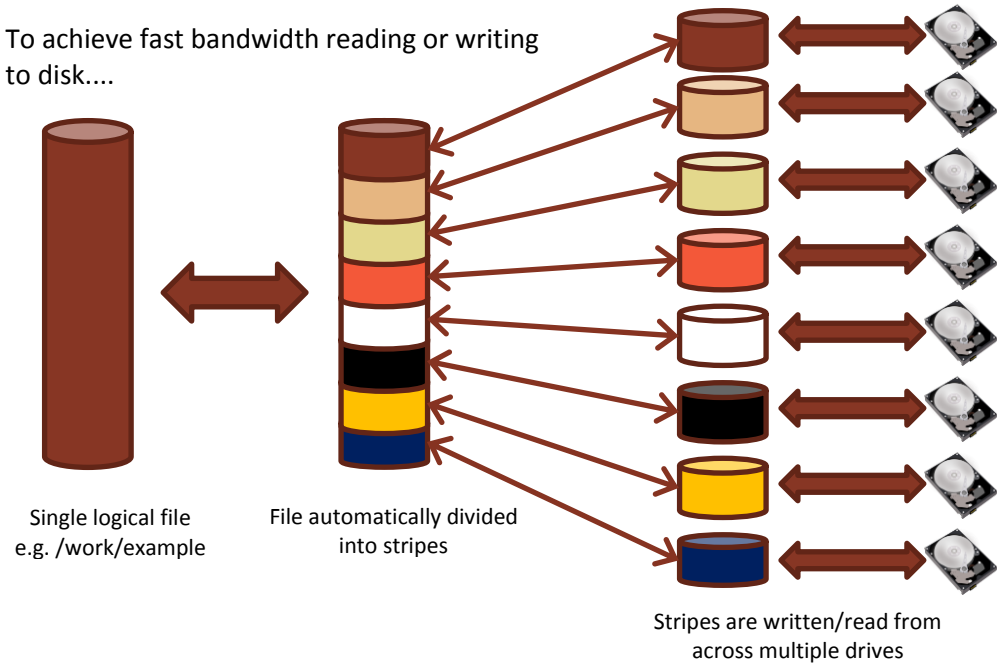
UNDERSTANDING LUSTRE STRIPING

Reading and writing data to disk (Input/Output) can be a time consuming part of any large parallel application. Writing to a single disk has a limited bandwidth, far lower than is practical for a large parallel application, so to improve overall bandwidth files are streamed to multiple disks in parallel. The Lustre parallel file system provides an automatic way of streaming data over multiple sets of disks under a single namespace. Each file is striped in regular chunks over a set of Object Storage Targets (OSTs), the size and number of which is configurable by users.



EVALUATING LUSTRE STRIPE SIZES AND COUNTS

Each Lustre file system consists of a single Metadata Server (MDS) and a fixed number of OSTs. Each file is striped into user-specified chunks over a random subset of the OSTs when data is first written into it. These two options give the user control over the amount of parallelism when reading or writing the file, the stripe count and the stripe size.

Stripe size and stripe count are set via the `lfs` command:

```
lfs –c <stripe_count> -s <stripe_size> <file|dir>
```

Choosing the right file size and stripe count is important to achieving optimal application I/O bandwidth and while experimentation and measurement are the only ways to be sure, a convenient rule of thumb for choosing the number of stripes on Lustre is:

# of Files	# of OSTs	Command
1 per PE	1 OST per file	`lfs –c 1 <file|dir>`
1 per run	All OSTS	`lfs –c -1 <file|dir>`
1 < #files < #PEs	#OSTS / #files	`lfs –c N -<file|dir>`

(however, for large configurations using all OSTs probably does not make much sense)

PRACTICAL

EVALUATING THE EFFECT OF STRIPE SIZE AND COUNT WITH VH1

The provided benchmark, VH1, performs *"file-per-process"* I/O, like many applications, writing one NetCDF file per PE to disk. We can assess the impact of changing stripe size and count on application performance using the CrayPAT-lite tools to report I/O performance for an individual run.

Once you have unpacked the source code provided by the tutor you can build the executable by running:

```
cd src
module load cray-netcdf
module load perftools-lite
make
```

We need to make sure that both the CrayPAT-lite (`perftools-lite`) and NetCDF modules are loaded for the application to correctly compile. This will now create a binary in the `bin` directory depending upon the currently loaded Programming Environment.

```
bin/vh1-mpi-<prgenv>
```

Now enter the run directory and inspect the `run_XXX.pbs` script (choose the appropriate one for your environment).

```
cd ../run/
vim run_XXX.pbs
```

You may need to edit the value of the `EXE` environment variable to point to the executable that you have just built (if you are not using the system defaults). You will also find two other variables, `STRIPE_SIZE` and `STRIPE_COUNT` which control the stripe size and stripe count of the output NetCDF files (these set the default values for files in the output directory when they are created). Run the benchmark by running:

```
qsub run_XXX.pbs
```

Once the job has run check the contents of the output file vh1.o<jobid>. This will contain job output, but will also contain profiling information from the CrayPAT-lite suite (if it does not, check that you had the perftools-lite module loaded before all source was compiled). You will find two tables relating to I/O: Table 2 contains the ten slowest file reads and Table 3 for the ten slowest file writes. e.g.

```
Table 3:  File Output Stats by Filename

    Write |       Write | Write Rate | Writes |     Bytes/ |File Name[max10]
     Time |      MBytes | MBytes/sec |        |       Call | PE=HIDE

 3.315631 | 1731.612668 | 522.257315 |  482.0 | 3767069.47 |Total
|-------------------------------------------------------------------------------
| 0.262812 |   71.504402 | 272.074611 |   18.0 | 4165433.33 |output/NCState_1002.0001.nc
| 0.227137 |   71.504402 | 314.807912 |   18.0 | 4165433.33 |output/NCState_1002.0003.nc
| 0.165433 |   71.504402 | 432.225494 |   18.0 | 4165433.33 |output/NCState_1002.0002.nc
| 0.153452 |   71.504402 | 465.970923 |   18.0 | 4165433.33 |output/NCState_1003.0003.nc
| 0.139180 |   71.504402 | 513.754982 |   18.0 | 4165433.33 |output/NCState_1000.0003.nc
| 0.137193 |   71.504402 | 521.194379 |   18.0 | 4165433.33 |output/NCState_1004.0001.nc
| 0.137123 |   71.504402 | 521.462672 |   18.0 | 4165433.33 |output/NCState_1004.0000.nc
| 0.136001 |   71.504402 | 525.762035 |   18.0 | 4165433.33 |output/NCState_1000.0000.nc
| 0.134345 |   71.504402 | 532.245964 |   18.0 | 4165433.33 |output/NCState_1003.0001.nc
| 0.132812 |   71.504402 | 538.386987 |   18.0 | 4165433.33 |output/NCState_1001.0000.nc
|===============================================================================
```

Example output from a VH1 run with CrayPAT-lite profiling.

EXERCISES

Investigate which combination of Lustre parameters gives the best I/O performance for the slowest performing file in Table 3 (File Output Stats by Filename), you may find this table helpful for keeping track of your results:

Size/Count	1	2	4	6	8
1m					
2m					
4m					
8m					
16m					
32m					

Look at Table 2 (File Input Stats by Filename): Why might the performance of the reads be so much higher than that of the writes (remember, no binary data is supplied to this benchmark in the setup script, where is the data the benchmark is reading coming from)?

ADVANCED EXERCISE

For programmers into NETCDF only.

Try adapting VH1 to use parallel NetCDF4 to write data files. Compare the performance with file-per-process. What are the advantages and disadvantages for the developer and the user of these approaches?