding I/O requirements of the ASC program has driven our involvement with the Lustre file system for our ever-expanding realm of Linux clusters.

Our extensive testing program has been crucial for the local success of the Lustre file system. The program includes nearly continuous testing on 450 nodes of a large-scale Linux cluster as well as on some smaller clusters. Beyond that there is remedial testing as required to diagnose and understand newly discovered file system issues.

We have developed a suite of test programs including performance and stress tests of both data transfer and metadata operations. The codes are based on the MPI parallel programming model, which is ubiquitous in our computer center. These tests originate both from our understanding of the I/O usage patterns of our customers and from problems discovered by them. The resulting test tools also serve us well in modeling customer I/O situations and in evaluating present and future file system configurations.

Let's review our suite of tests against the characteristics of an "ideal" benchmark as envisioned by Chen and Patterson:

- Spends most of its time doing I/O – **most definitely**
- Scales gracefully over a wide range of current and future machines – **yes: our tests are designed to scale to larger systems in the same manner as the codes in the general workload.**
- Allows fair comparison across machines - **across machines that would interest us, yes.**
- Relevant to a wide range of applications **– across ASC applications, yes.**
- Helps designers to understand why the system behaves as it does **– yes: we are certainly able to develop compelling stories about why Lustre behaves the way it does on our application mix.**

We have successfully accomplished these objectives in developing our file system tests in the context of our own systems and workloads. The utility of our particular tests for general environments or very different workloads may fairly be challenged. In that case, a higher-level view of our approach, that of a technical liaison and interpreter between developers and end users, may be applied.

Acknowledgements

We thank our colleagues from the LLNL Lustre team and CFS, Inc with whom we have near daily contact regarding these topics. Also we acknowledge our partners from Hewlett-Packard and Intel and our Tri-Lab collaborators at the Los Alamos and Sandia National Laboratories. Last but not least, we thank our customers for the opportunity to serve them, for the extra efforts that they have made in assisting our diagnoses, and for their great patience.

Auspices Statement

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

References

[1] http://www.lustre.org/ www.lustre.org/docs/whitepaper.pdf h

[2] http://www.llnl.gov/asci-pathforward/pf-overview.html

[3] Bonnie++[2001]. http://www.coker.com.au/bonnie++/

[4] dbench [2001]. http://freshmeat.net/projects/dbench

[5] fsx [1991]. http://www.codemonkey.org.uk/cruft/fsx-linux.c

[6] IOzone[2003]. http://www.iozone.org

[7] http://www-unix.mcs.anl.gov/mpi/

[8] IOR [2003]. UCRL-CODE-2003-016. http://www.llnl.gov/icc/lc/siop/downloads/download.html

[9] mdtest [2003]. # UCRL-CODE-155800. http://www.llnl.gov/icc/lc/siop/downloads/download.html

[10] simul[2003].UCRL-CODE-2003-019 http://www.llnl.gov/icc/lc/siop/downloads/download html

[11] An Overview of the BlueGene/L Supercomputer http://www.research.ibm.com/people/g/gupta/bgl_sc2002.pdf

[12] MCR Background http://www.llnl.gov/linux/mcr/background/mcr_background.html

[13] Scott Brandt, Private communication

[14] Lustre Testing https://wiki.clusterfs.com/lustre/LustreTesting

[15] SPEC SFS Suite http://www.spec.org/sfs97r1/

[16] SPECweb99 Benchmark http://www.spec.org/web99/

[17] Postmark: a New File System Benchmark http://www.netapp.com/tech_library/3022.html

[18] The Global Unified Parallel File System (GUPFS) Project: FY 2003 Activities and Results http://repositories.cdlib.org/lbnl/LBNL-52456_2003/

[19] Nov. '94 ACM Transactions on Computer Systems -- "A New Approach to I/O Performance Evaluation -- Self-Scaling I/O Benchmarks, Predicted I/O Performance", Peter Chen and David Patterson

[20] GPFS: A Shared-Disk File System for Large Computing Clusters: Frank Schmuck and Roger Haskin, IBM Almaden Research Center http://www.usenix.org/publications/library/proceedings/fast02/full_papers/schmuck/schmuck_html/index.html

[21] File System Workload Analysis For Large Scale Scientific Computing Applications: Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long and Tyce T. McLarty http://www.cse.ucsc.edu/~qxin/mypapers/msst04.pdf

Appendix Ia. Livermore Laboratory Computer Center Major Systems (Open Computing Facility)

Machine	Compute Power	Memory Capacity	Disk Capacity	File System Bandwidth
Purple	100 TFLOP/s	48 TiB	2 PB	100 GB/s
White	12.3 TFLOP/s	8 TB	100 TB	6.4 GB/s
UM & UV	12.3 TFLOP/s	4 TB	140 TB	12 GB/s
Lilac	9.2 TFLOP/s	3.1 TB	n/a *	9 GB/s
PCR Clusters	2.86 TFLOP/s	1 TB	7.0 TB	300 MB/s
SGI Vis Platforms	139 GFLOP/s	140 GB	20 TB	2.8 GB/s
Linux Vis Platform	717 GFLOP/s	272 GB	n/a *	2.8 GB/s

* n/a indicates that these platforms share a single "site-wide" lustre file system of approximately 150 TB.

Appendix Ib. Livermore Laboratory Computer Center Major Systems (Secure Computing Facility)

Machine	Compute Power	Memory Capacity	Disk Capacity	File System Bandwidth
BG/L	360 TFLOP/s	16 TB	n/a *	40 GB/s
Thunder (M&IC)	20.1 TFLOP/s	7.2 TB	n/a *	6.4 GB/s
MCR (M&IC)	11.1 TFLOP/s	4.6 TB	n/a *	4.8 GB/s
ALC (ASCI)	9.2 TFLOP/s	3.84 TB	n/a *	6 GB/s
Frost (ASCI)	1.6 TFLOP/s	1.0 TB	20 TB	1.6 GB/s
PVC/ Sphere	1.8 TFLOP/s	404 GB	n/a	1.6 GB/s
Cube	2.5 TFLOP/s	544 GB	n/a	1.6 GB/s

* n/a indicates that these platforms share a single "site-wide" lustre file system of approximately 250 TB.

Appendix II: Usage of IOR

Command lines for the throughput studies in section 5.2

file per process (Figure 2)

```
./IOR  -q -v -a POSIX -d 1 -w -r -o /p/gt1/rhedges/IOR/iorData -t 65536 -b 2684354560 -F -i 3 –C
```

shared segmented (Figure 3)

```
./IOR  -q -v -a POSIX -d 1 -w -r -o /p/gt1/rhedges/IOR/dir_max/iorData -t 65536 -b 2684354560  -i 3 –C
```

shared strided

```
./IOR  -q -v -a POSIX -d 1 -w -r -o /p/gt1/rhedges/IOR/dir_max/iorData -t 65536 -b 655360 -s4096  -i 3 –C
```

Command Line Options:

-a S	api -- API for I/O [POSIX|MPIIO|HDF5|NCMPI]
-b N	blockSize -- contiguous bytes to write per task (e.g.: 8, 4k, 2m, 1g)
-B	useO_DIRECT -- uses O_DIRECT for POSIX, bypassing I/O buffers
-c	collective -- collective I/O
-C	reorderTasks -- changes task ordering to n+1 ordering [!HDF5]
-d N	interTestDelay -- delay between reps in seconds
-e	fsync -- perform fsync after POSIX write close
-E	useExistingTestFile -- do not remove test file before access
-f S	scriptFile -- test script name
-F	filePerProc -- file-per-process
-g	intraTestBarriers – use barriers between open, read/write, and close
-h	showHelp -- displays options and help
-H	showHints -- show hints
-i N	repetitions -- number of repetitions of test
-I	individualDataSets -- datasets not shared by all procs [not working]
-k	keepFile -- keep testFile(s) on program exit
-l	storedFileOffset -- use file offset as stored signature
-n	noFill -- no fill in HDF5 file creation
-o S	testFileName -- full name for test
-O	options -- options string
-p	preallocate -- preallocate file size
-P	useSharedFilePointer -- use shared file pointer [not working]
-q	quitOnError -- during file error-checking, abort on error
-r	readFile -- read existing file
-R	checkRead -- check read after read
-s N	segmentCount -- number of segments
-S	useStridedDatatype -- put strided access into datatype [not working]
-t N	transferSize -- size of transfer in bytes (e.g.: 8, 4k, 2m, 1g)
-T	maxTimeDuration -- max time in minutes to run tests
-u	uniqueDir -- have each task in file-per-process use unique directory
-U	hintsFileName -- full name for hints file
-v	verbose -- output information (repeating flag increases level)
-V	useFileView -- use MPI_File_set_view
-w	writeFile -- write file
-W	checkWrite -- check read after write
-x	singleXferAttempt -- do not retry transfer if incomplete

NOTE: S is a string, N is an integer number.

Appendix III: Usage of simul

Usage: simul [-h] -d <testdir> [-f firsttest] [-l lasttest] [-n #] [-N #] [-i "4,7,13"] [-e "6,22"] [-s] [-v] [-V #]

-h: prints this help message
-d: the directory in which the tests will run
-f: the number of the first test to run (default: 0)
-l: the number of the last test to run (default: 39)
-i: comma-separated list of tests to include
-e: comma-separated list of tests to exclude
-s: single-step through every iteration of every test
-n: repeat each test # times
-N: repeat the entire set of tests # times
-v: increases the verbositly level by 1
-V: select a specific verbosity level

The available tests are:
Test #0: open, shared mode.
Test #1: close, shared mode.
Test #2: file stat, shared mode.
Test #3: lseek, shared mode.
Test #4: read, shared mode.
Test #5: write, shared mode.
Test #6: chdir, shared mode.
Test #7: directory stat, shared mode.
Test #8: statfs, shared mode.
Test #9: readdir, shared mode.
Test #10: mkdir, shared mode.
Test #11: rmdir, shared mode.
Test #12: unlink, shared mode.
Test #13: rename, shared mode.
Test #14: creat, shared mode.
Test #15: truncate, shared mode.
Test #16: symlink, shared mode.
Test #17: readlink, shared mode.
Test #18: link to one file, shared mode.
Test #19: link to a file per process, shared
Test #20: open, individual mode.
Test #21: close, individual mode.
Test #22: file stat, individual mode.
Test #23: lseek, individual mode.
Test #24: read, individual mode.
Test #25: write, individual mode.
Test #26: chdir, individual mode.
Test #27: directory stat, individual mode.
Test #28: statfs, individual mode.
Test #29: readdir, individual mode.
Test #30: mkdir, individual mode.
Test #31: rmdir, individual mode.
Test #32: unlink, individual mode.
Test #33: rename, individual mode.
Test #34: creat, individual mode.
Test #35: truncate, individual mode.
Test #36: symlink, individual mode.
Test #37: readlink, individual mode.
Test #38: link to one file, individual mode.
Test #39: link to a file per process, individual

Appendix IV: Usage of mdtest

Usage: mdtest [-h] [-f first] [-i iterations] [-l last] [-s stride] [-n #] [-p seconds] [-d testdir] [-t] [-u] [-v] [-D] [-F] [-N #] [-S] [-V #]

-h: prints this help message
-c: collective creates: task 0 does all creates
-d: the directory in which the tests will run
-f: first number of tasks on which the test will run
-i: number of iterations the test will run
-l: last number of tasks on which the test will run
-n: every process will creat/stat/remove # directories and files
-p: pre-iteration delay (in seconds)
-s: stride between the number of tasks for each test
-t: time unique working directory overhead
-u: unique working directory for each task
-v: verbosity (each instance of option increments by one)
-w: bytes to write to each file after it is created
-D: perform test on directories only (no files)
-F: perform test on files only (no directories)
-N: stride # between neighbor tasks for file/dir stat (local=0)
-S: shared file access (file only, no directories)
-V: verbosity value