

Course Title: Using the Lustre File System

1 Lustre Overview

1.1 Introduction

When using a Lustre file system, it is important to understand its underlying characteristics so you can take advantage of its possible optimizations. Otherwise, you may unnecessarily spend a large percentage of your allocation performing I/O.

This lesson will cover basic information about the Lustre file system to help you understand how to use it for your application. Covered topics include Lustre file system components, file striping, basic Lustre utilities, and Lustre best practices.

Upon completing this lesson you will be able to:

- Identify the purpose of each of the Lustre file system components.
- Understand how Lustre uses file striping to increase IO performance.
- Recognize the benefits and drawbacks of file striping.
- Recognize the impact of various Lustre stripe settings.
- Recognize how stripe alignment affects I/O performance.
- Use basic Lustre utilities when running your application.
- Follow best practices when using a Lustre file system.

A self test is provided at the end of this lesson to let you test your understanding of the topics covered.

1.2 File System Components

The Lustre file system is made up of an underlying set of I/O servers called **Object Storage Servers (OSSs)** and disks called **Object Storage Targets (OSTs)**. The file metadata is controlled by a **Metadata Server (MDS)** and stored on a **Metadata Target (MDT)**. A single Lustre file system consists of one MDS and one MDT. The functions of each of these components are described in the following list:

- **Object Storage Servers (OSSs)** manage a small set of OSTs by controlling I/O access and handling network requests to them. OSSs contain some metadata about the files stored on their OSTs. They typically serve between 2 and 8 OSTs, up to 16 TB in size each.
- **Object Storage Targets (OSTs)** are block storage devices that store user file data. An OST may be thought of as a virtual disk, though it often consists of several physical disks, in a RAID configuration for instance. User file data is stored in one or more objects, with each object stored on a separate OST. The number of objects per file is user configurable and can be tuned to optimize performance for a given workload.
- The **Metadata Server (MDS)** is a single service node that assigns and tracks all of the storage locations associated with each file in order to direct file I/O requests to the correct set of OSTs and corresponding OSSs. Once a file is opened, the MDS is not involved with I/O to the file. This is different from many block-based clustered file systems where the MDS controls block allocation, eliminating it as a source of contention for file I/O.
- The **Metadata Target (MDT)** stores metadata (such as filenames, directories, permissions and file layout) on storage attached to an MDS. Storing the metadata on a MDT provides an efficient division of labor between computing and storage resources. Each file on the MDT contains the layout of the associated data file, including the OST number and object identifier and points to one or more objects associated with the data file.

Figure 1 shows the interaction among Lustre components in a basic cluster.

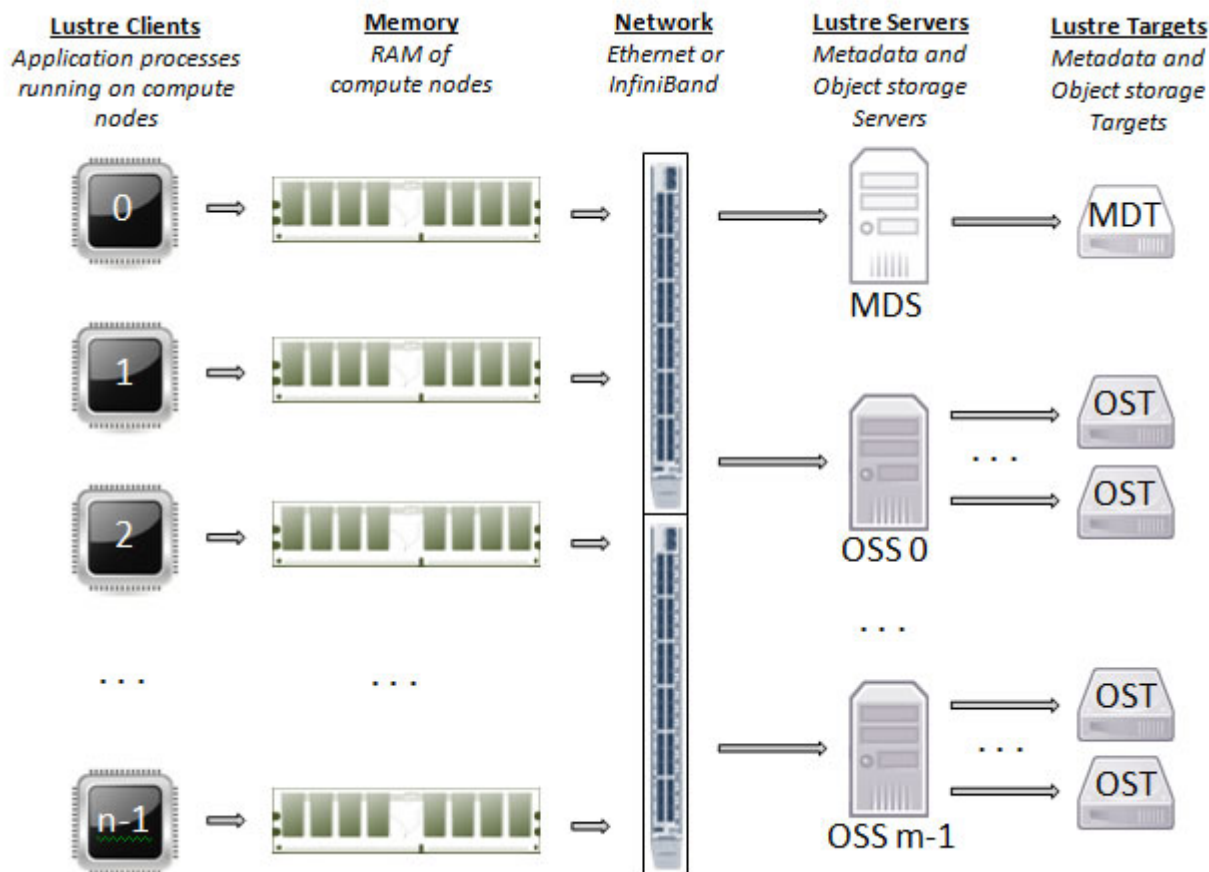


Figure 1: View of the Lustre File System

When a compute node needs to create or access a file, it requests the associated storage locations from the MDS and the associated MDT. I/O operations then occur directly with the OSSs and OSTs associated with the file bypassing the MDS. For read operations, file data flows from the OSTs to memory. Each OST and MDT maps to a distinct subset of the RAID devices. The total storage capacity of a Lustre file system is the sum of the capacities provided by the OSTs.

1.3 File Striping

1.3.1 File Striping Basics

A key feature of the Lustre file system is its ability to distribute the segments of a single file across multiple OSTs using a technique called *file striping*. A file is said to be *striped* when its linear sequence of bytes is separated into small chunks, or stripes, so that read and write operations can access multiple OSTs concurrently.

A file is a linear sequence of bytes lined up one after another. Figure 2 shows a logical view of a single file, File A, broken into five segments and lined up in sequence.

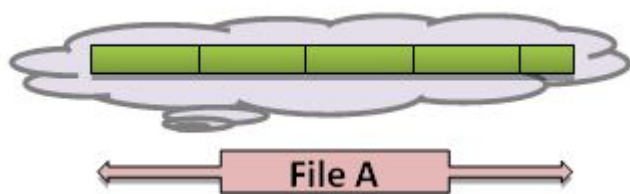


Figure 2: Logical view of a file.

A physical view of File A striped across four OSTs in five distinct pieces is shown in Figure 3.

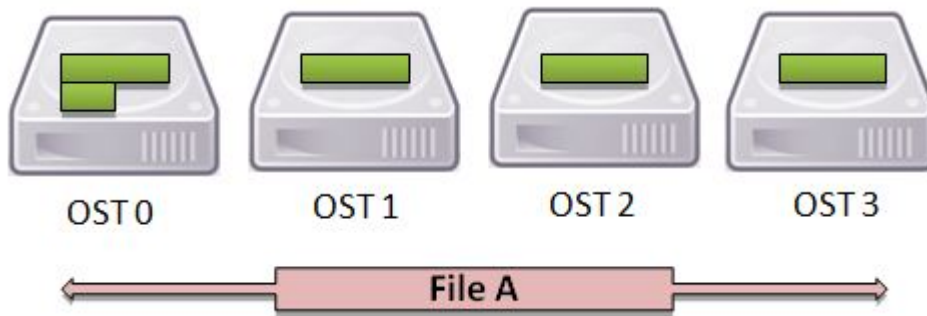


Figure 3: Physical view of a file.

The following values affect how a file will be striped:

- *Stripe count* - the number of OSTs over which a file is distributed.
- *Stripe size* - the number of bytes written on one OST before cycling to the next.

Lustre's file striping method is the same principal used in striping for a RAID array of disks in which data is striped across a certain number of objects. Lustre can stripe files across up to 160 objects and each object can be up to 2 TB in size. This leads to a maximum file size of 320 TB.

1.3.2 Benefits and Drawbacks

File striping increases IO performance since writing or reading from multiple OSTs simultaneously *increases the available IO bandwidth*. Many applications require high-bandwidth access to a single file, more than can be provided by a single OSS. Examples include a scientific application writing to a single file from hundreds of nodes, or a binary executable loaded by many nodes when an application starts. In addition, file striping *provides space for very large files*. This is possible since files that are too large to be written to a single OST can be striped across multiple OSTs.

However, file striping is not without drawbacks. In some cases, using it *can increase both the overhead and risk associated with I/O operations*. For example, the increased overhead of common file system operations, such as stat, can decrease I/O performance when file striping is used unnecessarily. When file striping is used correctly, any increase in overhead is completely hidden by file system parallelism. Additionally, *files stored across many hardware devices are at increased risk* if there is a hardware failure. When file striping is not necessary this risk may not be warranted. Typically though, the OSTs are configured in such a way as to minimize any risk due to hardware failure.

1.3.3 Stripe Alignment

Aligned Stripes

In Figure 3 we gave an example of a single file spread across four OSTs in five distinct pieces. Now, we add information to that example to show how the stripes are aligned in the logical view of File A. Since the file is spread across 4 OSTs the stripe count is 4. If File A has 9 MB of data and the stripe size is set to 1 MB it can be segmented into 9 equally sized stripes that will be accessed concurrently. The physical and logical views of File A are shown in Figure 4.

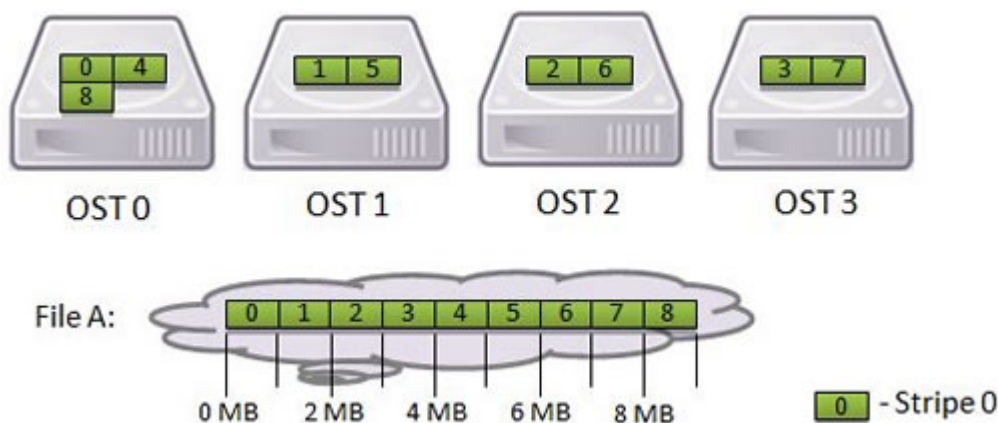


Figure 4: Physical and Logical Views of File A with Striping.

In this example, the I/O requests are *stripe aligned*, meaning that the processes access the file at offsets that correspond to stripe boundaries.

Non-aligned Stripes

Next, we give an example where the stripes are not aligned. Four processes write different amounts of data to a single shared File B that is 5 MB in size. The file is striped across 4 OSTs and the stripe size is 1 MB, meaning that the file will require 5 stripes. Each process writes its data as a single contiguous region in File B. No overlaps or gaps between these regions should be present; otherwise the data in the file would be corrupted. The sizes of the four writes and their corresponding offsets are as follows:

- Process 0 writes 0.6 MB starting at offset 0 MB
- Process 1 writes 1.8 MB starting at offset 0.6 MB
- Process 2 writes 1.2 MB starting at offset 2.4 MB
- Process 3 writes 1.4 MB starting at offset 3.6 MB

The logical and physical views of File B are shown in Figure 5.

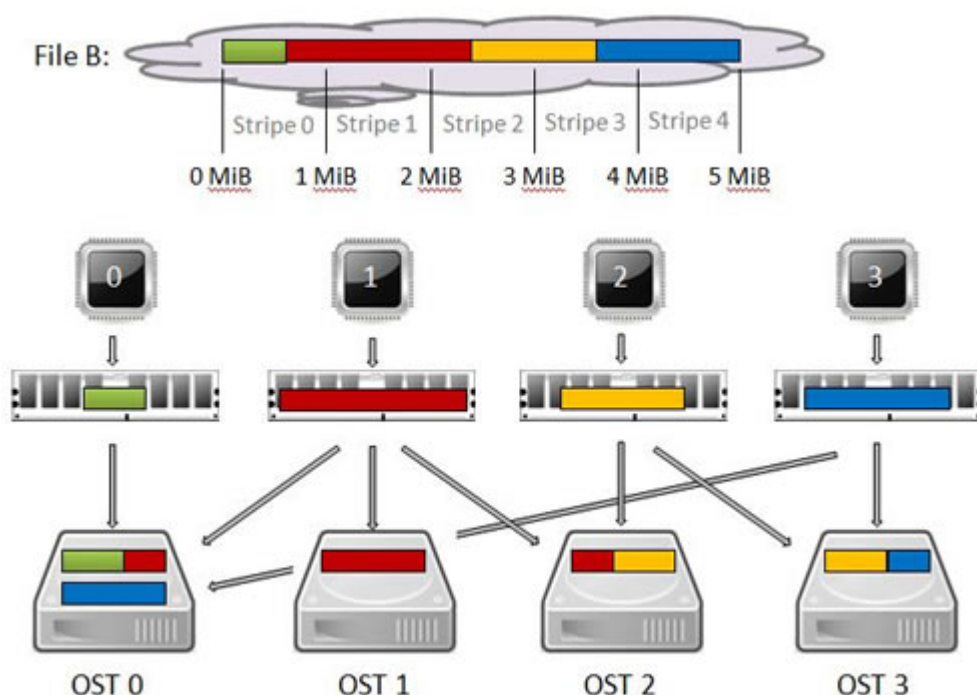


Figure 5: Logical and Physical Views of File B.

None of the four writes fits the stripe size exactly so Lustre will split each of them into pieces. Since these writes cross an object boundary, they are *not stripe aligned* as in our previous example. When they are not stripe aligned, some of the OSTs are simultaneously receiving data from more than one process. In our non-aligned example, OST 0 is simultaneously receiving data from processes 0, 1 and 3; OST 2 is simultaneously receiving data from processes 1 and 2; and OST 3 is simultaneously receiving data from processes 2 and 3. This creates resource contention on the OST and request contention on the OSS associated with the OST. This contention is a significant performance concern related to striping. It is minimized when processes (that access the file in parallel) access file locations that reside on different stripes as in our stripe aligned example.

1.3.4 Lustre Stripe Settings

You can set both the stripe count and size when using Lustre. In addition, you can specify the index value to set the starting number of the OST on which to place the first stripe. Lustre then uses a round-robin or weighted algorithm to assign which OST the next stripe is to be written. The descriptions and default values for these settings are given in the following table:

Setting	Default	Description
size	1 MB	Number of bytes in each stripe
index	-1, allows system to select	Starting number of OST to place the first stripe

count 4

Number of OSTs to stripe the file across

Note: Using the default index value is strongly recommended as it allows space and load balancing to be done by the MDS as needed.

Figure 6 below shows how 3 different sized files will be placed across 3 OSTs using varying stripe counts. The default stripe size (1MB) and the default stripe index value (-1) are used. The size of each file is:

- File A is < 1 MB
- File B is > 1 MB and < 2 MB
- File C is > 3 MB and < 4 MB

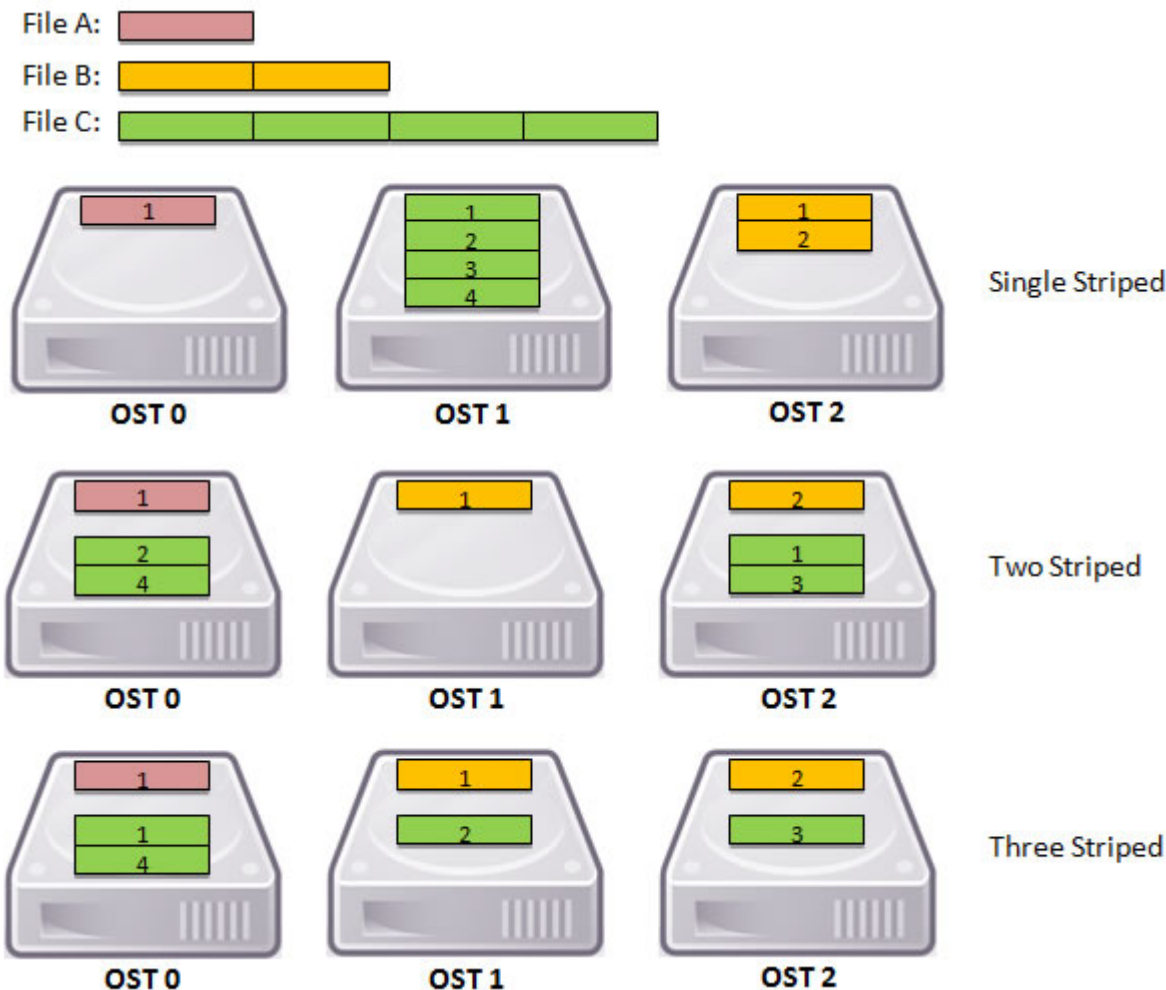


Figure 6: Example of three different striping patterns.

1.3.5 Choosing a Stripe Size

Your application's I/O pattern is the main factor in choosing a stripe size but there are some default values that you must consider. You will learn how to choose stripe sizes based on your application's I/O pattern in the section on Achieving Efficient I/O. Here, we provide some recommendations based on Lustre's default values.

- **The stripe size must be a multiple of the page size, which is 64 KB.** Lustre enforces this so users on platforms with smaller pages do not accidentally create files that might cause problems for users on platforms with larger pages.
- **The smallest recommended stripe size is 512 KB.** Although you can create files with a stripe size of 64 KB, the smallest practical stripe size is 512 KB. This is because Lustre sends 1MB chunks over the network so choosing a smaller stripe size may result in inefficient I/O to the disks and reduced performance.
- **The maximum stripe size is 4GB.** Using a large stripe size can improve performance when accessing very large files since it allows each client to have exclusive access to its own part of a file. However, if it does not match your I/O pattern it can be counterproductive.
- **For sequential I/O using high-speed networks, a good stripe size is between 1 MB and 4 MB. In most situations, stripe sizes larger than 4 MB may result in longer lock hold times and contention on shared file**

access.

1.4 Basic Lustre User Utilities

1.4.1 The lfs Utility

Lustre's **lfs** utility provides several options for monitoring and configuring your Lustre environment. In this lesson, we describe the basic options that enable you to:

- List OSTs in the File System
- Search the Directory Tree
- Check Disk Space Usage
- Get Striping Information
- Set Striping Patterns

For a complete list of available options, type help at the **lfs** prompt.

```
$ lfs help
```

To get more information on a specific option, type help along with the option name.

```
$ lfs help option-name
```

1.4.2 List OSTs in the File System

The **lfs osts** command lists all OSTs available on a file system, which can vary from one system to another. The usage for the command is:

```
lfs osts [path]
```

If a path is specified, only OSTs belonging to the specified path are displayed.

The **lfs osts** command displays the IDs of all available OSTs in the file system along with the default path to the file system, stripe count, stripe size, and stripe offset. The listing below shows the output produced by the **lfs osts** command on NICS's Kraken supercomputer:

```
$ lfs osts

OBDS:
0: scratch-OST0000_UUID ACTIVE
1: scratch-OST0001_UUID ACTIVE
.....//.....
334: scratch-OST014e_UUID ACTIVE
335: scratch-OST014f_UUID ACTIVE
/lustre/scratch
stripe_count: 4 stripe_size: 0 stripe_offset: -1
```

From this output you can see that Kraken (as of 5/11/11) has 336 OSTs, numbered from 0 to 335. In addition, the **lfs osts** command gives the path to the file system, /lustre/scratch, default stripe count (4), stripe size (0), and stripe offset (-1). This means that the files will be striped over 4 OSTs, the stripe size will be 1MB, and the MDS will choose the starting index. These are the default stripe setting values as described in the section on Lustre Stripe Settings.

Now, we give an example of the Lustre file system configuration for NCCS's Jaguar supercomputer (as of 5/11/11). Jaguar is composed of three Lustre file systems that can be used simultaneously. The listing below shows the output produced by the **lfs osts** command on Jaguar:

```
$ lfs osts

OBDS:
0: widow1-OST0000_UUID ACTIVE
...
671: widow1-OST029f_UUID ACTIVE
/lustre/widow1
(Default) stripe_count: 4 stripe_size: 1048576 stripe_offset: -1
OBDS:
0: widow2-OST0000_UUID ACTIVE
```

```
...
335: widow2-OST014f_UUID ACTIVE
/lustre/widow2
(Default) stripe_count: 4 stripe_size: 1048576 stripe_offset: -1
OBDS:
0: widow3-OST0000_UUID ACTIVE
...
335: widow3-OST014f_UUID ACTIVE
/lustre/widow3
(Default) stripe_count: 4 stripe_size: 1048576 stripe_offset: -1
```



Given the output shown above, what is the total number of OSTs on Jaguar?

[View Answer](#)

1.4.3 Search the Directory Tree

The **lfs find** command searches the directory tree rooted at the given directory/filename for files that match the specified parameters. The usage for the **lfs find** command is:

```
lfs find [[]] --atime|-A [-+ ]N [[]] --mtime|-M [-+ ]N
        [[]] --ctime|-C [-+ ]N [--maxdepth|-D N] [--name|-n ]
        [--print|-p] [--print0|-P] [[]] --obd|-O ]
        [[]] --size|-S [+ -]N[kMGTPe] --type | -t {bcdflpsD}]
        [[]] --gid|-g|--group|-G |]
        [[]] --uid|-u|--user|-U |]
        <dirname|filename>
```

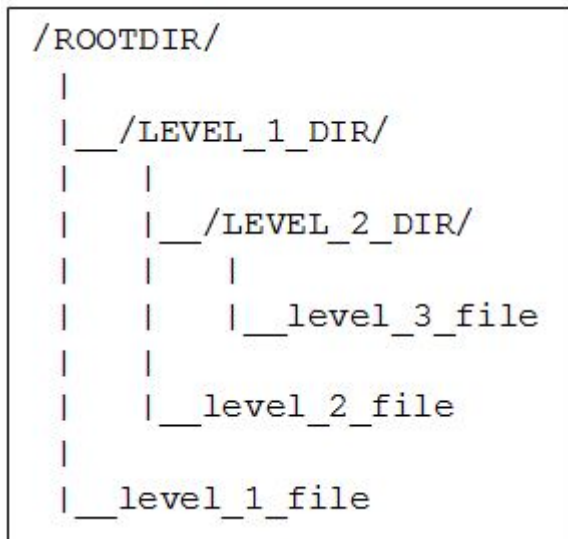
Note that it is usually more efficient to use **lfs find** rather than the **GNU find** when searching for files on Lustre.

Descriptions of the optional parameters for the **lfs find** command are given in the following table:

Parameter	Description
--atime	File was last accessed N*24 hours ago. (There is no guarantee that atime is kept coherent across the cluster.)
--mtime	File status was last modified N*24 hours ago.
--ctime	File status was last changed N*24 hours ago.
--maxdepth	Limits find to descend at most N levels of the directory tree.
--print / --print0	Prints the full filename, followed by a new line or NULL character correspondingly.
--obd	File has an object on a specific OST(s).
--size	File has a size in bytes or kilo-, Mega-, Giga-, Tera-, Peta- or Exabytes if a suffix is given.
--type	File has the type (block, character, directory, pipe, file, symlink, socket or Door [Solaris]).
--gid	File has a specific group ID.
--group	File belongs to a specific group (numeric group ID allowed).
--uid	File has a specific numeric user ID.
--user	File is owned by a specific user (numeric user ID is allowed).

Using an exclamation point “!” before an option negates its meaning (files NOT matching the parameter). Using a plus sign “+” before a numeric value means files with the parameter OR MORE. Using a minus sign “-” before a numeric value means files with the parameter OR LESS.

Consider an example of a 3-level directory tree shown below:



Results from using the **lfs find** command with various parameters are given in the following table:

Command	Result
lfs find /ROOTDIR	/ROOTDIR /ROOTDIR/level_1_file /ROOTDIR/LEVEL_1_DIR /ROOTDIR/LEVEL_1_DIR/level_2_file /ROOTDIR/LEVEL_1_DIR/LEVEL_2_DIR /ROOTDIR/LEVEL_1_DIR/LEVEL_2_DIR/level_3_file
lfs find /ROOTDIR --maxdepth 1	/ROOTDIR
or	/ROOTDIR/level_1_file
lfs find /ROOTDIR --maxdepth 1 --print	/ROOTDIR/LEVEL_1_DIR
lfs find /ROOTDIR --maxdepth 1 --print0	/ROOTDIR/ROOTDIR/level_1_file/ROOTDIR/LEVEL_1_DIR

The following example of using the **-mtime** parameter will result in a recursive list of all regular files in the directory /mnt/lustre that are more than 30 days old:

```
$ lfs find /mnt/lustre -mtime +30 -type f -print
```

1.4.4 Check Disk Space Usage

The **lfs df** command displays the file system disk space usage. Additional parameters can be specified to display inode usage of each MDT/OST or a subset of OSTs. The usage for the **lfs df** command is:

```
lfs df [-i] [-h] [--pool|-p <fsname>[.<pool>] [path]
```

By default, the usage of all mounted Lustre file systems is displayed. Otherwise, if a path is specified the usage of the specified file system is displayed.

Descriptions of the optional parameters are given in the following table:

Parameter	Description
-i	Lists inode usage per OST and MDT.
-h	Output is printed in human-readable format, using SI base-2 suffixes for Mega-, Giga-, Tera-, Peta-, or Exabytes.
--pool -p <fsname>[.<pool>]	Lists space or inode usage for the specific OST pool.

The **lfs df** command executed on NCCS's Jaguar supercomputer produces the following output:

```
$ lfs df

UUID                1K-blocks      Used Available  Use% Mounted on
widowl-MDT0000_UUID  7339473908   87887740 6832155768    1% /lustre/widowl[MDT:0]
widowl-OST0000_UUID  7433192488 3485286000 3570321764   46% /lustre/widowl[OST:0]
...
```



```
widow1-OST029f_UUID 7433192488 2644385276 4411222716 35% /lustre/widow1[OST:671]

filesystem summary: 4995105351936 2162036300436 2579331865060 43% /lustre/widow1

UUID      1K-blocks      Used Available Use% Mounted on
widow2-MDT0000_UUID 7339473908 25213776 6894829732 0% /lustre/widow2[MDT:0]
widow2-OST0000_UUID 7433192488 3214384884 3840980868 43% /lustre/widow2[OST:0]
...
widow2-OST014f_UUID 7433192488 3286417824 3768911620 44% /lustre/widow2[OST:335]

filesystem summary: 2497552675968 1083983770432 1286654861872 43% /lustre/widow2

UUID      1K-blocks      Used Available Use% Mounted on
widow3-MDT0000_UUID 7339473908 10452492 6909591016 0% /lustre/widow3[MDT:0]
widow3-OST0000_UUID 7433192488 2263463532 4792136852 30% /lustre/widow3[OST:0]
...
widow3-OST014f_UUID 7433192488 2000551164 5055054772 26% /lustre/widow3[OST:335]

filesystem summary: 2497552675968 683554460812 1687127537264 27% /lustre/widow3
```

You can see from this output that the file system is fairly balanced with none of the OSTs near 100% full. However, there are times when a Lustre file system becomes unbalanced meaning that one of the file's associated OSTs becomes 100% utilized. An OST may become 100% utilized even if there is space available on the file system. Examples of when this may occur include when stripe settings are not specified correctly or very large files are not striped over all of the OSTs. If an OST is full and you attempt to write to the file system, you will get an error message.

Lustre's OST pool feature is used to group OSTs together making object placement more flexible. A **pool**, also referred to as file system, is a name associated with an arbitrary subset of OSTs in a Lustre cluster. For example, the NCCS's Jaguar supercomputer currently (as of 5/11/11) has three OST pools (or file systems). Using three file systems allows NCCS to spread metadata load over multiple servers; it also allows a portion of users to run during periods when one file system is degraded or unavailable.

The following example shows limiting the scope to a specific OST pool. In addition, the `-h` parameter is used to give the disk space usage in human readable format for the OST pool `/lustre/widow1`:

```
$ lfs df -h -p /lustre/widow1

UUID      bytes      Used Available Use% Mounted on
widow1-MDT0000_UUID 6.8T 83.8G 6.4T 1% /lustre/widow1[MDT:0]
widow1-OST0000_UUID 6.9T 3.2T 3.3T 46% /lustre/widow1[OST:0]
...
widow1-OST029f_UUID 6.9T 2.5T 4.1T 35% /lustre/widow1[OST:671]

filesystem summary: 4.5P 2.0P 2.3P 43% /lustre/widow1
```

1.4.5 Get Striping Information

The **lfs getstripe** option lists the striping information for a file or directory. The syntax for the `getstripe` option is:

```
lfs getstripe [--obd|-o ] [--quiet|-q] [--verbose|-v]
               [--count|-c] [--index|-i | --offset|-o]
               [--size|-s] [--pool|-p] [--directory|-d]
               [--recursive|-r] ...
```

When querying a directory, the default striping parameters set for files created in that directory are listed. When querying a file, the OSTs over which the file is striped are listed.

Several parameters are available for retrieving specific striping information. These are listed and described in the table below:

Parameter	Description
<code>--obd <uuid></code>	Lists only files that have an object on a specific OST.
<code>--quiet</code>	Lists details about the file's object ID information.

- verbose** Prints additional striping information.
- count** Lists the stripe count (how many OSTs to use).
- index** Lists the index for each OST in the file system.
- offset** Lists the OST index on which file striping starts.
- pool** Lists the pools to which a file belongs.
- size** Lists the stripe size (how much data to write to one OST before moving to the next OST).
- directory** Lists entries about a specified directory instead of its contents (in the same manner as `ls -d`).
- recursive** Recurses into all sub-directories.

The following example shows that *file1* has a stripe count of six on OSTs 19, 59, 70, 54, 39, and 28. The **--quiet** parameter is used to list only information about a file's object ID.

```
$ lfs getstripe --quiet dir/file1

19          28675008      0x1b58bc0          0
59          28592466      0x1b44952          0
70          28656421      0x1b54325          0
54          28652653      0x1b5346d          0
39          28850966      0x1b83b16          0
28          28854363      0x1b8485b          0
```

The next example shows the output when querying a directory.

```
$ lfs getstripe dir1

...
dir1
stripe_count: 6 stripe_size: 0 stripe_offset: -1
...
```

1.4.6 Set Striping Patterns

Files and directories inherit striping patterns from the parent directory. However, you can change them for a single file, multiple files, or a directory using the **lfs setstripe** command. The **lfs setstripe** command creates a new file with a specified stripe configuration or sets a default striping configuration for files created in a directory. The usage for the command is:

```
lfs setstripe [--size|-s stripe_size] [--count|-c stripe_cnt]
              [--index|-i|--offset|-o start_ost_index]
              [--pool|-p <pool>]
              <dirname|filename>
```

Descriptions of the optional parameters are given in the following table:

Parameter	Description
--size stripe_size	Number of bytes to store on an OST before moving to the next OST. A stripe_size of 0 uses the file system's default stripe size, (default is 1 MB). Can be specified with k (KB), m (MB), or g (GB), respectively.
--count stripe_cnt	Number of OSTs over which to stripe a file. The default value for start_ost is -1 , which allows the MDS to choose the starting index. A stripe_cnt of 0 uses the file system-wide default stripe count (default is 1). A stripe_cnt of -1 stripes over all available OSTs, and normally results in a file with 80 stripes.
--index --offset start_ost_index	The OST index (base 10, starting at 0) on which to start striping for the file. The default value for start_ost is -1 , which allows the MDS to choose the starting index.
--pool <pool>	Name of the pre-defined pool of OSTs that will be used for striping. The stripe_cnt, stripe_size and start_ost values are used as well. The start-ost value must be part of the pool or an error is returned.

Shorter versions of these sub-options are also available, namely -s, -c, -i, -o, and -p as given in the usage above.

Although when using **lfs setstripe** you can specify option values based on position, it is best to use the explicit rather than the positional options. Using the positional options are error-prone and often misused. For example, it is best to use the following command:

```
lfs setstripe $NAME -s 1m -c 16
```

rather than

```
lfs setstripe $NAME 1m -l 16
```

Note that not specifying an option keeps the current value.

Setting the Striping Pattern for a Single File

You can specify the striping pattern of a file by using the **lfs setstripe** command to create it. This enables you to tune the file layout more optimally for your application. For example, the following command will create a new zero length file named file1 with a stripe size of 2MB, and a stripe count of 40:

```
$ lfs setstripe file1 -s 2m -c 40
```

Note: You CANNOT alter the striping pattern of an existing file with the **lfs setstripe** command. If you try to execute this command on an existing file, it will fail. Instead, you can create a new file with the desired attributes using **lfs setstripe** and then copy the existing file to the newly created file. The sample session below shows how to alter the striping pattern of an existing file using this method.

```
vbetro@krakenpf3:/lustre/medusa/vbetro/test> mv testfile testfile.orig
```

```
vbetro@krakenpf3:/lustre/medusa/vbetro/test> lfs setstripe -c 2 testfile
```

```
vbetro@krakenpf3:/lustre/medusa/vbetro/test> mv testfile.orig testfile
```

The session above shows how to change the stripe settings on an existing file using the **lfs setstripe** command. First, the UNIX **mv** command is used to rename the existing file from testfile to testfile.orig. This is done to save the contents of the file so they may be transferred to the newly striped file. Next, a new file named testfile is created with the desired stripe settings using the **lfs setstripe** command. Note that by using the **lfs setstripe** command to create this file all new files created in this directory will have these same stripe settings. Finally, the original file's contents are moved into the striped file by renaming it from testfile.orig to testfile. This works because we have essentially created a new file that now contains the original content and has the new stripe settings.

Setting the Striping Pattern for a Directory

Invoking the **lfs setstripe** command **on an existing directory** sets a default striping configuration for any new files created in the directory. Existing files in the directory are not affected. For example, to limit the number of OSTs to 2 for all new files to be created in an existing directory dir1 you can use the following command:

```
$ lfs setstripe dir1 -c 2
```

Note: The directory must exist before using **lfs setstripe** to set its stripe pattern. This is the opposite of what you just learned about using **lfs setstripe** to set the stripe pattern on a single file. You can alter the stripe pattern of an existing directory but cannot alter the stripe pattern of an existing file using the **lfs setstripe** command.

Setting the Striping Pattern for Multiple Files

You can't directly alter the stripe patterns of a large number of files with **lfs setstripe** but you can do it by taking advantage of the fact that files inherit the directory's settings. First, create a new directory setting its striping pattern to your desired settings using the **lfs setstripe** command. Then copy the files to the new directory and the files will inherit the directory settings that you specified.

Using the Non-striped Option

There are times when striping will not help your application's I/O performance. In those cases, it is recommended that you use Lustre's non-striped option. You can set the non-striped option by using a stripe count of 1 along with the default values for stripe index and stripe size. The **lfs setstripe** command for the non-striped option is as follows:

```
$ lfs setstripe dir1 -c 1
```

Striping across all OSTs

You can stripe across all or a subset of the OSTs by using a stripe count of -1 along with the default values for stripe index and stripe size. The **lfs setstripe** command for striping across all OSTs is as follows:

```
$ lfs setstripe dirl -c -1
```

1.5 Lustre Best Practices

A Lustre file system is shared among many users and application processes and its overall performance is dependent on their behavior. Some practices, such as choosing inappropriate stripe settings or excessive use of operations with high overhead, will cause contention for its resources. There are a number of best practices you should follow to reduce contention and hence improve performance. The most common of these are listed and described below.

Use the **ls -l** command only when absolutely necessary.

The **ls -l** command must communicate with every OST that is assigned to a file being listed and is done for every file listed. This makes it a very expensive operation that causes excessive overhead. This overhead can impact other users who will perceive the file system as unresponsive.

Open files read-only whenever possible.

If a file to be opened is not subject to write(s), it should be opened as read-only. Furthermore, if the access time on the file does not need to be updated, the open flags should be `O_RDONLY | O_NOATIME`. If this file is opened by all files in the group, the master process (rank 0) should open it `O_RDONLY` with all of the non-master processes (rank > 0) opening it `O_RDONLY | O_NOATIME`.

Read small, shared files from a single process.

If a shared file is to be read, and the data to be shared among the process group is less than approximately 100 MB, it is preferable to have one reader and then broadcast the contents rather than have everyone read the file. The MPI function used for broadcasting is `MPI_Bcast()`.

Limit the number of files in a single directory using a directory hierarchy.

For large scale applications that are going to write large numbers of files using private data, it is best to implement a subdirectory structure to limit the number of files in a single directory. A suggested approach is a two-level directory structure with \sqrt{n} directories each containing \sqrt{n} files, where n is the number of tasks.

Consider using available I/O middleware libraries.

For large scale applications that are going to share large amounts of data, one way to improve performance is to use a middleware library, such as ADIOS, HDF5, or MPI-IO.

1.6 Self Test

Lustre Overview



Question 1

Which Lustre file system component stores user data?

- ☐ Metadata Server
- ☐ Metadata Target
- ☐ Object Storage Servers
- ☐ Object Storage Targets



Question 2

What is the capacity of a Lustre file system with 32 OSS servers, each with two 16-TB targets?

[View Answer](#)

32 X (2 x 16 TB) = 1024 TB or approximately 1 PB. (The total storage capacity of a Lustre file system is the sum of the capacities provided by the OSTs.)



Question 3

Which one of the following stripe settings determines the degree of parallelism that can be achieved?

- ☐ stripe size
- ☐ stripe count
- ☐ start index




Question 4

The Figure below shows how 3 files are distributed across 3 OSTs. The default stripe size of 1MB and the default stripe index of -1 are used. Each file has the following size:

- File A is < 1 MB
- File B is > 1 MB and < 2 MB
- File C is > 2 MB and < 3 MB

File A: 

File B: 

File C: 



What stripe count would describe this pattern?

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ None of the above



Question 5

Which of the following index values would allow the system to automatically select where to place the first stripe?

- ☐ 1
- ☐ 0
- ☐ -1



Question 6

Match the following stripe settings with its description from the list below. (Enter the description's number in the box to the left.)

Stripe Size

Stripe Count

Stripe Index

1. Number of OSTs across which a file is distributed over.
2. Starting number of OST to place the first stripe
3. Number of segments in a striped file.
4. Number of bytes written on one OST before cycling to the next.

[View Answer](#)

[Restart](#)

[Show Answers](#)



Question 7

Consider the following two applications:

Application 1: Four processes write data in 1 MB chunks to a single shared file that is 100 MBs in size. Each process writes its data as a single contiguous region.

Application 2: Four processes write data to a single shared file that is 100 MBs in size. All processes write data in 1 MB chunks except for Process 3 which writes 2MBs. Each process writes its data as a single contiguous region.

Given a stripe count of 4 and stripe size of 1 MB, which application will achieve the best performance and why?

[View Answer](#)

Application 1 will have the best performance since the I/O requests are stripe aligned. Application 2 is not stripe aligned since Process 3 writes data in 2 MB chunks. This causes its data to be written across more than one OST creating resource contention on the OST and request contention on the OSS associated with the OST.



Question 8

Fill-in-the-blanks with the name of the Lustre lfs utility option that enables the action listed.

Set Striping Patterns - lfs

List OSTs in the File System - lfs

Search the Directory Tree - lfs

Get Striping Information - lfs

Check Disk Space Usage - lfs

[View Answer](#)

[Restart](#)

[Show Answers](#)



Question 9

You have just gotten a user account on a compute resource with a Lustre file system. Which command would you enter to list the available OSTs along with the default path to the file system, stripe count, stripe size, and stripe offset?

\$lfs

[View Answer](#)

[Restart](#)

[Show Answers](#)



Question 10

What command would you enter to create a file named astronomy1 in the directory /kraken/lustre that will be striped on six OSTs with 256 KB on each stripe?

[View Answer](#)

\$ lfs setstripe -s 256k -c 6 /kraken/lustre/astronomy1



Question 11

What command would you enter to **efficiently** list all files in the directory /kraken/lustre/ and its subdirectories?

[View Answer](#)

\$ lfs find /kraken/lustre

Although you can use the GNU find when searching for files on Lustre, it is usually more efficient to use the lfs find command.



Question 12

Given the following output from the lfs osts command on Kraken, what command would you enter to recursively list all files in the directory /kraken/lustre/ that have objects on OST number 334?

\$ lfs osts

OBDS:

0: scratch-OST0000_UUID ACTIVE

1: scratch-OST0001_UUID ACTIVE

.....//.....

334: scratch-OST014e_UUID ACTIVE

335: scratch-OST014f_UUID ACTIVE

/lustre/scratch

stripe_count: 4 stripe_size: 0 stripe_offset: -1

[View Answer](#)

\$ lfs find --obd scratch-OST014e_UUID /kraken/lustre/

The lfs find --obd option is used to list files that have an object on a specific OST(s). The unique identifier of the OST can be found from the listing given by the lfs osts command.



Question 13

You attempt to write to the file system and you get an OST full error message. To further investigate the problem you decide to check the disk usage on the system. What command would you enter to get this information in human readable

format?

[View Answer](#)

```
$ lfs df -h
```

The **lfs df** command displays the file system disk space usage. Specifying the optional **-h** parameter gives the output in human-readable format.



Question 14

What command would you enter to find the striping pattern for the file `/kraken/lustre/astronomy1`?

[View Answer](#)

```
lfs getstripe /kraken/lustre/astronomy1
```

When querying a file, the `lfs getstripe` command will list the OSTs over which the file is striped.



Question 15

You obtained the striping information below for the directory `dir1` by issuing the `lfs getstripe` command:

```
$ lfs getstripe dir1
```

```
...
dir1
stripe_count: 6 stripe_size: 0 stripe_offset: -1
...
```

You decide that you want to change the striping pattern for all of the files in the directory to a stripe count of 12 and stripe size of 256KB. How would you do that?

[View Answer](#)

You cannot directly change the stripe pattern on files within a given directory using the `lfs setstripe` command. However, you can create a new directory with the desired stripe pattern and then copy all of the files to the new directory. This approach works since the files will inherit the settings of the directory to which they are copied.

To create the new directory with the desired striping pattern, enter the command:

```
$ lfs setstripe newdir -c 12 -s 256KB
```

2 Achieving Efficient I/O

2.1 Introduction

In the previous lessons you learned about file striping and how it contributes to the high performance of the Lustre file system. You also learned how you can control striping by configuring the number of stripes, the size of each stripe, and the OSTs that are used. Now it's time to learn how you can use striping to make the most effective use of the Lustre file system so you can achieve efficient I/O performance from your application.

Upon completing this lesson you will be able to:

- Recognize the impact of various stripe settings on an I/O pattern(s)
- Select appropriate stripe settings for basic I/O patterns
- Choose stripe settings that minimize resource contention
- Manage free space to keep OST usage balanced

A self test is provided at the end of this lesson to let you test your understanding of the topics covered.

2.2 Application I/O Patterns

2.2.1 Introduction

Your choice of stripe settings is dependent on the I/O pattern(s) used in your application. Therefore, you need to understand the impact of various stripe settings on your application's performance so you can select the best settings for your application. It is important to realize that even though choosing the appropriate stripe settings will ensure the most effective use of the file system; they will not enhance, alter, or otherwise improve your application's I/O performance beyond what is possible from the I/O pattern it utilizes. Also note that selecting inappropriate stripe settings for your application's I/O pattern will degrade its I/O performance on the file system.

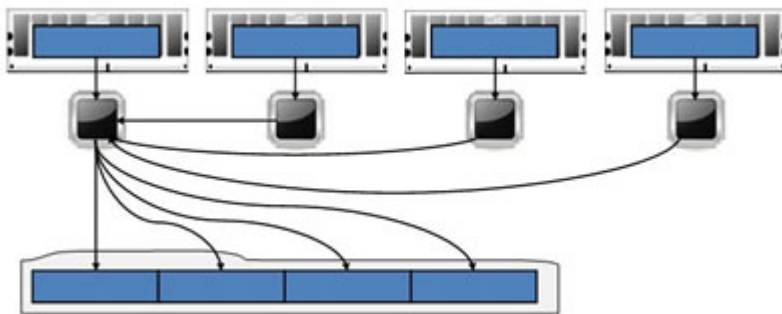
I/O patterns are characterized by the number of processes performing I/O and how the I/O is performed. In this lesson we look at the results of performance tests conducted on the Cray XT5 for the following three distinct application I/O patterns:

1. **Single Writer:** an application which only uses one process to perform I/O.
2. **Single Shared File:** an application in which all processes perform I/O while sharing access to a single file.
3. **File Per Process:** an application in which all processes perform I/O by writing an individual file per process.

These tests illustrate the impact of various stripe settings on application performance. Note that none of these I/O patterns should be considered as optimal. Depending on the application, it is most likely that some combination of the above patterns will be used.

2.2.2 Single Writer I/O

In **single writer I/O** one process aggregates data from all other processes and performs I/O operations to one or more files. This is depicted in the following image.



In general, single writer I/O is not scalable since I/O performance is limited by the single process performing I/O. Although you can use Lustre when using this I/O pattern, Lustre's parallelism cannot be exploited to increase I/O performance. The following test results illustrate the effectiveness of file striping on a single file when using single writer I/O.

Case 1: Write performance for single writer I/O at various Lustre stripe counts.

This test was performed to determine *the effect of stripe size on I/O performance*. A file size between 32 MB to 5 GB is utilized to keep the load per OST constant at 32MB. Transfer size of the write operations is 32 MB and the stripe size is changed between 1 MB and 32 MB. The graph in Figure 1 depicts the resulting I/O performance:

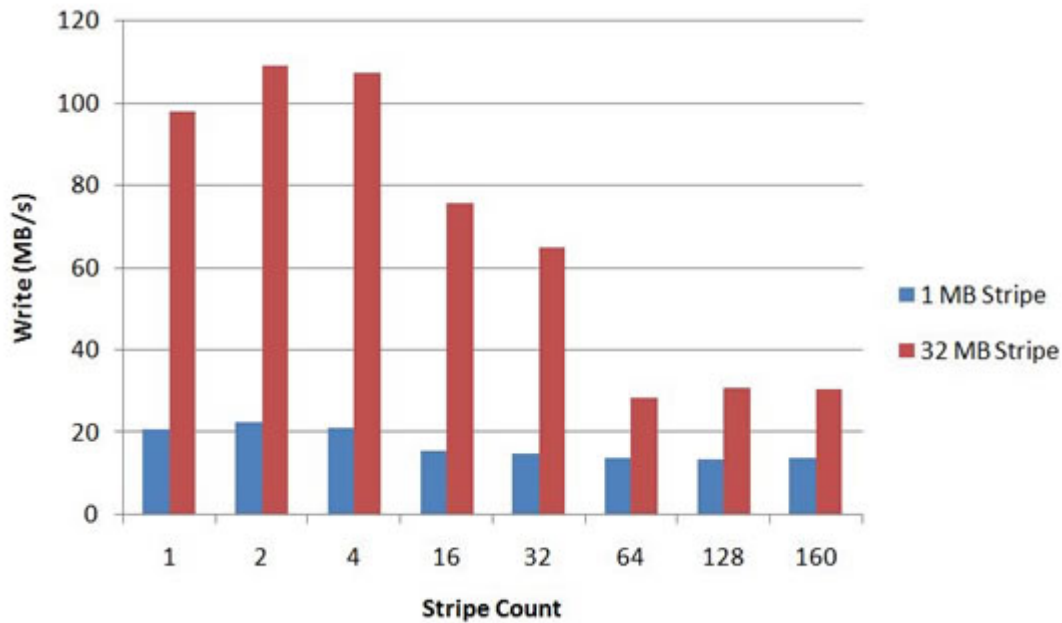


Figure 1: Single writer write performance.

These results suggest that utilizing more OSTs than the number of processes performing I/O does not increase write performance. In fact, performance decreases as the stripe count increases. This is because of the increased overhead from a single process accessing multiple OSTs for write operations. This illustrates the importance of minimizing the number of OSTs in which a process must communicate. The best performance is seen when the stripe size is set to 32 MB. Using this larger stripe size leads to a more contiguous write on the particular OST as compared to a 1MB stripe which may be distributed throughout the device.

Case 2: Write Performance for single writer I/O at various Lustre stripe sizes and I/O operation sizes.

This test was performed to see if there is a *correlation between the stripe size and transfer size*. In the previous test, the best performance was obtained with the larger stripe size of 32MB, which also matches the size of the write operations. Using a 256MB file, the transfer size is varied from 1, 8, and 32MB while the stripe size is varied from 1 to 128MB. I/O operations are performed on a single OST. The graph in Figure 2 depicts the resulting I/O performance:

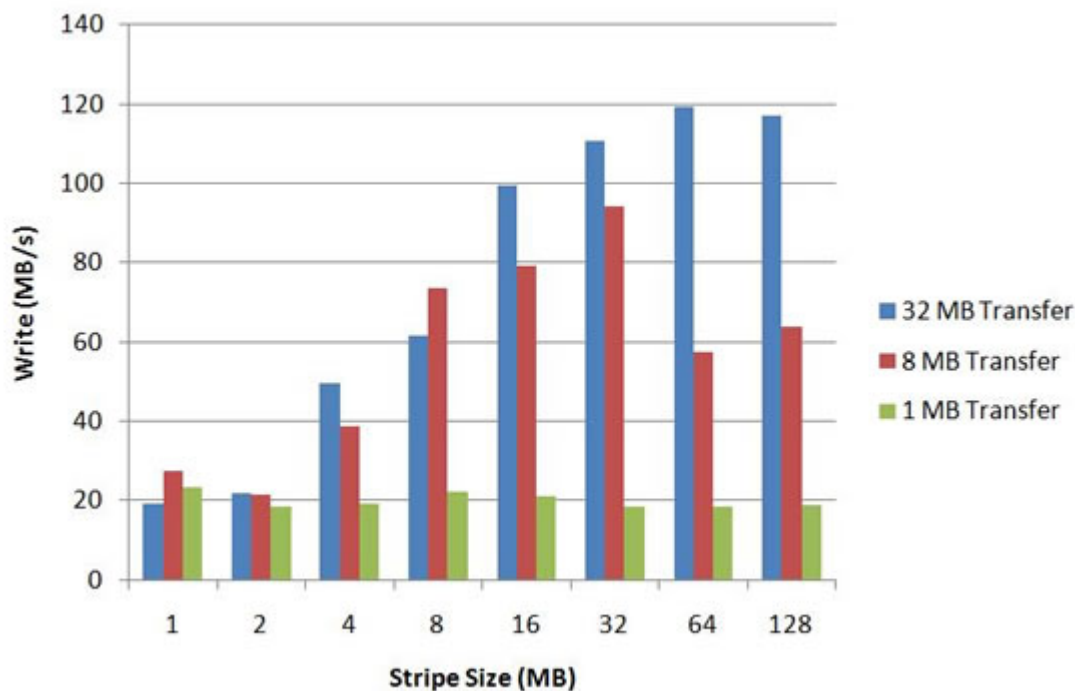
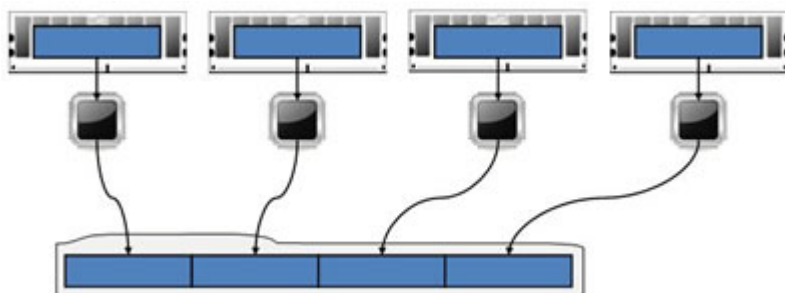


Figure 2: Single writer transfer versus stripe size.

The results show the importance of data continuity. For a 1MB transfer size, performance is unaffected by changes in the stripe size since the performance is effectively limited by the 1MB transfer size. For the 8 and 32MB transfer sizes, performance increases with the stripe size up to a point. The maximum performance for the 8 MB and 32MB transfer size occurs at a stripe size of 32 MB and 64 MB, respectively, and then decreases. Although maximum performance was achieved at stripe sizes slightly larger than the transfer sizes, *using a stripe size roughly equal to the transfer size is a good compromise* to reduce the possibility that either one will cause a decrease in performance.

2.2.3 Single Shared File I/O

In a single shared file application I/O pattern multiple processes perform I/O operations either independently or concurrently to the same file. This is depicted in the following image.



By using this I/O pattern you can take advantage of both Lustre's process and file system parallelism to achieve high levels of performance. There are still factors you must consider though since at large process counts contention for OSTs can hinder performance gains. The following I/O test results are used to illustrate the effectiveness of file striping on a single shared file.

Case 1: Write performance of a single shared file.

For this test, important application I/O characteristics include the transfer size and layout of the single shared file. From previous experience and results of the single writer test, we know to choose a stripe count that is at least equal to the number of processes performing I/O. This will minimize both the overhead associated with splitting an operation between OSTs and contention between processes over a single OST. The figure below shows the two file layouts used.

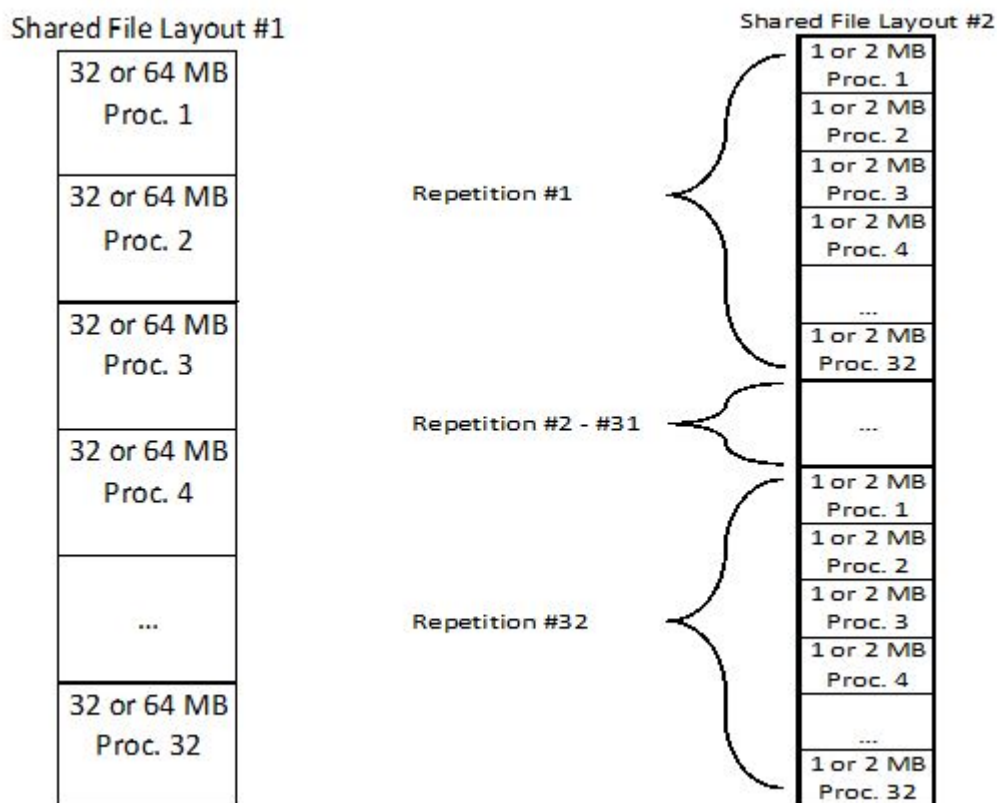


Figure 7: Two shared file layouts.

The major difference in file layouts is the locality of the data from each process. Layout #1 keeps data from a process in a contiguous block, while Layout #2 strides this data throughout the file. Thirty-two (32) processes will concurrently access this shared file.

In this test, processes write to a 1 GB and 2 GB shared file and the file is striped over 32 and 64 OSTs, respectively. To keep the load on each OST constant, the total file size is altered by increasing the amount of I/O performed by each process. The transfer size used is 32MB. The graph below depicts the resulting I/O performance:

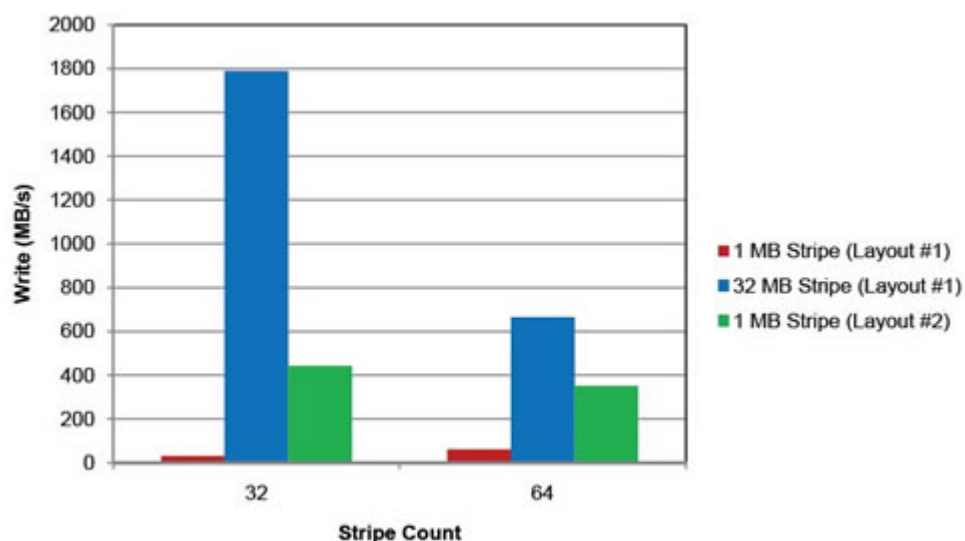


Figure 8: Single Shared File (32 Processes) 1 GB and 2 GB file

From these results we see widely varying performance for the differing file layouts and stripe patterns. The worst performance is seen with a 1MB stripe using file layout #1. This is because with the 1MB stripe size the 32MB block is broken into 1MB chunks for every process. Every process is writing to every OST causing process contention for OSTs as well as additional overhead from switching OSTs. This contention can be overcome by using a 32MB stripe size with file layout #1 which localizes each process's I/O to a single OST. Performance is reduced when increasing the stripe count to 64 due to the overhead of switching between OSTs for each process. The improved performance from using the 32MB stripe size comes from both data localization and continuity. To show the importance of these two factors a 1 MB stripe size is used with layout #2 providing for data localization but not data continuity. This gives better performance than the 1MB stripe size with layout #1 since each OST is accessed by only one process. However, the overall performance is lower due to the increased latency in the write (smaller I/O operations).

Case 2: Scaling of the Single Shared File

This test looks at the effect of file striping as the single shared file I/O pattern is scaled to larger process counts. A file size of between 1.25 and 252 GB and layout #1 given in the previous test with a 32 MB block size is used. The stripe count is set equal to the number of processes with a maximum setting of 160 (the maximum stripe count for the Lustre file system). The transfer size is set to 32 MB to maintain data locality and continuity. The graph below depicts the resulting I/O performance:

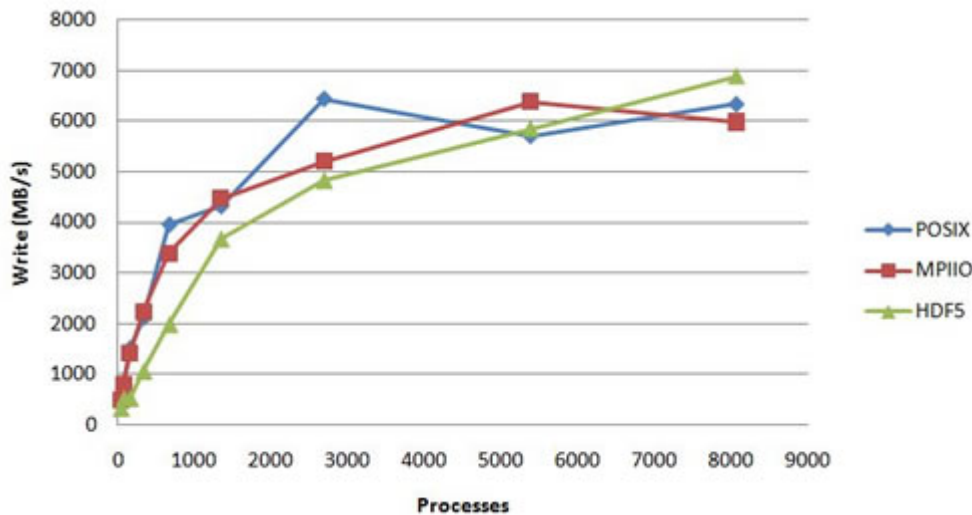


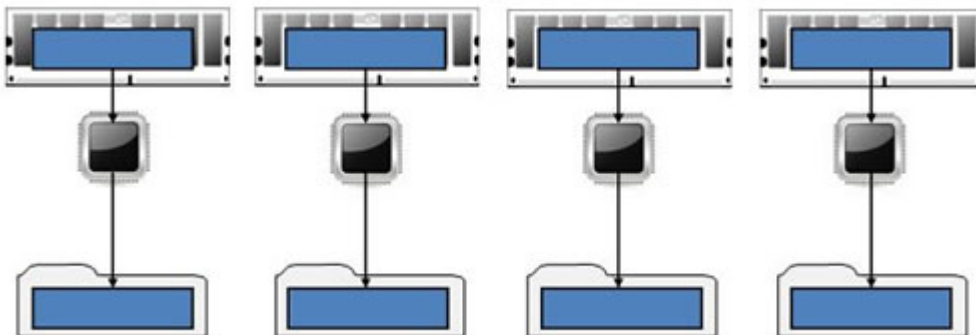
Figure 9: Scaling of a Single Shared File

Three parallel I/O libraries were used in this test - POSIX, MPI-IO, and HDF5. Their performance is very similar, peaking near 8000 processors at 5983 to 6886 MB/s write bandwidth. Note how the write performance levels off at large process counts. Performance increases dramatically at low process counts and continues to grow past 160 processes. Thus, the major limiting factor is the restriction of 160 OSTs.

These results also show the importance of data locality and continuity. By changing the stripe size to 1 MB, data from each process is spread across OSTs. Even though the overall trend of using a 1 MB stripe size is similar to that of the 32 MB stripe size, it causes a very large negative impact on write performance.

2.2.4 File Per Process I/O

The file per process I/O pattern scales the single writer I/O pattern to encompass all processes instead of just one process. In this I/O pattern each process performs I/O operations on their own separate file. Thus, for an application run of N processes, N or more files are created. This is depicted in the following image.



As with single writer I/O, this pattern is limited by each individual process which performs I/O. File Per Process I/O constitutes the simplest implementation of parallel I/O enabling the possibility of improved I/O performance from a parallel file system.

The graph below shows the write performance of this I/O pattern as a function of the number of processes or files.

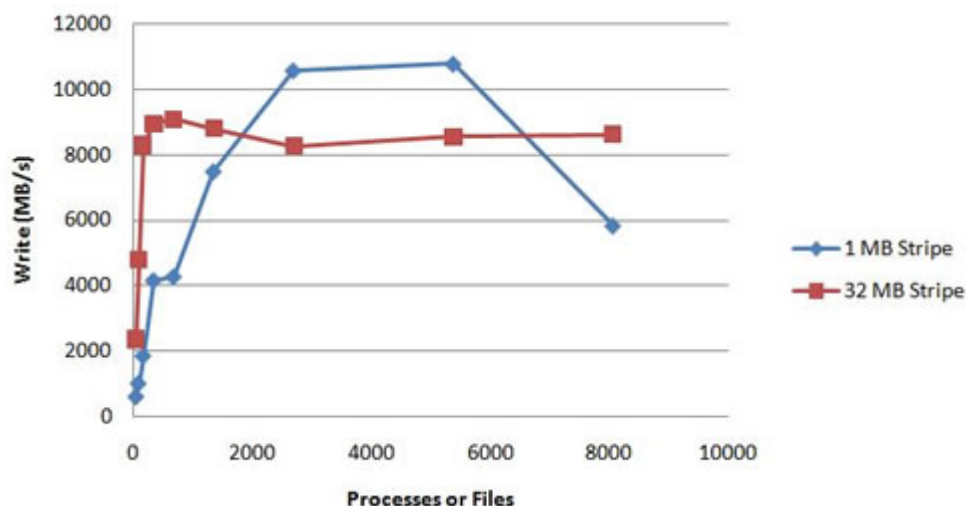


Figure 10: File per process write performance.

To be consistent with previous tests, each file is striped over a single OST chosen using a round-robin or load-based metric. Each file is 128 MB in size and utilizes a 32 MB transfer size. Both 1 MB and 32 MB stripe sizes are utilized. In both cases, performance increases as the number of processes or files increases until OST and metadata contention hinder performance improvements. The 1MB stripe size generally has lower performance but is the most scalable. This is mainly because it doesn't fill up its bandwidth as quickly. However, as you increase to 5, 6, 7000 processes, or files, the contention for OSTs and metadata from opening all of these files at the same time causes a decrease in performance. With the 32MB stripe size the performance peaks sooner because the bandwidth is filling up a lot quicker but performance still decreases due to file system contention. Although the 1MB stripe actually achieves better performance in the mid-range than the 32MB stripe it is much worse at lower and higher process or file counts. You may be able to predict where your application can get this higher performance with a 1 MB stripe on a dedicated machine but you can't on a shared resource. It is far more likely that performance will be on the lower end of the curves. Using the 32MB stripe more or less hits roughly 85% of the performance expectation and holds it pretty much level, which is consistent with the performance of applications running on Kraken in capability mode.

2.3 Minimizing Resource Contention

2.3.1 Introduction

In the section on file striping you learned that striping has its drawbacks. One of these is that it can increase overhead due to resource contention on the OST. To minimize resource contention you need to choose the best stripe settings for your application's I/O pattern. Otherwise, resource contention can seriously degrade your application's I/O performance.

The resource contention on the OST is the result of many processes trying to simultaneously access that OST. This contention is minimized when processes, which access the file in parallel, access file locations that reside on different OSTs. Additionally, performance can be improved by minimizing the number of OSTs in which a process must communicate. This can be done by stripe aligning the I/O requests and by setting the striping so that a single process does not access more than one or two OSTs.

Answering the following three questions when choosing your stripe settings will help you minimize contention:

1. How can I limit the number of processes simultaneously accessing one OST?
2. How can I limit the number of OSTs accessed by each process?
3. How can I limit the number of processes accessing each stripe?

In the following sections we answer these questions with respect to the application I/O patterns previously described.

2.3.2 Single Writer I/O

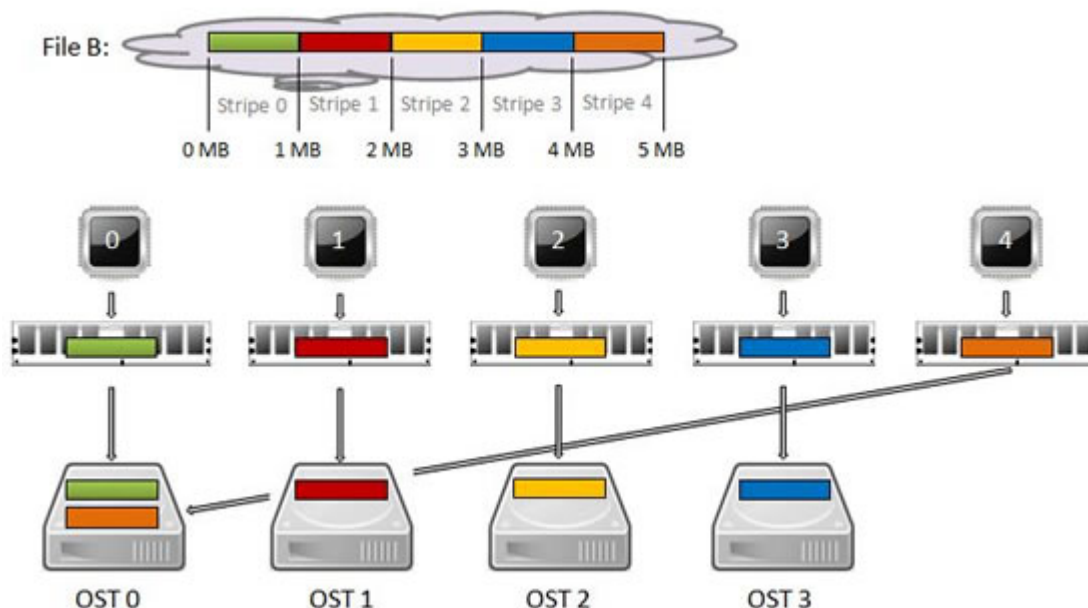
1. *How can I limit the number of processes simultaneously accessing one OST?* The limit is automatically imposed since in the single writer I/O pattern there is only one process performing I/O. Therefore, none of the OSTs are accessed by more than one process.
2. *How can I limit the number of OSTs accessed by each process?* You can limit the number of OSTs accessed by each process by striping the files across only one OST. However, if the file is large (> 1GB) multiple OSTs may be used to

ensure that sufficient resources are allocated to the file. As a general rule, we recommend using a stripe count of 1 for files less than 10 GB and an additional stripe for each additional 100 GB. For example, a 1 TB file would have a stripe count of 10.

3. *How can I limit the number of processes accessing each stripe?* The limit is automatically imposed since in the single writer I/O pattern there is only one process performing I/O. Therefore, the stripes will always be accessed by only one process.

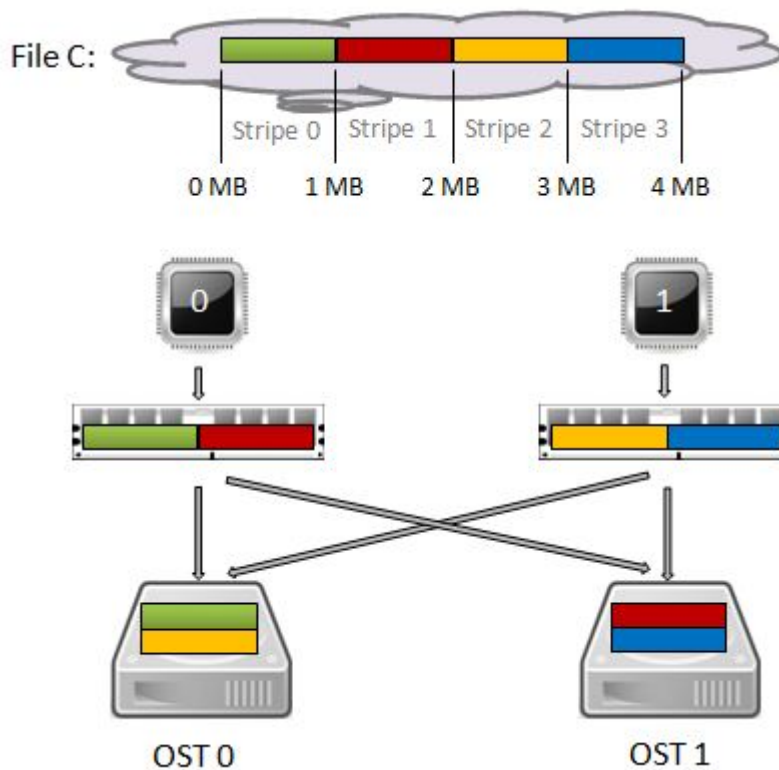
2.3.3 Single Shared File I/O

1. *How can I limit the number of processes simultaneously accessing one OST?* You can limit the number of processes simultaneously accessing a single OST by both setting the stripe count to the number of processes and the stripe size to the contiguous write size for all processes. A stripe count limit of 160 OSTs is imposed by the Lustre file system. By setting the stripe size equal to the amount of contiguous data written by a process, you localize each process's contiguous I/O to a single OST. By setting the stripe count equal to the number of processes, you ensure that each of these stripes resides on a separate OST. Consider an example of a single shared File B spread across four OSTs in five distinct pieces. Five processes write the same amount of data (1 MB each) to this shared file. If File B has 5 MBs of data and the stripe size is set to 1 MB, then the file will be segmented into 5 stripes that will be accessed concurrently. The logical and physical views of File B in this example are shown below.



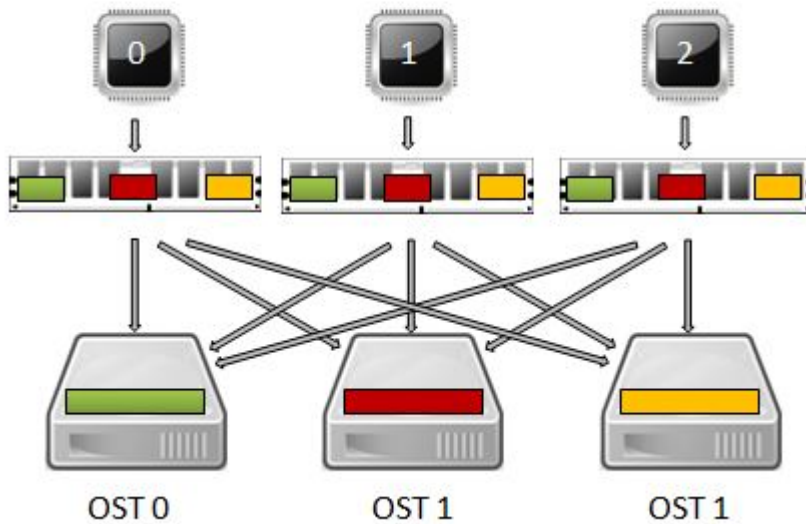
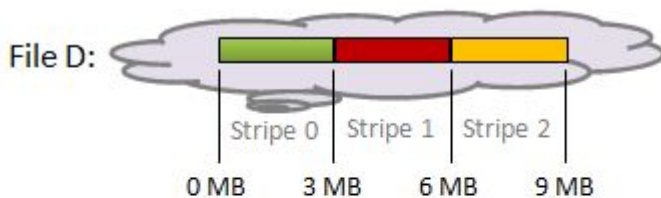
Even though the I/O requests are stripe aligned, OST 0 is accessed by process 0 and process 4 simultaneously. To reduce the contention caused by the two processes simultaneously accessing OST 0 the stripe count should be increased from 4 to 5.

2. *How can I limit the number of OSTs accessed by each process?* You can limit the number of OSTs accessed by each process by setting the stripe size equal to the contiguous write size for that process. If the amount of data written contiguously by each process is not uniform, then using the largest contiguous write size will ensure that each operation will only at most access two OSTs. Consider an example of a single shared File C spread across two OSTs, where the transfer size on each processor is greater than the stripe size. Two processes write the same amount of data (2 MB each) to this shared file. If File C has 4 MB of data and the stripe size is set to 1 MB, then the file will be segmented into 4 stripes, at least two of which will be accessed concurrently. The logical and physical views of File C are shown below. Not only do each of the two processes have to communicate with two OSTs, they also have to use the same OSTs concurrently, creating resource contention on those OSTs. The contention would get even worse if the transfer size was increased. Increasing the stripe size setting to 2 MB will make all I/O requests stripe aligned and immediately fix both

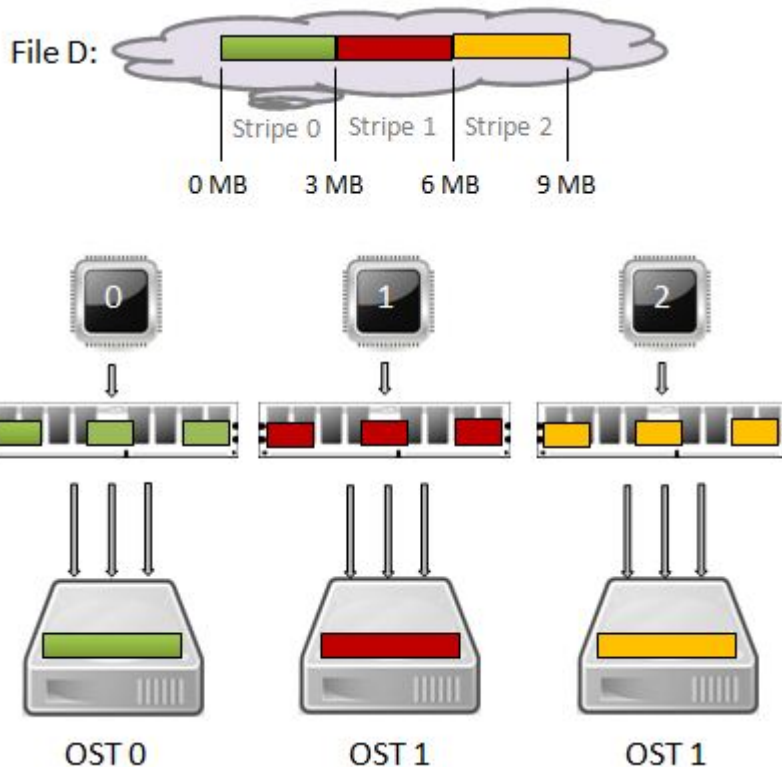


problems.

3. *How can I limit the number of processes accessing each stripe?* You can limit the number of processes that access a single stripe by setting the stripe size to the contiguous write size for all processes. If all processes use a uniform contiguous write size, then both OST access and stripe access are minimized. If the write size is not uniform across processes, choosing the minimum stripe size will minimize stripe accesses and choosing the maximum stripe size will minimize OST accesses. Consider an example of a single shared File D spread across three OSTs where the transfer size on each processor is smaller than the stripe size. Three processes write the same amount of data (3 MB each) to this shared file, and the transfer size is 1 MB. If File D has 9 MB of data and the stripe size is set to 3 MB, then the file will be segmented into 3 stripes that will be accessed concurrently. The logical and physical views of File D are shown below for two different data layouts in memory. In the first example each stripe is simultaneously accessed by three processes creating resource contention on that OST. Also each of the three processes have to communicate with three different OSTs creating some communication overhead as well. If code modification is not allowed, you could decrease the stripe size setting to 1 MB to make all I/O requests stripe aligned. Although the number of writes would still be the same (nine in this case) the contention caused by many processes simultaneously accessing one OST will be avoided.



The situation is better in the second example shown below. The data layout in memory allows each processor to communicate with only one OST. The only improvement you could make is to increase the transfer size on each processor to 3 MB. However, this would require



code modification.

2.3.4 File Per Process I/O

1. *How can I limit the number of processes simultaneously accessing one OST?* You can limit the number of processes that access a single OST by striping the files across only one OST. A stripe count of 1 ensures that each file utilizes a separate OST. However, this setting may need to be increased due to file size. If the number of processes (equal to the number of files) times the stripe count is greater than the number of available OSTs then overlap will occur and multiple files will be written on the same OST. At large process counts (about 10 times the number of OSTs), the contention on the OSTs becomes prohibitive. In this case, it is best to limit the number of files that are written simultaneously.
2. *How can I limit the number of OSTs accessed by each process?* The number of OSTs accessed by each process can be limited by striping the files across only one OST. However, if the file is large (> 1GB) multiple OSTs may be used to

ensure that sufficient resources are allocated to the file. As a general rule, we recommend using a stripe count of 1 for files less than 10 GB and an additional stripe for each additional 100 GB. A 1 TB file would have a stripe count of about 10 in this case.

3. *How can I limit the number of processes accessing each stripe?* The limit is automatically imposed since in the file per process I/O pattern a single process accesses each file. Therefore, the stripes will always be accessed by a single process.

2.4 Managing Free Space

In Lustre, the MDS assigns file stripes to OSTs using a mix of load balancing and free space considerations. This is done to optimize file system performance and OSS space utilizations. Emptier OSTs are preferred and stripes are distributed evenly over OSSs to increase network bandwidth utilization.

Lustre uses two methods for selecting the next OST to which a stripe is to be written: round-robin and weighted. The *round-robin allocator* alternates stripes between OSTs on different OSSs, so the OST used for stripe 0 of each file is evenly distributed among OSTs regardless of the stripe count. The *weighted allocator* uses a weighting algorithm to influence OST ordering based on size and location. Note that these are weightings for a random algorithm, so the OST with the most free space is not necessarily chosen each time. On average, the weighted allocator fills the emptier OSTs faster.

Lustre determines which method to use by the amount of free-space imbalance on the OSTs. The round-robin method maximizes network balancing and is used by default until free space across the OSTs differs by more than 20%. The MDS will then use weighted random allocations with a preference for allocating objects on OSTs with more free space. This can reduce I/O performance until space usage is rebalanced to within 20% again.

Normally, the usage of OSTs is well balanced and the more efficient round-robin allocator is used. However, if a small number of exceptionally large files are created or striping parameters are incorrectly specified, imbalanced OST usage may result. To keep this from happening, *it is important that you use the maximum number of OSTs available while writing large single shared files (160 is Lustre's maximum) and a reasonable number of OSTs while writing large file per process files.* Otherwise you may overwhelm and/or imbalance an OST.

Lustre's `lfs df` command displays the file system disk space usage. For more information about this command see the section about checking disk space usage.

2.5 Self Test

Achieving Efficient I/O



Question 1

For an application using single writer I/O, why does a stripe count greater than 1 generally lead to decreased performance?

[View Answer](#)

Using a stripe count greater than 1 will result in increased overhead from the single process accessing multiple OSTs for write operations.

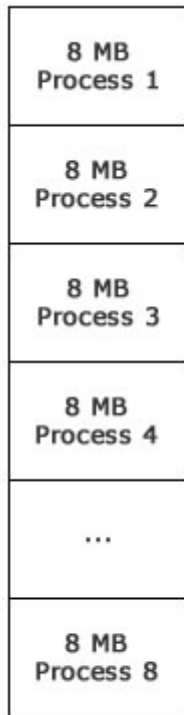


Question 2

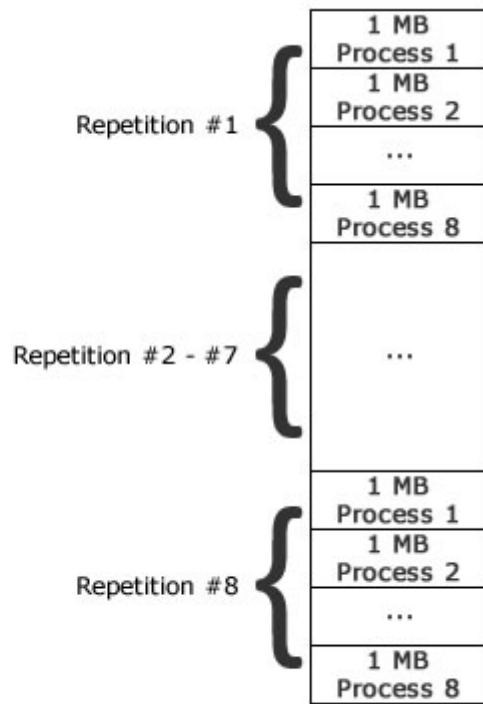
In your application a single process writes data in 16MB chunks to one OST. On a shared resource, which of the following stripe size values will most reliably lead to maximum performance?

- ☐ 8
- ☐ 16
- ☐ 32
- ☐ 64

Two different applications have 8 processes that write to a single shared file of the same size in 8 MB chunks. Application 1 uses file layout #1 which keeps data from a process in a contiguous block. Application 2 uses file layout #2 which strides this data throughout the file. These layouts are shown below.



Shared File Layout #1



Shared File Layout #2

Answer the following questions pertaining to these applications.



Question 3

If the stripe count is 8 and stripe size is 1 MB for each application, which one will get the worst performance?

[View Answer](#)

Application 1 will have the worst performance. Using a 1MB stripe size each process will write 1 MB of data to each of the 8 OSTs. This will cause process contention for OSTs as well as the additional overhead from a single process switching

between OSTs. In Application 2, each process will write data to only one of the OSTs, albeit in non-contiguous blocks.



Question 4

If the stripe count is 8 for both applications and stripe size is 1 MB for Application 2, what stripe size for Application 1 would improve its performance over Application 2? Why?

[View Answer](#)

A stripe size of 8 MB would give Application 1 better performance than Application 2. In both cases, each process writes data to a single, separate OST. Application 1 experiences better performance because it writes its 8 MB of data in a single contiguous block. Application 2 writes its data in eight contiguous 1MB blocks resulting in increased latency from smaller write operations.



Question 5

What would happen if a stripe count of 8 and a stripe size of 8 MB were used for Application 2? Why?

[View Answer](#)

Application 2 would experience worse performance. Each process will write data to all 8 OSTs. However, unlike previous examples, the processes not only share an OST they share a single 8 MB stripe. Additional overhead will be incurred for each process to switch between OSTs and to provide shared access within the single 8 MB stripe.

What stripe settings would be optimal for each of the applications described below?



Question 6

I/O is performed by a single processor that writes three different files. The files are 3 GB in size each and written in 32 MB chunks.

[View Answer](#)

This is an example of a “Single Writer” I/O pattern. Since all three files are accessed by a single process and the size of each file is not very big, these files should not be striped across more than one OST. Therefore the stripe count should be set to 1. Also, since all of the I/O requests are the same with a transfer size of 32 MB it is best to stripe align them by setting the stripe size to 32 MB.



Question 7

I/O is performed by a single processor that writes a single file. The file size is 15 GB and it is written in 100 MB chunks.

[View Answer](#)

This is an example of a Single Writer I/O pattern. Since there is only one process accessing a single file the file should not generally be striped across more than one OST. However, the size of the file is big enough to possibly cause an imbalance on a single OST. Therefore, the stripe count should be set to 2. Also, since all of the I/O requests are the same with a transfer size of 100 MB it is best to stripe align them by setting the stripe size to 100 MB.



Question 8

Every process in an application writes the same amount of data to a shared file. The application utilizes 100 processors and the size of the shared file is 8 GB so each process writes 80 MB to it. All write operations are 2 MB in size.

View Answer

This is an example of a Single Shared File I/O pattern. The number of processes accessing the file is 100, therefore the file should be striped across 100 OSTs (stripe count is 100). All write operations are 2 MB in size, therefore in order to stripe align them the stripe size should be set to 2 MB.



Question 9

Every processor in an application running on 100 processors simultaneously writes its data to a separate file. Fifty of those hundred files are written in 1 MB chunks with a total size of 50 MB per file. The rest of the files are written in 8 MB chunks with a total size of 400 MB per file.

View Answer

This is an example of a File per Process I/O pattern. Although the files differ in size, you should always remember that file striping is a property of a file or a folder. In other words, when Lustre stripe settings are being set, they are being set for individual files and folders and not for the Lustre file system itself. Since each file is accessed by only one process and the files are small (50 MB or 800 MB), they should not be striped across multiple OSTs. The stripe count for all files should be set to 1. The stripe size, however, should be set differently. In order to stripe align all the I/O requests to all files, the first 50 files should have a stripe size set to 1MB, and the rest of the files should have it set to 8 MB.



Question 10

Select the I/O patterns from the list below that will have optimal performance with a stripe count of 1.

- ☐ Multiple processes perform I/O simultaneously to one large shared file.
- ☐ A single node performs all of the I/O to one or more small files (less than 10 GB each).
- ☐ Multiple processes perform I/O simultaneously to different small files.
- ☐ A single process performs all of the I/O to one or more large files (greater than 10 GB).
- ☐ Multiple processes perform I/O to a single small file, but the accesses are only performed by one process at a time.

Show Answer