

# I/O Guide for LC

# Table of Contents

Preface .....	3
Introduction .....	4
I/O Glossary .....	5
Hierarchical Data Format .....	7
HDF Features .....	7
HDF Availability .....	8
HDF5 File Structure .....	9
HDF Operations .....	11
MPI-IO .....	14
MPI-IO Role .....	14
MPI-IO Data Access .....	15
MPI-IO Issues at LC .....	17
GPFS at LC .....	18
GPFS, NFS, DFS Compared .....	19
How GPFS Works .....	21
GPFS Usage Advice .....	25
GPFS Purge Policy .....	25
GPFS Usage Data .....	26
I/O Analysis of FLASH .....	27
Lustre Parallel File System .....	29
Lustre Goals and Roles .....	29
Lustre and GPFS Compared .....	30
Feature Comparison .....	30
Purge Comparison .....	31
Name and Mount Comparison .....	32
Lustre LLNL Implementation .....	34
LLNL's Lustre Strategy .....	34
LLNL's Lustre Service .....	36
Lustre Operational Issues .....	37
Directory Names and Aliases .....	37
Lustre Purge Policy .....	38
MPI-IO Interaction with Lustre .....	39
Lustre Backup Policy .....	41
Lustre Striping .....	42
Lustre Groups .....	44
Disclaimer .....	45
Keyword Index .....	46
Alphabetical List of Keywords .....	48
Date and Revisions .....	49

# Preface

**Scope:** After an overview of I/O issues, this guide provides an alphabetical glossary of (just those key) I/O terms needed to understand the later treatments of LC-related I/O tools and techniques. One major section explains the features and local use of the Hierarchical Data Format (HDF), designed to promote efficient large-scale I/O (HDF use by the I/O-intensive FLASH hydrodynamics code is also analyzed, later). Portable, parallel I/O with the MPI library is discussed as well, both in terms of general data-access routines and local implementation constraints. Another section contrasts the General Parallel File System (GPFS, which supports parallel I/O on LC IBM clusters) with NFS and DFS, also in use on LC machines. And another explains how GPFS handles parallel writes (to clarify potential performance bottlenecks and I/O strategies). A separate discussion introduces the design features (such as data and metadata separation), local implementation details (such as file system names and sizes), and known pitfalls of Lustre, the open-source parallel file system that LC deploys on its Linux/CHAOS clusters. A purge-policy comparison for GPFS and Lustre is also included.

**Availability:** GPFS is an IBM product available at LC on IBM SP machines. Lustre is designed for and available only on LC Linux/CHAOS clusters. The availability of other I/O features and software may vary (details and restrictions are in the text related to each feature).

**Consultant:** For help contact the LC customer service and support hotline at 925-422-4531 (open e-mail: [lc-hotline@llnl.gov](mailto:lc-hotline@llnl.gov), secure e-mail: [hotline@pop.llnl.gov](mailto:hotline@pop.llnl.gov)).

**Printing:** The print file for this document can be found at:

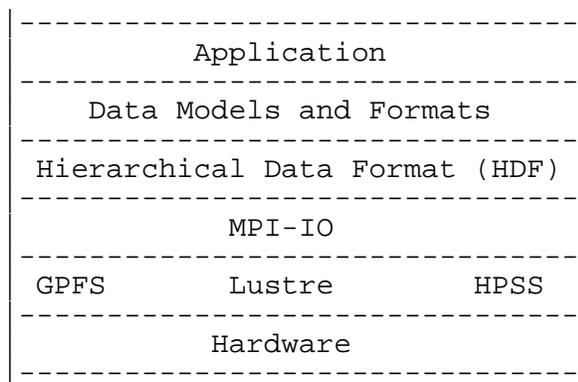
OCF: <http://www.llnl.gov/LCdocs/ioguide/ioguide.pdf>  
SCF: [https://lc.llnl.gov/LCdocs/ioguide/ioguide\\_scf.pdf](https://lc.llnl.gov/LCdocs/ioguide/ioguide_scf.pdf)

# Introduction

High-performance input and output (I/O), including parallel I/O, is crucial for the overall success of large-scale computer simulations, such as the ASCI simulations on which LC's production computing resources are focused. Planning for effective, scalable I/O by application codes is important because:

- Expensive computer time spent performing I/O is lost for the primary task of simulating physical processes. For example, when wall-clock time for I/O grows beyond 25% of a code's total run time, then I/O techniques are often causing serious overall performance problems.
- Simulation data, and the time and cycles spent creating them, are wasted if not well managed and effectively saved for future reuse (such as visualization).
- Even with LC's large local disks and storage media, I/O can be a bottleneck in high-performance computing unless applications use the most appropriate techniques for reading and writing their data. The separation of compute nodes and I/O nodes can make this bottleneck worse.

LLNL's Scalable I/O Project has developed an "end-to-end model of the I/O stack" to make clear the layers that data pass through from an application code to physical storage:



This I/O Guide follows this same general path to introduce I/O terminology and to discuss LC-relevant I/O techniques and resources (after an introductory glossary).

# I/O Glossary

This section provides an alphabetical set of brief explanations for the unusual technical terms that appear throughout this guide as I/O issues and features are discussed. The glossary here is intended to make the text of this manual easier to use, not to offer a comprehensive I/O dictionary.

- Disk striping** distributes file data across multiple disks for speed and safety. The amount of consecutive data stored on each disk is the "striping unit" or "strip width," which may be the block size (as on IBM's GPFS (page 18)), or multiple blocks, or just a few bits. "Declustering" is sometimes a synonym for disk striping. See also the "Lustre Striping" section (page 42) below for some relevant user tools.
- Diskless nodes** are compute nodes in a Linux/CHAOS cluster that have no local hard disks. At LC, the "Peleton machines" (such as Atlas and Zeus on OCF, Rhea and Minos on SCF) have diskless nodes. The advantage is that there are fewer disk drives to fail, increasing overall reliability. But diskless nodes have no swap space, so any application that runs out of memory on those nodes will be terminated by the CHAOS OOM (Out Of Memory) killer. Also, /tmp and /var/tmp on diskless nodes use RAM, not disk. So CHAOS purges those file systems completely between jobs on diskless nodes to reclaim the memory used. Jobs there must use HPSS, Lustre, or /nfs/tmp\* for output to survive after the job ends.
- F/b ratio** measures the effectiveness of an I/O system. F is the rate of executing floating-point operations and b is the rate of performing I/O (so  $F/b = 1$  means one bit of I/O occurs for every floating-point operation). While  $F/b = 1$  is sometimes thought to be the ideal for supercomputers, real-life F/b ratios are often closer to 100 for scientific applications, and closer to 10 for I/O intensive applications.
- GPFS** is IBM's commercial General Parallel File System. This is the file system installed for parallel I/O on LC's IBM/AIX massively parallel production computers (see the separate section (page 18) below for local implementation details).
- HDF** is Hierarchical Data Format, a standard way to organize files internally and a supporting I/O library. Both are designed to promote efficient large-scale I/O for scientific applications (see the separate section (page 7) below for LC's local implementation details).
- Lustre** is an open-source parallel file system from Cluster File Systems, Inc. This is the file system installed for parallel I/O on LC's Linux/CHAOS massively parallel production computers (see the separate section (page 29) below for LC's design constraints, a feature comparison with GPFS, and local implementation details).
- Parallel file system**  
is a file system specifically designed to allow  
(a) simultaneous reads and writes to nonoverlapping regions of the same (logical) file,

(b) simultaneous reads and writes of different files, and  
(c) distribution (striping) of file data across several I/O nodes or disks (or both), especially for large files. GPFS (page 18) is an example of a parallel file system in use at LC; Lustre (page 29) is another example.

#### Parallel I/O subsystem

is a way to transfer data in parallel between compute nodes and dedicated I/O nodes within the same massively parallel machine. The parallel I/O subsystem takes advantage of the machine's high-speed internal switch to handle small requests efficiently, yet it can scatter I/O operations among many nodes to efficiently distribute large files too. I/O occurs internally across the parallel I/O subsystem, then externally across high-bandwidth channels to mass-storage servers.

#### RAID

is a redundant array of inexpensive (or independent) disks. RAID technology provides high reliability for stored data by striping the data across several disks in a way that uses more disk space than without striping but maintains parity so that lost data can be reconstructed even if one disk in the array fails. LC's globally mounted NFS disks (such as for the global home directories) use RAID.

# Hierarchical Data Format

## HDF Features

Hierarchical Data Format (HDF) is a (specification for a) file format and a supporting I/O library for storing technical data. HDF is designed to promote efficient large-scale I/O for scientific applications running in high-performance computing environments. Hierarchical Data Format 5 (called HDF5) replaces an earlier, and incompatible, attempt to meet similar goals (called HDF4).

This table introduces key HDF5 features by comparing them with the corresponding aspects of HDF4:

	HDF5	HDF4 (replaced)
Developers:	NCSA	NCSA
Originated:	1998	1980s
Limits on stored objects:	none	20,000
file size:	none	2 Gbyte
Data model:	simple and comprehensive	some limits, inconsistencies
Supports parallel I/O?	yes	no
Supports threaded applications?	in theory	no

## HDF Availability

HDF5 is available for both AIX (IBM Unix) and LINUX operating systems, but some supported features differ between them. This chart summarizes the most important differences:

	AIX (IBM)	LINUX
Supports C?	yes	yes
C parallel?	yes	yes (MPICH)
F90?	yes	no
F90 parallel?	yes	no
C++?	no	yes
Tools available(*)?	yes	yes
Thread safe?	no	maybe

(\*)To manipulate HDF files, such as H5DUMP.

Your program manipulates HDF5 files by using calls to the HDF5 I/O library. On LC production machines this library resides in

```
/usr/local/hdf5
```

and each library version has its own child directory. The C language (libhdf5.a) and Fortran (libhdf5\_fortran.a) alternatives for the latest version reside in directory hdf5-1.4.3 (down several levels), for example, except on Linux machines, where the latest version is in hdf5-1.4.2 and there are no Fortran files.

The HDF5 tools reside in a /bin directory several layers (depending on the host) below /usr/local/hdf5-*version* on each LC production machine. See the "HDF Operations" [section below](#) (page 11) for details on using the HDF5 library and tools locally. See the "I/O Analysis of FLASH" [section below](#) (page 27) for discussion of the benefits and pitfalls of trying to use HDF5 to support a hydrodynamics code that performs extensive I/O.

## HDF5 File Structure

HDF5 files are binary containers for efficiently holding scientific data in an organized way, with explicit supporting metadata to facilitate later reuse.

### FEATURES.

HDF5 files consist of:

- groups            HDF5 groups behave like UNIX directories: they organize data hierarchically. Groups can contain other, child groups or point to other groups. Every HDF5 group has three *attributes* that overtly declare its:  
NAME (one "root group" is always named "/").  
PATHNAME (called its "OBJ-XID").  
IMMEDIATE PARENT GROUP.
  
- datasets            HDF5 datasets within groups behave like UNIX files within directories, except that they too have overt structures and supporting metadata. Every HDF5 dataset includes these features:
  - dataspace            is a way to overtly declare the number and range of the *dimensions* needed for the subsequent scientific data.
  
  - datatype            is a way to specify how to interpret the data, such as array, compound, or atomic. "Atomic" HDF5 datasets can have their byte order, size, and sign declared, along with any of these subtypes:  
INTEGER  
FLOAT  
STRING  
TIME  
BITFIELD  
OPAQUE  
OBJECT REFERENCE  
REGION REFERENCE  
ENUM(ERATION)
  
  - data                is the actual "lowest level" output from or input to your application program (optionally empty).

### XML ROLE.

Because HDF5 files are hierarchically organized and encoded with overt attributes, they can be represented by and manipulated using XML (the ISO standard "eXtensible Markup Language"). NCSA, in collaboration with LLNL, LANL, and SNL, has already developed and published an XML "document type definition" (DTD, or element inventory) for HDF5, spelling out all the elements, attributes, and interrelationships needed to adequately, accurately represent HDF5 files as "XML instances." For details see

<http://hdf.ncsa.uiuc.edu/DTDs/HDF5-File.dtd>

No mere intellectual exercise, this representation of HDF5 files in XML offers seven benefits to HDF5 users:

- Viewing.  
You can use any web browser to inexpensively view (survey the contents of) the XML version of an HDF5 file, either directly or when served from a site that generates HTML from XML on the fly.
- Cataloging.  
You can use XML attributes to group or identify your datasets for faster, more reliable subsequent searches or extraction by specified topic.
- Application Exchange.  
Just as RTF facilitates the exchange of files among different word processors, so XML (as an intermediate format) facilitates exchange of HDF5 files among application programs or HDF5 editors that may be very different otherwise.
- Validation.  
Standard XML parsers can validate the syntactic correctness of any HDF5 file in XML format (but of course they cannot validate the *data* inside).
- Transformation.  
Standard programming tools such as Javascript and XSL ("eXtensible Stylesheet Language") already easily transform valid XML into other formats or languages, and these now apply to HDF5 files as well.
- Database Exchange.  
XML facilitates insertion of HDF5 files into formal databases or other archival systems that recognize XML input.
- Templates.  
XML can create templates or skeleton files for reliably making new HDF5 files consistent with previous ones, an aid in standardizing data handling among collaborators or different projects.

## HDF Operations

Your program manipulates HDF5 files by using calls to the HDF5 I/O library. On LC production machines this library resides in

```
/usr/local/hdf5
```

and each library version has its own child directory. The C language (libhdf5.a) and Fortran (libhdf5\_fortran.a) alternatives for the latest version reside in directory hdf5-1.4.3 (down several levels), for example, except on Linux machines, where the latest version is in hdf5-1.4.2 and there are no Fortran files.

NCSA provides a fairly elaborate tutorial on HDF5 file operations (and hence on relevant library components) online at

<http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/index.html>

Among the most important operations on HDF5 files are these:

**File Creation.** The include file hdf5.h (for C) or the module HDF5 (for Fortran) contains definitions and declarations that you must use in any program that invokes the HDF5 library. A call to routine H5Fcreate (C) or h5fcreate.f (Fortran) creates a new HDF5 file, returns its file identifier, and lets you specify its:

Filename

Access Mode (to control reads and writes)

Creation Property List (to control metadata; defaults available)

Access Property List (to control methods of performing I/O).

**File Display.** Once you have created, expanded, or altered an HDF5 file, you can display its contents (groups, attributes, etc.) in human-readable form by invoking any of several software tools provided by NCSA for this purpose. The HDF5 tools are:

h5debug

h5dump

h5gif

h5import

h5ls

h5repart

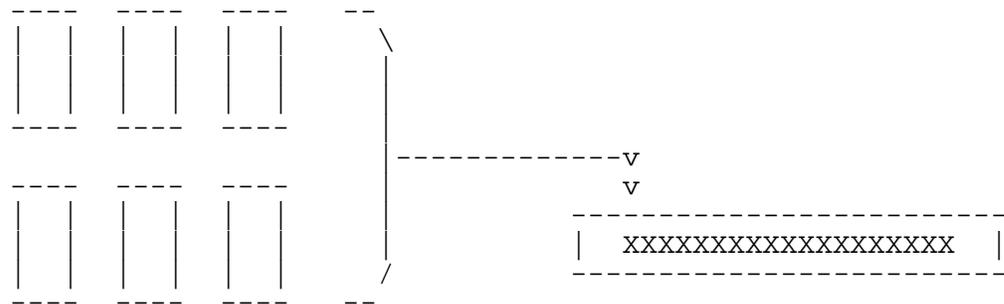
and they reside in a /bin directory several layers below /usr/local/hdf5/hdf5-*version* on each LC production machine. (Unfortunately, the exact path varies considerably in detail and length among LC's AIX, Compaq, and Linux platforms.) Each HDF5 tool, if run without options, displays several screens of text summarizing its usage syntax and available options, then ends. One of the most helpful tools is H5DUMP, which outputs an ACSI text display in Backaus-Naur Form by default or encoded in XML if you request with its -x option. (See also the comments on using the IDL library for HDF5 output, below.)

## Hyperslab Selection.

One way to read existing HDF5 files is by invoking `H5Sselect_hyperslab` (C) or `h5sselect_hyperslab.f` (Fortran), which extracts a "hyperslab" from an HDF5 dataset. A hyperslab can be

- (a) a logically contiguous set of points, or
- (b) a regular pattern of points or blocks even if noncontiguous.

Hyperslab selection from HDF5 datasets is so flexible that you can read from a dataset with one size, shape, and datatype, and then write into a dataset with a different size, shape, and datatype. For example, you can read blocks from a 2-D array of 32-bit floats and then write that data into a contiguous sequence of 64-bit floats at a specified offset in a 1-D array, as shown here:



## Parallel HDF5.

A parallel HDF5 API is supported on some but not all environments where the HDF5 library is available. For example, both AIX (IBM) and Linux support parallel C HDF5, but parallel Fortran HDF5 is only available under AIX (see the "HDF5 Availability" section [above](#) (page 8) for a summary chart). Parallel I/O on HDF5 files always involves the MPI concept of a "communicator," a specified set of processes that pass messages to each other. For parallel HDF5 I/O, each process in an MPI communicator

- (a) invokes `H5Pcreate` (C) or `h5pcreate.f` (Fortran) to create an "access template" and obtain a file's access property list, and
- (b) invokes `H5Pset_fapl_mpio` (C) or `h5pset_fapl_mpio.f` (Fortran) to initiate parallel I/O access.

With parallel HDF5 I/O,

- All parts of the file are accessible by all MPI processes.
- All objects in the file are accessible by all processes.
- Multiple processes can write to the same dataset (or, optionally, to individual datasets).

The HDF5 tutorial referenced at the start of this section includes annotated programming examples (in C and Fortran) of performing parallel I/O on HDF5 files.

## HDF5 Support in IDL.

On all production machines, including the Linux clusters, LC offers a licensed commercial library and tools together called "Interactive Data Language" (IDL). IDL is really a general data exploration and visualization language designed for writing high-level data-analysis programs much more compactly than with C or Fortran. HDF is not even mentioned in the index of the 210-page IDL "Getting Started" manual. But HDF5 is indeed one of four "self-describing scientific data formats" that IDL routines can read and query (but *not* write). IDL acknowledges the following limitations when reading HDF5 files:

- (1) No datatype conversion (until after the data is read).
- (2) Only the topmost HDF5 error message is printed from the stack.
- (3) No support for variable-length, reference, or opaque datatypes.
- (4) No property-interface support.
- (5) No writes.

On LC machines, the IDL library resides at

```
/usr/global/tools/RSI/idl_5.5/lib
```

and the alphabetical descriptions of the 74 IDL HDF5 library routines appear in a PDF manual at

```
/usr/global/tools/RSI/idl_5.5/docs/HDF5.pdf
```

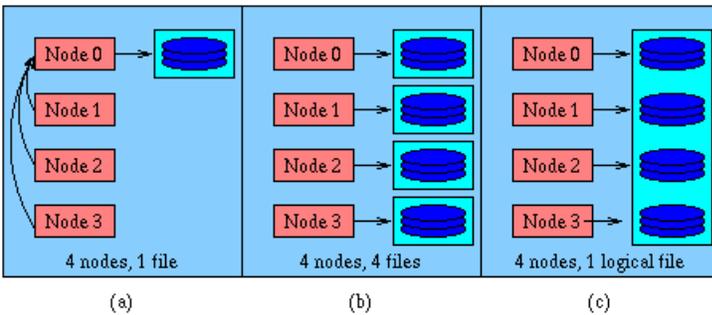
(See also the comments on using native HDF5 tools for output instead, in the "File Display" paragraphs above.)

# MPI-IO

## MPI-IO Role

A parallel I/O interface allows programs with many processes on many nodes to coordinate their I/O read and write operations for greater efficiency. When the Message Passing Interface (MPI) Forum revised and expanded the MPI standard in the mid 1990s, they added support for such parallel I/O. This portable, parallel interface is called MPI-IO. Although not fully implemented by most vendors, LLNL was an early supporter of MPI-IO.

Implementing successful parallel I/O, including MPI-IO, requires the underlying support of a parallel file system (such as IBM's GPFS (page 18) or Linux-oriented Lustre (page 29)). This diagram shows why a traditional file system causes expensive inefficiency when users attempt parallel I/O without proper hardware support:



One inefficient approach concentrates all read and write requests at a single I/O node (a). Another allows several nodes to read or write data (b), but only to separate (logically distinct) files that have to be somehow merged later. A parallel file system (c) not only supports I/O from many nodes at the same time, but also transfers the data to (different parts of) a single logical file, even if the file is "striped" across multiple physical disks (for safety, convenience, or speed).

## MPI-IO Data Access

With MPI-IO, data moves between processes and files using a variety of specific read and write calls, all variations on `MPI_FILE_READ` and `MPI_FILE_WRITE`. You select from these many routines to address three independent aspects of parallel data access: positioning, synchronism, and coordination.

**Positioning:** MPI-IO data access routines provide three types of positioning (which you can mix safely within the same program):

- **Explicit Offsets.**  
Explicit offset routines perform data access at a file position that you give explicitly as an argument (no file pointer used). All such routine names end in `_AT` (such as `MPI_FILE_WRITE_AT`).
- **Individual File Pointers.**  
Each I/O operation with a file pointer leaves the pointer pointing to the *next* data item after the one last accessed by the operation (example: `MPI_FILE_WRITE` unqualified).
- **Shared File Pointers.**  
These behave much like individual file pointers (above), but each routine ends in `_SHARED` (noncollective) or in `_ORDERED` (collective).

**Synchronism:** MPI-IO offers both blocking and nonblocking I/O routines.

- **Blocking.**  
Blocking I/O calls will not return until their I/O request is completed. Blocking is the default approach (e.g., with `MPI_FILE_WRITE` or `MPI_FILE_WRITE_AT`); special routines request nonblocking I/O (next).
- **Nonblocking.**  
Nonblocking I/O calls start an I/O operation but do *not* wait for it to complete. This can allow data transfer simultaneous with computation if hardware permits. To confirm that your data was actually read or written, however, you must use a separate "request complete" call (such as `MPI_WAIT`). Nonblocking versions of MPI routines all have names of the form `MPI_FILE_iaaa` (where I is for "immediate," such as `MPI_FILE_IWRITE`).

Coordination: MPI data access routines may be noncollective or collective, a measure of their dependence on other members of their process group.

- Noncollective.  
Noncollective call completion depends only on the calling process itself. The default MPI routines above (such as `MPI_FILE_WRITE`) perform noncollective data access.
- Collective.  
Completion of a collective call (made by all members of a process group) may depend on the activity of every process making the call. But sometimes collective calls perform better than noncollective ones because they can be automatically optimized. Collective MPI routines end in `_ALL` (such as `MPI_FILE_WRITE_ALL`) or in the pair `_ALL_BEGIN/END` (such as the explicit-offset pair `MPI_FILE_WRITE_AT_ALL_BEGIN` and `MPI_FILE_WRITE_AT_ALL_END`). The noncollective `MPI_FILE_aaa_SHARED` pointer routines map to the collective routines called `MPI_FILE_aaa_ORDERED`.

For more details on MPI-IO routines, on routine naming patterns for handling every combination of positioning, synchronism, and coordination, and for a summary of MPI-IO data access conventions in general, consult this specific part of the MPI Forum web site:

<http://www.mpi-forum.org/docs/mpi-20-html/node186.htm>

## MPI-IO Issues at LC

This section summarizes and compares MPI-IO issues, problems, and implementation constraints that specifically affect local users of LC production machines.

### NFS Incompatible with MPI-IO.

Successfully performing MPI-IO on NFS-mounted file systems requires that (a) NFS is at version 3, and (b) each NFS shared directory is mounted with the "no attribute caching" (NOACC) option enabled. However, all NFS-mounted file systems on LC production machines (such as `/nfs/tmpn` or the global home directories) are installed with attribute caching enabled (so NOACC is disabled and does *not* appear in their attribute list in `/etc/fstab`). This means that attempts to perform parallel MPI I/O to LC's NFS-mounted disks will fail. (NFS normally caches modified file pages on each client node that performs a write, without promptly updating the master copy on the file server. When multiple parallel clients write to the same file, this shortcut means that NFS will probably not correctly update the master copy.) Furthermore, when a globally mounted NFS file system (such as those supporting LC's common home directories) is flooded with MPI I/O traffic, service slows, often dramatically, not only on the machine running the MPI code but on *all machines* on which that file system is mounted. This is not a responsible use of shared computing resources.

### ROMIO Problems.

LC users of the vendor-independent MPICH libraries for MPI should note that the MPICH "ROMIO" implementation of parallel I/O is not standard compliant in the way it manages file handles, which are used for nonblocking I/O requests.

### MP\_SINGLE\_THREAD Role.

On LC IBM machines, the environment variable `MP_SINGLE_THREAD` is an optimization flag. At LC, it is NO by default, a setting that assumes multiple message-passing threads and can improve the performance of the threaded MPI library. If this flag is set to YES, then your program cannot use MPI-IO.

### Portability Issues.

Tests show that for some MPI-IO operations GPFS is much more efficient than Lustre, while for other operations Lustre performs much better than does GPFS. See the "[MPI-IO Interaction](#)" (page 39) section below for a discussion of the difficulties of predicting how MPI-IO operations tuned to one parallel file system will behave when moved to a different parallel file system.

## GPFS at LC

A file system is the software (or sometimes, the software with the collection of data that it manages) that allows you as a user to manipulate hierarchically organized, access-controlled files and directories rather than just the countless raw blocks of data that comprise them (on disk). This section

- compares the LC-installed IBM General Parallel File System (GPFS), designed for large-scale parallel I/O, with traditional file systems,
- explains just enough about how GPFS works to help you use it effectively, and
- offers basic usage advice for GPFS.

See the section above on [MPI-IO](#) (page 14) for a diagram that shows why some parallel file system (such as IBM's GPFS) is necessary to support efficient parallel I/O from many processes running concurrently on many separate compute nodes. See also the usage warnings and concerns [above](#) (page 17) about how careless parallel I/O can not only hurt your code's performance but also undermine I/O service for many users on many machines at once.

During 2007 LC is changing the names of its parallel file systems (including those based on GPFS) and experimenting with mounting the same parallel file system on multiple clusters for greater file-handling convenience. For a summary of the consequences of these changes for GPFS users, see the "Name and Mount Comparison" [section](#) (page 32) below.

## GPFS, NFS, DFS Compared

The Network File System (NFS) and the Distributed File System (DFS) are commercial file-system products designed to minimize file transfers by letting multiple computers all access a collection of files as if it were local them. LC uses NFS extensively and DFS experimentally. But neither NFS nor DFS is designed to

- let multiple (parallel) processes simultaneously read from or write to the same file from different compute nodes,
- scale across many I/O servers to avoid I/O bandwidth bottlenecks, and
- transparently balance incoming I/O data across all disks in the file system with a built-in striping algorithm (but RAID disks do provide behind-the-scenes striping at LC).

To support these special goals, LC has installed IBM's General Parallel File System (GPFS) on its AIX IBM SP machines (but *not* on its Linux machines, even though GPFS is also available for Linux). The parallel file system on LC Linux/CHAOS machines is Lustre, discussed in its own [section](#) (page 29) below.

This table summarizes the most interesting, user-relevant differences between the more familiar NFS and DFS file systems on the one hand and the less familiar but more parallel-friendly GPFS on the other. (NOTE: DFS is *not* supported on any LC Linux/CHAOS machine nor on any LC IBM/AIX machine running any version of AIX later than 5.2.)

<b>File-System Features</b>	<b>NFS</b>	<b>DFS(*)</b>	<b>GPFS</b>
Introduced:	1985		1998
Original vendor:	Sun	Open Software Foundation	IBM
Example at LC:	/nfs/tmpn	/dfs/proj	/p/gx1
Primary role:	Share files among machines	Fine-grained access control	Fast parallel I/O for large files
Easy to scale?	No	No	Yes
Network needed:	Any TCP/IP network	Any TCP/IP network	Only IBM SP "switch"
Access control method:	UNIX permission bits (CHMOD)	Access control lists (ACLs)	UNIX permission bits (CHMOD)
Block size:	256 byte		512 Kbyte (White)
Stripe width:	Depends on RAID	Depends on RAID	256 Kbyte
Maximum file size:	2 Gbyte (longer with v3)	2 Gbyte	26 Gbyte
File consistency:			
.....uses client buffering?	Yes	Yes	Yes (see diagram)
.....uses server buffering?			Yes (see diagram)
.....uses locking?	No	Yes	Yes (token passing)
.....lock granularity?		Whole file	Byte range
.....lock managed by?		I/O server node	Requesting compute node
Purged at LC?	Home, No; Tmp, Yes	No	<u>Yes</u>
Supports file quotas?	Yes	Yes	No

(See the glossary (page 5) above for definitions of the I/O terms in this table. See the next section (page 21) for an explanatory diagram showing how GPFS works.)

(\*)WARNING: DFS is not supported on any LC Linux/CHAOS machine and, starting in 2007, not supported on any LC IBM AIX machine running any version of AIX later than 5.2.

## How GPFS Works

This section explains in basic terms how the General Parallel File System (GPFS) works by tracing the data flow when a typical application writes a file to GPFS. The goal is to reveal just enough of GPFS's complex internal mechanisms to appreciate its strengths and weaknesses for I/O practice. The focus is on GPFS users, not system designers.

See the [next section](#) (page 25) for usage advice based on the local GPFS machinery and GPFS performance tests. See the general [glossary](#) (page 5) above for definitions of the I/O terms used in these steps.

This diagram shows what happens on the compute node (where your application runs) and on the I/O node (that services your write request) when you write to a GPFS file.



is available, the oldest used buffer is written to disk). Your data then moves from the application's data buffer into the GPFS pagepool buffer. This ends your "application I/O time," because now the application completes its write system call. Privately, GPFS schedules a worker thread to continue the write if and when the pagepool block is full, a technique called "write-behind caching."

3. Second CPU copy occurs.

The GPFS worker thread requests that the Virtual Shared Disk (VSD) client write the data to disk in GPFS-blocksized chunks. This request passes to the IP layer, where the write is broken up into 60-Kbyte IP message packets (called mbufs). Your data then moves to the local switch communications send pool (spool) buffers, the second time that the compute node CPU has copied it.

4. Your data moves across the switch.

The VSD client sends the IP packets with your data over IBM's high-speed interconnect among all SP nodes (called the SP switch). The VSD server on the I/O node collects the incoming packets in its receive pool (rpool) buffer.

5. Third CPU copy occurs.

Once the VSD server on the I/O node receives all the packets in your application's request, it allocates a buddy buffer (or queues the data in the switch receive pool until buffer space opens up). The buddy buffer reassembles the original chunk of data from the packets, making the third CPU copy through which it has passed.

6. Your data moves to disk.

When the buddy buffer is allocated, the VSD server releases all mbufs in the receive pool and calls the Logical Volume Manager to schedule a disk write through the device driver. The disk driver performs the write, perhaps waiting just enough to combine this data with other sequential writes to form an entire storage block (which on RAID disks, such as at LC, will equal the size of the RAID stripe).

7. Server notifies the client.

The VSD server releases the no-longer-needed buddy buffer on the I/O node and notifies the VSD client on the compute node that the write has safely completed.

8. Client completes the process.

The VSD client ends this multi-stage write process by making the previously committed pagepool buffer available for the next application call.

### Reading a File.

GPFS disk reads follow the same pattern as writes, but with data flowing in the reverse direction. Also, during reads GPFS tries to guess which data your application will request next and prefetch those blocks to the pagepool (if the guess is right, the performance gain is substantial).

# GPFS Usage Advice

## GPFS Purge Policy

Using GPFS (at LC) effectively and appropriately requires storing your files so that you avoid needlessly clogging the file system, and especially so that you avoid losing valuable data to the GPFS file purge.

- **Threshold:**  
LC purges GPFS when and only when usage exceeds 80%.  
Purging of files continues (oldest files first) until the file system is no more than 70% full (see schedule below).
- **Scope:**  
All GPFS files at least 13 weeks old (= 3 months) are eligible to be purged if the purge threshold is reached.  
All GPFS files 100 Kbyte or less in size are exempt from the purge regardless of their age.
- **Schedule:**  
If usage reaches the purge threshold during any month, then LC will start purging eligible GPFS files on the third Tuesday of that month (and continue until usage sinks to 70%).  
On the first Tuesday of every month, pre-purge logs are available for each user in a personal directory called

*/p/gxx/purgelogs/username*

(to help you anticipate which of your files are vulnerable for purge that month). Remember that every listed vulnerable file may not actually be purged, because the purge works through the list oldest to youngest only so far as needed to reduce usage to 70%.

See a later section for a [chart](#) (page 31) that compares the very different purge policies that apply to GPFS (AIX) and Lustre (Linux) parallel file systems.

## GPFS Usage Data

If you make use of GPFS on LC's AIX machines (/p/gb1, etc.), you may need to coordinate your disk space needs with other users on the same machine. Every LC AIX machine therefore offers a system file called `/usr/local/etc/pfs_status.machinename` that reports for each available parallel file system its current total size, space already used, percentage used, percentage of possible inodes (roughly, files) used, and an ordered list of users and their current space usage (in both Tbytes and number of files).

# I/O Analysis of FLASH

## What is FLASH?

FLASH is an adaptive-mesh parallel hydrodynamics code developed at the University of Chicago's Center for Astrophysical Thermonuclear Flashes ([flash.uchicago.edu](http://flash.uchicago.edu) (URL: <http://flash.uchicago.edu>)), a DOE "ASCI Alliance" site. FLASH simulates astrophysical thermonuclear flashes (such as supernovae and x-ray bursts) in two or three dimensions. Written in Fortran90, this code uses MPI for interprocess communication, relies on [HDF5](#) (page 7) for handling output data, and solves the compressible Euler equations on a block-structured adaptive mesh.

Because I/O is important for overall FLASH performance, this is an excellent test case for I/O optimization strategies (each FLASH run often generates 0.5 Tbyte of data, and I/O sometimes takes up as much as half of the total FLASH run time on 1024 processors).

## What is the FLASH I/O Benchmark?

The FLASH I/O benchmark tests FLASH's I/O performance independently of using the entire code. It sets up the same data structures as FLASH, fills them with dummy data, and then performs I/O through the HDF5 interface (or alternatives). The benchmark tests I/O performance on three kinds of files:

- Checkpoint Files--  
used to restart after a failed run, these files store all variables, the tree structure, the current simulation step, and the number of steps. Computational blocks account for more than 95% of the data written during each checkpoint, and 24 separate I/O operations (one per variable) are needed to write all of the computational blocks.
- Plot Files--  
are used for visualization runs. Once again, a separate I/O operation per variable is involved, but not all variables are stored and precision is reduced to 4-byte reals instead of 8-byte reals.
- Plot Files With Corners--  
similar to plot files but with an extra step added to generate a 9-by-9-by-9 interpolated block instead of the normal 8-by-8-by-8 block (to facilitate subsequent visualization).

FLASH performs I/O in this way to minimize the memory needed (a buffer to hold all of the variables for one single write would be very large) and because later data analysis is greatly aided by storing each variable separately.

## Current I/O Issues.

Possible general I/O optimization strategies (most related to HDF use by FLASH) that are currently under study at LLNL and other ASCI sites include:

- Storage Density--  
To store each variable in a separate record, single variables are extracted from the array of blocks, where the values are not contiguous in memory. FLASH extracts these values using the "hyperslab" feature of the HDF5 library.

- Record Size--  
The small records that FLASH writes at the beginning of its output may be as expensive as the large chunk that it dumps at the end. Packing small records, either in the code itself or by instructing the HDF5 library to buffer them before writing to disk, could significantly improve performance.
- Write Calls--  
FLASH issues only a single call to H5Dwrite for each variable stored. But within the library, HDF could make one compound MPI object to address the data or it could issue many separate write calls itself. Setting the "data transfer property" to use collective I/O should force HDF to use the first strategy instead of the second, perhaps making a significant difference in overall I/O performance.
- Two-Phase I/O--  
Experiments on ASCI Red show a fivefold increase in I/O rate by using two-phase I/O: first, collect output across processors into a buffer, then write a large contiguous chunk of memory to disk. This requires careful interaction among FLASH, the HDF5 library, and the MPI\_file\_open command, and the portability of that interaction remains to be tested on other machines (HDF5 features are known to vary among platforms).
- Split I/O--  
Normally the metadata for an XML-encoded HDF5 file resides in the same file as the data stored. Splitting the metadata into a file separate from the FLASH data itself might improve I/O, especially for situations (such as writing checkpoint files) where the likelihood of ever *reading* the file later is small so reading inefficiencies can be ignored.

# Lustre Parallel File System

## Lustre Goals and Roles

A parallel file system is part of any complete massively parallel computing environment (in fact, failure to use an available parallel file system and instead running parallel I/O to a traditional global file system such as `/nfs/tmpn` will degrade I/O performance for all users across all the machines that share that traditional file system).

In general terms, such a parallel file system:

- mounts on *every* compute and login node across the cluster that it serves,
- stores *very large* files efficiently, such as application-code data sets or restart dumps of runs that encounter trouble, and
- uses *high-speed* local communication paths to move data quickly to minimize I/O delays during code execution.

At LC, a parallel file system tailored to LLNL's specific computational needs and resource design policies must also:

- *scale up* to effectively serve clusters with over 1,000 nodes (and eventually those with over 10,000 nodes),
- rely on *open source* software (to maximize vendor flexibility and encourage collaboration with university researchers worldwide), and
- be *independent* of any single brand of storage-device hardware. We want to be able to change hardware vendors as new design features become available, and to make the most of our hardware funds.

LC's (collaborative) attempt to develop a practical parallel file system that meets these criteria is called Lustre (for "Linux Cluster"). The prime contractor is Cluster File Systems, Inc., whose own technical description of Lustre appears at its web site:

[www.clusterfs.org](http://www.clusterfs.org)

The other subsections of this section compare Lustre with GPFS (IBM's proprietary "general parallel file system"), describe the unusual implementation features that Lustre includes (as installed for production use on LC Linux machines), and explain how to cope with the currently known pitfalls or complexities that Lustre presents to users. LC's point of contact for users needing technical advice about the local Lustre file systems is Richard Hedges ([hedges1@llnl.gov](mailto:hedges1@llnl.gov)).

# Lustre and GPFS Compared

## Feature Comparison

LLNL deploys both GPFS (page 18) and Lustre parallel file systems (to support AIX and Linux/CHAOS compute clusters, respectively). Here is a feature-by-feature comparison of these two alternative solutions to the parallel file-system problem.

<b>File-System Features</b>	<b>Lustre</b>	<b>GPFS</b>
Introduced:	2003	1998
Vendor:	Cluster File Systems Inc.	IBM
Hardware:	Many brands	IBM only
Software:	GNU public license, open source	IBM proprietary
Switches allowed:	TCP/IP (Ethernet), Quadrics Elan 3 or 4, Infiniband	Storage Area Network (SAN), Network Shared Disk, or combination
Networking protocol:	Sandia's Portals API (open)	IBM proprietary
File locking:	Intent based (request + reason sent together)	Token passing
Lock granularity:	Byte range	Byte range
Data/metadata operations:	Separated, by different servers	Together, by the same servers
Scalability strategy:	(1) "object storage targets" manage data moves to actual disks, (2) metadata servers manage namespace	(1) same disks attached to all nodes, (2) storage nodes manage both data and metadata
POSIX compliant?	Yes (but lacks ACLs)	Yes

## Purge Comparison

File systems at or near their capacity often show degraded performance, higher I/O error rates, or sometimes complete service failure. To make service more predictable and reliable, LC intentionally destroys ("purges") files on at-risk file systems intended for temporary storage (especially the large NFS-mounted temporary file systems and the GPFS and Lustre parallel file systems).

This chart summarizes and compares the current LC file-purge policies on those file systems where LC regularly purges user files (without backup):

Purge Policy	/nfs/tmp*	GPFS (AIX)	Lustre (Linux)
Purged?	Yes	Yes	Yes
Backed up?	No	No	No
Usage threshold that triggers a purge?	70%	80%	As needed
Purged down to what level?	50%	70%	As needed
Purge order?	Oldest files first	Oldest files first	Oldest files first
Eligible files:			
...Age (last accessed)?	Over 10 days(+)	Over 13 weeks = 91 days	Over 60 days
...Size?	Any size	Over 100 kbyte	Any size
Schedule:			
...Purge cycle?	Nightly (if needed)	Monthly, third Tuesday	As needed
...Prepurge inventory?	No	Yes, first Tuesday(#)	No

(+)Over 5 days if usage reaches 90% since the previous day.

(#)Prepurge logs for each user are available at `/p/gX1/purgelogs/username`, where X is the relevant machine abbreviation (e.g., gum1) and *username* is your login name. Each log lists your specific files that would be purged unless you store and delete them beforehand.

## Name and Mount Comparison

Parallel file systems are specially designed to efficiently support large-file parallel I/O from every compute node in a cluster. At LC, the installed parallel file systems are Lustre (page 29) for Linux/CHAOS machines and GPFS (page 18) for IBM/AIX machines. LC is currently changing its approach to parallel file systems to introduce:

- new file-system *names* that generalize more easily and that emphasize the temporary nature of data placed on these devices, and
- the *mounting* of (some) parallel file systems across multiple clusters for greater convenience with less need to move files between like machines.

Eventually, this section will offer a unified summary of the parallel file system features (as changed by the two adjustments mentioned above, if they succeed) that are now discussed separately for IBM (AIX) and Linux (CHAOS) clusters for historical reasons. During this (probably year-long) transition, however, this section explains the in-coming naming scheme for parallel file systems. See also the warnings at the end of this section.

### Old Names:

Under the old naming scheme used through 2006, each LC parallel file system had a name of the form

*/p/gabbrnum*

where *abbr* was a one- or two-letter abbreviation for the machine on which the file system was mounted (e.g., b for BlueGene/L, um for UM) and *num* was the digit 1 or 2. For example, */p/gum1* was the parallel file system on UM.

### New Names:

Under the new naming scheme phased in starting in late 2006, each LC parallel file system gets a name of the form

*/p/[l|g]scratch[ocfletter|scfnumber]*

where

- |                  |   |
|------------------|---|
| <i>l</i>         | (literal lowercase el) indicates a Lustre (Linux/CHAOS) file system,              |
| <i>g</i>         | (literal) indicates a General Parallel File System or GPFS (IBM/AIX) file system, |
| <i>ocfletter</i> | is a unique one-letter identifier for OCF systems (a, b, c, etc.), and            |
| <i>scfnumber</i> | is a unique one-digit identifier for SCF systems (1, 2, 3, etc.).                 |

Example:

	OCF (Thunder)	SCF (Purple)
Old name	/p/gt1(*)	/p/gp1
New name	/p/lscratcha	/p/gscratch1
Alias	/p/glocal1	/p/glocal1

(\*)Discontinued as a separate file system in August, 2007.

Note that the script-stabilizing alias names continue to follow the old, not the new, scheme (on the use of g and of numbers for OCF).

Warnings:

(A) To discover which naming scheme, old or new, is currently in place on any particular machine where you plan to run jobs, log on to that machine and try to CD into a directory with each alternative name (or CD to /p and run LS). Note that just before each name change, the old directory remains for a few weeks in *read-only* status, so also check that you can actually create files once you are in any parallel file system.

(B) Further complicating this naming and cross-mounting transition are numerous reliability and performance problems with the Lustre file systems at LC. Underlying hardware failures are causing lost files and long, often unpredictable down times for repairs and rebuilds. Some lscratch*n* file systems will be unavailable for a month or more at a time to allow for part replacements and debugging during parts of 2007.

# Lustre LLNL Implementation

## LLNL's Lustre Strategy

### DESIGN.

Two key design features distinguish Lustre's implementation from other parallel file systems:

- Division of Labor.

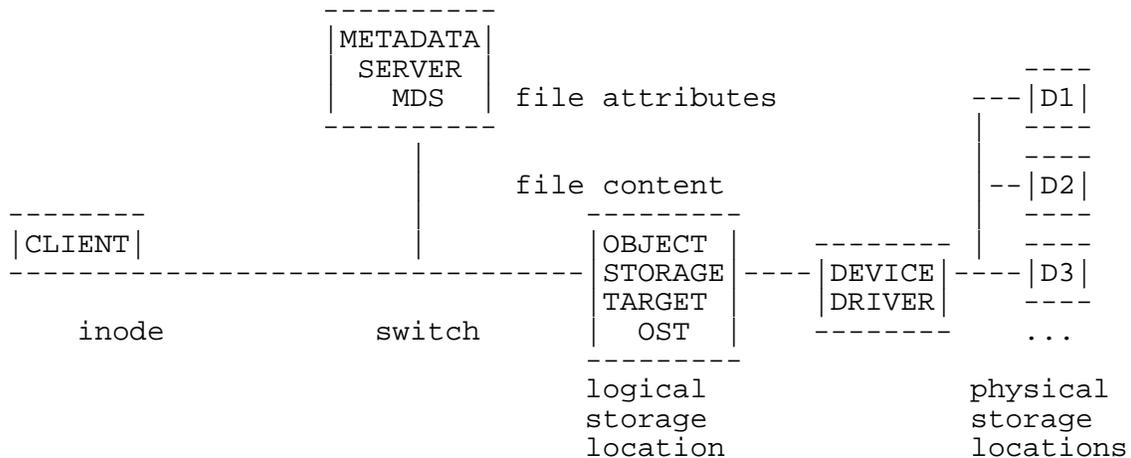
Lustre separates a file's *metadata* (its attributes and status information) from the file's *data* (its "content" that a program wants to get or put). Each is handled by its own separate server(s) for efficiency.

- Object Approach.

All actual file I/O is mediated by one or more "object storage targets" (OSTs), on which files seem to reside but which really mask the character of the underlying physical storage devices (could be multiple disks or other "file systems" of various sizes or brands). This approach promotes flexibility, reliability, and scalability.

### FRAMEWORK.

This diagram shows how the logical parts of Lustre fit together (from a user's viewpoint):



An "inode" is the standard UNIX data structure for a file (or directory or symbolic link to either one). When your application "creates a file" (inode) on Lustre, the system's unusual design features come into play, as described below.

### METADATA SERVER.

A Lustre Metadata Server (MDS; there are usually several for speed and failover redundancy) manages all "namespace operations" relevant to your file, such as assigning or updating references to the file's

- name,
- owner,
- permissions,
- access (conflict-control) locks, and

- (apparent) location on an OST (or striped across several OSTs).

The MDS does *not* participate in data transfers between the program and storage devices, however (those occur directly between the client node and the OSTs involved to save overhead). Nevertheless, LC Lustre users often encounter slow metadata performance (such as slow response to LS when executed with options like -l).

#### OBJECT STORAGE TARGET (OST).

Each Lustre OST is a server devoted to file I/O operations. It handles reads and writes of file *data*, but it talks to the Metadata Server (MDS) only if namespace changes for the file are needed. OSTs can "fill up" as if they were disks and can share files by striping (as if they were RAID disks). But files actually reside on lower-level physical devices managed indirectly by device drivers. These underlying storage devices are sometimes called "object-based disks" (OBDs) to emphasize how much their physical traits are hidden from your application program: they can be generic or customized and branded, can include nondisk storage, and can be upgraded to newer technology "below" an OST without disturbing that OST's consistent, reliable service to your program I/O requests. At LLNL, some OSTs come not from Cluster File Systems but from Blue Arc, an alternative brand. OSTs provide not only storage "abstraction," but also modular expandability: more OSTs can be easily added to an existing Lustre file system to expand the pool of logical locations for incoming files.

One drawback of allowing different brands and kinds of physical storage device to hide below the OST level is that your application program writing to Lustre may receive different exit or return codes at different times. Users often interpret these return-code differences as file system errors, when they usually just reflect subtle between-brand differences allowed within the POSIX specification. If you use Lustre extensively, change your application to overtly check I/O return codes so that you can appropriately ignore unimportant differences yet still detect file-corrupting genuine I/O errors.

#### NETWORK INDEPENDENCE.

At LLNL, Lustre uses either the Quadrics Elan or the InfiniBand network switch (depending on the Linux cluster where it is installed). Network independence is another Lustre design strength, facilitated by its use of the "Portals protocol stack," an abstract approach to networking originally developed at Sandia National Laboratory (but now available as open-source software).

#### SYSTEM ADMINISTRATION.

Lustre streamlines its own system administration by again relying on or coordinating with open standards, including these:

- "eXtensible Markup Language" (XML) to encode its configuration files in plain text.
- Light-Weight Directory Assistance Protocol (LDAP) to promote redundancy and easy recovery from infrastructure failures.
- Simple Network Management Protocol (SNMP).

## LLNL's Lustre Service

For each Linux/CHAOS production cluster at LC that currently has Lustre service, the comparison chart below shows:

- the name(s) of the public parallel file system(s) by which computer users access Lustre disk space on that cluster (but see the announcement about name changes during 2007, [above](#) (page 32)),
- the current total size of each parallel file system (but this may change because of on-going repairs or hardware failures; run `BDF pathname` for a current report on any specific file system),
- the number of "object-storage server" (OSS) nodes or gateway nodes that enable Lustre service on that cluster, and
- the underlying, often shared but hidden, "storage cluster" that contains the physical storage devices offered to users as a public Lustre parallel file system.

The covertly shared but hidden underlying "storage clusters" mentioned in the last item above provide a basis for mounting Lustre file systems *across* production Linux clusters. LC is experimenting with just such across-machine Lustre service during 2007. Gradual technical changes are underway that will yield different names and more cross-mounted Lustre parallel file systems, such as happened on Thunder in August, 2007 (all with the `lscratchn` [format](#) (page 32)).

Current Lustre File Systems at LC.

Cluster Name	Parallel File System Name(s)	Total Capacity (tentative)	Access Nodes	Underlying Storage Cluster
YANA, ZEUS, etc.	lscratcha	175 Tbyte		
	lscratchb	338 Tbyte		
	lscratchc	338 Tbyte		
	lscratchd	4 Gbyte		
ALC	lscratch[a-d]	see above		
THUNDER	lscratch[a-c]	see above		
LILAC, BG/L, etc.	gl1	4 Gbyte	32 OSS	(self,
	gl2	4 Gbyte	(Lilac)	Lilac)
	gb1 = lscratch1	403 Tbyte	1024 direct	BLC
	gb2 = lscratch2	403 Tbyte	connections	(224 OSS)
	lscratch3	930 Tbyte	(BG/L)	
Gauss(viz)	shares with BG/L	shares with BG/L	256 direct connections	BLC

# Lustre Operational Issues

This section briefly describes known Lustre usage (operational) issues or pitfalls, and it suggest ways to cope with each one.

## Directory Names and Aliases

### NAMES.

On each LC Linux/CHAOS cluster, some Lustre parallel file systems are mounted *locally* (like /usr/tmp) but now most are mounted globally (like your common home directory). The former (now being phased out) can be seen by every node in the cluster, but *only* by applications running on the specific cluster (such as LILAC) where it resides. Each remaining local Lustre file system has standard directory names of the form

```
/p/gX1/username  
[example: /p/g11/smith]
```

where

- /p indicates that this is a *parallel* file system (allows many nodes to interact appropriately with a file at the same time),
- /g indicates that all nodes on this cluster but *only those* can access this directory (but see the section [above](#) (page 32) for new name changes here),
- X is a lowercase one- or two-letter abbreviation for the single cluster that this file system serves (e.g., l for LILAC), and
- username* is your login name on the relevant cluster.

This Lustre naming scheme is the same as that used for GPFS directories that serve the IBM/AIX clusters at LC (but see the "Name and Mount Comparison" section [above](#) (page 32) for how these older names have largely been made obsolete with the arrival of cross-mounted Lustre file systems with global names).

### ALIASES.

One obvious drawback with machine-specific directory names such as /p/gm1/smith is that using them in job-control scripts will prevent moving those scripts to other Linux clusters without renaming all of the parallel directories mentioned. To circumvent this problem, LC automatically provides for each parallel directory a symbolic link (an alias) of the form

```
/p/glocal1/username
```

You can use this name in a portable script because files created in directory /p/glocal1/abc will (also) appear in the corresponding local /p/gX1/abc directory on each cluster X where the script runs.

## Lustre Purge Policy

Using Lustre (at LC) effectively and appropriately requires storing (archivally) your files so that you avoid needlessly clogging the parallel file system, and especially so that you avoid losing valuable data in case of a Lustre file purge. Once files are purged from Lustre they cannot be recovered, so use archival storage (see EZSTORAGE (URL: <http://www.llnl.gov/LCdocs/ezstorage>)) to protect your important content.

Starting in August, 2006, LC uses the policy below to purge all Lustre (Linux) parallel file systems (open and secure):

- **Threshold:**  
LC purges Lustre file systems on an on-going, as-needed basis, *without* promising that any specific usage level must be reached first to trigger a purge (this is different from both the GPFS and /tmp purges, which involve a prespecified usage threshold).
- **Scope:**  
All Lustre files not *accessed* for at least 60 days eligible to be purged at any time *regardless of size* (for GPFS the time scope is 90 days).
- **Schedule:**  
LC purges Lustre as soon as needed to maintain efficiency, *not* on a monthly or other periodic schedule (GPFS purges occur only on the third Tuesday of each month).

See an earlier section for a chart (page 31) that compares the very different purge policies that apply to GPFS (AIX) and to the Lustre (Linux) parallel file systems.

## MPI-IO Interaction with Lustre

Each implementation of MPI-IO (parallel I/O using the message passing interface library) depends for success, and certainly for whatever scalability it offers, on the underlying parallel file system that performs its requested I/O operations. On AIX systems, MPI-IO is strongly affected by IBM's GPFS support, and likewise on LC's Linux/CHAOS clusters, where Lustre supports MPI-IO requests. (WARNING: attempting MPI-IO to a standard shared file system such as /nfs/tmpn, or worse, to your common home directory, will severely degrade I/O performance for *all users* of that file system across *all machines* where it is mounted.)

### GENERAL ISSUES:

MPI-IO efficiency varies greatly depending on its underlying parallel file system for three reasons:

- File-system software ("middleware") often reorganizes (to suit itself) how programmatic I/O operations appear to the actual hardware that services them. For example, multiple small noncontiguous file requests may coalesce into one large(r) I/O step to reduce network traffic.
- File-locking is crucial for reliable simultaneous reads from or writes to (different parts of) one file, yet locking availability and grain size vary from one parallel file system to another.
- Management operations (open, close, resize) depend on each file system's API for their implementation (and hence for their efficiency) details.

The features of a parallel file system most likely to influence how well MPI-IO works and how easily it scales up as the number of nodes grows large include:

- Just how the file system supports noncontiguous I/O to distributed files.
- The system's "consistency semantics," that is, just when data and metadata are locally cached and when changes quickly propagate to all clients (after you write to a file, for example).
- Whether (and how) client-independent (across-node) "handles" (references to files) are available.

Some MPI-relevant features are advertised by parallel file-system vendors, while others are hidden or even proprietary. So as a user, you should expect significant and sometimes inexplicable differences in MPI-IO performance as you move your applications from one parallel file system to another (even within the LC computing environment). Sometimes simple changes from one login node to another cause major differences in resource contention and hence in file-transfer rates. See also the "Lustre Striping" subsection (page 42) below.

For example, in April 2004, a team at Argonne National Laboratory compared (URL: <http://www-unix.mcs.anl.gov/~thakur/papers/scalable-mpi-io.pdf>) several (ROMIO MPICH2) MPI-IO operations across six different parallel file systems, including two in use at LLNL (Lustre on ALC, and GPFS on IBM clusters). They found that the average time per file create (MPI\_File\_open) using 128 clients was one third *less* on Lustre than on GPFS (although both systems took much longer than some other competitors). But the average time to resize a file (MPI\_File\_set\_size) using 128 clients was six times *greater* on Lustre than on GPFS. Unknown internal mechanisms apparently account for these divergent scalability results. Caution and careful testing are therefore vital to manage such MPI-IO surprises.

#### CHANGES WITH LUSTRE 1.4.8:

Deploying CHAOS 3.2 (and later) starting in May, 2007, which includes Lustre 1.4.8, addressed three serious MPI-IO operational problems with earlier Lustre versions--

- Page Cache Flush--  
Under CHAOS 3.2, the SLURM epilog script that always executes immediately after your job script now flushes the page cache of Lustre pages (clean and dirty) after every job. This guarantees that the next job will start with all memory available and with no interference from delayed I/O.
- Assertion Failures--  
Under CHAOS 3.2 (Lustre 1.4.8) any Lustre assertion failures on a compute node cause the node to panic and jobs to completely terminate. Previous Lustre versions allowed nodes with assertion failures to lapse into a strange, partly failed state.
- FLOCK and FCNTL--  
Under CHAOS 3.2, system calls to FLOCK(2) and FCNTL(2) to lock Lustre files always return an error. This may affect some MPI-IO and HDF5 software. Previously, separate tasks running on different clients could use FLOCK or FCNTL to simultaneously obtain exclusive locks on the same file, clearly an operational mistake.

## Lustre Backup Policy

Because of the volume of material involved and the computationally high overhead for parallel file operations, LC does *not* backup its Lustre file systems. Should power failures or other unscheduled hardware problems occur, all of your data residing on any Lustre file system could be lost with no possibility for recovery. Also, LC currently does not provide redundant (failover) "object storage servers" (OSSs). So each OSS failure makes some data unavailable until hardware is repaired or replaced.

Hence, you should thoughtfully move or copy important Lustre files to duplicate (and safer) locations yourself. Moving large numbers of small files can be tedious, error-prone, and very network congestive, however. LC provides a special software tool, called HTAR, specifically designed to efficiently transfer very large file sets either directly to archival storage or (if you request) to another file system on another LC machine. Using HTAR to self-backup your Lustre files thus benefits you as well as other users (who avoid the congestion you could cause by using slower manual transfers).

HTAR resembles traditional TAR in many ways (but *not* in requiring duplicate local disk space to create its target archive file, a great benefit). To take full advantage of HTAR's efficient backup potential, consult the feature explanations and examples in the [HTAR Reference Manual](http://www.llnl.gov/LCdocs/htar) (URL: <http://www.llnl.gov/LCdocs/htar>).

## Lustre Striping

### DEFINITIONS.

Like many high-performance storage systems, Lustre uses *disk striping* to improve I/O speed: the system automatically divides the data to be stored into "stripes" and spreads those stripes across (some) available storage locations so that they can be processed in parallel. Since Lustre places files on logical "object storage targets" (OSTs), which manage the physical disk interactions hidden from the user, user data is striped across multiple OSTs to improve performance and to better balance the storage load.

Lustre is fairly fault tolerant (compared to GPFS and NFS), and it continues to operate even if one (or more) specific OST goes offline. Striping data widely across OSTs works against this reliability, of course. If any portion of your data resides on an OST that is down, then attempted access of that file returns an I/O error until the faulty OST returns to service. So it may be important to know which OSTs a particular file is spread across, or even to influence that spread. Two tools (see below) address this need on LC machines served by Lustre.

### DEFAULTS.

*Stripe width* is the number of devices (or, in the case of Lustre, the number of OSTs) across which a file is divided. LC assigns different default stripe widths to Lustre on different computers to take advantage of different storage resources as well as differences among each machine's "object storage servers" (OSSs, the Linux nodes that communicate with OSTs). BlueGene/L, for example, runs two OST processes on each OSS node. The current default Lustre stripe widths include:

BlueGene/L	1
ALC, ATLAS, etc.	2

### TOOLS.

To discover the distribution of file segments across Lustre OSTs, use LFIND. If you type

```
lfind filename
```

where *filename* resides on any Lustre /p/gXn file system, then LFIND returns a list of locally available OSTs (one per line, in order by their index numbers 0, 1, 2...n), the status of each OST (active/inactive), and then the list of specific OSTs on which the segments of *filename* currently reside (again, ordered by their index numbers). The first list may be long (for example, 18 OSTs support /p/lscratcha on ATLAS).

To change the default Lustre striping characteristics for a *new* (empty, not yet written) file *fname* or for a directory *dname* (so that new files written to it inherit those characteristics), use LSTRIPE. On a Lustre-served machine, type

```
lstripe fname | dname stripesize stripesize stripewidth
```

where

<i>stripesize</i>	specifies the number of bytes in each stripe (must be a multiple of 64 1-kbyte blocks or 65536 kbyte).
-------------------	--

*stripesstart* specifies the index number of the OST for the first stripe (randomly distributed for load balancing by default).

*stripewidth* specifies the number of OSTs over which to spread the stripes (default varies by machine, -1 specifies all OSTs).

(LSTRIPE creates *fname* when it runs; you must create *dname* with MKDIR.) Before using LSTRIPE consider the strategy issues below, because inappropriate choices here can cost you dearly in parallel I/O performance.

#### STRATEGY.

- (1) Creating a very large file (for example, a large TAR bundle of already large files) on a small number of OSTs will result in very suboptimal performance. Striping over more OSTs will use a larger fraction of the available storage devices (or simply avoid TARing files that are already quite large).
- (2) If your application program writes one file per process, then letting Lustre place these small separate files on different OSTs "round robin" (the default) will beneficially balance the load on the underlying storage devices. Striping over many (or all) OSTs here degrades performance.
- (3) If your application program instead has all of its parallel processes write to different parts of a single shared file then you will probably need to help Lustre widely distribute this load. In this case, striping the big shared file over many (or all) OSTs is probably very desirable.

## Lustre Groups

### PROBLEM.

Support for group access to Lustre files is sometimes faulty or absent for groups other than your primary group. You can always share Lustre files with others in your primary group. But Lustre may not give you access to files assigned to other groups to which you belong, files that you could routinely share on other file systems.

### INTERACTIVE SOLUTION.

You can (temporarily) change your primary group to one of your secondary groups, to enable more flexible access to Lustre files, by running

```
newgrp group2
```

where *group2* is any one of your nonprimary group names. This spawns a new shell, in which you have a different "real and effective" group ID. Remember, however, that all variables that you have not explicitly EXPORTed to this new shell will revert to null or to their default values. Typing NEWGRP with no argument restores your primary group to its original value (as specified in your password-file entry).

### BATCH SOLUTION.

To adapt this same strategy for use in a batch script, insert the line

```
newgrp group2 << EOFMARK
```

as the first executable line in your script (immediately after your #PSUB directives and any introductory comments). Then insert the line

```
EOFMARK
```

as the very last line of your batch script.

# Disclaimer

---

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government thereof, and shall not be used for advertising or product endorsement purposes.

(C) Copyright 2007 The Regents of the University of California. All rights reserved.

---

# Keyword Index

To see an alphabetical list of keywords for this document, consult the [next section](#) (page 48).

Keyword	Description
<u>entire</u>	This entire document.
<u>title</u>	The name of this document.
<u>scope</u>	Topics covered in this document.
<u>availability</u>	Where these programs run.
<u>who</u>	Who to contact for assistance.
<u>introduction</u>	Role and goals of this document.
<u>glossary</u>	Basic I/O terms and distinctions.
<u>hdf</u>	Hierarchical Data Format for I/O.
<u>hdf-features</u>	HDF5, HDF4 features compared.
<u>hdf-availability</u>	HDF under AIX and Linux at LC.
<u>hdf-file-structure</u>	HDF5 file organization, XML model.
<u>hdf-operations</u>	How to manipulate HDF files.
<u>mpi-io</u>	Portable parallel I/O library.
<u>mpi-io-role</u>	Parallel I/O interface in MPI-2.
<u>mpi-io-data-access</u>	Positioning, synchronism, coordination.
<u>mpi-io-issues</u>	MPI-IO local problems, features, tips.
<u>gpfs</u>	General Parallel File System (IBM)
<u>gpfs-compared</u>	GPFS, DFS, NFS compared.
<u>gpfs-steps</u>	How GPFS works.
<u>gpfs-advice</u>	GPFS usage advice and purge policy.
<u>gpfs-purge</u>	LC purge policy for GPFS.
<u>gpfs-usage</u>	Usage data reports for GPFS.
<u>flash</u>	I/O analysis of FLASH code.
<u>lustre</u>	Lustre parallel file system (Linux).
<u>lustre-goals</u>	Role of Lustre, design goals.
<u>lustre-compared</u>	Lustre compared with GPFS.
<u>feature-comparison</u>	Lustre and GPFS features compared.
<u>purge-comparison</u>	Lustre and GPFS purges compared.
<u>name-comparison</u>	Lustre, GPFS new name and mount comparison.
<u>mount-comparison</u>	Lustre, GPFS new name and mount comparison.
<u>lustre-design</u>	How Lustre is implemented at LC.
<u>lustre-strategy</u>	Components and roles for LLNL Lustre.
<u>lustre-service</u>	Current Lustre file systems at LC.
<u>lustre-advice</u>	Lustre usage advice, pitfalls.
<u>lustre-directory-names</u>	Names, aliases for Lustre directories.
<u>lustre-purge</u>	LC purge policy for Lustre.
<u>lustre-mpi-io</u>	How MPI-IO interacts with Lustre.
<u>lustre-backup</u>	Do your own backup with HTAR.
<u>lustre-striping</u>	Striping defaults, tools, strategy.
<u>lustre-groups</u>	Lustre group-support problems.

index

a

date

revisions

The structural index of keywords.

The alphabetical index of keywords.

The latest changes to this document.

The complete revision history.

# Alphabetical List of Keywords

Keyword	Description
-----	-----
<u>a</u>	The alphabetical index of keywords.
<u>availability</u>	Where these programs run.
<u>date</u>	The latest changes to this document.
<u>entire</u>	This entire document.
<u>flash</u>	I/O analysis of FLASH code.
<u>feature-comparison</u>	Lustre and GPFS features compared.
<u>glossary</u>	Basic I/O terms and distinctions.
<u>gpfs</u>	General Parallel File System (IBM)
<u>gpfs-advice</u>	GPFS usage advice and purge policy.
<u>gpfs-compared</u>	GPFS, DFS, NFS compared.
<u>gpfs-purge</u>	LC purge policy for GPFS.
<u>gpfs-steps</u>	How GPFS works.
<u>gpfs-usage</u>	Usage data reports for GPFS.
<u>hdf</u>	Hierarchical Data Format for I/O.
<u>hdf-availability</u>	HDF under AIX and Linux at LC.
<u>hdf-features</u>	HDF5, HDF4 features compared.
<u>hdf-file-structure</u>	HDF5 file organization, XML model.
<u>hdf-operations</u>	How to manipulate HDF files.
<u>index</u>	The structural index of keywords.
<u>introduction</u>	Role and goals of this document.
<u>lustre</u>	Lustre parallel file system (Linux).
<u>lustre-advice</u>	Lustre usage advice, pitfalls.
<u>lustre-backup</u>	Do your own backup with HTAR.
<u>lustre-compared</u>	Lustre compared with GPFS.
<u>lustre-design</u>	How Lustre is implemented at LC.
<u>lustre-directory-names</u>	Names, aliases for Lustre directories.
<u>lustre-goals</u>	Role of Lustre, design goals.
<u>lustre-groups</u>	Lustre group-support problems.
<u>lustre-mpi-io</u>	How MPI-IO interacts with Lustre.
<u>lustre-purge</u>	LC purge policy for Lustre.
<u>lustre-service</u>	Current Lustre file systems at LC.
<u>lustre-strategy</u>	Components and roles for LLNL Lustre.
<u>lustre-striping</u>	Striping defaults, tools, strategy.
<u>mount-comparison</u>	Lustre, GPFS new name and mount comparison.
<u>mpi-io</u>	Portable parallel I/O library.
<u>mpi-io-data-access</u>	Positioning, synchronism, coordination.
<u>mpi-io-issues</u>	MPI-IO local problems, features, tips.
<u>mpi-io-role</u>	Parallel I/O interface in MPI-2.
<u>name-comparison</u>	Lustre, GPFS new name and mount comparison.
<u>purge-comparison</u>	Lustre and GPFS purges compared.
<u>revisions</u>	The complete revision history.
<u>scope</u>	Topics covered in this document.
<u>title</u>	The name of this document.
<u>who</u>	Who to contact for assistance.

## Date and Revisions

Revision Date -----	Keyword Affected -----	Description of Change -----
22Aug07	<u>lustre-service</u> <u>lustre-directory-names</u> <u>lustre-striping</u>	THUNDER, ALC now use lscratch[a-c d]. THUNDER uses global directory names now. THUNDER no longer different.
10Jul07	<u>glossary</u> <u>gpfs-compared</u> <u>lustre-mpi-io</u> <u>lustre-striping</u>	Diskless nodes explained. DFS no longer supported at LC. Lustre 1.4.8 changes explained. Atlas replaces MCR in example.
13Feb07	<u>gpfs</u> <u>gpfs-compared</u> <u>lustre-service</u> <u>lustre-directory-names</u> <u>name-comparison</u> <u>index</u>	Name, mount changes underway. DFS access limited now. Details updated, name and mount changes. Cross refs added to new section. New section on name, mount changes. New keywords for new section.
16Aug06	<u>lustre-compared</u> <u>purge-comparison</u> <u>lustre-purge</u> <u>gpfs-purge</u> <u>index</u>	Section subdivided for better access. New subsection (detailed chart) added. Cross reference to chart added. Cross reference to chart added. New keyword for new section.
23May06	<u>lustre-strategy</u> <u>lustre-service</u> <u>index</u> <u>lustre-compared</u>	Design section subdivided. New section compares available systems. New keyword for new section. GPFS vs. Lustre details expanded.
22Mar06	<u>mpi-io-issues</u> <u>gpfs</u> <u>lustre-mpi-io</u>	Warning added about parallel I/O to globally mounted file systems. Cross ref to warning added. Warning enhanced.
26Sep05	<u>lustre-striping</u> <u>lustre-groups</u> <u>index</u> <u>lustre-goals</u> <u>lustre-design</u>	New section on striping issues. New section on group problems. New keywords for new sections. Warnings, cross ref added. OST return-code issues.
03Feb05	<u>lustre</u> <u>index</u> <u>introduction</u> <u>glossary</u>	New sections on Lustre file system. New keywords for new sections. Lustre added to stack. Lustre added to glossary.

	<u>mpi-io</u>	Lustre cross refs added.
	<u>gpfs-compared</u>	Lustre cross refs added.
08Jul03	<u>flash</u>	New section on FLASH use of HDF.
	<u>index</u>	New keyword for new section.
	<u>gpfs-advice</u>	GPFS usage/status files noted.
16Sep02	<u>hdf-availability</u>	Comparison chart updated.
	<u>gpfs-compared</u>	Comparison chart expanded.
	<u>gpfs-advice</u>	Purge policy clarified.
	<u>index</u>	Keyword error corrected.
13Aug02	entire	Expanded edition of I/O Guide for LC.
23Jul02	entire	First edition of I/O Guide for LC.
TRG (22Aug07)		

UCRL-WEB-201482

Privacy and Legal Notice (URL: <http://www.llnl.gov/disclaimer.html>)

TRG (22Aug07) Contact: [lc-hotline@llnl.gov](mailto:lc-hotline@llnl.gov)