Sebastian Oeste
Center for Information Services and High Performance Computing (ZIH)

Introduction on parallel I/O and distributed file systems

NHR-Lecture

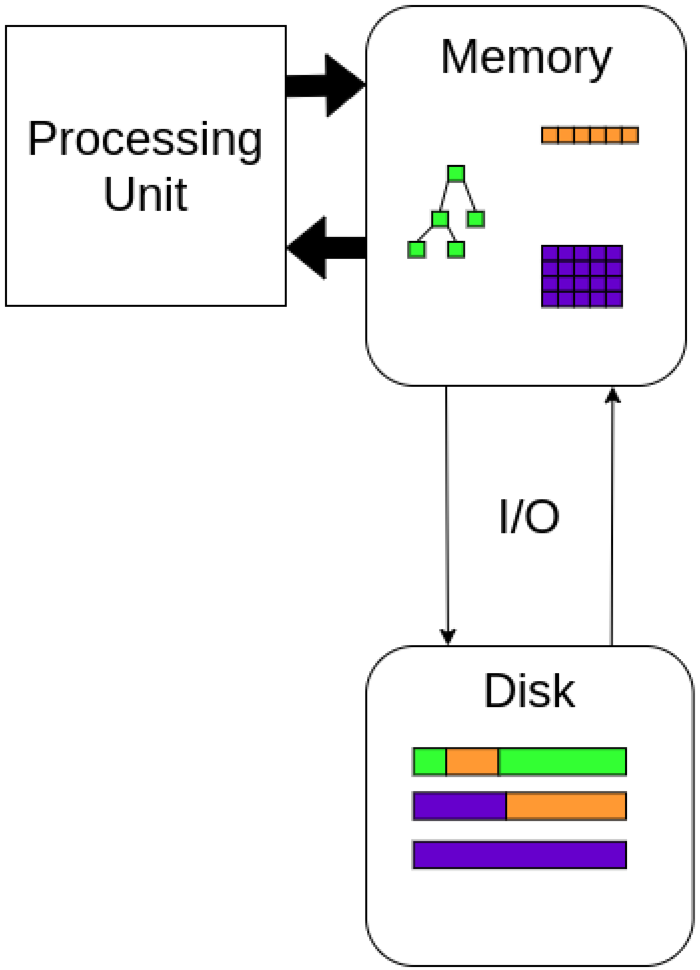
Agenda

1. Basic I/O
2. Parallel file systems
3. Parallel I/O
4. I/O performance analysis
5. Best practice for parallel I/O

What is I/O?

- **I/O is data migration!**
- Between data in memory and a storage medium (e.g. a disk).

- Application libraries gives in-memory data an application defined structure.
- On disks data is typically stored in files.
- What is a File?
  - Unix philosophy says „*Everything is a file*“.
  - Linus Torvalds says „*Everything is a stream of bytes*“*[1].*
  - POSIX says „*An object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. […]*”*[2].*
- A File is just an unstructured stream of bytes.
- Applications must manage I/O to close this semantical gap!

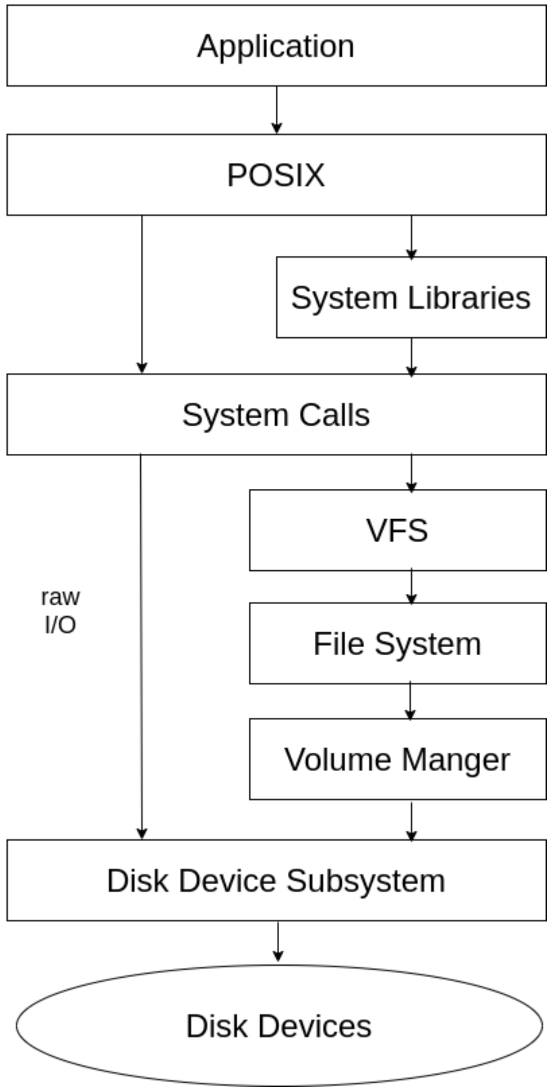


[1] https://yarchive.net/comp/linux/everything_is_file.html

[2] 2018. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (Jan 2018), 1–3951.

I/O Stack

- Most applications using POSIX-I/O API to do I/O.
- POSIX API may be Implemented in OS syscalls or additional system libraries are used for mapping (e.g. libc).
- System Calls are entry into the OS Kernel.
- VFS – virtual file system interface.
- File system works with abstract devices.
- Disk device subsystem does the disk dependent part.



POSIX and POSIX-I/O

- POSIX is the IEEE Portable Operating Standard Interface for Computing Environments.
- POSIX defines a standard way for an application program to obtain basic services from the operating system.
    - Tools, API's but also semantical requirements, such as consistency.
    - Not just I/O also for Processes, Signals, etc …
- Linux is mostly POSIX-compliant.
- POSIX-I/O defines an interface to work with files.
    - read(), write(), open(), close(), stat(), mkdir(), …
    - Defines also strong consistency requirements for data and metadata.
- POSIX at all was **not** designed with parallelism in mind.

[3] 2018. IEEE Standard for Information Technology–Portable Operating System
Interface (POSIX(R)) Base Specifications, Issue 7. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (Jan 2018), 1–3951.

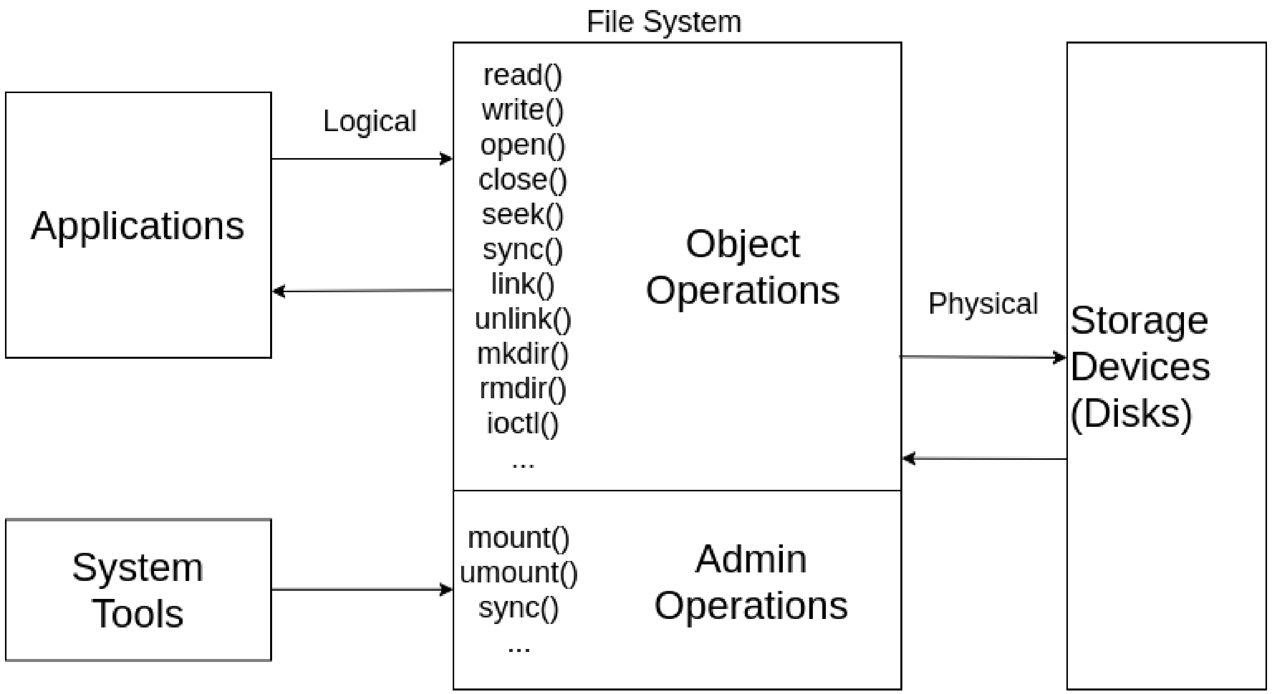
Files

- Files consists of data and metadata.
- Metadata include
  - Type of file
  - Premissions (rwxrwxrwx)
  - Timestamps
  - Size
  - Owner / Group
- Most files provide random access.
- Data is commonly stored in regular files.
- Files are organized by file systems.

Type of Files
Regular file (-)
Directory (d)
Block device (b)
Character device (c)
Symbolic link (l)
Socket (s)
FIFO (named pipe) (p)

Role of the file system

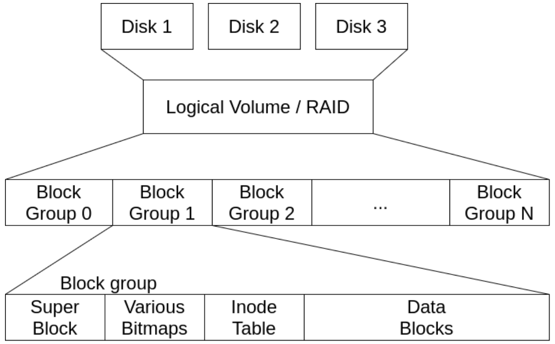
- A file system holds a collection of files.
- Maintain the file namespace (mostly directory hierachy)
- Storing contents of the files.
- Can be added or removed to a namespace.
- Map logical I/O requests from the applications to physical I/O requests to the disks.

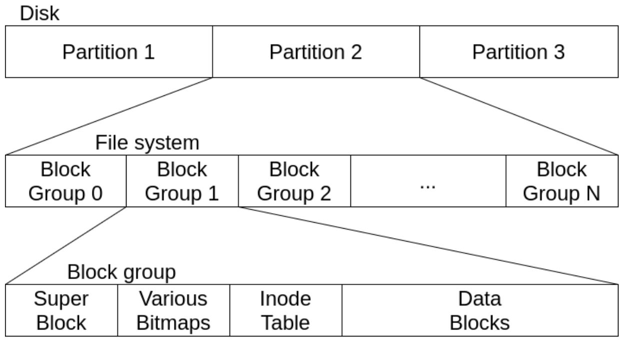


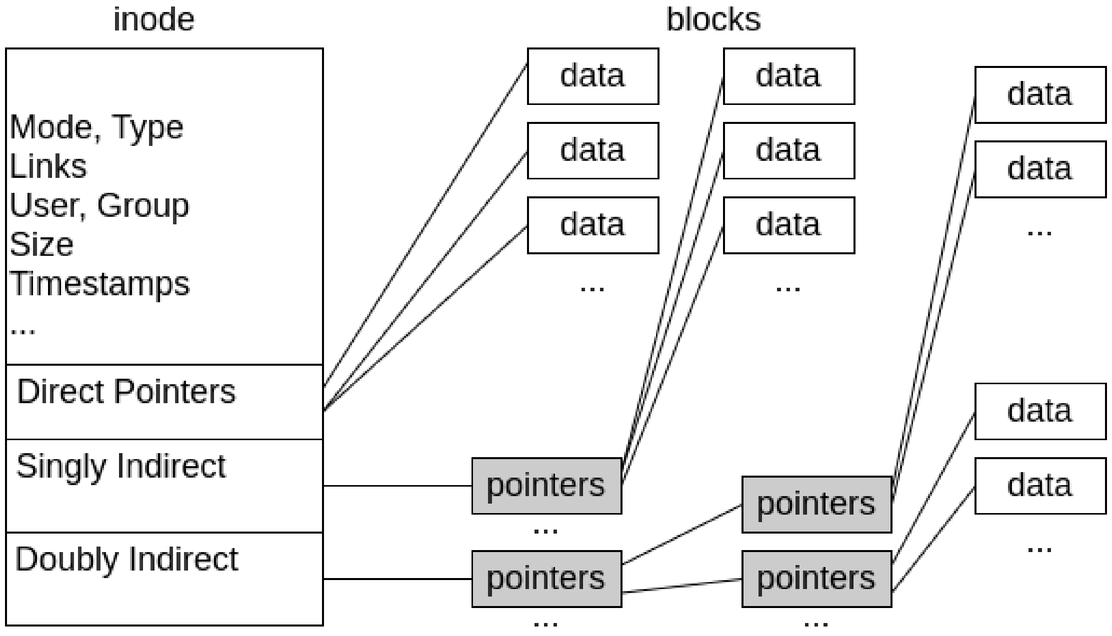
Inode Structure

- Disks can be partitioned with different file systems.
- Disks can be grouped together to one file system.
- Superblock stores general file system metadata.
- File metadata is stored in inode structure.
  - Record for metadata
  - Information to data blocks
  - Each inode within a file system has an unique number.
- Number of used inodes == number of files.
- No free space for inodes → ENOSPC.
- How inodes are allocated depends on file system.

- Hint: Check for free inodes: *df -i*



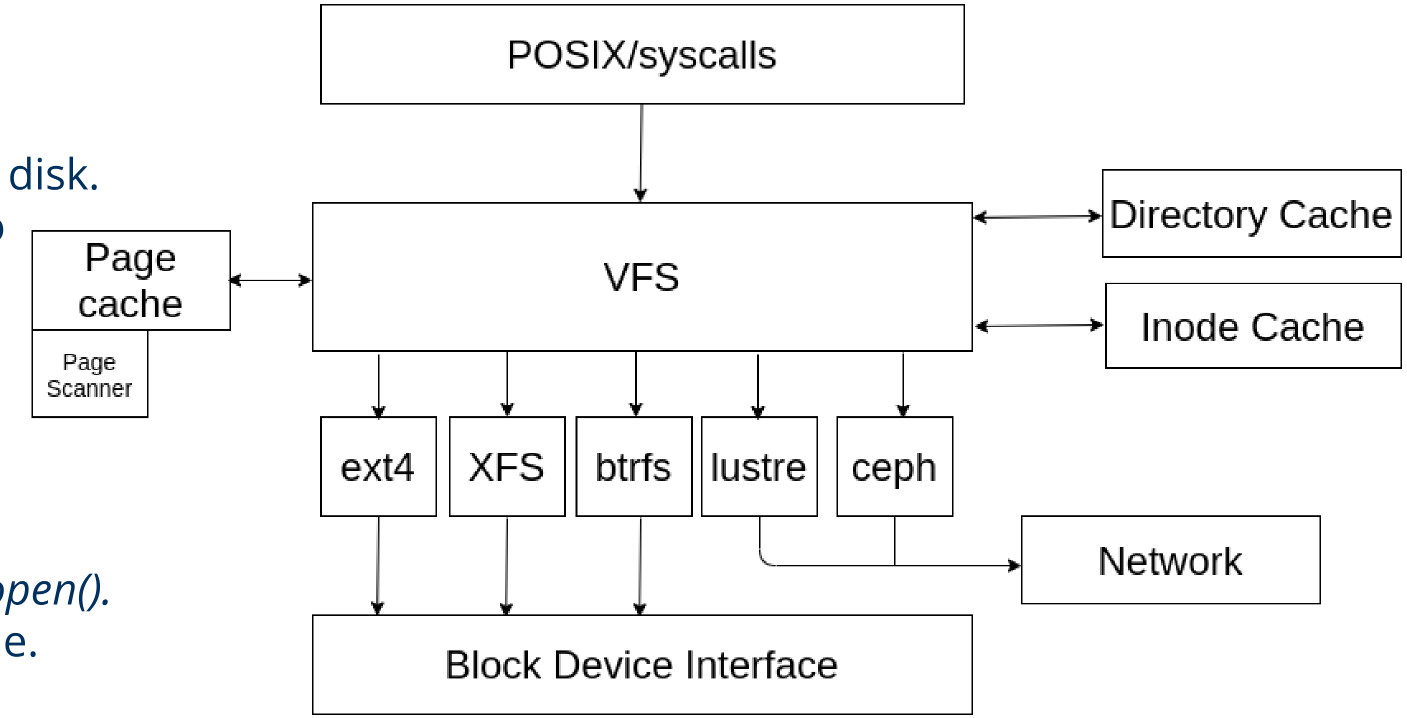




[1] Mathur, Avantika, et al. "The new ext4 filesystem: current status and future plans." Proceedings of the Linux symposium. Vol. 2. 2007.

Linux file system caches

- Multiple caches boosts I/O performance locally.
- Page Cache
  - Caches file system pages.
  - Dynamically sized
  - Modified pages „dirty“ are written back to disk.
  - Flushed after: interval, (f)sync(), dirty_ratio
- Directory Cache (dentry cache)
  - Remembers mappings from directories.
  - Improves performance for path lookups.
  - Dynamically sized
- Inode Cache
  - Frequently used Inodes
  - Improves performance e.g. for *stat()* and *open().*
  - Most lookups will be done via dentry cache.



[1] Gregg, Brendan. Systems performance: enterprise and the cloud. Pearson Education, 2014.

I/O Metrics

- **Bandwidth (GB/s)** – how many data can be moved within one second.
- **Operations (IOP/s)** – how many I/O operations can be done within one second.

- Both metrics depend on the latencies of involved components.

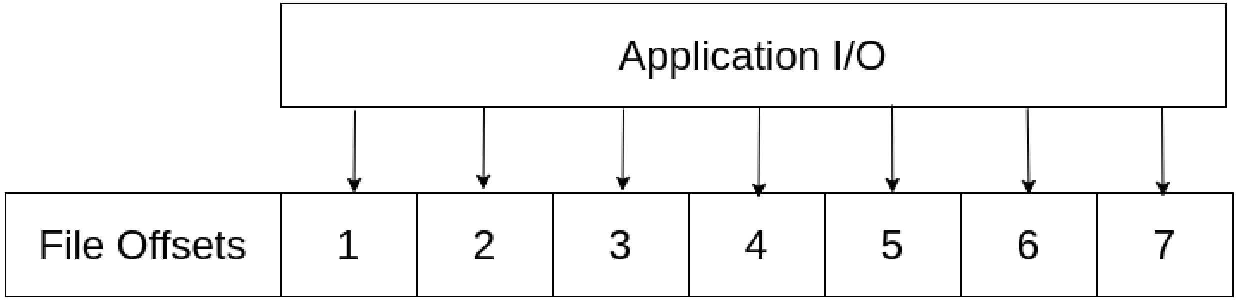
Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Table 2.2 Example Time Scale of System Latencies

[1] Gregg, Brendan. Systems performance: enterprise and the cloud. Pearson Education, 2014.

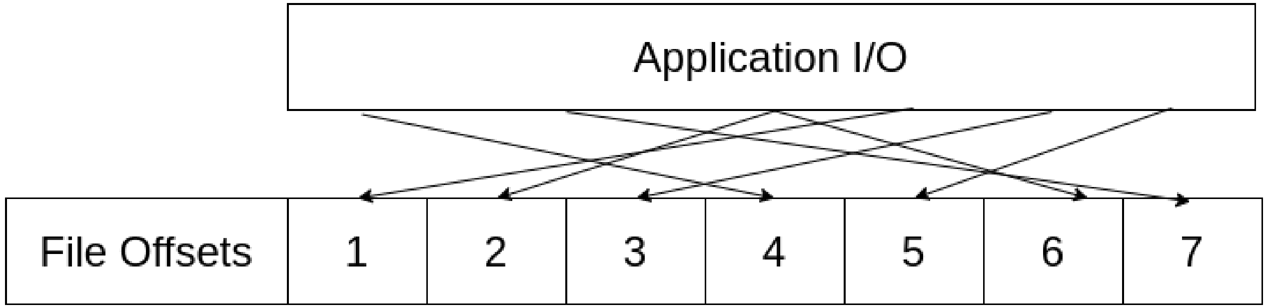
I/O-Pattern

- Different I/O Patterns depending on file offset.
- File systems may try to prefetch data.
- **Sequential I/O:** next I/O begins at the end of the previous I/O.
- **Random I/O:** no apparent relationship between I/O, offsets changes randomly.



Sequential I/O

POSIX defines a File offset as:
"The byte position in the file where the next I/O operation begins. Each open file description associated with a regular file, block special file, or directory has a file offset."



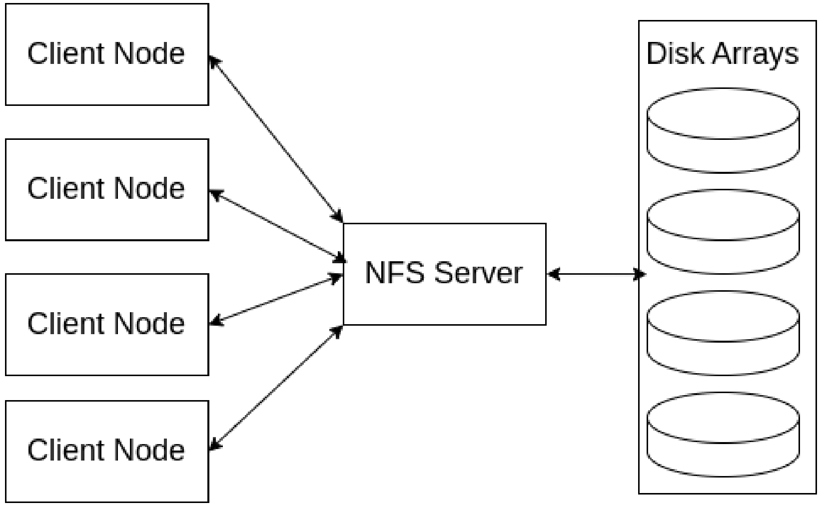
Random I/O

Parallel file systems

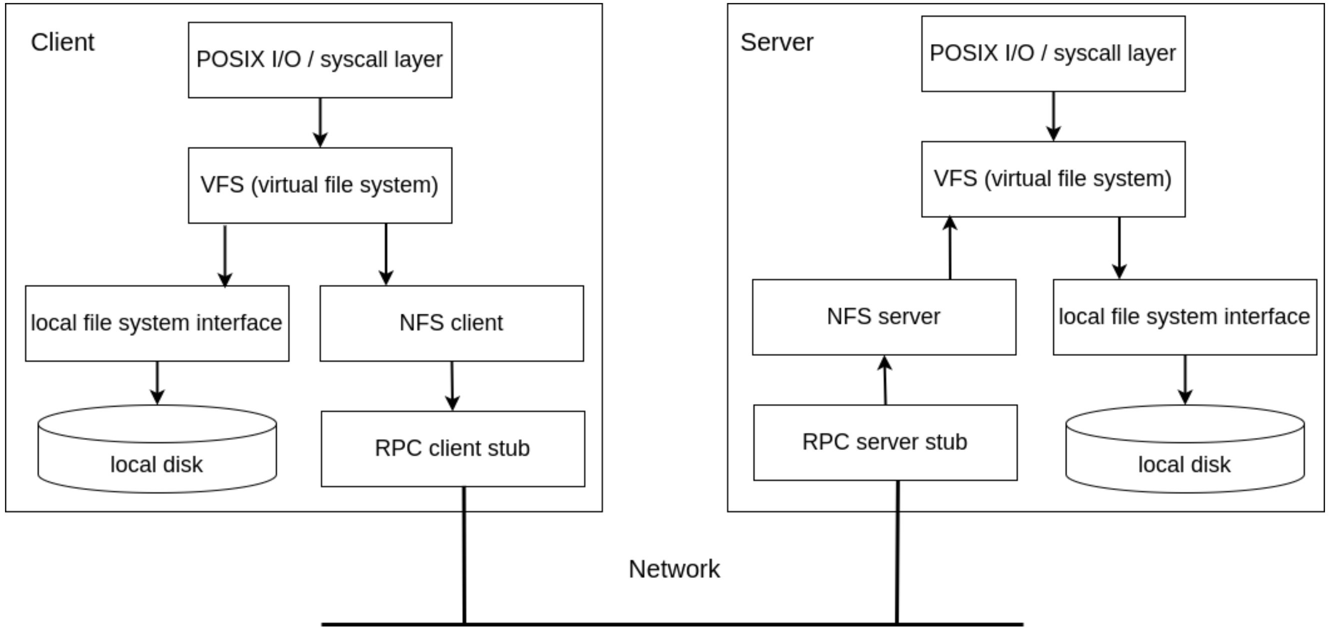
Network file systems and parallel file systems

- Network file systems are file systems attached to clients via a network.
- Network file systems provides access to one or more clients who might not have direct access to the disk.
- Maintains a globally shared namespace for data (single view).
- Transparent: files accessed over the network can be treated the same as files on local disk by programs and users.
- Network file systems maintain metadata and data on a single server.
- Network file systems e.g. NFS are not designed for parallel access on the same files.

Network file system: Architecture and I/O Stack



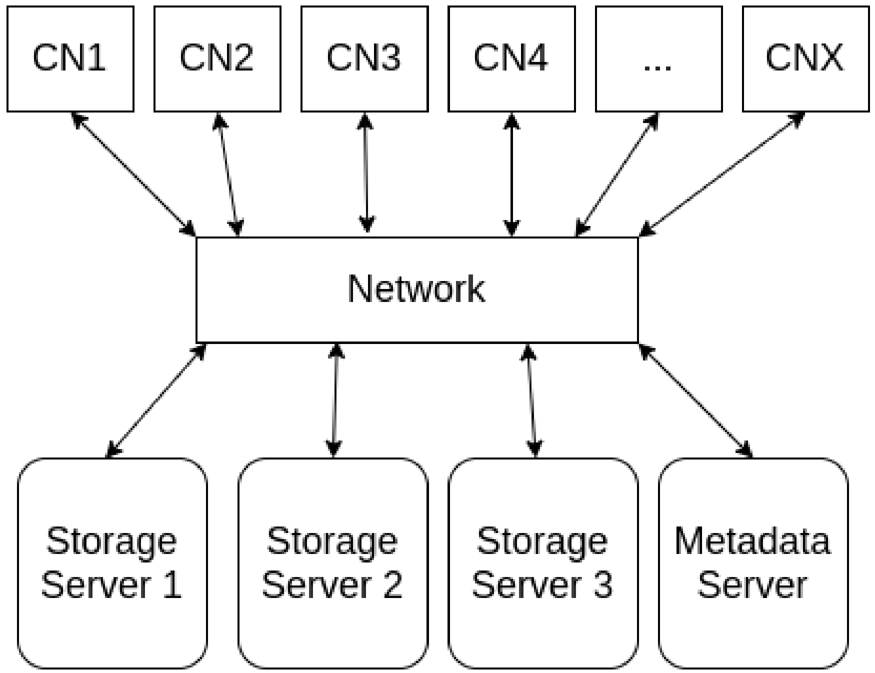
Single-Server architecture of a network file system



Example of an I/O Stack for network file systems

Parallel file systems

- Support for parallel access to files.
- Metadata and data is seperated on different servers.
- „stripes“ data accross multiple disks/server to utilize parallel bandwidth.
- Focus on concurrent, independent access.
- Using Remote-Direct-Memory-Access (RDMA) for data access.

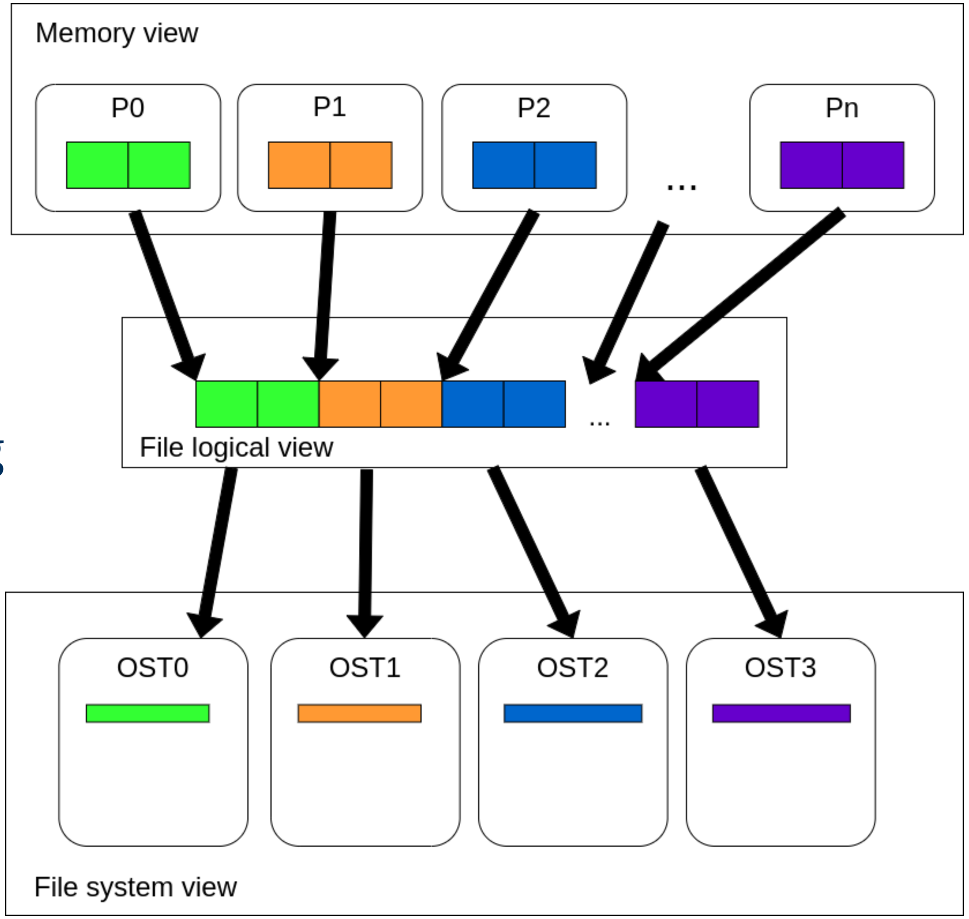


Parallel file system Glossary

Abbreviation	Meaning
MDS	Metadata server
MDT	Metadata target
MGS	Management server
MGT	Management target
OSS	Object storage server
OST	Object storage target – Typically the block device where data chunks are stored.
Chunk	Striped piece of data on an OST (part of a file).

Striping

- A single file may be split into multiple chunks.
- A chunk is then striped on one or more OSTs.
- Advantages:
  - An increase in the bandwidth available when accessing the file.
  - An increase in the available disk space for storing the file.
- Disadvantages:
  - Increased overhead due to network operations and server contention.



Most parallel file systems allows user to specify the striping policy for each file or directory of files.

Striping Considerations

Using more OSTS does not increase write performance.

- **Single writer**
  - Unable to take advantage of file system parallelism.
  - Access to multiple disks adds overhead which hurts performance.
- **File per process**
  - Performance increases as the number of processes/files until OST and Metadata contention hinder performance improvements.
- Best performance when the I/O operation and stripe size are similar.
- Larger I/O and matching stipe sizes may improve performance (reduces the latency of I/O op.).
- If each OST is accessed by every process → OST contention, better perf. If each OST is accessed by only one process.

Lustre

Management Service:

- Provide registry of all components.
- Store configuration information.
- Not involved in I/O.

Metadata Service:

- Provide file system namespace.
- Storing inodes for the file system.

Object Storage Service:

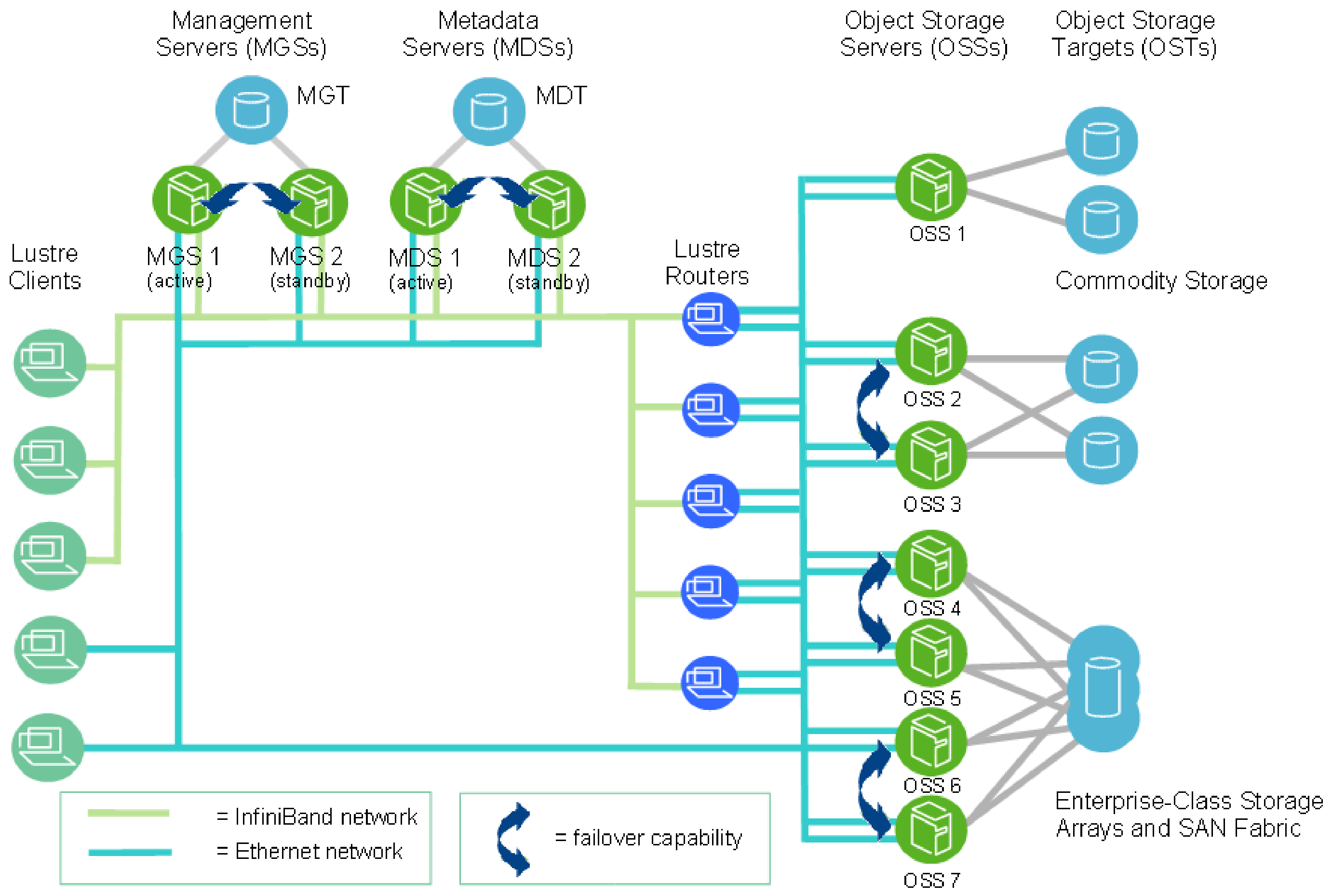
- Provide bulk storage data.
- Files can be written across multipe targets.
- OSSs are the primary scalable service unit.

Clients:

- Mount lustre file system using LNet protocol.
- Presents POSIX-compliant FS to the OS.

Network:

- Lustre network I/O using LNet protocol.
- LNet can aggregate I/O accross independent interfaces.
- LNet routers provide a gateway between different LNet networks.

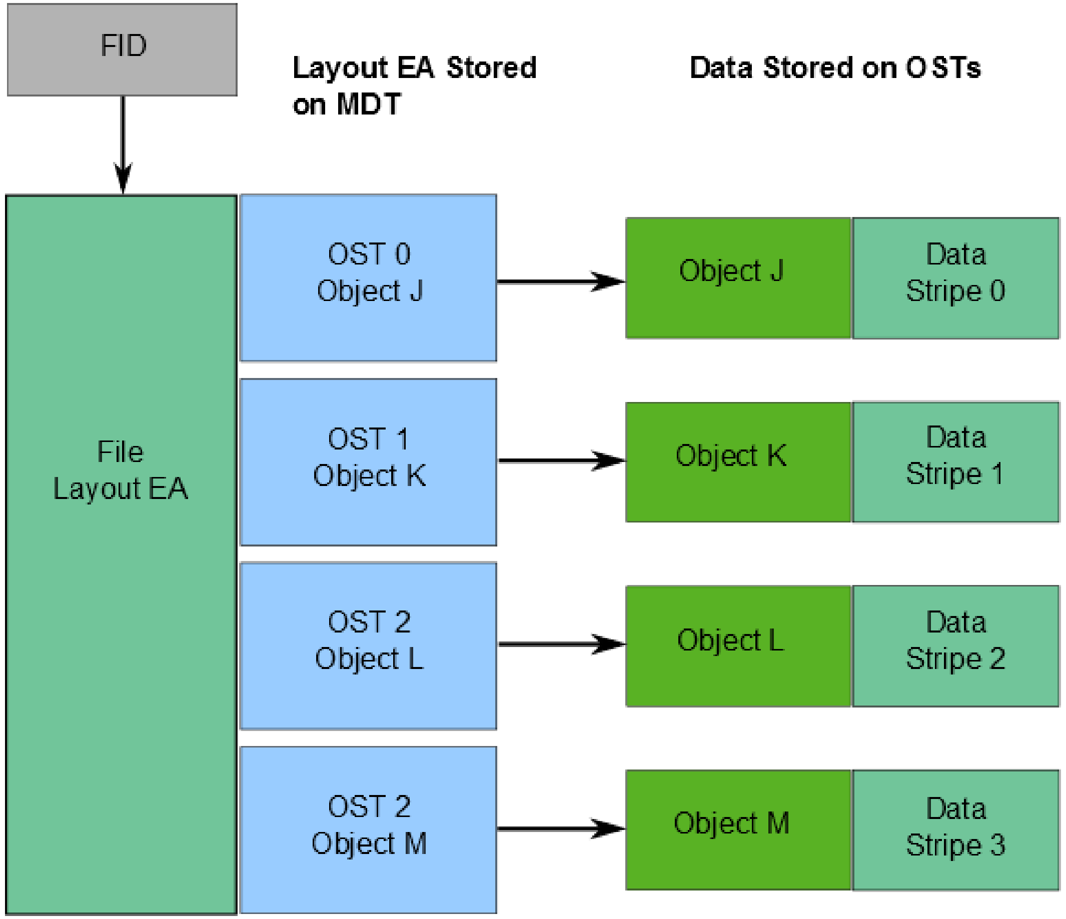


https://doc.lustre.org/figures/Scaled_Cluster.png

[1] https://wiki.lustre.org

Metadata

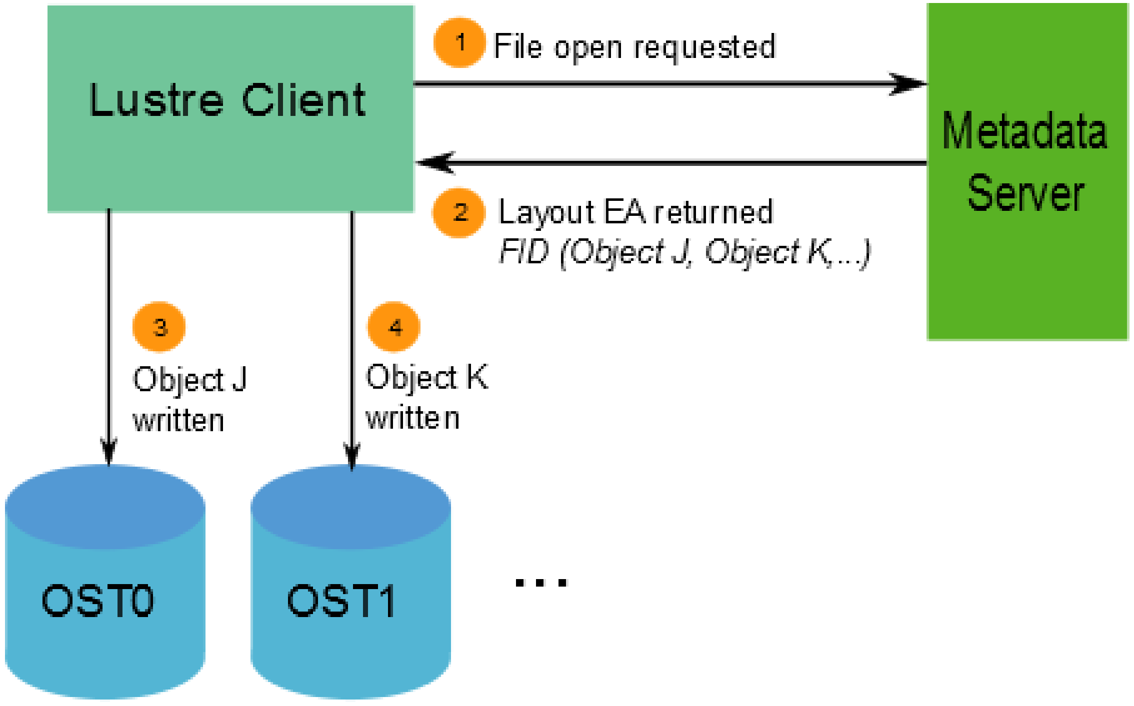
- Metadata Server (MDS) stores:
  - File Metadata (size, owner, permissions, …)
  - File layout information.
  - How data is distributed over OSTs.
- Client can query by File-Identification (FID).



https://doc.lustre.org/figures/Metadata_File.png

File I/O

- Clients must talk to **both** MDS and OSS servers.
- Opening files, listing directories, … go to MDS.
- File I/O goes directly to one or more OSSs.



1. Client ask MDS for file information.
2. MDS tells the client layout and object information of the file.
3. Client can directly read/write to OST.
4. Client can directly read/write to OST.

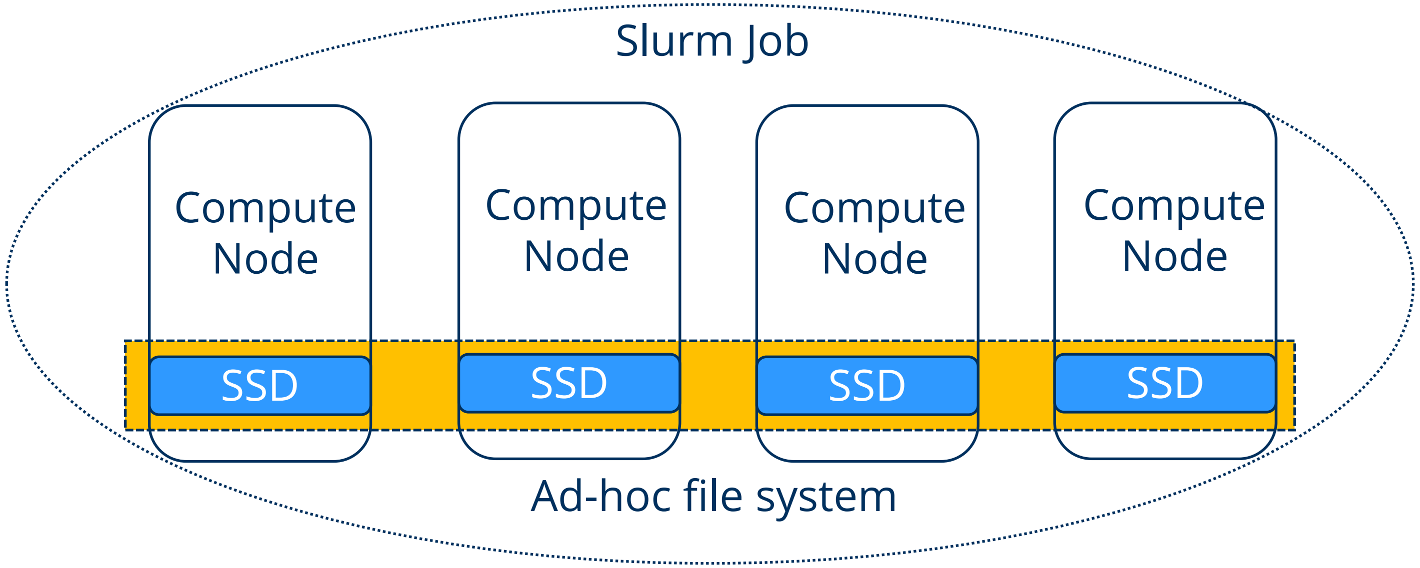
https://doc.lustre.org/figures/File_Write.png

Working with lustre (lfs) commands

- Show available lfs commands.
  - *lfs –list-commands*
- Show capacity information of OSTs of a file system
  - *lfs df /lustre/scratch2*
- Query stripe/layout information of a file or directory.
  - *lfs getstripe <file|dir>*
- Setting the stripe layout for a file or directory.
  - *lfs setstripe <file|dir> -s <bytes/OST> -o <start OST> -c <#OSTs>*
  - E.g. to stripe across two OSTs with 4MB stripes, you would call:
  - *lfs setstripe myfile –s 4m –o -1 –c 2*

Ad-hoc file systems for HPC*

- Isolation of challenging I/O from PFS and the Network
- Using node local fast storages (e.g. SSDs, NVRAM, …)
- Provide a global file system view in a shared namespace
- Job-temporal life time → requires Data Staging



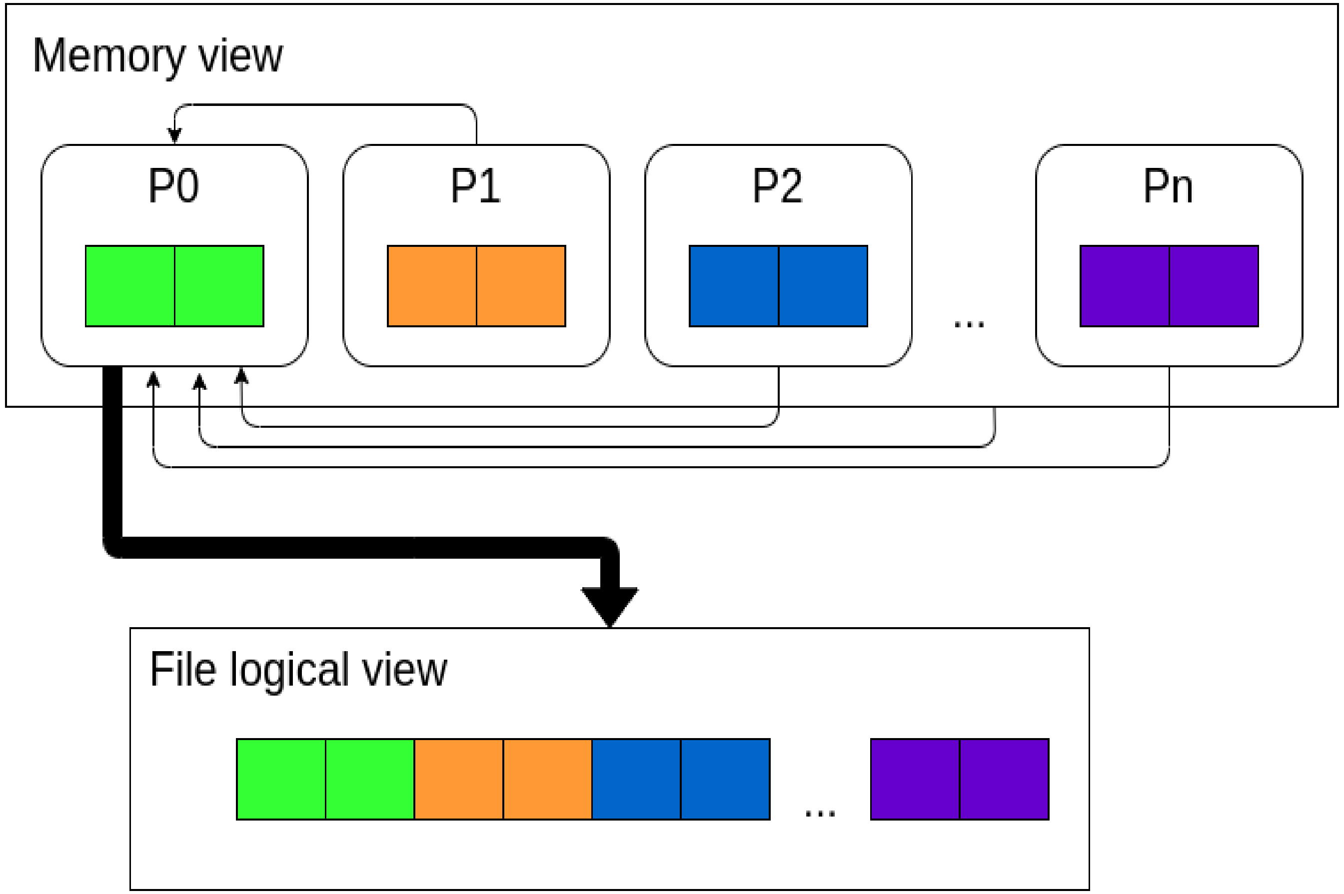
* Brinkmann, André, Mohror, Kathryn, Yu, Weikuan, Carns, Philip, Cortes, Toni, Klasky, Scott A., Miranda, Alberto, Pfreundt, Franz-Josef, Ross, Robert B., and Vef, Marc-André. Ad Hoc File Systems for High-Performance Computing. United States: N. p., 2020. Web. https://doi.org/10.1007/s11390-020-9801-1.

Parallel I/O

Serial I/O

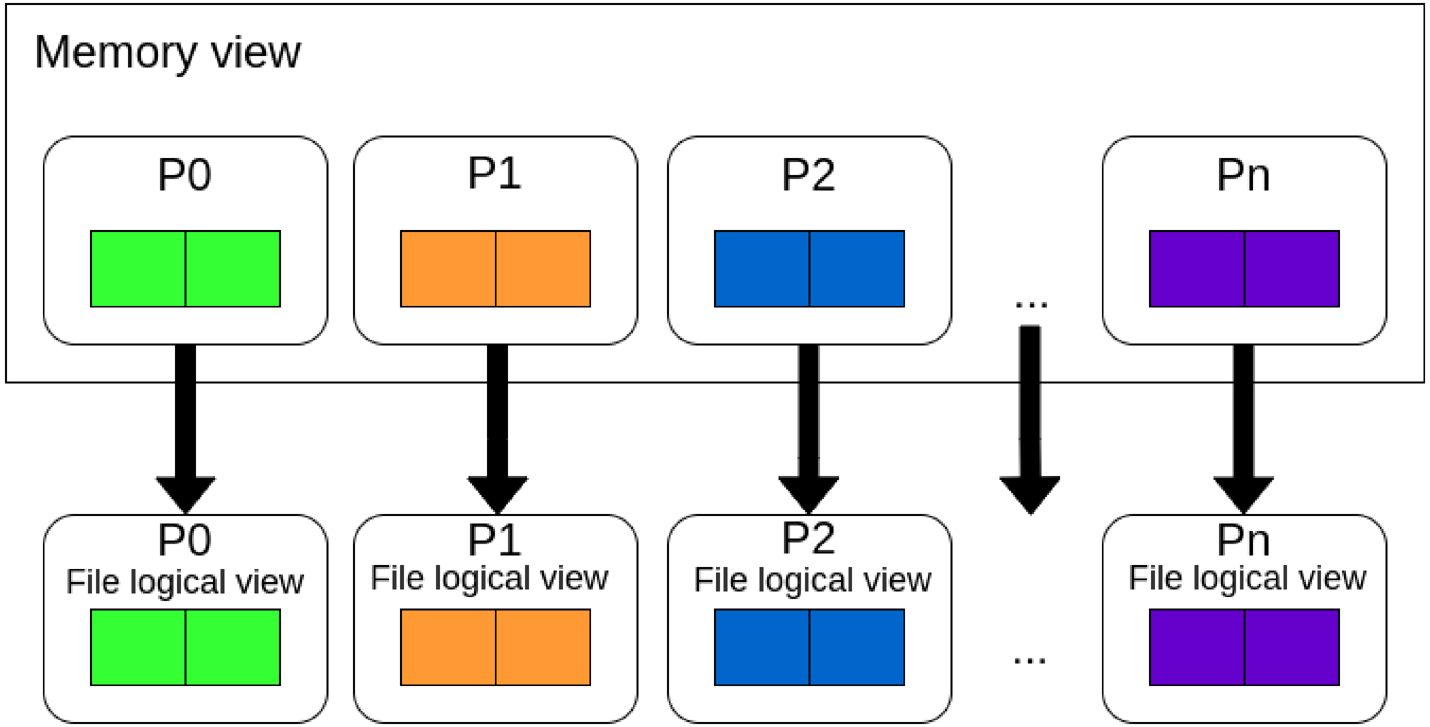
- First collective call to gather the data on one Process.
- Then this process writes the data to a single file.
- Memory of a single node might be a limitation.
- No utilization of parallel bandwidth.

- Simple solution, easy to manage but does not scale
  - Time increases linearly with amount of data
  - Time increases with number of processes



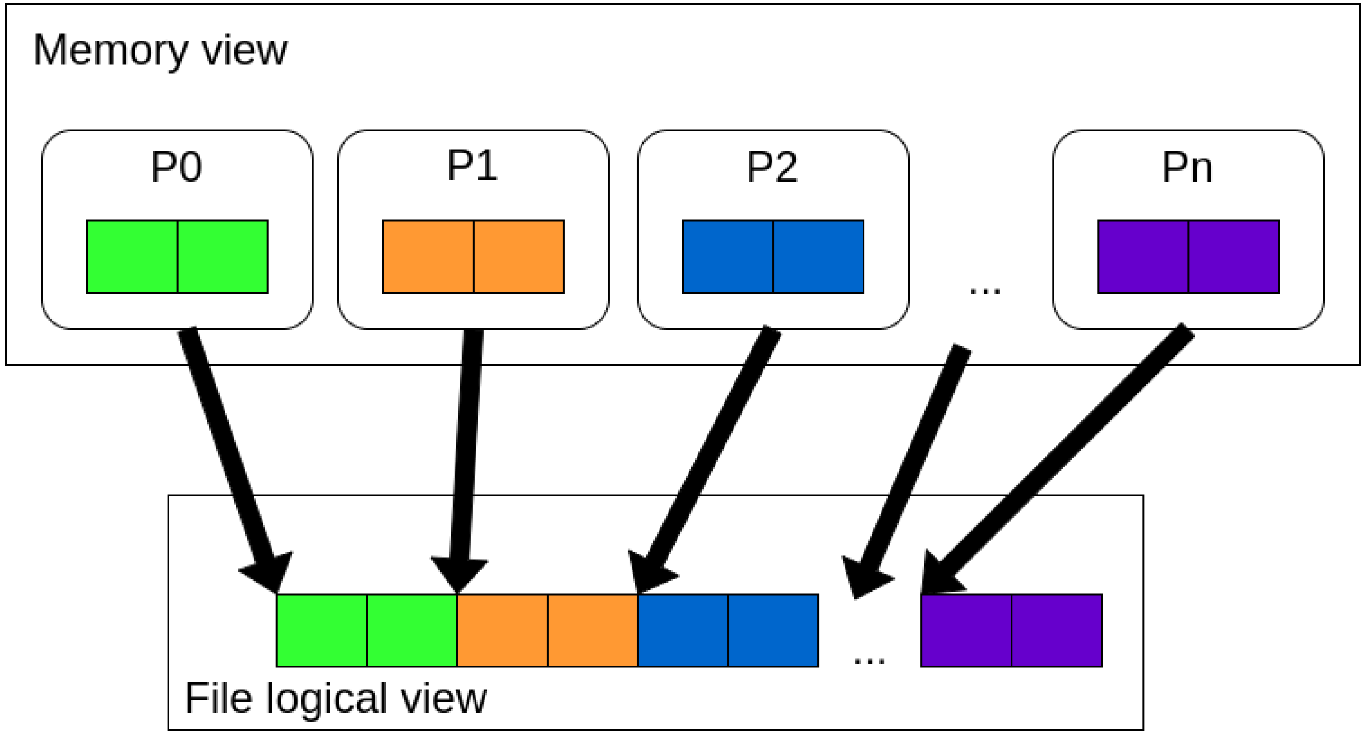
File-per-Process

- Each process writes its own file.
- A single distributed data is spread out in different files.
- Files not portable
- Multiple output files can result in more prost processing work.
- **Advantages:**
  - Easy to implement.
  - Can utilize parallel bandwidth.
- **Disadvantages:**
  - Number of files creates bottleneck with metadata operations.
  - Number of simultaneous disk accesses creates contention for file system resources.



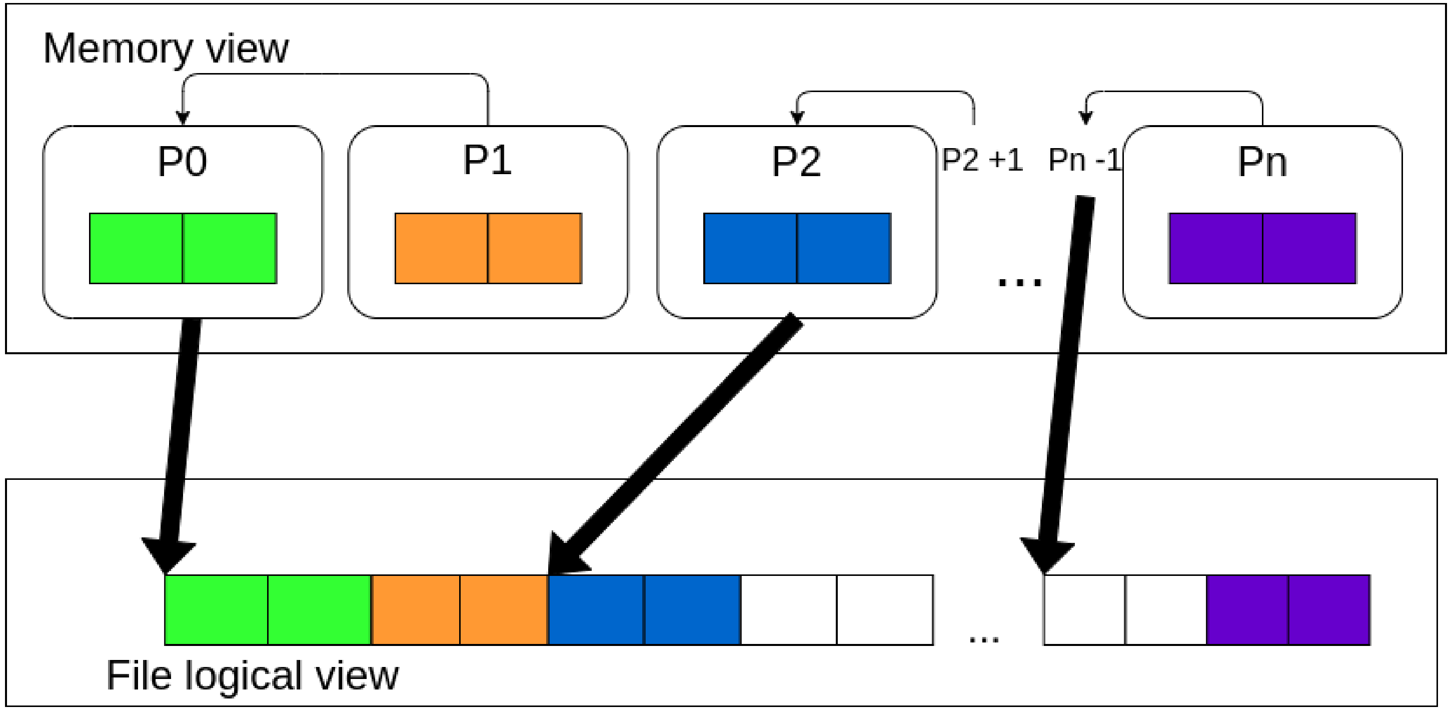
Shared-file (single file, multiple writers)

- Each process performs I/O to a single file which is shared.
- Data layout within the shared file is important.
- At large process counts contention can build for file system resources.



Shared-file (single file, collective writers)

- Subset of processes which perform I/O.
- Aggregation of a group of processes data.
- Serializes I/O in group.

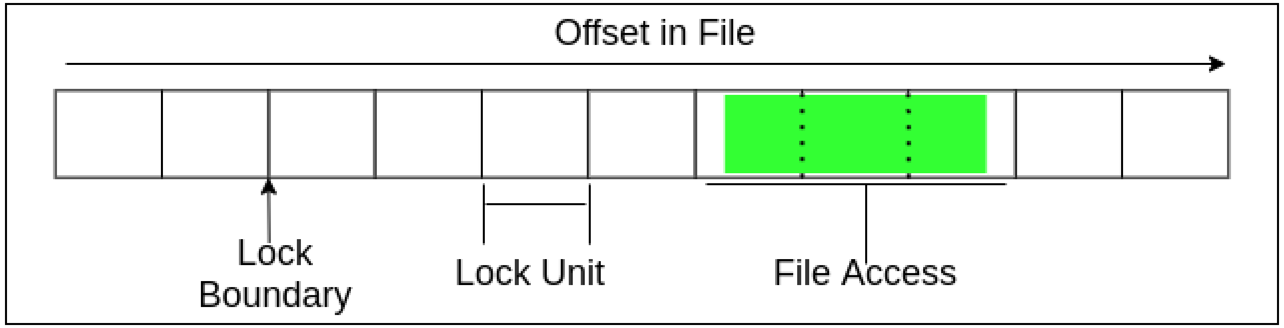


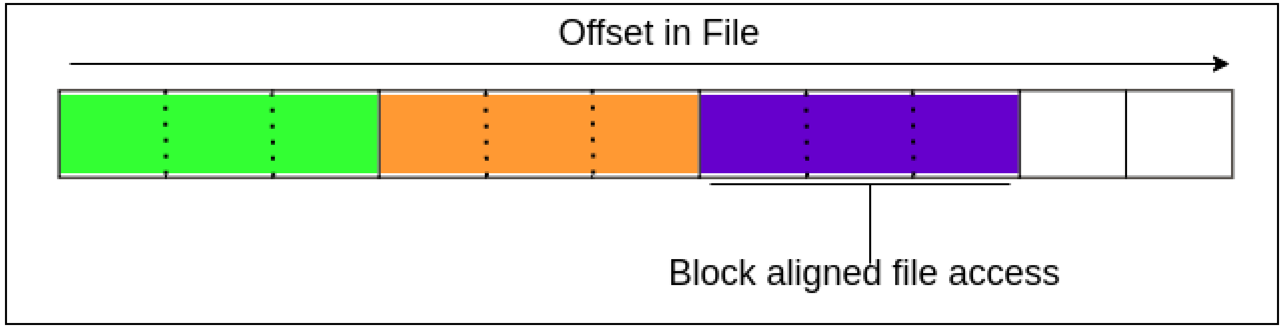
How to choose the right number of I/O processes?

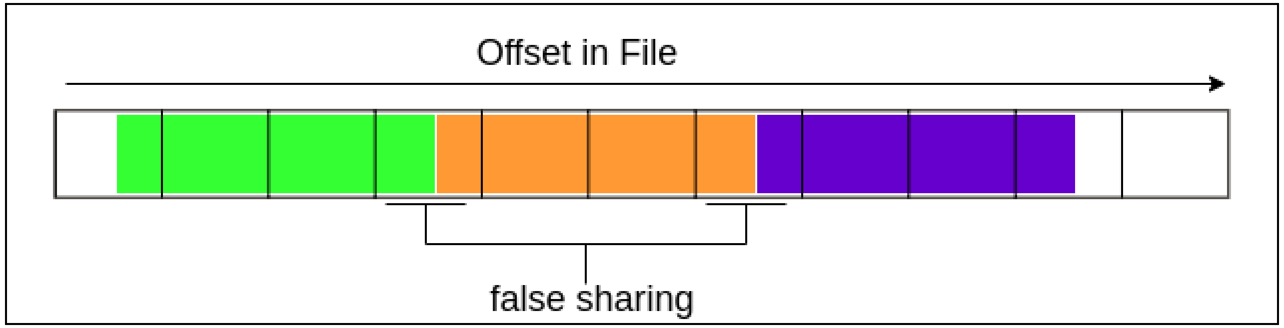
Need to saturate memory bandwidth within node.

Managing Concurrent Access

- Files are treated as random global shared memory regions.
- Locks are used to manage concurrent access.
- Unit boundaries are dictated by the storage system regardless of access pattern.
- Clients will obtain locks on units before I/O occurs.
- Enables caching on clients as well – as long as client has a lock, it knows its cached data is valid.
- Locks are reclaimed from clients when other desire access.







MPI-IO

- Provides a low-level interface to carrying out parallel I/O.
- Facilitate concurrent access by groups of processes.

MPI-IO can be done in 2 basic ways:

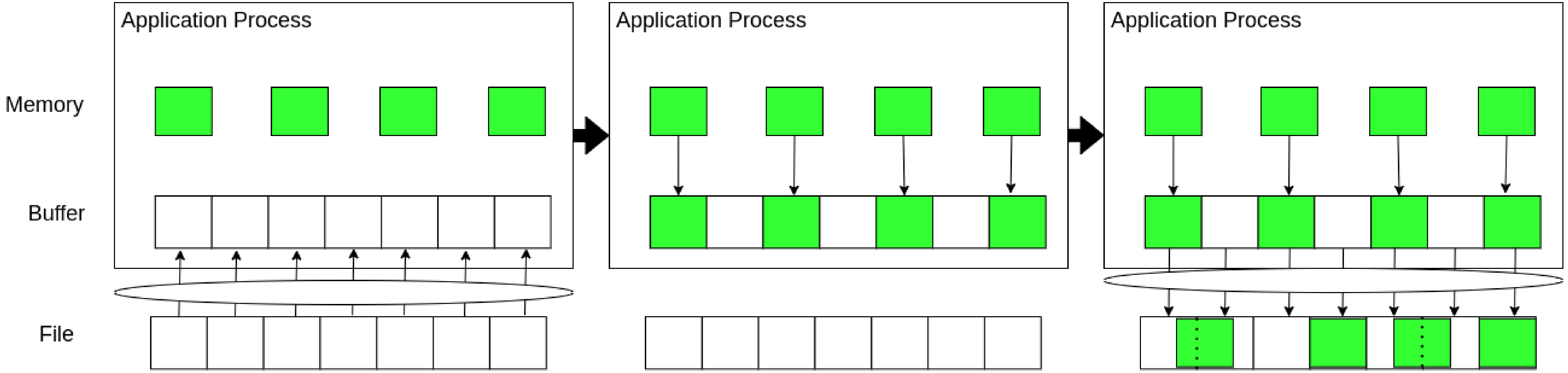
- **Independent MPI-I/O:**
  - Each MPI rank is handling the I/O independently using non-collective calls like *MPI_File_write()* and *MPI_File_read().*
  - Similar to POSIX-I/O, but supports derived datatypes and thus noncontigous data and nonuniform strides and can take advantage of MPI_Hints.
- **Collective MPI-I/O:**
  - When doing collective I/O all MPI tasks participating in I/O has to call the same routines. E.g. *MPI_File_write_all()/MPI_File_read_all().*
  - This allows the MPI library to do I/O optimization.

Collective I/O with MPI-I/O

- All processes specified in the group by the communicator passed to *MPI_File_open()* will call this function.
- Each process specifies only its own access information.
- MPI-I/O library is given a lot of information.
  - Collection of processes reading or writing data.
  - Structured description of the regions.
- When writing in collective mode, the MPI library carries out a number of optimizations
  - Using fewer processes to actually do the writing – typically one per node.
  - It aggregates data in appropriate chunks before writing.
- MPI-IO Hints can be given to improve performance by supplying more information to the library. This information can provide the link between application and file system.

Data Sieving

- Technique to address I/O latency by combining operations.
- Larger read and write requests → higher bandwidth, lower latency!
- Doing extra I/O to avoid contention.



Step 1: Read region to be modified into intermeidate buffer.

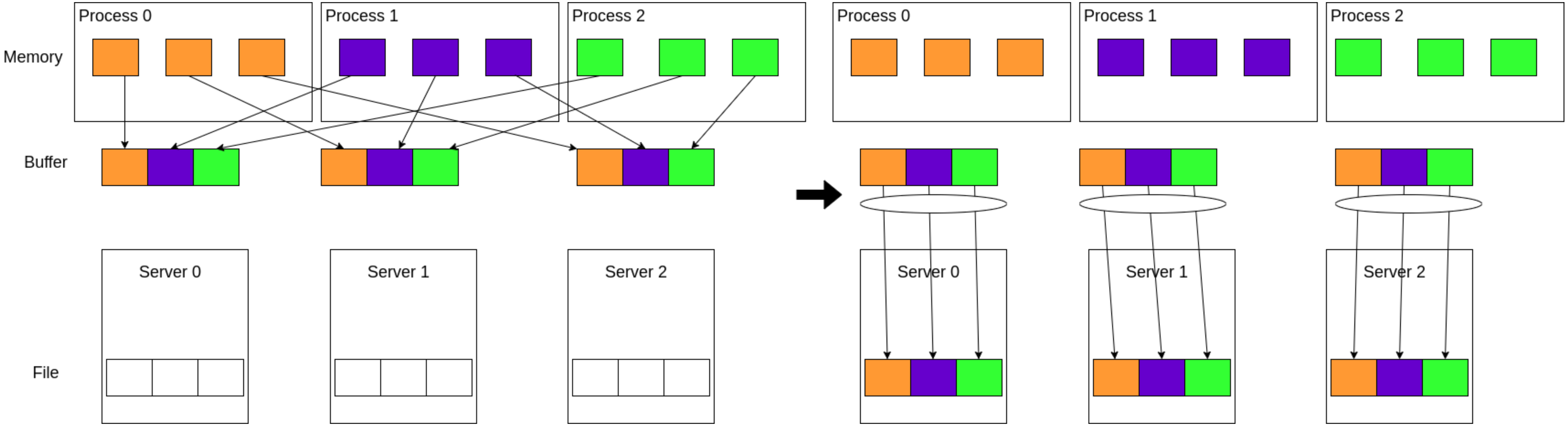
Step 2: Elements to be written to file are replaced in the buffer.

Step 3: Entire region is written back to file with a single write.

Two-Phase I/O

[1] Liao, Wei-keng, and Alok Choudhary. "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols." SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. IEEE, 2008.

- Reorder data among processes to avoid lock contention.
- Two-Phase I/O splits I/O into a data reorganization phase and interaction with the file system.
- Data exchanged between processes to match file layout.



Step 1: Data are exchanged between processes on organization of data in file.

Step 2: Data are written to file, with large writes and avoid contention.

Performance Analysis

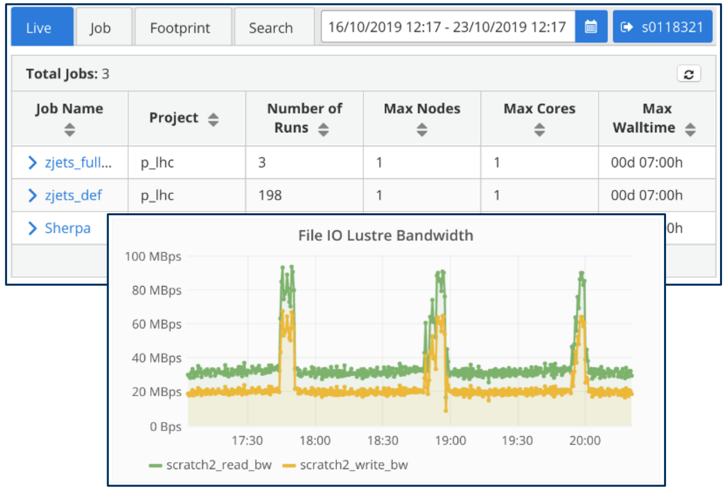
Performance Analysis

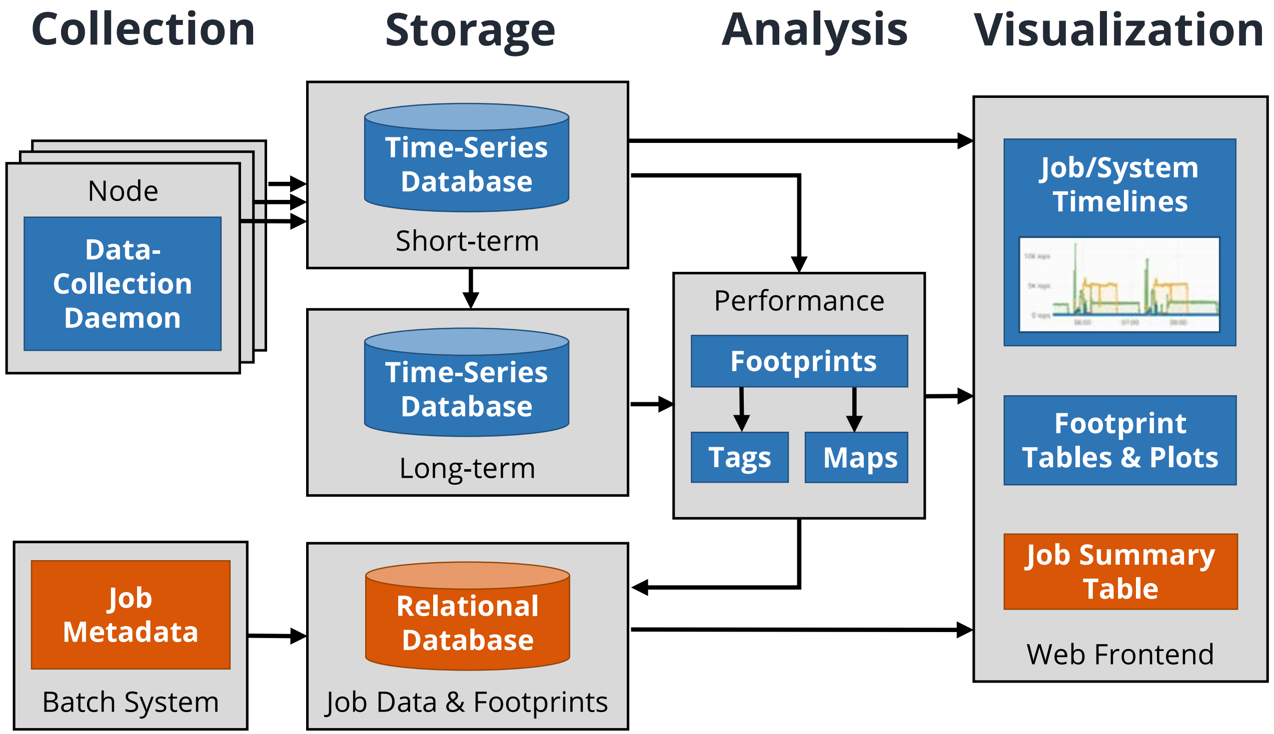
- I/O Performance depends on many factors.
  - Access pattern – Application dependent
  - Scale / Volume – Number of processes, data volume
  - File system – shared medium
  - Disk and network type and speed – Hardware dependent
  - Network topology – platform dependent + shared medium

PIKA: Center-Wide and Job-Aware Cluster Monitoring

- Non-intrusive data acquisition on all cluster nodes.
- Continous data collection.

- Web frontend for live and post-mortem visualization.





Software project available at https://gitlab.hrz.tu-chemnitz.de/pika

PIKA Data Collection and Metrics

- Uses collectd collection daemon[1]
- One collector/plugin for each metric source.
- CPU Counters collected with LIKWID[2].
- All other metrics are collected every 30s.
- Lustre collector: read/write bandwidth and metadata IOPS.

Metric	Proposed Name	Data Source	Hardware Unit
CPU			
Usage	cpu_usage	/proc/stat	hardware thread
Main memory utilization	mem_used	/proc/meminfo	node
IPC	ipc	LIKWID	hardware thread
FLOPS (SP-normalized)	flops_any	LIKWID	hardware thread
Main memory bandwidth	mem_bw	LIKWID	CPU/socket
Power consumption	rapl_power	LIKWID	CPU/socket
Network bandwidth	net_bw		
Infiniband bandwidth	ib_bw	/sys/class/infiniband/...	Infiniband device
Ethernet bandwidth	eth_bw	/sys/class/net/eth*/...	ethernet device
I/O bandwidth & metadata			
Local disk	read_bw, write_bw & read_ops, write_ops	/proc/diskstats	disk
Lustre	read_bw, write_bw & open, close, create, seek, fsync, read_requests, write_requests	/proc/fs/lustre/llite/*/stats	Lustre instance
GPU			
Usage	gpu_used		
Memory Utilization	gpu_mem_used	NVML	GPU
Power Consumption	gpu_power		
Temperature	gpu_temperature		

[1] https://github.com/collectd/collectd

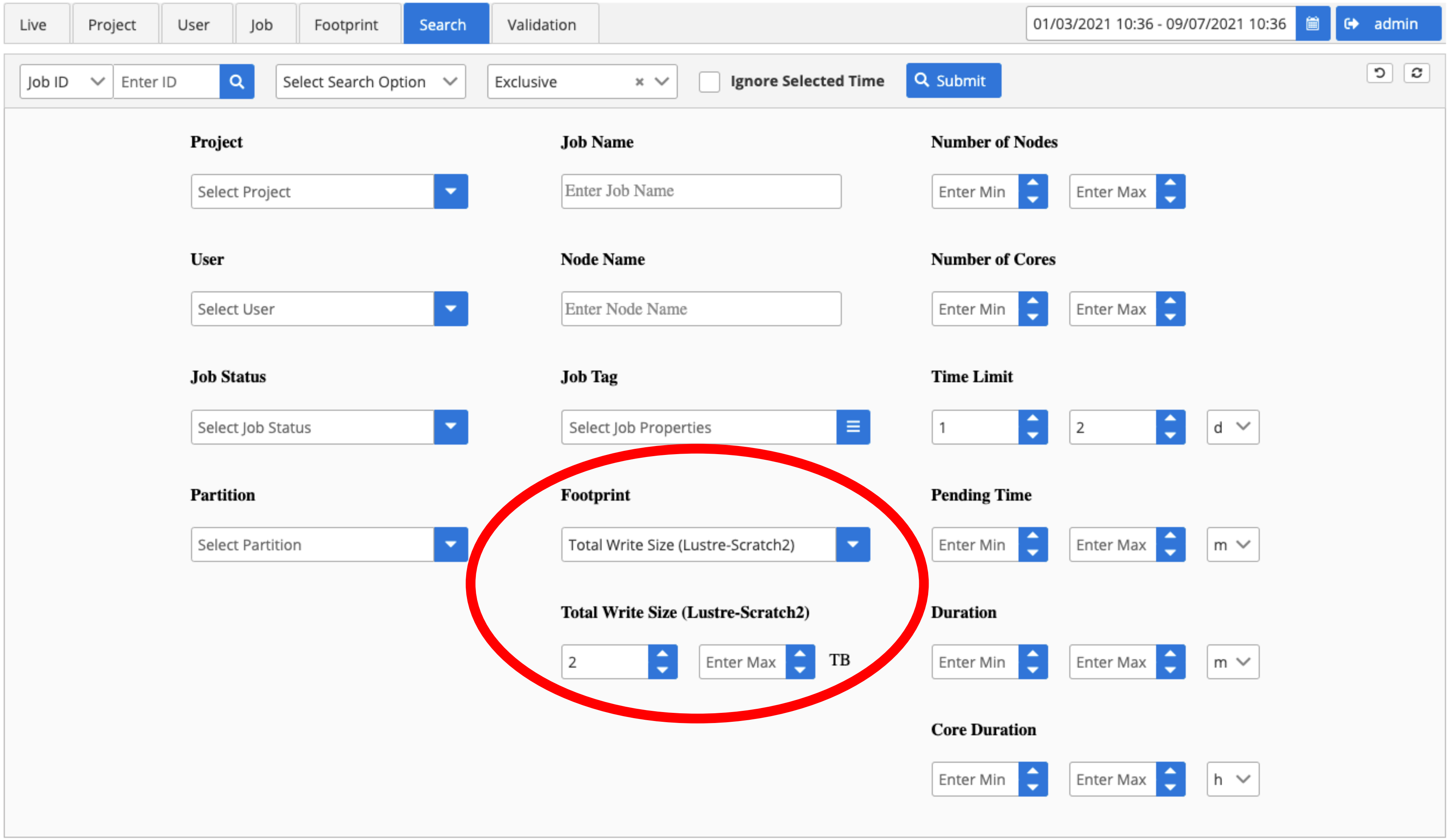
[2] https://github.com/RRZE-HPC/likwid

PIKA Job Visualization

Search: JID=20837141 Job ID: 20837141

Project	User	Start	End	State	#Nodes	#Cores	Exclusive	Walltime	Pending	Duration	Core Duration	Used Walltime	Partition
dasupport	soeste	16/11/21 17:09:10	16/11/21 17:21:43	completed	4	96	1	00d 01:30:00h	00d 00:00:00h	00d 00:12:33h	0000y 000d 20:04h	13.94%	haswell64

PIKA Job Footprint Analysis – Search Jobs



PIKA Job Footprint Analysis – Search Jobs

Live | Project | User | Job | Footprint | **Search** | Validation | admin

Sort jobs by largest write size from scratch2.

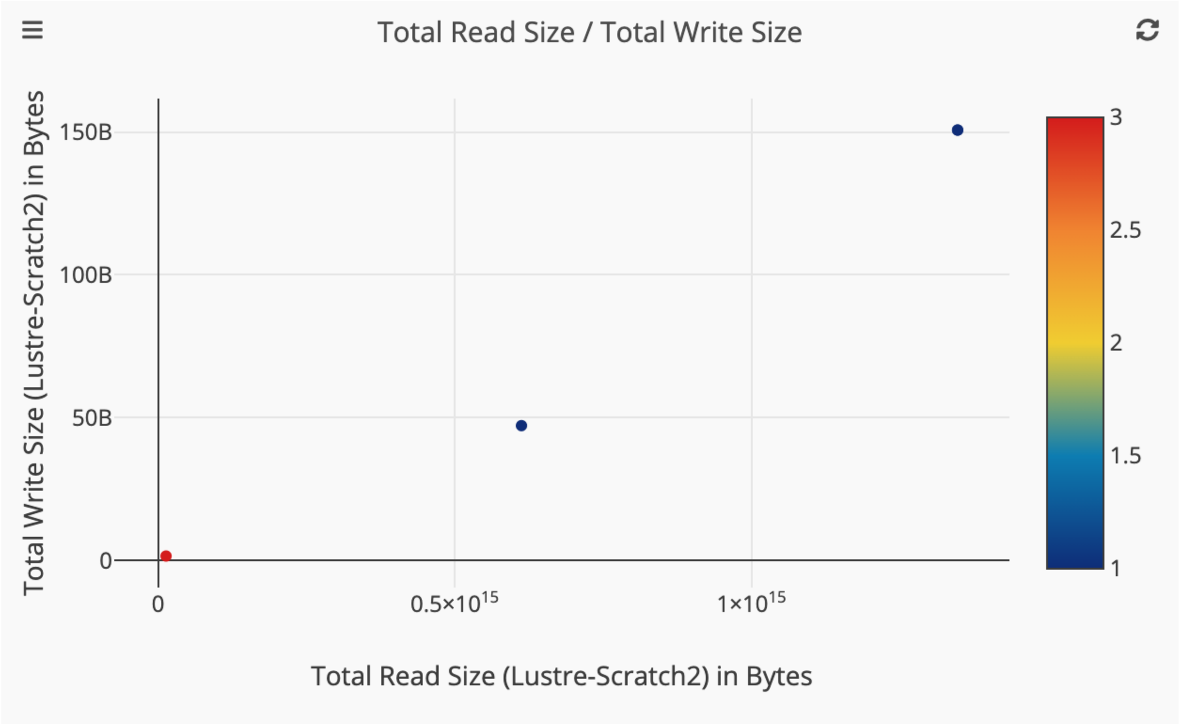
< Search: **Detail** **Total Results:** 47

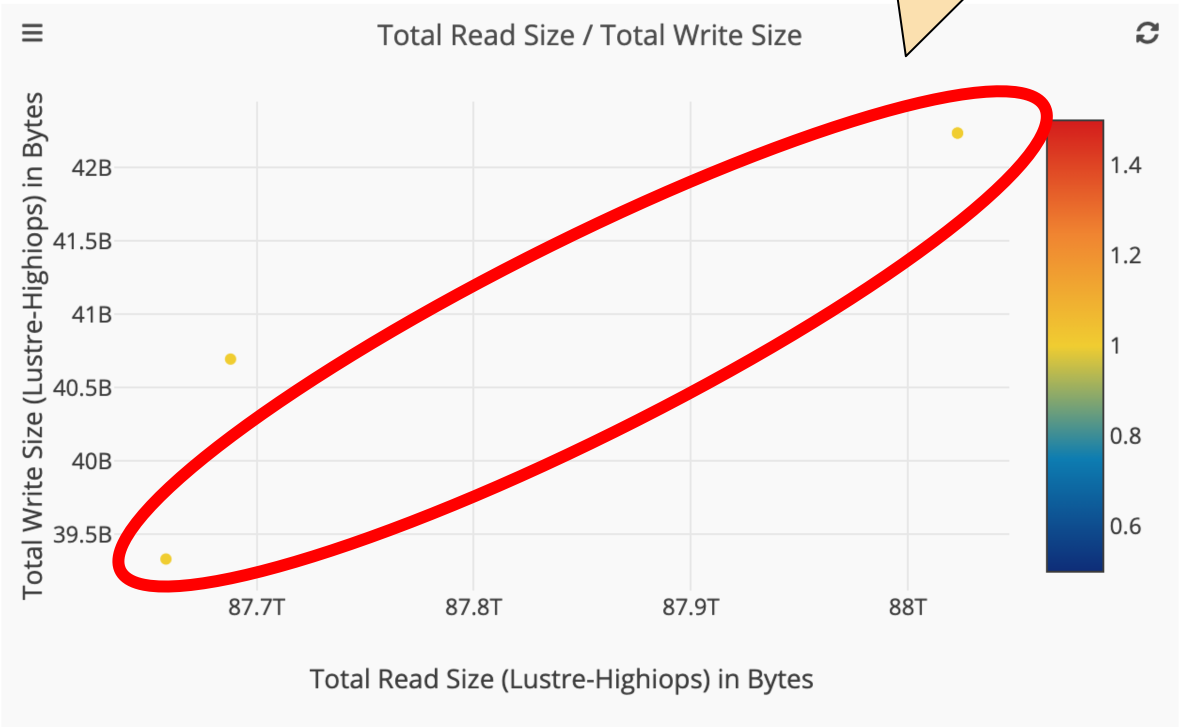
Job ID	Project	User	Job Name	Start	End	State	#Nodes	#Cores	Exclusive	Walltime	Pending	Duration	Core Duration	Used Walltime	Partition	lustre scratch2 write bytes
12919...	p_rna	zwein...	RunC...	29/03/21 14:55:54	30/03/21 01:44:43	compl...	1	128	1	01d 00:00:00h	00d 00:01:10h	00d 10:48:49h	0000y 057d 16:08h	45.06%	romeo	7.4e+12
12352...	p_func...	vankova	1p90_...	19/03/21 23:07:11	21/03/21 00:07:22	timeout	1	24	1	01d 00:00:00h	00d 06:08:10h	01d 01:00:11h	0000y 025d 00:04h	104.18%	haswe...	7.2e+12
12352...	p_func...	vankova	Mst_sc...	19/03/21 19:45:48	20/03/21 13:07:26	compl...	1	24	1	01d 00:00:00h	00d 03:07:15h	00d 17:21:38h	0000y 017d 08:39h	72.34%	haswe...	6.0e+12
12352...	p_func...	vankova	rot-Ms...	19/03/21 23:07:12	20/03/21 16:35:41	compl...	1	24	1	01d 00:00:00h	00d 06:10:35h	00d 17:28:29h	0000y 017d 11:23h	72.81%	haswe...	6.0e+12
12768...	p_func...	vankova	St_sc-...	27/03/21 07:55:59	28/03/21 09:56:24	timeout	1	24	1	01d 00:00:00h	01d 15:40:54h	01d 01:00:25h	0000y 025d 00:10h	104.2%	haswe...	4.6e+12

1 2 3 4 5 | 5

PIKA Job Visualization - Footprints

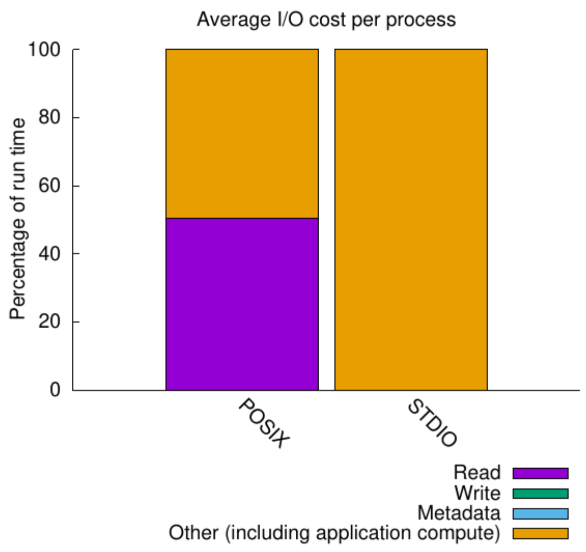
Different runs from the same code.

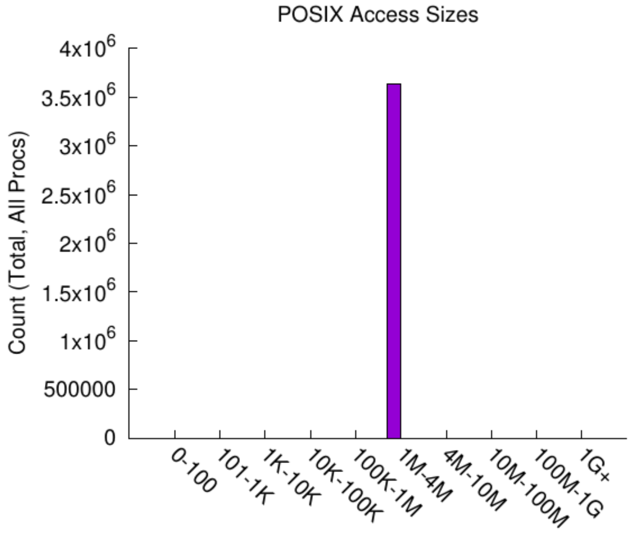


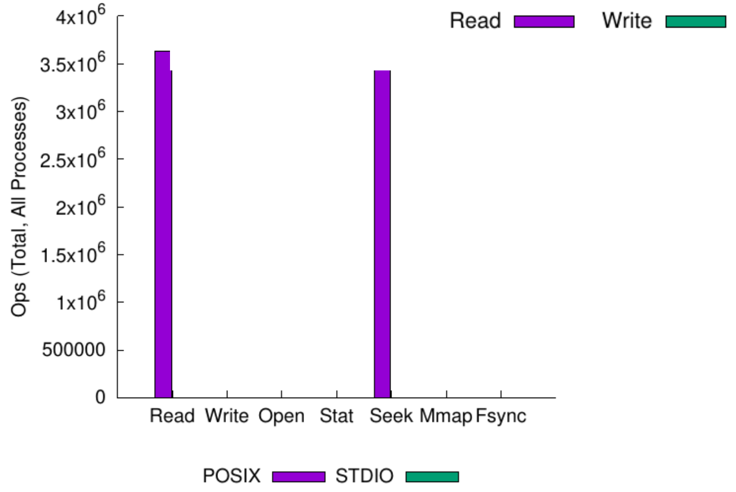


The darshan I/O-Characterization Tool

- Developed at ANL.
- Uses compiler or LD_PRELOAD based intstrumentation.
- Writes a log file with aggregated I/O metrics.
- Post-processing script generates report.
- Since version 3.0 extended trace support with:
  DXT_ENABLE_IO_TRACE=1







Darshan: https://github.com/darshan-hpc/darshan

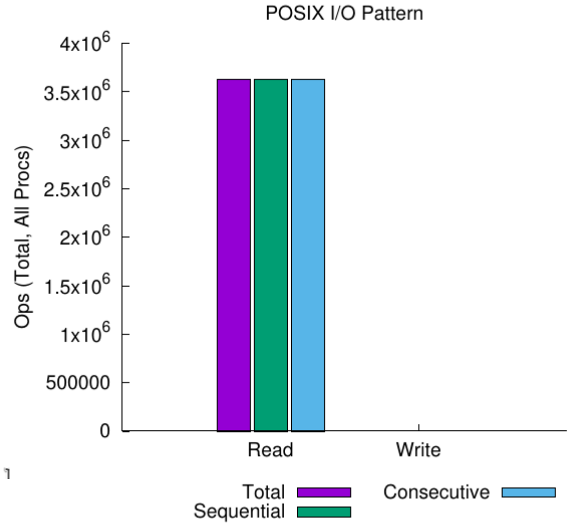
The darshan I/O-Characterization Tool

Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	2097152	3629728

File Count Summary
(estimated by POSIX I/O access offsets)

type	number of files	avg. size	max size
total opened	34	209G	222G
read-only files	33	215G	222G
write-only files	1	2.4K	2.4K
read/write files	0	0	0
created files	1	2.4K	2.4K



sequential: An I/O op issued at an offset greater than where the previous I/O op ended.
consecutive: An I/O op issued at the offset immediately following the end of the previous I/O op.

I/O Recording and Analysis with Score-P and Vampir

- Score-P uses event tracing for data acquisition.
- Applications must be instrumented during compilation.
- Support for multi-layer I/O instrumentation.
- For available layers see: *score-p –io=help*
- Python support.
- OTF2 – Open Trace Format 2.
- Vampir Analysis Tool.
- Provides a lot displays for performance analysis of OTF2 trace files.

Vampir: https://vampir.eu/
Score-P & OTF2: https://score-p.org
Score-P Python: https://github.com/score-p/scorep_binding_python

Scalable performance measurement infrastructure for parallel codes

Open Trace Format 2

Using Score-P for your application?

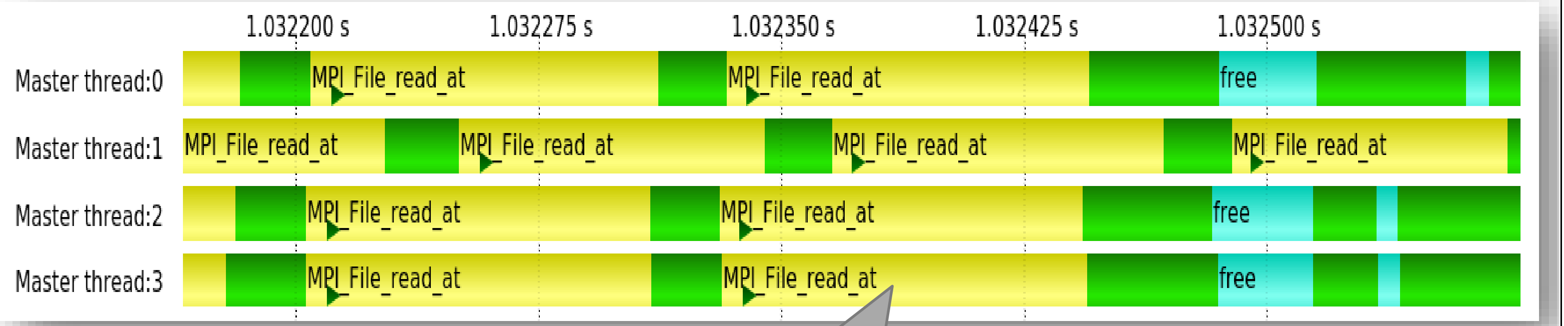
In your makefile:

```
PREP = scorep --dynamic --io=runtime:netcdf --io=runtime:posix
CC = $(PREP) gcc
CFLAGS = -Wall -Wextra

instrumented: foo.c
        $(PREP) $(CC) $(CFLAGS) -o foo foo.c
```

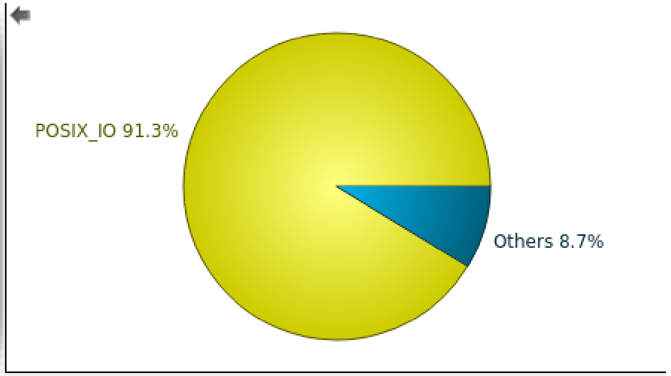
In your batch file:

```
#! /bin/bash
#SBATCH -nodes=256
#SBATCH -ntasks=256
#SBATCH ...

export SCOREP_ENABLE_TRACING=true
export SCOREP_ENABLE_PROFILING=false
export SCOREP_TOTAL_MEMORY=256MB

srun -n 256 ./your-app
```

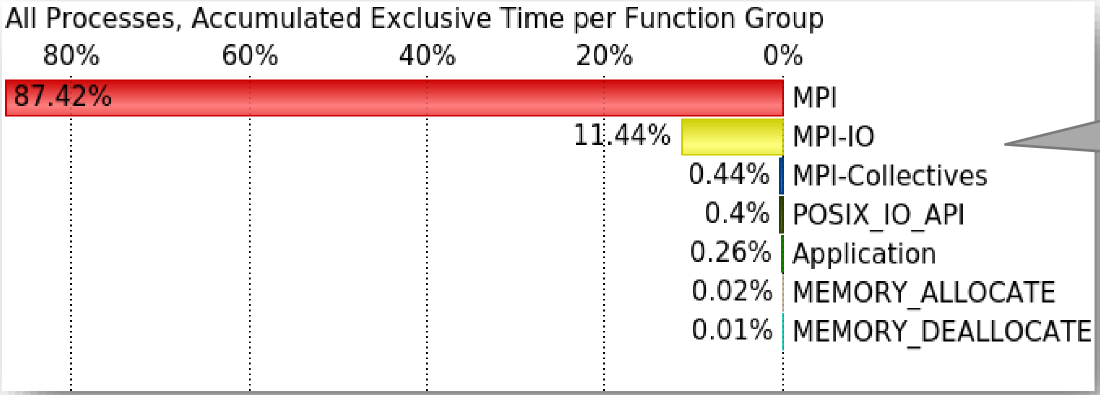
I/O operations over time



Individual I/O Operation

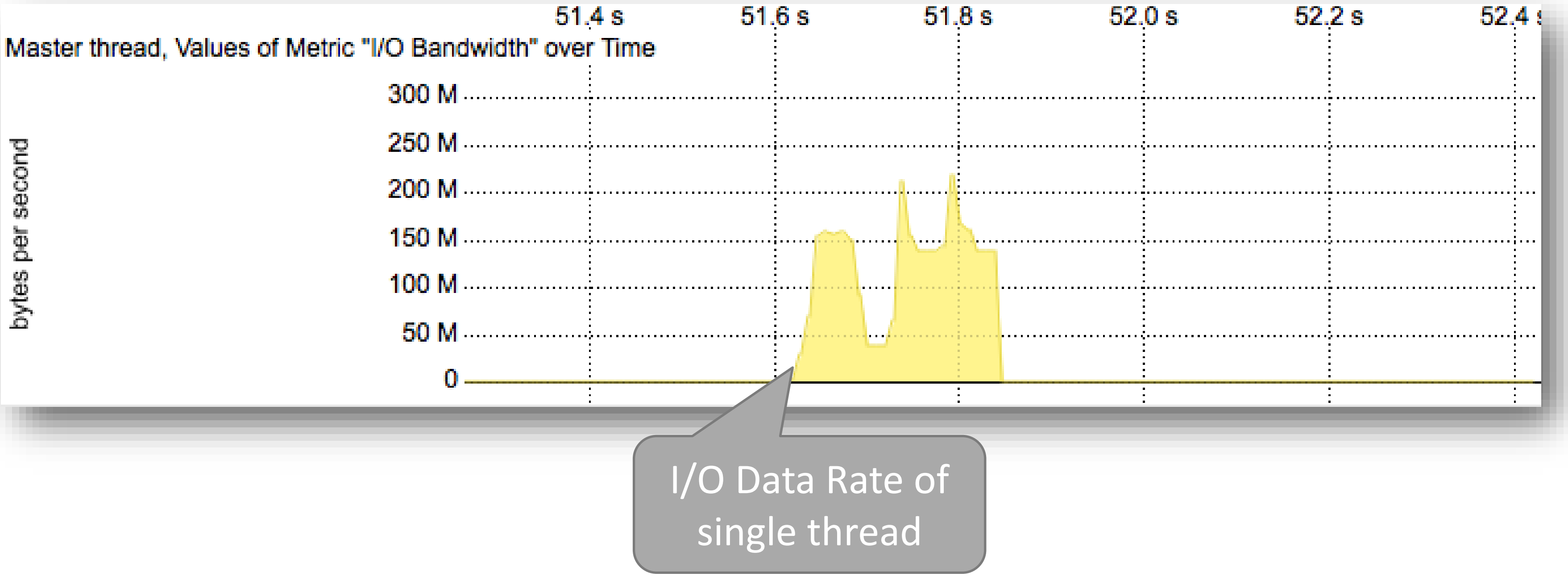


Functions	
Number of called Functions	118
Number of Invocations	5,199,963
Messages	
I/O Events	
POSIX I/O	
Number of I/O Transactions	1,139,309
Aggregated I/O Transaction Size	121.7 GiB
Average I/O Transaction Size	112.01 KiB
Aggregated I/O Transaction Time	970.35 ns
Average I/O Transaction Time	0.00 ps
ISO C I/O	
Number of I/O Transactions	4,016,749
Aggregated I/O Transaction Size	121.8 GiB
Average I/O Transaction Size	31.8 KiB
Aggregated I/O Transaction Time	970.64 ns
Average I/O Transaction Time	0.00 ps
MPI-IO	
Number of I/O Transactions	5,000,275
Aggregated I/O Transaction Size	188.58 GiB
Average I/O Transaction Size	39.55 KiB
Aggregated I/O Transaction Time	3.02 µs
Average I/O Transaction Time	0.00 ps

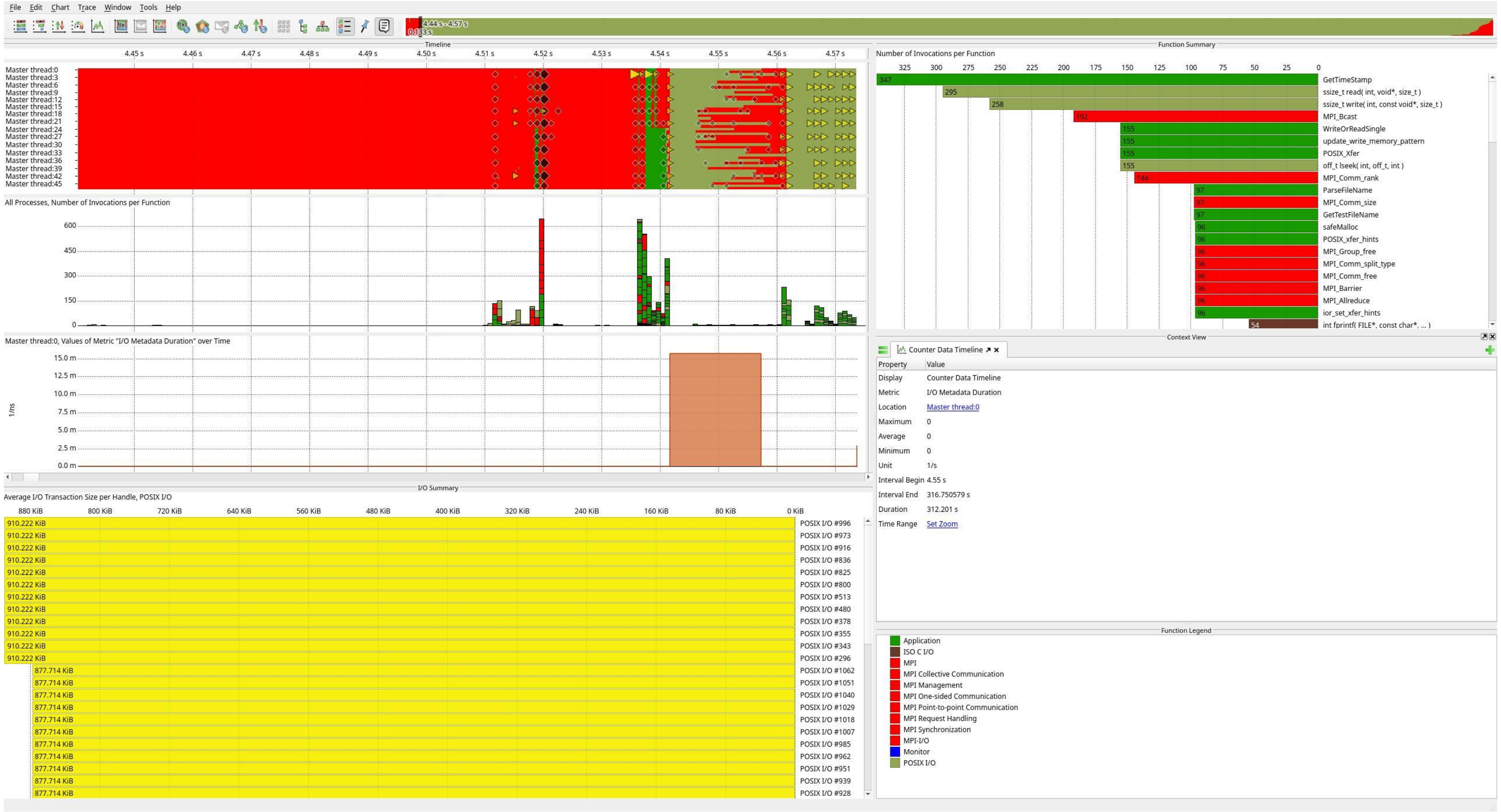


I/O Runtime Contribution

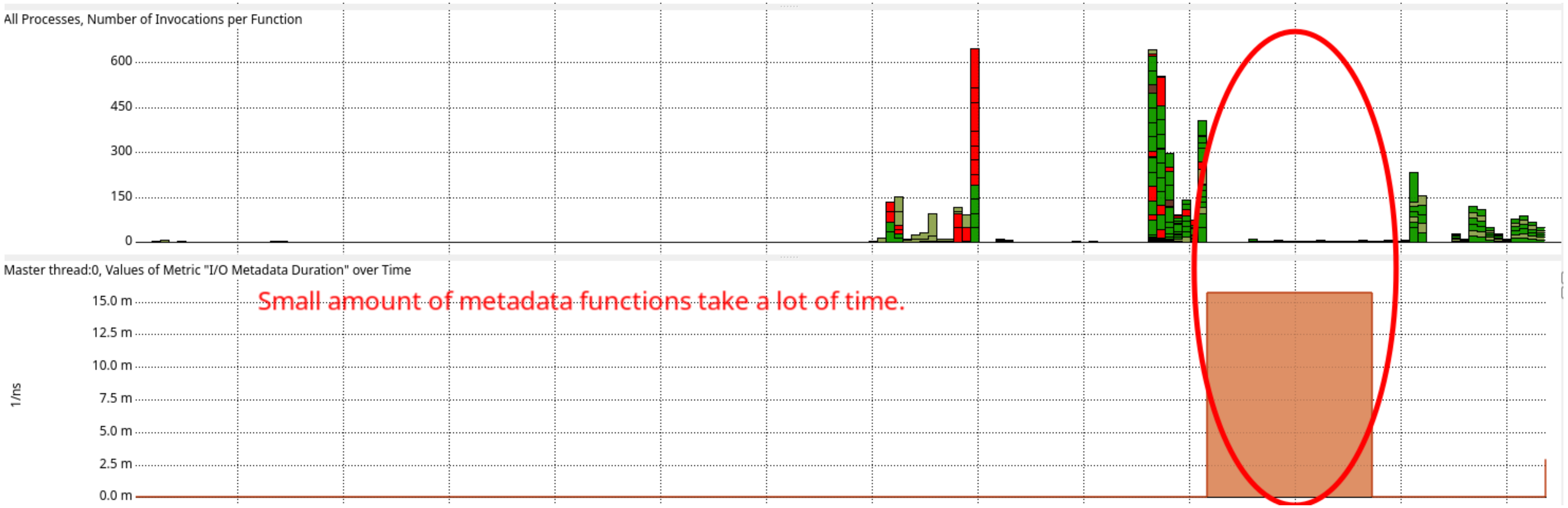
I/O data rates over time



Support for Metadata Operations



Allows for detailed analysis of I/O



I/O Best Practice

I/O Best Practices

- Read small, shared files from a single task.
  - Instead of reading a small file from every task, it is advisable to read the entire file from one task and broadcast the contents to all other tasks.
- Small file (<1GB) accessed by a single process (set stripe count of 1).
- Medium sized files (>1GB) accessed by a single process – set to utilize a stripe count of no more than 4.
- Large files (>10GB)
  - Stripe count should be adjusted to a value larger than 4.
  - Such files should never be accessed by serial I/O or file-per-process I/O pattern.

I/O Best Practices (2)

- Limit the number of files within a single directory.
  - Incorporate additional directory structure.
  - Set stripe count of directories that contain many small files to 1.
- Place small files on single OSTs.
  - If only one process will read/write the file and the amount of data in the file is small (<1GB), performance will be improved by limiting the file to single OST on creation.
- Place directories containing many small files on single OSTs.
  - If you are going to create many small files in a single directory, greater efficiency will be achieved if you have the directory default to 1 OST on creation.

I/O Best Practice (3)

- Avoid opening and closing files frequently → this creates excessive metadata overhead
- Use `ls –l` only where absolutely necessary
  - Consider that `ls –l` must communicate with every OST that is assigned to a file being listed and this is done for every file listed. `lfs find` is more efficient solution
- Consider I/O middleware libraries such as SIONlib, ADIOS, (HDF5), (p)netCDF, or MPI-IO.
- Limit the number of files (less Metadata and easier to post process).
- Make large continous requests, group operations → increase bandwidth, decrease latency.
- Prefer collective I/O to independent I/O, especially if operations can be aggregated.
- Use derived datatypes and file views.

I/O Best Practice (4)

- Open files in correct mode, allows the system to apply optimisations.
- Write/read arrays datastructures in one call rather then element per element.
- Avoid excessive stdout / stderr output.
- Flush buffers only if necessary.
- Create files independent from number of processes → easier to post process, easier to scale.
- Write/ read only if necessary.
- If you work with a lot of data plan your I/O before writing the code.

Thank you!

Contact: sebastian.oeste@tu-dresden.de