Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bericht

I/O Aware Slurm with PFL

vorgelegt von

Felix Pusch

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: M.Sc. Informatik
Matrikelnummer: 7539944

Hamburg, 2023-04-19

Abstract

Recent versions of the Lustre parallel distributed file system have introduced the progressive file layout (PFL) feature. This enables a single file to have an evolving striping configuration allowing files to be stored with a growing number of stripes as the file grows. Determining and setting a decent PFL configuration is non-trivial. Scientific computing jobs are generally submitted to clusters running a scheduling software such as SLURM which already exposes a number of tweakable parameters to end users to control job execution.
This project explores the possibility of automatically setting a usable PFL configuration on cluster job submission depending on job information like the number of requested nodes and the type of application. Apart from this the plugin considers environmental cluster information such as the bandwidth between compute nodes and Lustre storage backend. The project shows that setting a PFL configuration through a SLURM job submission plugin is possible. Additionally it discusses considerations to be made when determining the PFL configuration and limitations encountered when setting this via a plugin.

Contents

1 Introduction **4**
1.1 Slurm . . . 4
1.2 Lustre . . . 5
1.2.1 Striping . . . 6
1.2.2 Progressive File Layout . . . 7
1.2.3 Real Life Systems . . . 9

2 Implementation **10**
2.1 Local Development Setup . . . 10
2.1.1 Usage . . . 10
2.1.2 Noteworthy Modifications . . . 11
2.2 Setup on Test Cluster . . . 12
2.2.1 Installing SLURM . . . 12
2.2.2 Configuring Munge . . . 13
2.2.3 Configuring SLURM . . . 13
2.3 Job Submit Plugin . . . 15
2.3.1 Requirements . . . 16
2.3.2 Setting the Progressive File Layout . . . 16
2.3.3 Implementation Details . . . 17

3 Testing **21**
3.1 Performance Baseline . . . 22
3.2 Layout Selection . . . 23

4 Conclusion **25**
4.1 Discussion & Limitations . . . 25
4.2 Future Work . . . 26

Bibliography **27**

Appendices **29**

List of Figures **30**

List of Listings **31**

List of Tables **32**

1 Introduction

1.1 Slurm

The **S**imple **L**inux **Utility** **R**esource **M**anagement (SLURM) is an open-source highly scalable resource management system for large high performance computer clusters [YJG03]. SLURM was originally developed at Lawrence Livermore National Laboratory but has since been adopted in a wide range of commercial and research HPC clusters. As an example the system is used for both the Perlmutter and Cori supercomputers in the National Energy Research Scientific Computing Center at LLNL [Cen23], as well as the Levante supercomputer located at the Deutsches Klimarechenzentrum (DKRZ). These supercomputers are ranked 8, 45 and 53 respectively in the TOP500 list of November 2022 [Pro22].

A SLURM cluster is composed of a single primary node running the SLURM controller daemon `slurmctld` and a set of compute nodes running the SLURM daemon `slurmd` as outlined in Figure 1.1. End users can query the cluster and submit jobs via a set of client command line tools, most prominently `srun` for individual jobs and `sbatch` for batch scripts. Batch jobs and individual job steps are then allocated among the compute nodes by the controller according to the job allocation rules configured in the cluster.

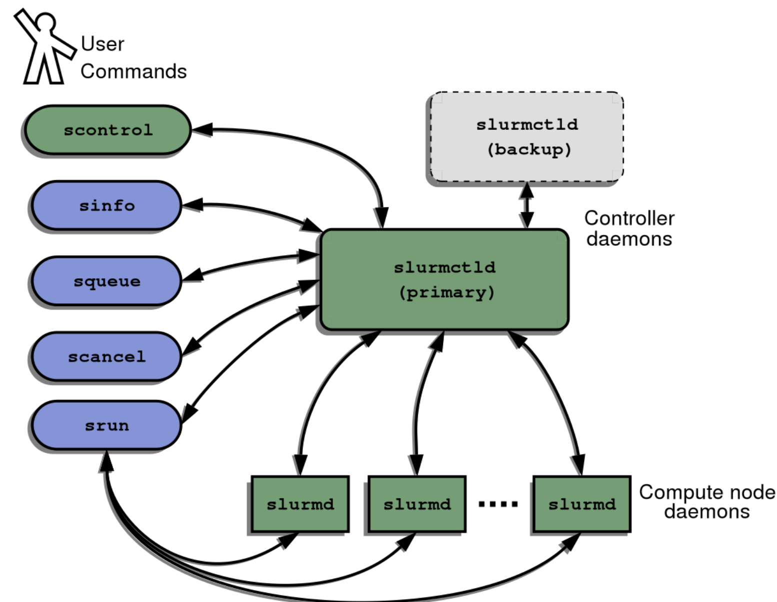

Figure 1.1: SLURM Architecture [YJG03]

1.2 Lustre

Lustre is a common file system choice in High Performance Computing (HPC) deployments as it answers to the high scalability requirements while being open source and configurable to suit a wide number of deployment options. Lustre is a distributed file system meaning data is stored in a distributed fashion across a number of storage targets and offers a parallel storage API allowing parallel file I/O operations through APIs like MPI-IO. A Lustre deployment is general composed of three high level system components as illustrated in Figure 1.2 [Bra19].

- Lustre Clients running the Lustre filesystem
- Object Storage Targets (OSTs) storing the actual end user data
- Metadata Servers (MDS) storing object metadata like permissions, the directory, filenames and the individual object location information

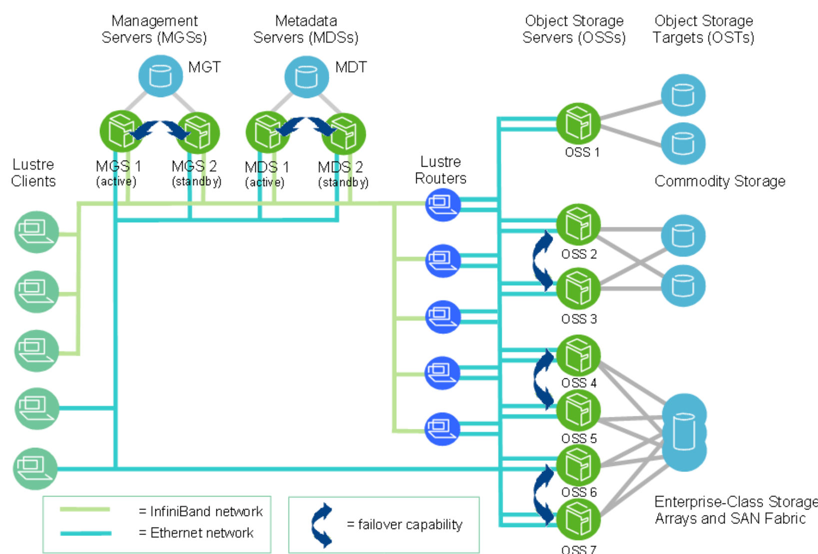

Figure 1.2: A typical distributed Lustre deployment [OE23]

A typical write operation on a Lustre file system is therefore composed of two stages as illustrated in Figure 1.3. First the metadata server is contacted to allocate the layout of the requested objects on the OSTs. Secondly the client then writes the stored objects to the respective OSTs by interacting with the relevant Object Storage Servers. It is easy to see that due to the distributed nature of the storage on the OSTs a client can theoretically benefit from the aggregated bandwidth of all OSTs involved in the file operation assuming sufficient bandwidth is available between the client and the OSSs.

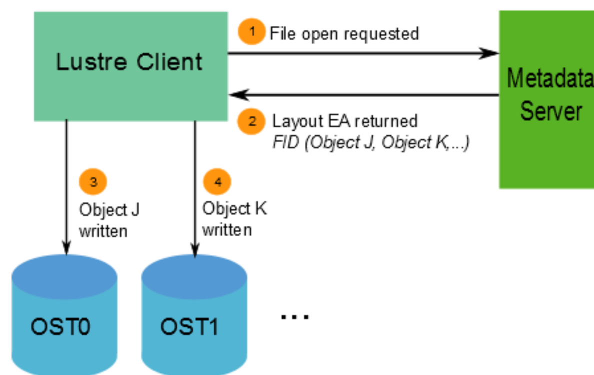

Figure 1.3: Lustre File Write Operation [OE23]

1.2.1 Striping

The Lustre filesystem supports striping of the data of a single file across multiple Object Storage Targets. This feature provides the following benefits

- ability to store files that are larger than the total (or remaining) physical capacity of a single OST
- enable higher bandwidth access to a file than a single OST (or OSS) is able to provide

Striping is configured via two main parameters: the stripe size and stripe count [OE23]. The stripe size determines the number of bytes stored before moving on to the next stripe (on the next OST) with the stripe count determining the total number of OSTs the file should be striped across. Figure 1.4 illustrates this with File A being striped across three OSTs while File B and C are only striped across a single OST.

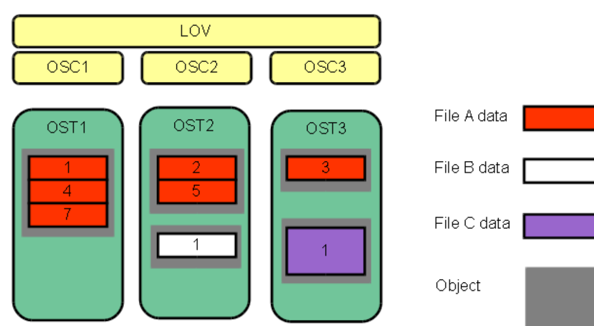

Figure 1.4: Lustre File Striping [OE23]

Choosing the right striping configuration is not trivial, as it depends largely on the I/O pattern of the given application. A poor striping configuration can have adverse effects on performance not just for the given application, but for the entire storage cluster, as there may be increased disk contention by multiple processes across the (shared) cluster

resources. It is therefore important to evaluate the performance at cluster level analyzing the aggregate storage performance for all users.

For example, striping very small (≤1GB) files across many OSTs will increase contention of those disks and incur a high overhead leading to poorer performance than if those small files would be striped across just one or two OSTs. Therefore it is generally recommend to stripe small files over a single OST, especially if the number of clients is larger than the number of OSTs [Bra19].

In contrast large files typically benefit the most from striping allowing an application to use the aggregated bandwidth of many OSTs across multiple client processes each reading different parts of the file. Similarly, if there are only a few active client processes, it is useful to stripe across as many OSTs as practical to ensure the available bandwidth between the clients and OSTs is properly used [Bra19]. Striping appropriately also ensures that the different OSSs and OSTs are used equally and that the storage utilization is balanced across the cluster avoiding situations where individual, very large, files create fragmentation in the storage pool.

In Lustre the striping layout of a file is configured either via system or directory wide defaults or manually at file creation time. Once a file has been created the striping configuration can only be changed by completely rewriting the entire file (effectively creating a new one).

1.2.2 Progressive File Layout

Lustre 2.10 introduced the ability to specify striping in a more dynamic fashion with a feature called „Progressive File Layout (PFL)“. This enables the specification of a sort of template consisting of multiple components (called extents) describing how the striping behavior should adapt as the file grows. Concretely this enables defaults set at cluster or directory level to be better matched to an adequate striping configuration for a wider variety of application use cases and file sizes. Consequently the striping defaults no longer have to „choose“ between being practical for small or large files.

In the example illustrated in Figure 1.5 the PFL specifies

- the first 2MB to be striped across one OST with a stripe size of 1MB (leading to two stripes being created on the same OST)
- the following 253MB to be striped across four OSTs with a stripe size of 1MB (leading to 253 stripes being created spread across four OSTs)
- the remainder of the file to be striped across 32 OSTs with a stripe size of 4MB

On a running system the PFL configuration of a file or directory can be retrieved with the `lfs getstripe` command as seen in Listing 1.1. In this case the first gigabyte is striped across one OST, the following three gigabytes are striped across four OSTs and the remainder of the file is striped across sixteen OSTs.

```
1 $ lfs getstripe /scratch/k/k203168
2 /scratch/k/k203168
```

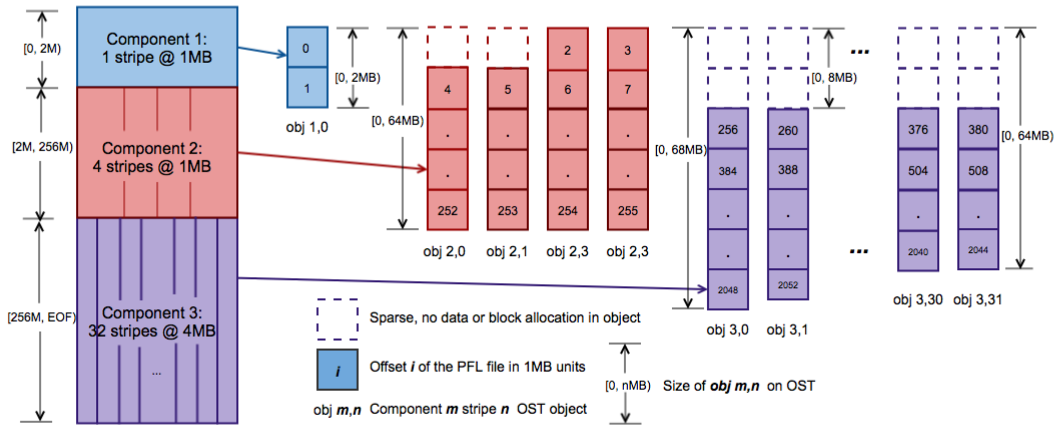

Figure 1.5: Lustre Progress File Layout [OE23]

```
 3  lcm_layout_gen:     0
 4  lcm_mirror_count:   1
 5  lcm_entry_count:    3
 6  lcme_id:              N/A
 7  lcme_mirror_id:       N/A
 8  lcme_flags:           0
 9  lcme_extent.e_start: 0
10  lcme_extent.e_end:    1073741824
11  stripe_count:   1         stripe_size:    1048576
      ↪ pattern:          raid0         stripe_offset:  -1
12
13  lcme_id:              N/A
14  lcme_mirror_id:       N/A
15  lcme_flags:           0
16  lcme_extent.e_start: 1073741824
17  lcme_extent.e_end:    4294967296
18  stripe_count:   4         stripe_size:    1048576
      ↪ pattern:          raid0         stripe_offset:  -1
19
20  lcme_id:              N/A
21  lcme_mirror_id:       N/A
22  lcme_flags:           0
23  lcme_extent.e_start: 4294967296
24  lcme_extent.e_end:    EOF
25  stripe_count:   16         stripe_size:    1048576
      ↪ pattern:          raid0         stripe_offset:  -1
```

Listing 1.1: Retrieving the Lustre PFL definition

1.2.3 Real Life Systems

Prior to the implementation of Progressive File Layouts in Lustre, analysis of large scale deployments performed at the Oak Ridge and Lawrence Livermore National Laboratories show that the majority of users and applications on HPC systems tend not to choose a reasonable striping layout for their use case [WSHO17]. The authors showcase that depending on the type of data stored on such a system a reasonable striping configuration may look different. As an example the Lustre deployment at Lawrence Livermore used a stripe count of one (1) as on this system around 88% of files are ≤1MB in size meaning these files would span only one stripe. On this system files with a size of less than 1GB make up roughly 57% of data stored. For comparison the Lustre deployment at Oak Ridge used a stripe count of four (4) with only around 15% of data stored being taken up by files less than 1GB in size [WSHO17].

More recent Lustre deployments, like the all-flash storage supercomputer Perlmutter at the National Energy Research Scientific Computing Center, make use of the PFL feature in Lustre. The Perlmutter supercomputer is configured with a default setting of 1 stripe count for data ≤1GB, a stripe count of 8 for data ≤10GB, a stripe count of 24 for data ≤100GB and a stripe count of 72 for data larger than 100GB [GEK+22]. The authors note that this default layout, in their experiments, shows „at least near best write performance among all the Lustre configurations [...] explored“[GEK+22]. In practice their experiments show that the PFL feature is applicable to both, disk based and flash based, Lustre deployments.

The Levante (HLRE-4) supercomputer recently deployed at the DKRZ is equipped with a 130 Petabyte storage cluster running Lustre [Kli23]. Here the users' scratch and project's work volumes are by default configured with a stripe size of 1MB and a striping setup of

- the first 1GB to be striped across one OST
- the the following 3GB to be striped across 4 OSTs
- the remainder of the file to be striped across 16 OSTs

2 Implementation

The goal of this project is to create a mechanism that automatically sets an appropriate Lustre PFL configuration in a more granular fashion than is possible with cluster level defaults. This configuration should take into account the storage model of the application, the available bandwidth within the cluster and the resources requested by the job. In order to have access to the resource requests and job description the implementation was chosen to be integrated in the SLURM cluster workload manager via the available plugin API. This enables the plugin to be automatically called upon job submission with the respective job information and avoids having to fork and create a custom version of SLURM. While forking SLURM would have allowed for more intricate control of the scheduling decisions for this project the information available via the lua plugin API is sufficient and results in a more portable and maintainable solution.

2.1 Local Development Setup

In order to perform the development of the lua job submit plugin a local SLURM setup is used. This enables easy debugging of the lua plugin code and the lua specific SLURM configuration. The local setup is realized using a set of *Docker* containers run via `docker-compose` mostly based upon [Tor22]. The code for the customized version of this is available at `https://github.com/fpusch/slurm/tree/master/docker`.

The cluster is composed of five individual containers each running a single component.

- a SLURM Controller running `slurmctld`
- two SLURM Compute Nodes *c1,c2* running `slurmd`
- a Database instance running `mysql`
- a SLURM Database daemon running `slurmdbd`

2.1.1 Usage

All SLURM Docker containers are started from the same Docker image which can be built by executing `docker build -t slurm-docker-cluster:<tag> .` with `tag` specifying the target SLURM version Docker tag to use. The docker image itself is built from the `rockylinux:8` baseline to be easily portable to Red Hat Enterprise Linux based installations. The detailed build script is defined in the `Dockerfile`. On a high level the build performs the following steps

Volume Name	Mount Path	Usage
etc_munge	/etc/munge	Munge Config
./slurm (bind)	/etc/slurm	SLURM Config
slurm_jobdir	/data	Shared Data Directory
var_lib_mysql	/var/lib/mysql	Database Data Directory
var_log_slurm	/var/log/slurm	SLURM Logs

Table 2.1: Docker Volumes

1. Install prerequisite packages
2. Clone SLURM repository
3. Configure and build SLURM
4. Copy entrypoint

The Docker cluster is defined using Docker compose in docker-compose.yml. The cluster is started by running docker-compose up and stopped by running docker-compose down. The cluster uses the Docker volumes described in Table 2.1 to persist state across restarts. The volume definition of the etc_slurm volume is setup as a bind mount to enable modifications to the job submission plugin code to be available immediately without having to restart the cluster or rebuild the image. The cluster can be completely reset by running docker-compose down -v which will also remove any data stored in the volumes.

After the cluster is started jobs can be submitted by logging into any SLURM container.

```
1 $ docker exec -it c1 bash
2 [root@c1 /]# cd /data/
3 [root@c1 data]# sbatch --wrap="hostname"
4 sbatch: slurm_job_submit: hello world
5 Submitted batch job 1
6 [root@c1 data]# cat slurm-1.out
7 c1
```

Listing 2.1: Submitting a Job on the local cluster

2.1.2 Noteworthy Modifications

Support for SLURM job submission plugins written in lua is enabled at compilation time by having the lua development libraries lua-devel present during ./configure. This check is performed by an autoconf macro in auxdir/x_ac_lua.m4 and checks for the presence of packages named lua-<major>.<minor> or lua<major>.<minor>. In Red Hat Enterprise Linux based distributions, like RockyLinux which is used for the local Docker setup, the lua development libraries do not follow this convention. To circumvent this issue the package configuration files lua.pc are symlinked to the expected names.

In addition to compiling SLURM with lua support the main configuration files `/etc/slurm/slurm.conf` needs to be modified to enable job submission plugins to use lua.

```
# PLUGINS - enable lua plugin support
JobSubmitPlugins=lua
```

Listing 2.2: Enabling lua support in slurm.conf

Afterwards, upon start, the SLURM Controller will look for a file named `/etc/slurm/job_submit.lua` which should contain the lua plugin code.

2.2 Setup on Test Cluster

As the local setup using Docker containers to run a SLURM cluster does not support the Lustre parallel and distributed filesystem a test cluster of five compute (west[1-5]) and five storage (sandy[1-5]) nodes was provisioned. The compute nodes were equipped with 24 core CPUs and a Gigabit/s Ethernet networking link to the storage nodes.

2.2.1 Installing SLURM

The installation and configuration of SLURM is done via a number of scripts available at `https://github.com/fpusch/slurm/tree/master/wr-cluster`

The SLURM controller is setup by installing the package dependencies, cloning the source code repository and then building SLURM similarily to what is done in the Docker based setup.

```
#!/usr/bin/env bash
set -e
apt install munge libmunge-dev lua5.3 liblua5.3-dev -y
git clone -b slurm-21-08-6-1 --single-branch --depth=1
  ↪ https://github.com/SchedMD/slurm.git
cd slurm
./configure --enable-debug --sysconfdir=/etc/slurm
make install
groupadd -r slurm
useradd -r -g slurm slurm
mkdir /var/spool/slurmd \
/var/spool/slurmctld \
/var/run/slurmd \
/var/lib/slurmd \
/var/log/slurm
chown -R slurm:slurm /var/spool/slurmd
chown -R slurm:slurm /var/spool/slurmctld
chown -R slurm:slurm /var/run/slurmd
chown -R slurm:slurm /var/lib/slurmd
```

```
19 chown -R slurm:slurm /var/log/slurm
20 chown -R slurm:slurm /usr/local/lib/slurm
21 chown -R slurm:slurm /etc/slurm/
```

Listing 2.3: Installing SLURM Controller Node on Test Cluster

The compute nodes are setup in the same way.

```
1  #!/usr/bin/env bash
2  set -e
3  apt install munge libmunge-dev lua5.3 liblua5.3-dev -y
4  git clone -b slurm-21-08-6-1 --single-branch --depth=1
     ↪ https://github.com/SchedMD/slurm.git
5  cd slurm
6  ./configure --enable-debug --sysconfdir=/etc/slurm
7  make install
8  groupadd -r slurm
9  useradd -r -g slurm slurm
10 mkdir /var/spool/slurmd \
11 /var/run/slurmd \
12 /var/lib/slurmd \
13 /var/log/slurm
14 chown -R slurm:slurm /var/spool/slurmd
15 chown -R slurm:slurm /var/run/slurmd
16 chown -R slurm:slurm /var/lib/slurmd
17 chown -R slurm:slurm /var/log/slurm
18 chown -R slurm:slurm /usr/local/lib/slurm
19 chown -R slurm:slurm /etc/slurm/
```

Listing 2.4: Installing SLURM Compute Node on Test Cluster

2.2.2 Configuring Munge

SLURM uses `munge` for authentication between the nodes of a cluster. In the subsection 2.2.1 `munge` was installed from the package repositories. To properly configure `munge` the `munge.key` file must be identical on all hosts participating within the cluster [Dun23]. Therefore the file is generated via `mungekey` on the slurm controller node and then transferred to all compute nodes via `scp`.

2.2.3 Configuring SLURM

SLURM is configured via the `/etc/slurm/slurm.conf` file present on all machines that are part of the cluster. The configuration file used was generated via the configurator tool available at `https://slurm.schedmd.com/configurator.html`

```
1  # slurm.conf file generated by configurator.html.
2  # Put this file on all nodes of your cluster.
3  # See the slurm.conf man page for more information.
4  #
5  ClusterName=cluster
6  SlurmctldHost=west1
7  JobSubmitPlugins=lua
8  MpiDefault=none
9  ProctrackType=proctrack/linuxproc
10 ReturnToService=1
11 SlurmctldPidFile=/var/run/slurmctld.pid
12 SlurmctldPort=6817
13 SlurmdPidFile=/var/run/slurmd.pid
14 SlurmdPort=6818
15 SlurmdSpoolDir=/var/spool/slurmd
16 SlurmUser=slurm
17 StateSaveLocation=/var/spool/slurmctld
18 SwitchType=switch/none
19 TaskPlugin=task/affinity
20 # TIMERS
21 InactiveLimit=0
22 KillWait=30
23 MinJobAge=300
24 SlurmctldTimeout=120
25 SlurmdTimeout=300
26 Waittime=0
27 # SCHEDULING
28 SchedulerType=sched/backfill
29 SelectType=select/cons_tres
30 # LOGGING AND ACCOUNTING
31 AccountingStorageType=accounting_storage/none
32 JobCompType=jobcomp/none
33 JobAcctGatherFrequency=30
34 JobAcctGatherType=jobacct_gather/none
35 SlurmctldDebug=info
36 SlurmctldLogFile=/var/log/slurm/slurmctld.log
37 SlurmdDebug=info
38 SlurmdLogFile=/var/log/slurm/slurmd.log
39 # COMPUTE NODES
40 NodeName=west[2-5] CPUs=24 Sockets=2 CoresPerSocket=6
       ↪ ThreadsPerCore=2 State=UNKNOWN
41 PartitionName=debug Nodes=ALL Default=YES MaxTime=INFINITE
       ↪ State=UP
```

Listing 2.5: SLURM Configuration

After the configuration is setup on all machines SLURM can be started.

```
1 systemctl start munge # ensure munge is running
2 slurmctld -D
```

Listing 2.6: Starting `slurmctld`

```
1 systemctl start munge # ensure munge is running
2 slurmd -D
```

Listing 2.7: Starting `slurmd`

2.3 Job Submit Plugin

Any job submission plugin defines two functions: `slurm_job_submit` and `slurm_job_modify`. Both functions must be defined for the plugin file to be accepted. Job submission plugins can be written using either the C or lua programming languages with roughly the same method signatures [Sch23b]. A simple *Hello World* implementation of this using lua can be found in Listing 2.8. For this project the implementation was done using lua as this required very few changes to the SLURM build and installation process while providing the functionality needed within this scope.

```
1  function slurm_job_submit(job_desc, part_list, submit_uid)
2      slurm.log_user("slurm_job_submit: hello world")
3      return slurm.SUCCESS
4  end
5
6  function slurm_job_modify(job_desc, job_rec, part_list,
     ↪ modify_uid)
7      slurm.log_user("slurm_job_modify: hello world")
8      return slurm.SUCCESS
9  end
10
11 slurm.log_info("job submit plugin initialized")
12 return slurm.SUCCESS
```

Listing 2.8: Hello World Job Submit Plugin

The data structure for the `job_desc` parameter matches the `job_descriptor` struct defined in `slurm.h` [Sch23a].

2.3.1 Requirements

The job submission script should

- detect the target directory to which the job needs to write data (i.e. using a flag, environment variable or parameter)
- be able to be deactivated by the user (i.e. using a flag, environment variable or parameter)
- set the PFL layout according to
  - the detected application / job submission script and its I/O storage mode
  - number of nodes / tasks requested by the job
  - available maximum bandwidth between storage and compute nodes
  - enforce a maximum number of stripes (to avoid striping across all available nodes)

2.3.2 Setting the Progressive File Layout

The PFL configuration can be specified either at file or directory level with the `lfs setstripe` command. Files automatically use the configuration specified in the parent directory unless a file specific layout is provided. The `lfs setstripe` command accepts the parameters (among others) outlined in Table 2.2.

Option Name	Behavior
-E <offset>	end offset of the following component (allows the use of `-1` to signify until end of file)
-c <stripe_count>	number of OSTs to stripe the current component over (allows the use of `-1` to signify all OSTs)
-S <stripe_size>	number of bytes within a single stripe

Table 2.2: lfs setstripe command

As an example the command in Listing 2.9 specifies the Levante default layout described in Listing 1.1.

```
1 $ lfs setstripe \
2     -E 1G -c 1 -S 1M \
3     -E 4G -c 4 -S 1M \
4     -E -1 -c 16 -S 1M \
5     /scratch/k/k203168/demo
```

Listing 2.9: Configuring a PFL setup with lfs setstripe

In the lua code of the Job Submit Plugin the `os.execute` function is used to execute the PFL command.

2.3.3 Implementation Details

The code for the job submission plugin is available at `https://github.com/fpusch/slurm/blob/master/wr-cluster/job_submit.lua`. The plugin code performs the following steps

1. establish global configuration, read SLURM job description and read cluster state
2. detect application storage mode
3. determine and set the progressive file layout

The global configuration is performed at the top of the plugin file via a set of constants outlined in Table 2.3.

Name	Behavior
DISABLE_FLAG	Controls the name of the flag used to disable the plugin from a job submission script. Placing the specified value anywhere inside the job submission script skips any further actions by the plugin for this job.
TOTAL_LUSTRE_NODES	Total number of Lustre object storage servers that are part of the cluster.
TOTAL_SLURM_NODES	Total number of SLURM nodes that are part of the cluster.
MAX_OSS_OST_BANDWIDTH_MBITS	Maximum bandwidth (in Megabit/s) between a single object storage server and its target, basically representing the sustained speed of the target. This is limited by the storage interconnect and disk technology (i.e. HDD / SSD / NVME).
MAX_COMPUTE_OSS_BANDWIDTH_MBITS	Maximum bandwidth (in Megabit/s) between a single compute server and a single object storage server. This is limited by the compute storage network interconnect (i.e. Ethernet or Infiband).
EXECUTABLE_STORAGE_MODE	Lua table containing the mapping of application / job script name to storage mode (see Table 2.4).

Table 2.3: Plugin Global Constants

The plugin distinguishes three types of application storage modes as outlined in Table 2.4. These modes are similar to the storage modes configurable via the `-F` flag of the IOR benchmark further detailed in chapter 3 in that they distinguish jobs writing to a single shared file from multiple processes to those where each process writes to its own file. The plugin determines the appropriate storage mode purely based on the name of the job submission script (in case of submission via `sbatch`) or the name of the executable (in case of submission via `srun`).

The available bandwidth between compute nodes, object storage servers and object storage targets is introduced as a configurable constant to capture multiple scenarios

Name	Code	Behavior
Single Process Single File	11	Application is not parallelized and writes to a single file
Multi Process Single File	21	Application is parallelized and all clients write to a single file
Multi Process Multi File	22	Application is parallelized and all clients write to their own file in a file per process mode

Table 2.4: Plugin Storage Modes

depending on the network and storage architecture of the cluster. A simple example can be visualized in Figure 2.1 with Ethernet being used for compute to object storage networking and regular hard drives used for object storage targets.

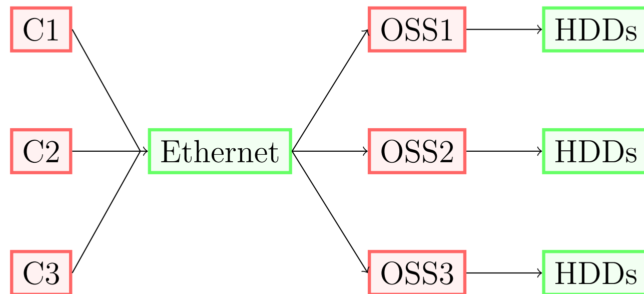

Figure 2.1: Simple Cluster Architecture

In order to enable decent utilization of the available resources it is important to distinguish jobs based on the application's storage mode listed in Table 2.4.

For *single process single file* type applications a striping configuration allowing maximum throughput to the storage cluster can be achieved by determining the number of object storage targets a single compute node can saturate as is done in Equation 2.1. This number is then capped by the total number of Lustre storage nodes (N_L) available in the cluster.

$$max_{s11} = min(max(1, floor(\frac{BW_{C_OSS}}{BW_{OSS_OST}})), N_L) \tag{2.1}$$

Taking a simple practical example using 1 Gigabit/s Ethernet compute to object storage server links with hard drives allowing for 250 Megabyte/s write performance as object storage targets shows that in this case the bottleneck is the Ethernet networking. A single compute node's available bandwidth is already saturated using a single object storage target. In this case striping a file across multiple servers is not going to yield any performance gain. Utilizing 10 Gigabit/s Ethernet instead would allow a single compute node to benefit from the aggregate performance of multiple object storage servers yielding a maximum stripe count of 5.

A similar behavior is applicable for *multi process multi file* type applications as in this case it is assumed that every process writes to its own file exclusively. Here every process can be treated as an instance of a single process single file mode job. This estimation is a „greedy“ upper limit in that it assumes all processes to run on their own compute node, which is unlikely for nodes with multiple CPU cores. Should jobs be co-located on the same node it assumes that these jobs do not perform I/O concurrently.

$$max_{s22} = min(max(1, floor(\frac{BW_{C_OSS}}{BW_{OSS_OST}})), N_L) \quad (2.2)$$

A more pessimistic approach could be taken by simply dividing the number of targets by the number of processes co-located on any single node. This calculation could be tricky to perform at job submission time, especially in clusters with non-homogenous compute nodes with varying CPU core counts, and therefore the previous, optimistic, calculation was chosen for this implementation.

For *multi process single file* applications it is assumed that all client processes write to the same file. This scenario is the most interesting as it is the most likely to benefit from striping, even in bandwidth limited scenarios. Here the maximum number of stripes can be determined by taking into account the total number of compute nodes requested by the job (N_J) as shown in Equation 2.3.

$$max_{s21} = min(N_J * max_{s11}, N_L) \quad (2.3)$$

Taking the same example of 1 Gigabit Ethernet and 125 Megabyte/s hard drives as above now yields a varying number of maximum stripes depending on the nodes requested by the job. An application parallelized across three nodes could benefit from the aggregate bandwidth of three object storage targets when writing to the same file striped across those targets.

Notably this scheme of assigning the maximum stripe count does not overreach in that it stripes across as many targets as are necessary to maximize bandwidth but no more. This can be desirable in cluster setups where the number of compute nodes is significantly larger than the number of object storage targets in that it does not introduce additional contention over those resources. In contrast using a scheme that always stripes across the maximum number of available OSTs would cause every *multi process single file* job to compete over all OSTs even if the job could not take advantage of the additional bandwidth available.

Expanding on this the job submission plugin assigns striping layouts in the following manner

- for *single process single file* and *multi process multi file* jobs
  - first 16MB striped across 1 OST
  - remainder of the file striped across max_{s11} OSTs
- for *multi process single file* jobs
  - first 16MB striped across 1 OST

- remainder of the file striped across max_{s21} OSTs depending on job allocation request

The first extent threshold is taken to prevent jobs writing many small files from creating unnecessary network and disk contention which would negatively impact performance for all jobs without providing a meaningful benefit for the given job [MBOD16].

3 Testing

LLNL's IOR benchmark (v4.1.0) is used to perform a baseline of I/O tests on the test cluster as it provides configuration options to perform parallel file I/O in both a file-per-process and shared file mode [SAS08]. IOR writes data sequentially according to a defined block (`-b`) and transfer (`-t`) size grouped into a number of segments (`-s`). This pattern is visualized in Figure 3.1.

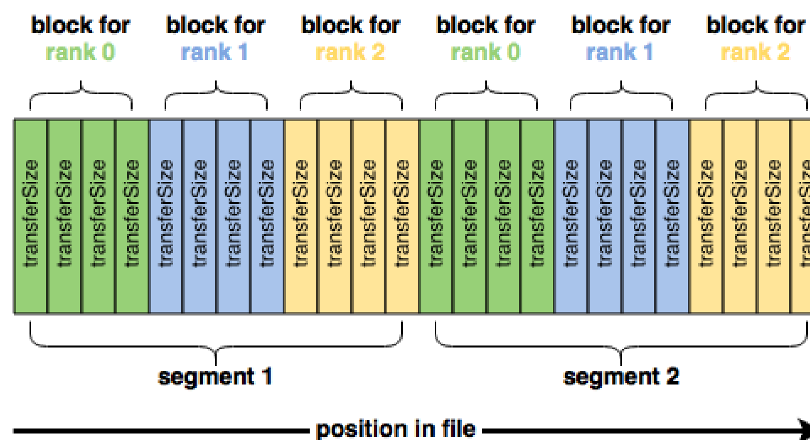

Figure 3.1: IOR Configuration Options [Aut23]

```
1  #!/bin/bash
2  # sample script running the IOR benchmark in file per
     ↪ process mode to simulate a parallelized application
3  #SBATCH --ntasks=96
4  #SBATCH --ntasks-per-node=24
5  #SBATCH --partition=debug
6
7  # this is needed to run as root
8  export OMPI_ALLOW_RUN_AS_ROOT=1
9  export OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1
10
11 filename=`date '+%Y_%m_%d__%H_%M_%S'`
12 filename="test_$filename"
13
14 # 96 tasks (24 per node)
15 # 1 megabyte transfer size
```

```
16 # 64 megabytes block size
17 # 32 segments
18 # 5 iterations
19 mpirun -n 96 ior -t 1m -b 64m -s 32 -k -F -C -e -i 5 -o
     ↪ $filename
```

Listing 3.1: IOR Benchmark Job Submission Script

3.1 Performance Baseline

A set of IOR benchmarks was run to establish a performance baseline of the storage setup available on the test cluster for different job and stripe configurations. For these baseline results IOR was called using a transfer size of 1MB and a blocksize of 16MB. Depending on the task configuration the segment number was adjusted to yield an aggregate filesize of 32GB ensuring the file does not fit into the operating system memory or cache. The tests were run sequentially to avoid any cross-influence and executed with five iterations (via the `-i` flag of IOR). Table 3.1 displays the mean read and write bandwidth results.

Configuration				File-Per-Process		Single-Shared-File	
#	Nodes	Tasks	Stripes	Read	Write	Read	Write
1	1	1	1	112 MB/s	111 MB/s	112 MB/s	111 MB/s
2	1	1	4	112 MB/s	111 MB/s	112 MB/s	111 MB/s
3	1	24	1	112 MB/s	111 MB/s	109 MB/s	111 MB/s
4	1	24	4	112 MB/s	111 MB/s	112 MB/s	111 MB/s
5	4	4	1	439 MB/s	432 MB/s	105 MB/s	110 MB/s
6	4	4	4	418 MB/s	433 MB/s	298 MB/s	268 MB/s
7	4	96	4	443 MB/s	440 MB/s	441 MB/s	430 MB/s
8	4	96	5	443 MB/s	443 MB/s	443 MB/s	437 MB/s

Table 3.1: Performance Baseline

The first two test results show that for non-parallelized tasks the IO bandwidth is limited by the gigabit network link between a single compute node and the Lustre objects storage servers. Increasing the number of stripes does not benefit throughput as a single target is able to saturate the gigabit link. In these tests switching between a file-per-process or a single-shared-file configuration leads to the same results as there is only ever a single process running as part of the job.

The same behavior is noticeable for test results three and four which perform parallelized IO from multiple tasks located on a single node. Here again the bandwidth is limited by the network so adding additional tasks on a single node does not improve pure I/O performance.

The test results for parallelized jobs across multiple nodes (tests five through eight) show improved performance over the non-parallelized jobs. In the file-per-process mode

tasks do not compete for access to a single file and as such this configuration better simulates multiple separate jobs running in parallel on the cluster. Here the performance is roughly equivalent to the number of nodes times the compute to storage bandwidth.

For parallelized jobs in a single-shared-file configuration the results are different, showing no gain (over single node execution) in a single stripe configuration. This is reasonable as all nodes compete over the gigabit network link between the respective OST and any given single compute node. With tests six through eight as the stripe size is increased to match the number of nodes the performance improves. It is noteworthy that more than a single task per node is needed to reach equivalent throughput to the file-per-process mode saturating the network bandwidth once again. In the same manner adding a fifth stripe, as is done in test eight, does not improve performance due to the bandwidth limitation.

3.2 Layout Selection

In order to verify that the plugin selects the desired layout the same tests where run with the plugin enabled. Here the striping layout was not set statically ahead of time but calculated by the plugin and as such is different depending on if the benchmark was launched in file-per-process or single-shared-file mode. The results are displayed in Table 3.2 and show the stripe size of the largest (last) extent. The tests include two additional cases of 48 & 72 tasks across 2 & 3 requested nodes to demonstrate that the selected layout is able to fully saturate the storage bandwidth available to the compute nodes.

Configuration			File-Per-Process			Single-Shared-File		
#	Nodes	Tasks	Stripes	Read	Write	Stripes	Read	Write
1	1	1	1	111 MB/s	110 MB/s	1	111 MB/s	109 MB/s
3	1	24	1	112 MB/s	111 MB/s	1	102 MB/s	110 MB/s
5	4	4	1	438 MB/s	425 MB/s	4	294 MB/s	252 MB/s
7	4	96	1	443 MB/s	439 MB/s	4	441 MB/s	426 MB/s
9	2	48	1	224 MB/s	221 MB/s	2	218 MB/s	218 MB/s
10	3	72	1	331 MB/s	328 MB/s	3	322 MB/s	325 MB/s

Table 3.2: Layout Selection

Listing 3.2 shows the job submission and script response for such a test.

```
1 root@west1:/mnt/lustre/work/tmp sbatch ior_mps.slurm
2 sbatch: slurm_job_submit: job from uid 0
3 sbatch: slurm_job_submit: job_desc.name ior_mps.slurm
4 sbatch: slurm_job_submit: job_desc.work_dir
    ↪ /mnt/lustre/work/tmp
5 sbatch: slurm_job_submit: nodes in use 0
6 sbatch: slurm_job_submit: storage_mode 21
```

```
7  sbatch: slurm_job_submit: slurm_requested_nodes 2
8  sbatch: slurm_job_submit: no running jobs detected,
     ↪ mode=eager
9  sbatch: slurm_job_submit: pfl_cmd lfs setstripe -E 16M -c 1
     ↪ -E -1 -c 2 /mnt/lustre/work/tmp
10 sbatch: slurm_job_submit: successfully set PFL
     ↪ configuration for mode 21
11 Submitted batch job 167
```

Listing 3.2: Sample Execution for Test 9

Querying the striping layout of the created test file as is done in Listing 3.3 shows the desired layout has been applied successfully by the script.

```
1  root@west1:/mnt/lustre/work/tmp lfs getstripe
     ↪ test_2023_04_15__11_08_52
2  test_2023_04_15__11_08_52
3  lcm_entry_count:   2
4  lcme_id:             1
5  lcme_extent.e_start: 0
6  lcme_extent.e_end:   16777216
7  lmm_stripe_count:  1
8  lmm_objects:
9  - 0: { l_ost_idx: 4, l_fid: [0x100040000:0x299e:0x0] }
10 lcme_id:             2
11 lcme_extent.e_start: 16777216
12 lcme_extent.e_end:   EOF
13 lmm_stripe_count:  2
14 lmm_objects:
15 - 0: { l_ost_idx: 3, l_fid: [0x100030000:0x298d:0x0] }
16 - 1: { l_ost_idx: 1, l_fid: [0x100010000:0x2980:0x0] }
```

Listing 3.3: Applied Layout for Test 9

4 Conclusion

4.1 Discussion & Limitations

The baseline tests executed in Table 3.1 showcase that three factors are important in choosing a relevant striping layout

- detecting the IO pattern and type of a given job: is the job executing IO in a file-per-process or in a single-shared-file mode?
- raw bandwidth / network bottlenecks: in the absence of other jobs, is a single object storage server able to saturate the network link to a single compute node and vice versa?
- activity on the cluster: are other running jobs already saturating the capabilities of the storage backend?

The job submission plugin introduced above is able to deal with the first two factors in a relatively straight forward way via global parameters and „world knowledge“. It is assumed that a single job submission script does not change storage modes post-submission and that the engineers setting up the system know the types ahead of time. Should this not be the case then the plugin could be extended by analyzing past executions and pulling information from a job database as is done with DCA-IO [KSW+19].

The third factor is significantly more difficult to adjust for in a job submission plugin as the plugin is executed when the job is submitted to the cluster not when it starts executing. It would be feasible to adjust for this to some extent by adding the logic to determine the appropriate PFL configuration to a short helper script whose execution the plugin places just before the MPI executions in the job script. This script could then query the cluster state and apply the configuration. Unfortunately this would not solve problems arising during the execution of the job. Lustre currently does not allowing modifying a file's PFL configuration without completely rewriting it, which would add considerable IO to the job just to account for changes in the cluster environment. As such, if the SLURM or Lustre cluster load changes drastically post submission, the mechanisms explored here are not able to adjust.

Additionally, it is important to note that the job submission plugin proposed in the project has some significant limitations.

- SLURM job submission scripts are executed as the `slurm` user and not as the user submitting the job. This can create issues if the user running SLURM does not

have write privileges in the target directory to which the job would need to write. In this case the script cannot default the progressive file layout at all.

- The plugin currently only supports a single PFL layout per job. This could be extended to enable nested jobs to trigger different layouts, however there would need to be an external way to communicate to the plugin which sub-steps perform IO for which file targets since the SLURM job descriptor only passes the working directory on submission. It could be feasible to perform this via user submitted parameters in the submission script or by keeping a mapping of applications to storage directories externally.
- The plugin currently does not consider storage or compute cluster utilization during execution in any way. Noticeable the plugin cannot adjust the configuration depending on active load after submission.

4.2 Future Work

In order to create a utility which is able to add I/O aware characteristics into SLURM in a production-ready setup additional work is required. The proposed implementation here would benefit from moving much of the static configuration, currently performed via constants in the script, to a dynamic setup read from a database. This implementation could expand the ideas discussed in DCA-IO [KSW+19] by measuring past executions performance and tweaking the execution parameters accordingly.

On the level of an individual job submission the simple I/O categorization performed in this project does not capture timing characteristics of job execution. It is likely that jobs perform compute and I/O intensive workloads in bursts, within the same job. Practically it is easy to imagine a job reading the input dataset, then performing a large amount of computation, writing a checkpoint, performing additional computation and eventually writing out the job results. Depending on the stage the parallelization (and storage mode) may be different and further analysis would be needed to accurately capture these patterns.

Additionally the functionality could be enhanced, in a similar fashion, by performing scheduling decisions based upon the current load of the storage cluster. In this way the execution order of I/O intensive tasks by SLURM could be delayed in favor of compute intensive tasks if the Lustre system is currently under heavy load. This could improve the overall task throughput (in terms task execution time) of the cluster by spacing out utilization more evenly over time.

Bibliography

[Aut23] IOR Authors. IOR - User Documentation. Online, 2023. `https://ior.readthedocs.io/en/latest/userDoc/tutorial.html`.

[Bra19] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[Cen23] National Energy Research Scientific Computing Center. NESRC Documentation - Running Jobs. Online, 2023. `https://docs.nersc.gov/jobs/`.

[Dun23] Chris Dunlap. Munge Installation Guide. Online, 2023. `https://github.com/dun/munge/wiki/Installation-Guide`.

[GEK+22] Junmin Gu, Greg Eisenhauer, Scott Klasky, Norbert Podhorszki, Ruonan Wang, and Kesheng Wu. Exploring large all-flash storage system with scientific simulation. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management*, pages 1–4, 2022.

[Kli23] Deutsches Klimarechenzentrum. DKRZ: HLRE-4 Levante. Online, 2023. `https://www.dkrz.de/de/systeme/hpc`.

[KSW+19] Sunggon Kim, Alex Sim, Kesheng Wu, Suren Byna, Teng Wang, Yongseok Son, and Hyeonsang Eom. Dca-io: a dynamic i/o control scheme for parallel and distributed file systems. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 351–360. IEEE, 2019.

[MBOD16] Rick Mohr, Michael Brim, Sarp Oral, and Andreas Dilger. Evaluating progressive file layouts for lustre. In *Cray User Group Conference (CUG 2016)*, 2016.

[OE23] OpenSFS and EOFS. Lustre Software Release 2.x Operations Manual. Online, 2023. `https://doc.lustre.org/lustre_manual.xhtml`.

[Pro22] TOP500 Project. TOP500 - November 2022. Online, 2022. `https://www.top500.org/lists/top500/2022/11/`.

[SAS08] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008.

[Sch23a] SchedMD. Slurm: A Highly Scalable Workload Manager. Online, 2023. https://github.com/SchedMD/slurm.

[Sch23b] SchedMD. Slurm: Job Submit Plugin API. Online, 2023. https://slurm.schedmd.com/job_submit_plugins.html.

[Tor22] Giovanni Torres. Slurm Docker Cluster. Online, 2022. https://github.com/giovtorres/slurm-docker-cluster.

[WSHO17] Feiyi Wang, Hyogi Sim, Cameron Harr, and Sarp Oral. Diving into petascale production file systems through large scale profiling and analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 37–42, 2017.

[YJG03] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper 9*, pages 44–60. Springer, 2003.

Appendices

List of Figures

1.1 SLURM Architecture [YJG03] 4
1.2 A typical distributed Lustre deployment [OE23] 5
1.3 Lustre File Write Operation [OE23] 6
1.4 Lustre File Striping [OE23] 6
1.5 Lustre Progress File Layout [OE23] 8

2.1 Simple Cluster Architecture 18

3.1 IOR Configuration Options [Aut23] 21

List of Listings

1.1 Retrieving the Lustre PFL definition 7

2.1 Submitting a Job on the local cluster 11
2.2 Enabling lua support in slurm.conf 12
2.3 Installing SLURM Controller Node on Test Cluster 12
2.4 Installing SLURM Compute Node on Test Cluster 13
2.5 SLURM Configuration 14
2.6 Starting `slurmctld` 15
2.7 Starting `slurmd` 15
2.8 Hello World Job Submit Plugin 15
2.9 Configuring a PFL setup with lfs setstripe 16

3.1 IOR Benchmark Job Submission Script 21
3.2 Sample Execution for Test 9 23
3.3 Applied Layout for Test 9 24

List of Tables

2.1 Docker Volumes 11
2.2 lfs setstripe command 16
2.3 Plugin Global Constants 17
2.4 Plugin Storage Modes 18

3.1 Performance Baseline 22
3.2 Layout Selection 23