Storage systems @TGCC and Lustre File Systems

Architecture and best practices

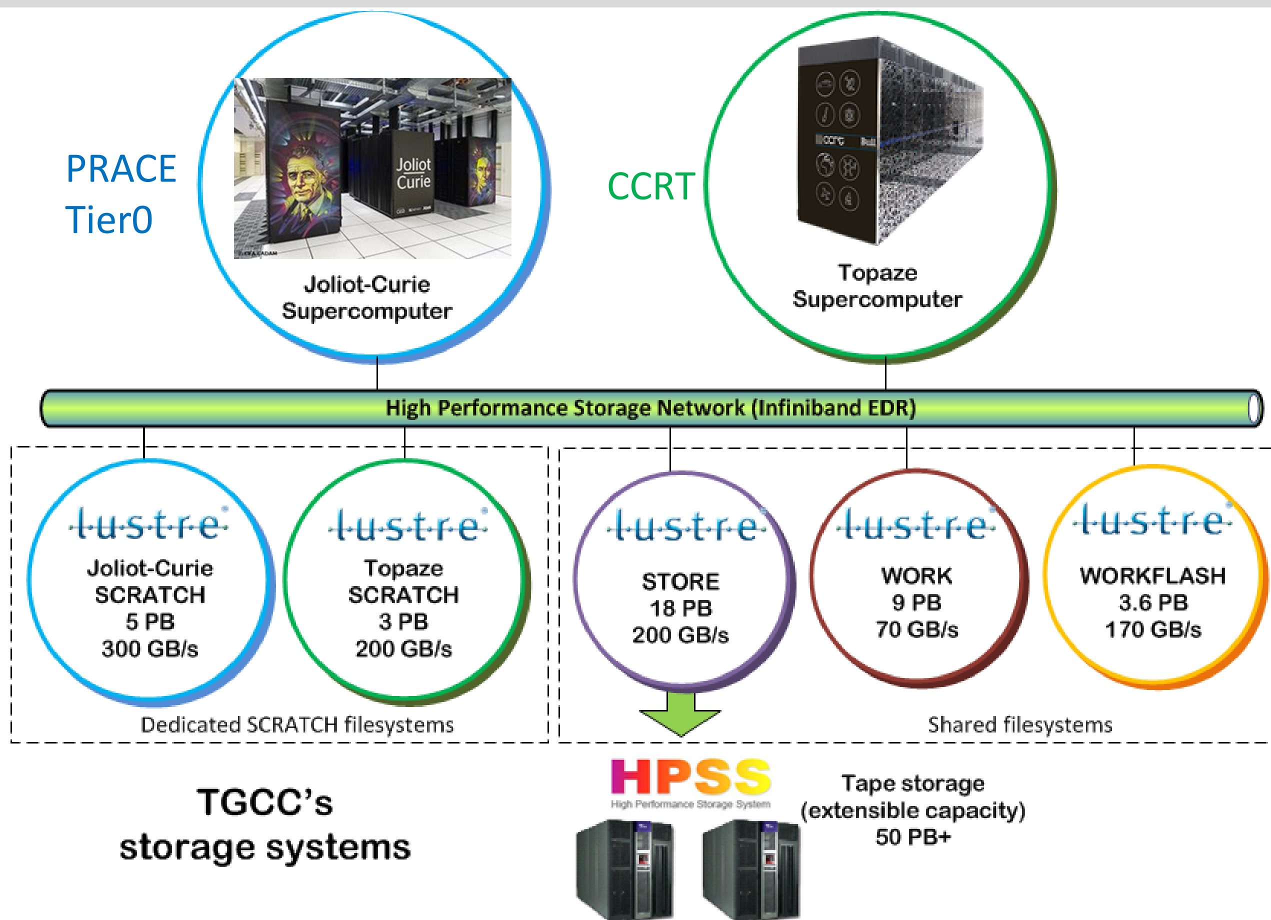
PATC - High Performance Parallel I/O and post-processing

Thomas Leibovici | CEA/DAM

Agenda

- TGCC storage architecture
- TGCC storage workspaces
- Lustre parallel file system
- Hierarchical Storage (STORE)
- Integration of SSDs and hybrid filesystems
- Object store and Swift interface

TGCC's storage architecture



Scratch = temporary storage

- Workspace for temporary data
- One dedicated *Scratch* per supercomputer
- Mount point: /ccc/scratch ($CCCSCRATCHDIR)
- Unused files deleted after 40 days
- Designed for throughput and performance

Work = permanent workspace

- Permanent workspace (no purge)
- Accessible from all compute clusters
- Larger that “home” directories
- Quotas : 1TB, 500k inodes per user
- Mount point: /ccc/work ($CCCWORKDIR)
- /!\ no backup

[new] Workflash

- Same management as *work* except it is made of SSDs
  - Speeds-up non-sequential I/O operations
- Also accessible via $CCCWORKDIR
- Not for all projects. Switch between WORK and WORKFLASH depends on:
  - Where your project is submitted (available through Prace-ICEI joint calls, HBP, DARI…)
  - Project submission must specifically request this workspace (actual need for it is assessed with the project)

Store = long-term storage

- To store final results
- Connected to a tape storage for larger capacity
- Recommended file size : 1GB-100GB
- Quotas : 100k inodes per user, no quota on volume
- Automated migration and staging with tape storage (see later slides)
- Mount point: /ccc/store ($CCCSTOREDIR)
- <u>Designed for data capacity</u>

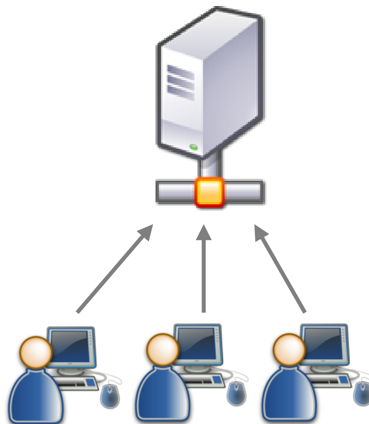
"Long term parking"

Lustre Parallel File System

From Local File Systems to Parallel File Systems

Local file system
1 disk, 1 client

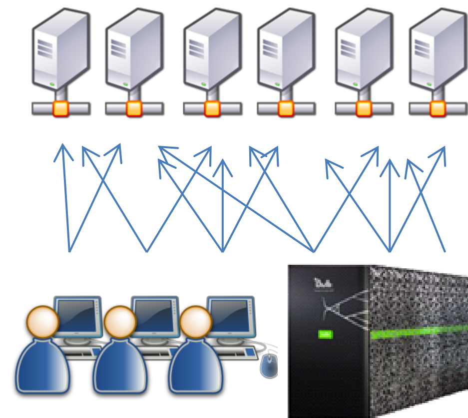
Examples:
- Personnal computer
- /tmp of compute nodes



File server
1 server, N clients

Example:
Login home

Interests:
sharing, access from any workstation

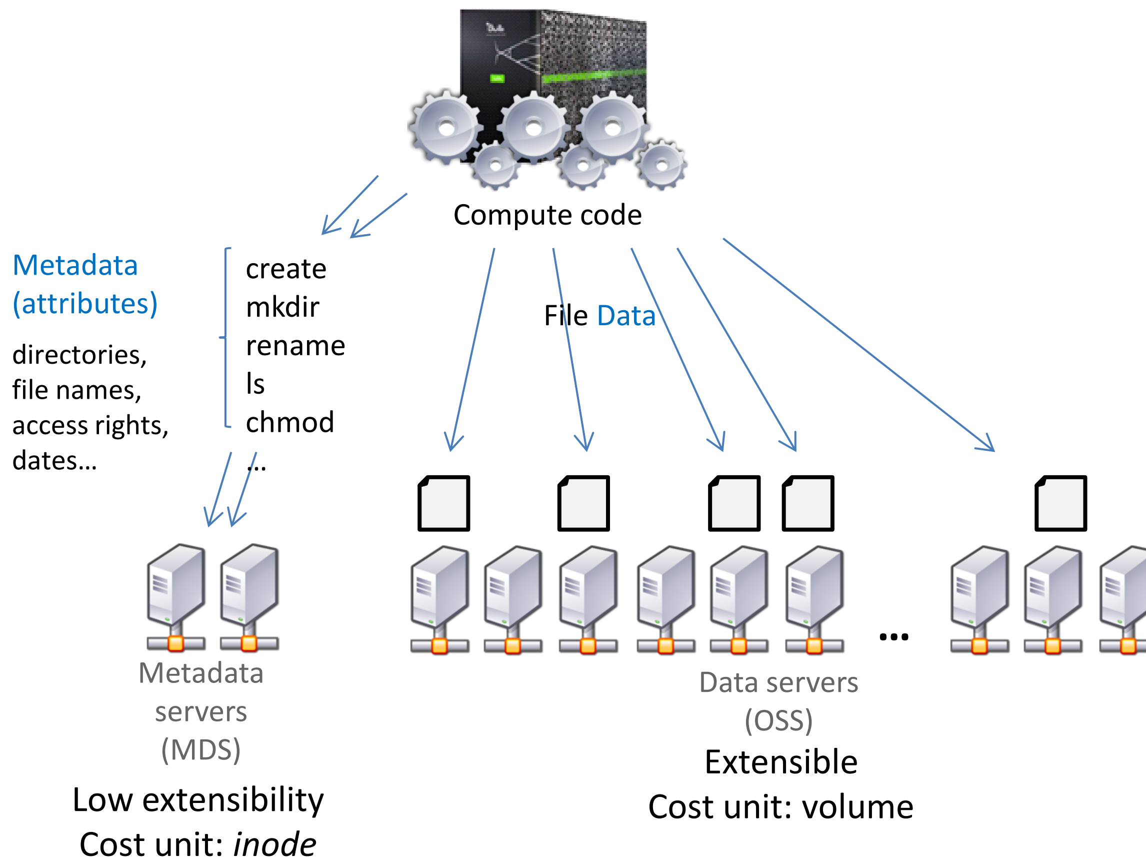


Parallel file system
M servers, N clients

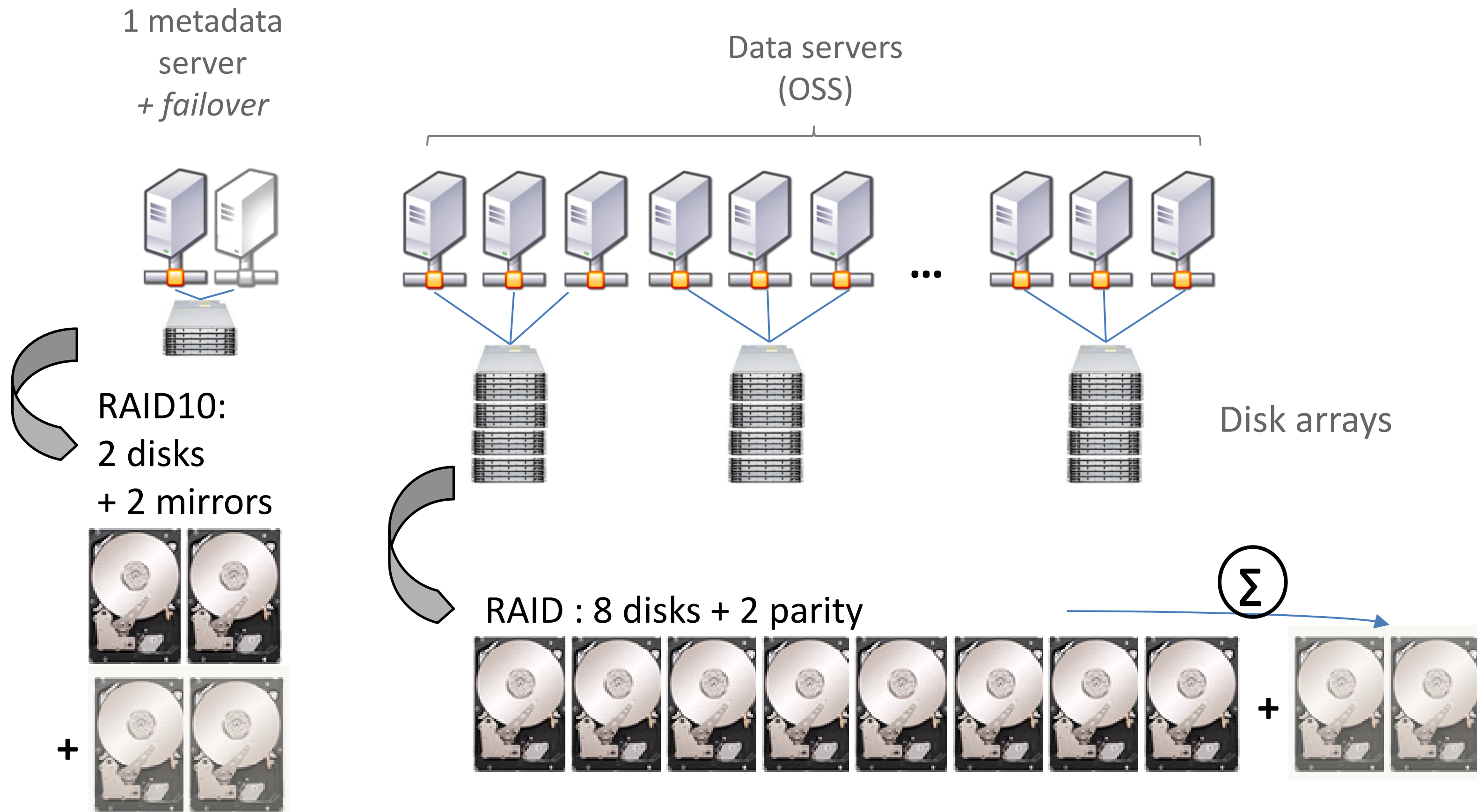
Example:
scratch of supercomputer

Interests:
scalability, performance, fault tolerance

Lustre: A Parallel File System

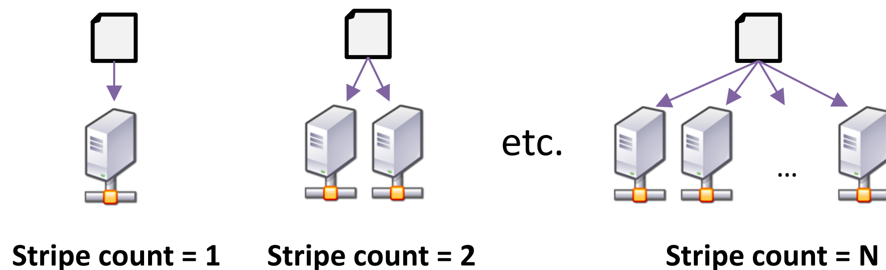


Hardware Redundancy for Lustre Filesystems (at TGCC)



What is striping?

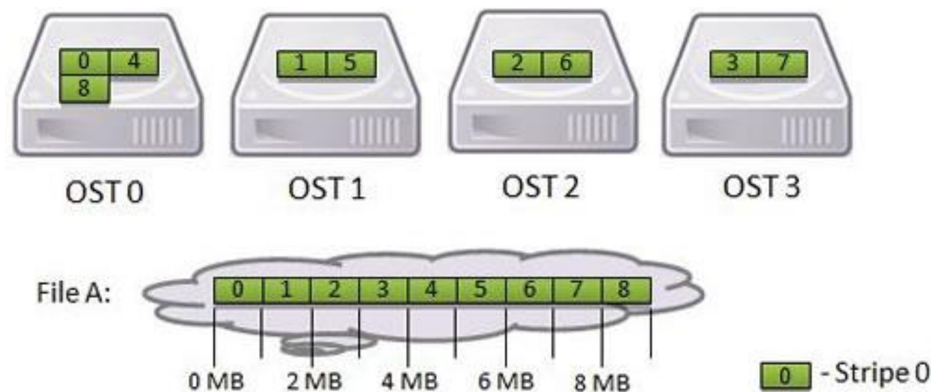
- To increase throughput, Lustre can paralellize file storage on several servers:



What is striping?

- Data is distributed between servers as blocks of ‘stripe size’ :

Example:
stripe_count=4
stripe_size=1MB



Striping examples

Querying directory and file striping

```
$ lfs getstripe -d /path/to/directory
stripe_count: 1 stripe_size: 1048576 pattern: 0 stripe_offset: -1

$ lfs getstripe /path/to/file
lmm_stripe_count:  2
lmm_stripe_size:   1048576
lmm_pattern:       raid0
lmm_layout_gen:    0
lmm_stripe_offset: 1
        obdidx           objid          objid          group
             1       132026355      0x7de8ff3      0x280000400
             3       130617540      0x7c910c4      0x2c0000400
```

Striping recommendations

What striping to set?

- Striping > 1 induces extra costs (N servers to communicate with) but results in an increased bandwidth
- Useless for small files (< a few MB)
- Worthwhile for bigger files (~ Gigabyte-sized)
- If accessed from a single client: stripe_count = 2 is enough to get the max throughput
- Increase stripe count if many clients write large volumes of data to the same file
- As much as possible, align writes with stripe_size -> best performance
- Mandatory for huge files (x100 GB): avoid having more than 500GB / IO server

Setting stripe

How to set stripe?

- Per directory:
  - File and sub-directories inherit when they are created
  - Only affects new file creation (not previously created files)
  - Command line:

```
lfs setstripe -c <stripe_count>  <directory>
```

- When creating a file with ROMIO:

```
MPI_Info_create(&info);
MPI_Info_set(info, "striping_factor", "32");
MPI_File_open(MPI_COMM_WORLD, "foo.chkpt", MPI_MODE_CREATE, info, &fh);
```

- Default striping @ TGCC:
  - 1 on scratch and work
  - 4 on store
  - 4 with MPI-IO

Lustre: Best Practices

TGCC-specific:

- Lustre filesystems @TGCC are not backed up
  → keep critical data (e.g. source code) in your home directory

Limiting metadata overhead

- Avoid using « ls –l » when « ls » is enough
- "ls" is purely sequential: avoid having a huge number of files in a single directory (<1000)
- Use a stripe count of 1 for directories with many small files (< few MB)
- Avoid repetitive open/close operations
- Example of inappropriate script:

```
while … do
        echo 'bla' >> my_file.out
done
```

Lustre: Best Practices

Limiting metadata overhead and optimizing throughput:

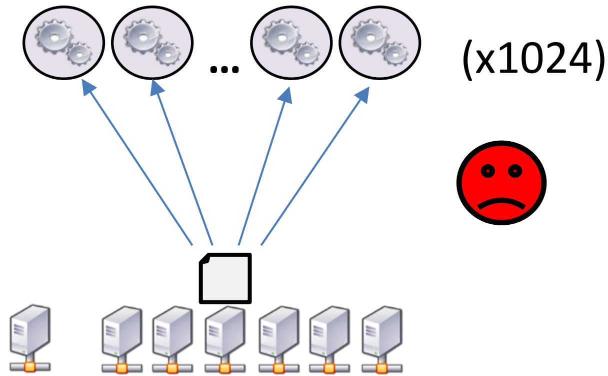
- Avoid small files on Lustre filesystems
  - Few KB is bad
  - Few MB is suboptimal
  - Several GB is good
  - Several TB may reach system limits (ulimit, timeouts…)

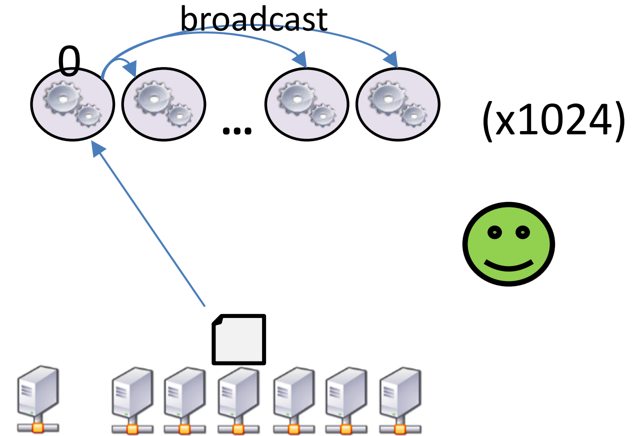
Parallelism and locking

- Limit the number of processes writing simultaneously to the same file (locking contentions)
- Open « read-only » when only reading a file
  - E.g. in Fortran, use ACTION='read' instead of the default **ACTION='readwrite‘**

Parallel I/O strategies

1) Parallel read

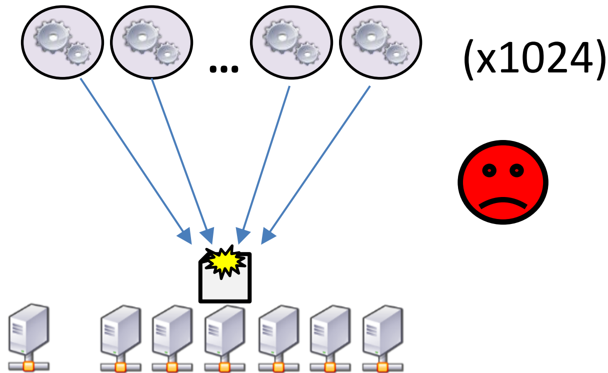




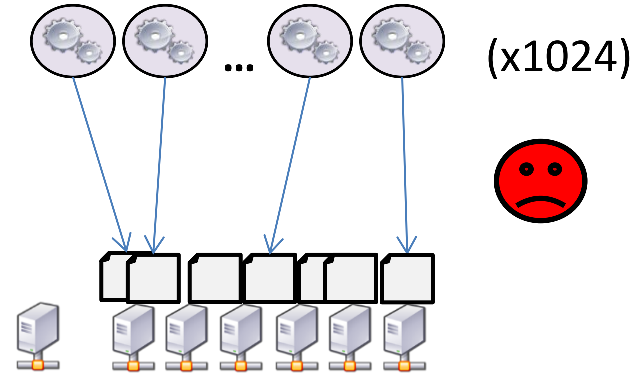
- Each task reads data once
- Data serveur load = 1024
- Metadata load = 1024 x open/close

- 1 task reads the file
- Data serveur load = 1
- Metadata load = 1 x open/close

2) Parallel write

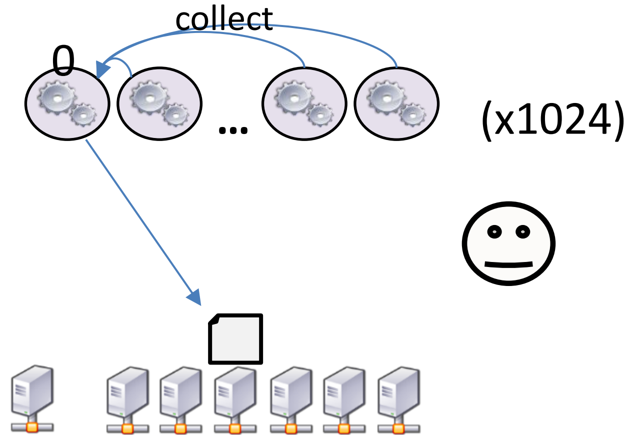


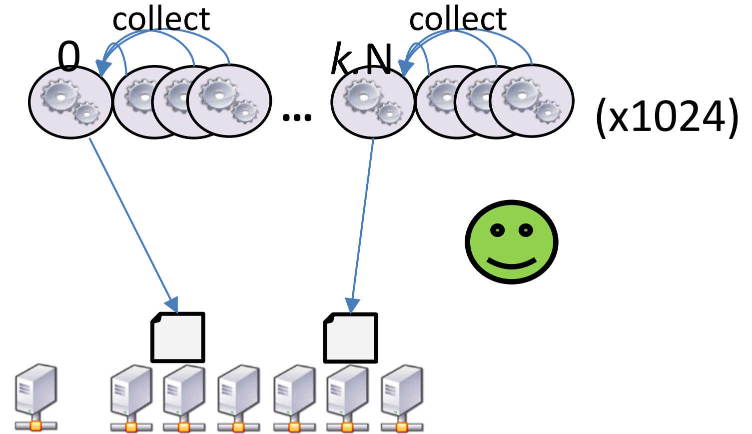
- Each task writes a part of the file
- Data server load = 1024 + « lock tempest »
- Metadata load = 1 create + 1024 x open/close



- Each task writes a file
- Load is distributed accross servers
  - Note: load may overlap on some servers, resulting in uneven performance between files
- Metadata load = 1024 create + 1024 x open/close
- /!\ Many small files
  - Increased metadata overhead
  - + « lock tempest » on the directory

2) Parallel write (continued)





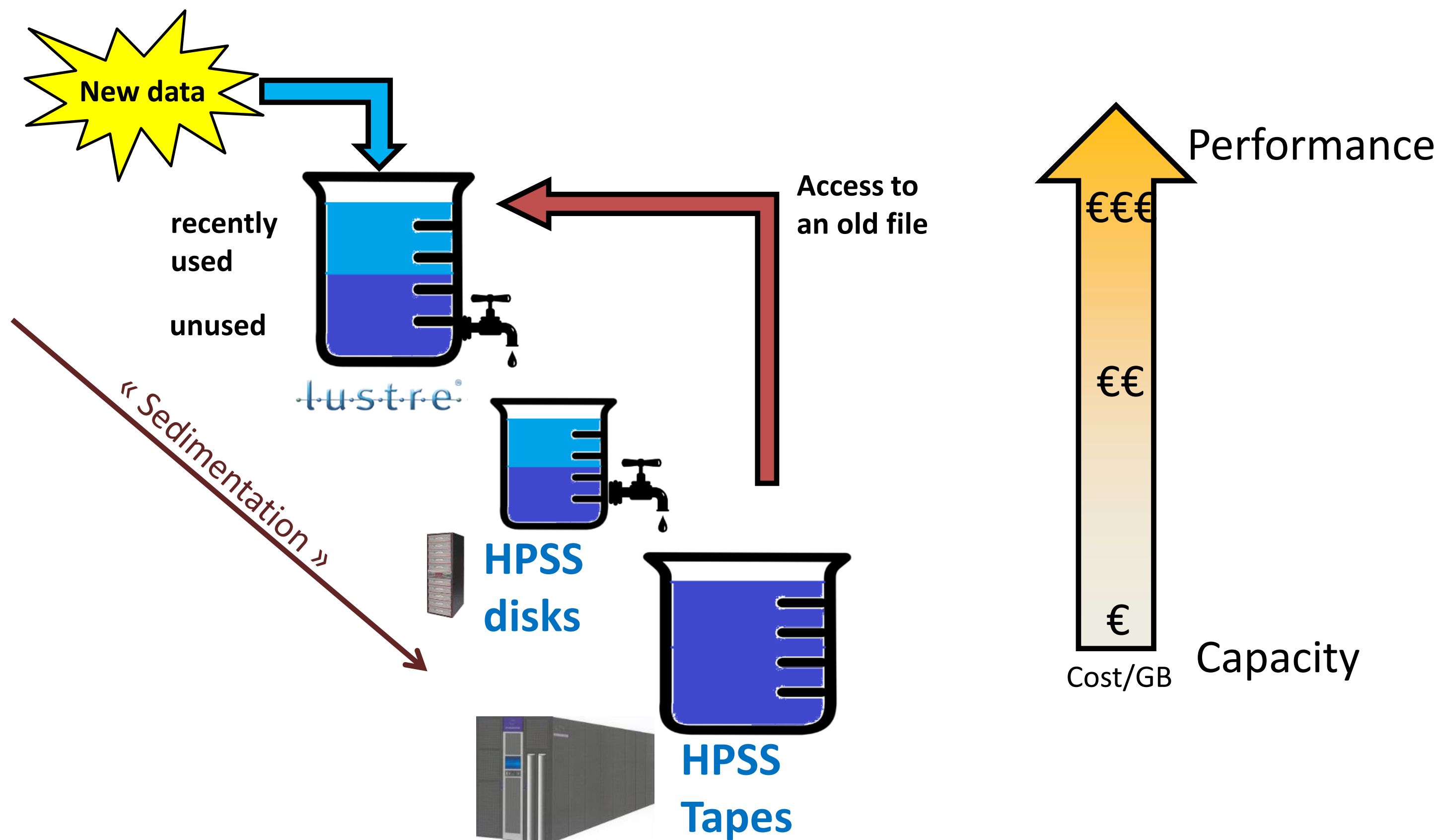
- 1 task collects all data
- Data server load = 1
- Metadata load = 1 create/open/close
- **BUT**: limited bandwidth (1 single write stream)

- Data is collected per group of tasks
- Data server load = 1024/group size
- Metadata load = 1024/group size
- **Parallel and scalable** (commulates several write streams)

Hierarchical Storage Management

and the STORE filesystem

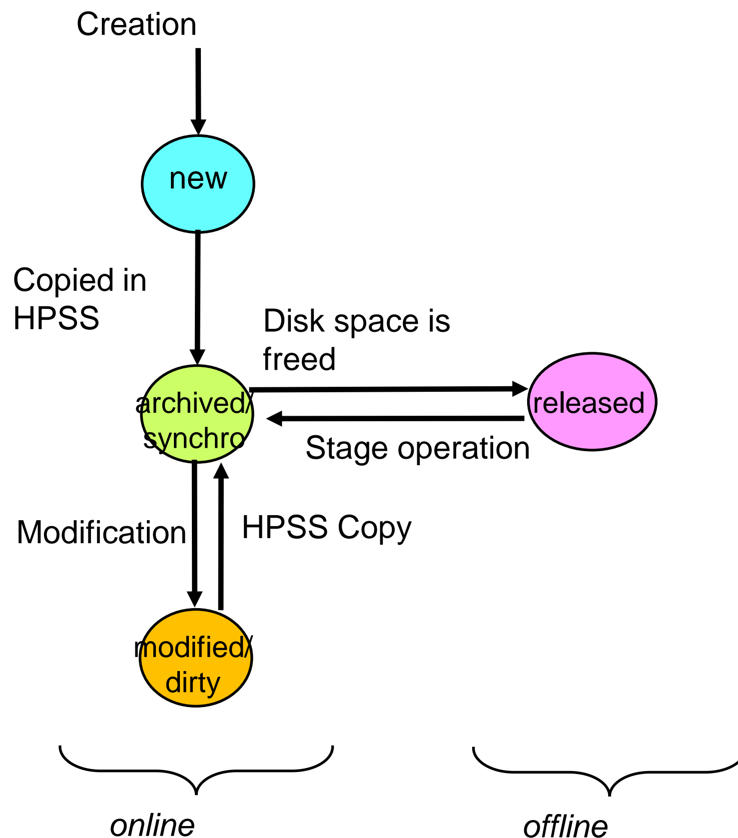
Principle of HSM: data “sedimentation”



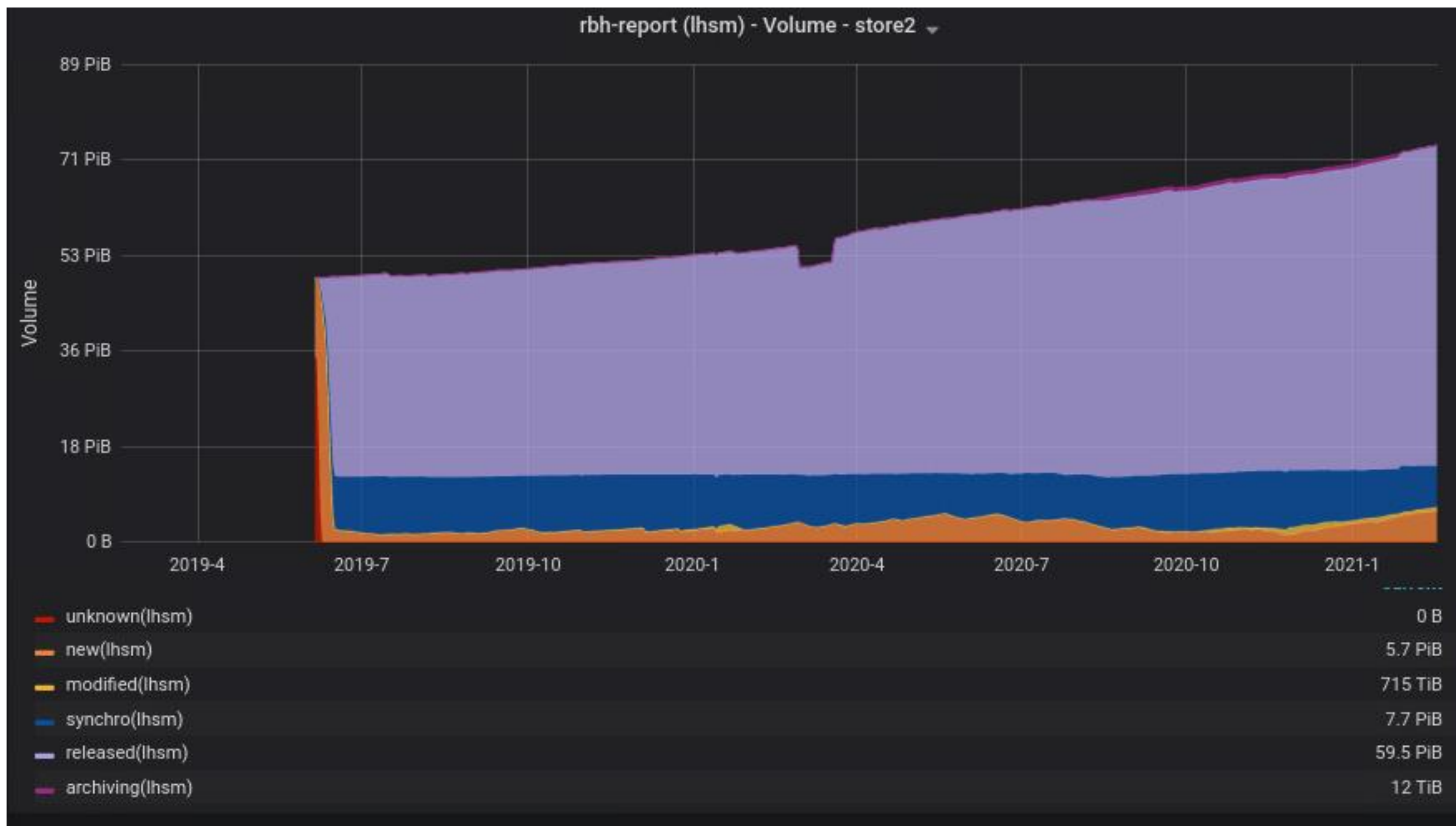
How Lustre HSM works?

- *store* is permanently watched by a *Policy Engine (Robinhood)*
- Eligible files for migration are automatically stored in HPSS
  - The filesystem is saved in the HSM
  - Possible recovery in case of crash, major hardware failure, FS been reformatted
- Older files are
  - Still visible in store with their original size
  - Their contents are out of store and kept in HPSS
  - This is fully transparent to the end-user
  - Freed space in store is available for new files
- Released files are staged back at first access
  - Transparent to the end-user
  - The first IO call is blocked until the stage operation is completed

A file's life on STORE



Overview of files status on STORE



STORE: user interface

Users' view:

- User has access to data via a standardized path: $CCCSTOREDIR =/ccc/store/contxxx/grp/usr
- No direct access to HPSS, it's « hidden » behind the Lustre filesystem
- Regular commands apply to store
- Accessing a released file brings it back to LUSTRE. (data access is blocked until the recall is completed).

ccc_hsm commands:

- **ccc_hsm status**: query file status (online, released, …)
- **ccc_hsm get**: explicitly reload files
- **ccc_hsm ls**: does « ls » but show hsm status (online, released) too
- **ccc_hsm hint**: give indications to the system about future use of a file (will access, won't access, won't change, don't archive…)

Preloading data from tapes

- Retrieving data from tapes can be **long**: mounting and reading magnetic tapes
- It is **advised** to preload data **before submitting** a job (to reuse or post-process an **old computation**)
- Preloading data **avoid wasting compute time**
- To preload ‘released’ files, use: **ccc_hsm get**

Tips: using “du” on STORE

What does ‘du’ reports on STORE?

- By default, ‘du’ displays space used on **Lustre disks**, i.e. only ‘online’ files:

  # du -sh $CCCSTOREDIR
  2T (?!)

- If you want to get the total usage for both Lustre and HPSS use the ‘–b’ option:

  # du -bsh $CCCSTOREDIR
  224T ☺

Good practice on STORE: Packing data

On store, pack your data into big files

- The time to reload each file from tape is significant:
  - Time to move & load the tape in a tape drive, time to rewind the tape…

➔ **Packing data into bigger files makes it possible to reduce the time to read-back data from tapes**

Example: Reading the same amount of data (100GB) from tapes

	10 files x 10GB	1000 files x 100MB
Time to read-back from tapes	A few minutes	Several hours
Overall system usage (tape drives)	Partial	Full

➔ **Recommended file size on store: 10GB to 500GB**

Common tool to pack data

TAR is dangerous… only in cigarettes

- Using “tar” command is an easy way of packing files

```
tar cf output.tar source_directory
```

- Tools exist to access tarballs from software
  - Tar files follow a well known standard
  - See libarchive for example
- TAR preserves metadata
  - Permissions
  - Owners/groups
- TAR preserves symlinks
- A TAR file can be appended
- Tar can be extracted (as a whole, or per file)
- A similar tool “**dar**” provides additional features like comparing, merging…

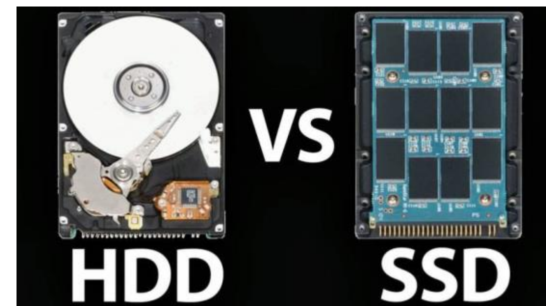
Thinking on a I/O middleware to perform I/Os in simulation codes is never a bad idea

Integration of SSDs in Lustre filesystems

Management of “hybride” filesystems

Flash Storage

- Flash devices (like SSDs) are very fast:
  - High bandwidth (recent models: a couple of GB/s)
  - High operation rate: x 100k IOPS
  - Random I/Os at no cost (no head movements)
  - Now in production at TGCC: workflash filesystem



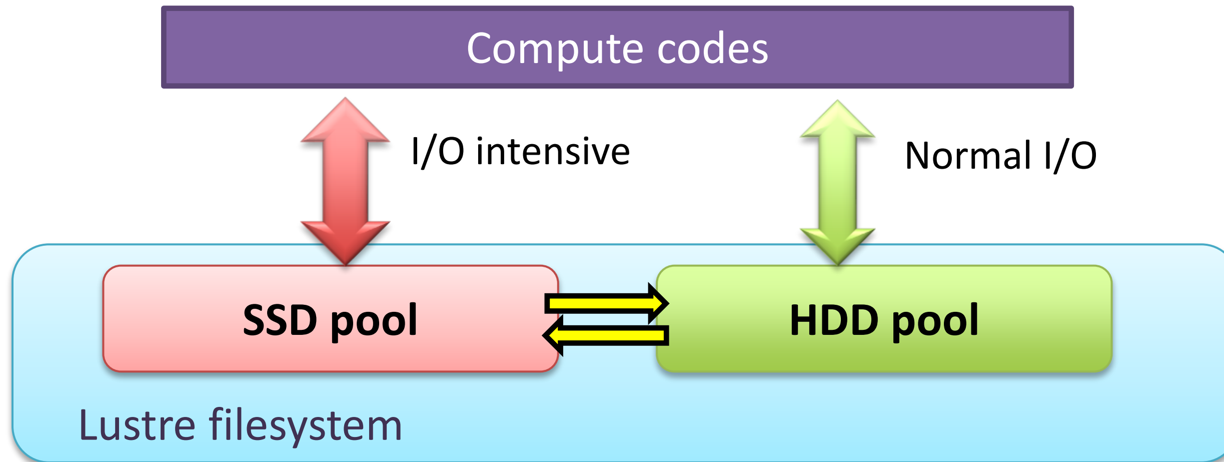
- Comparison of storage technologies:

Device type	Bandwidth	Latency	Price
Flash (SSD)	1-3 GB/s	~µsec	500€/TB
Disk (HDD)	250MB/s	~msec	50€/TB
Tape	350MB/s	~seconds	5€/TB

“Hybrid” filesystems

Integration of SSDs to Lustre filesystems

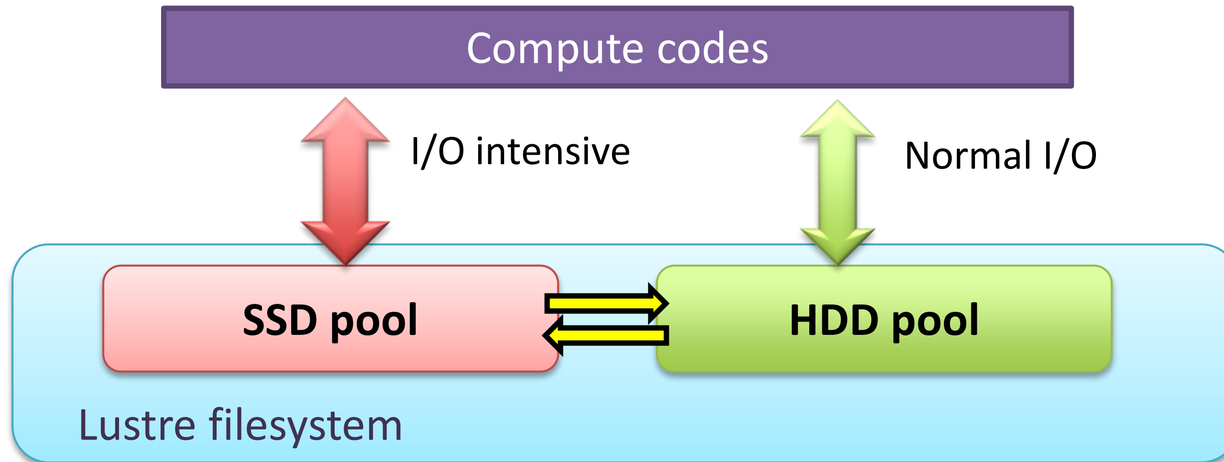
- To accelerate specific I/O patterns (e.g. data extraction…)
- To provide higher bandwidth
- CCRT’s Topaze scratch is such a hybrid system
- This is a common trend for future systems



"Hybrid" filesystems

Data flow

- Codes/users can select the target pool for a directory or a specific file
- Codes/users can explicitly request migrations from SSD to HDD or from HDD to SSD.
- Data from SSDs is automatically migrated to HDD after some time.



ccc_hsm for managing SSDs

- Management of SSD<->HDD migrations is also integrated to the ccc_hsm command
- Commands (also available as a C API):
  - **ccc_hsm status –T** : indicates the current location of the data (oneline/offline, ssd, hdd…)
  - **ccc_hsm ls –T** : like « ls » with information about the current storage location

```
$ ccc_hsm status -T my_file
my_file                      online (ssd)

$ ccc_hsm ls my_directory
online (ssd) file.1
online (sdd) file.2
online (hdd) file.3
```

ccc_hsm for managing SSDs

- **ccc_hsm mkdir** {--capacitive | --random-io} *directory_name*
  Creates a new directory for the given I/O pattern
  (capacitive -> HDD, random-io -> SSD)
- **ccc_hsm create** {--capacitive | --random-io} *file_name*
  Creates a single file for the given I/O pattern.

```
$ ccc_hsm mkdir --random-io my_flash_dir
$ cd my_flash_dir
$ touch file.1 file.2
$ ccc_hsm create --capacitive file.3

$ ccc_hsm ls -T
online (ssd) file.1
online (sdd) file.2
online (hdd) file.3
```

ccc_hsm for managing SSDs

- **ccc_hsm migrate** [--local] [--async] [--timeout *timeout*]
  [--capacitive | --random-io] [--] *file* [*file*...]

  Migrate a file from SSD to HDD or HDD from SSD.

  Migration can be delegated to a pool of workers that migrate files asynchronously, but currently at TGCC only local migration is possible.

```
$ ccc_hsm status -T  file.1
file.1               online (hdd)

$ ccc_hsm migrate --local --random-io file.1

$ ccc_hsm status -T file.1
file.1               online (ssd)
```

Object storage

with Swift/S3 interfaces

Object storage

- Object storage is a computer data storage architecture that manages data as objects (≠ files)
- Made popular by cloud computing services like Amazon AWS (S3 protocol)
- Many computing centers in Europe now provide such a storage service (BSC, CEA, CSCS, CINECA…)
- Allow access to data through HTTPS, from the supercomputer center, or from the Internet
- Convenient for sharing data (HTTPS links to download files)

Using Swift

- Example commands:

```
$ swift list
$ swift upload container object
$ swift download container object
…
```

- Client libraries available for most languages
  - e.g. python-swiftclient

Summary

KEY TAKEAWAYS

- Store the right data in the right space
  - Scratch = temporary data
  - Work = permanent workspace
  - Store = long-term (large files or packed data!)

- Flash storage speeds up random I/Os

KEY TAKEAWAYS

- To increase I/O performances:
  - Reduce the metadata overhead
    - The larger files, the better
    - Large I/Os better than small I/Os
  - Avoid locking contention (concurrent access to files)
    - Aggregating data per group of jobs is a good strategy
  - Be aware of the filesystem data alignment
    - E.g. with Lustre, align I/Os with the stripe size

DE LA RECHERCHE À L'INDUSTRIE

Thank you for your attention!

Other questions?

Other topics?