



703308 VO High-Performance Computing WS2022/2023

Parallel I/O

Philipp Gschwandtner

Overview

- ▶ Parallel filesystems and I/O concepts
 - ▶ Software and hardware characteristics
- ▶ MPI I/O
 - ▶ How to open, read & write files, performance aspects
- ▶ High-level I/O libraries
 - ▶ HDF5

Motivation: kmMountains (PRACE Access project) cont'd

- ▶ Simulates mountain climates in European Alps and Himalaya
 - ▶ Estimated project duration of 3 years
 - ▶ Total amount of data to be written: approx. 400 TB per year
- ▶ High I/O requirements
 - ▶ Up to 82 GB for checkpointing
 - ▶ Up to 1 TB of result data
 - ▶ per simulation run!

Run type	#Simulation steps	Wall time per step	#Nodes	Total node hours	Storage needs
Himalayas at 2.2 km	312 months	8.6 h	204	547373	198 TB
Himalayas at 12 km	372 months	1.7 h	8	5059	18.5 TB
Alps at 1.1 km	240 months	16.0 h	104	399360	86 TB
Alps at 2.2 km	240 months	12.0 h	40	115200	21.6 TB
Alps at 12 km	300 months	1.7 h	4	2040	7.8 TB
Processing				100000	
Total Year 1				1169032	332 TB
Estimated Year 2				~1300000	~400 TB
Estimated Year 3				~1300000	~400 TB

Motivation: PRACE technical guidelines

- ▶ Lists and describes technical properties and requirements of every PRACE Supercomputer
 - ▶ Min/max/average for job lengths, RAM, no. of cores, etc.
 - ▶ Also includes storage
 - ▶ size and number of files
- ▶ [https://prace-ri.eu/wp-content/uploads/Technical Guidelines Call 24.pdf](https://prace-ri.eu/wp-content/uploads/Technical_Guidelines_Call_24.pdf)

Number of Files

In addition to the specification of the amount of data, the number of files also has to be specified. If you need to store more files, [the project applicant must contact the centre beforehand for approval](#).

Field in online form	Machine	Max	Remarks
Number of files (Scratch) <number>	HAWK	n.a.	
	Joliot-Curie	2 million	10 000 files max per directory, without backup, files older than 90 days will be removed automatically
	JUWELS	4 million	Without backup, files older than 90 days will be removed automatically
	Marconi100	2 million	Without backup, files older than 50 days will be removed automatically
	MareNostrum 4	2 million	
	Piz Daint	1 million	No limit while running, but job submission is blocked if the max number of files left on scratch is reached
	SuperMUC-NG	1 million	Without backup, old files are removed automatically, Ideal file size: >100 GB
Number of files (Work) <number>	HAWK	100 000	
	Joliot-Curie	500 000	Extensible on demand, 10 000 files max per directory
	JUWELS	3 million	With backup
	Marconi100	2 million	Without backup
	MareNostrum 4	2 million	
	Piz Daint	50 000 per TB	With backup and snapshots
	SuperMUC-NG	1 million	Ideal file size: >100 GB

Number of files (Normal)	HAWK	100 000	
--------------------------	------	---------	--

Motivation

- ▶ Computation is parallel, why not I/O?
 - ▶ Single disks and sequential I/O are slow
 - ▶ I/O performance does not automatically scale with compute parallelism
- ▶ Why MPI or any other I/O library?
 - ▶ You could use e.g. POSIX to manually implement parallel I/O...
 - ▶ ...but that would mean reinventing the wheel for every new program
- ▶ Parallel I/O is very complex!
 - ▶ David Henty (EPCC): “IO is the HPC equivalent of printing”
 - ▶ Lots of platform-dependent properties, hard to give general advice

Why is I/O in HPC so complex?

- ▶ Data layout issues become much more significant
 - ▶ Data is persistent, can outlive job duration
 - ▶ Consider checkpoint/restart where the number of nodes changes in-between
 - ▶ Pre- or postprocessing using programs other than your main application
- ▶ RAM means Random Access Memory
 - ▶ Random file access (`fseek()`) is far more inefficient
 - ▶ Performance impact of linearization (e.g. depth-first tree indexing, etc.) is aggravated
- ▶ I/O software and hardware stacks are very complex
 - ▶ Hardware: Multiple components, network interfaces and properties
 - ▶ Software: Multiple libraries, I/O call patterns and tunable parameters

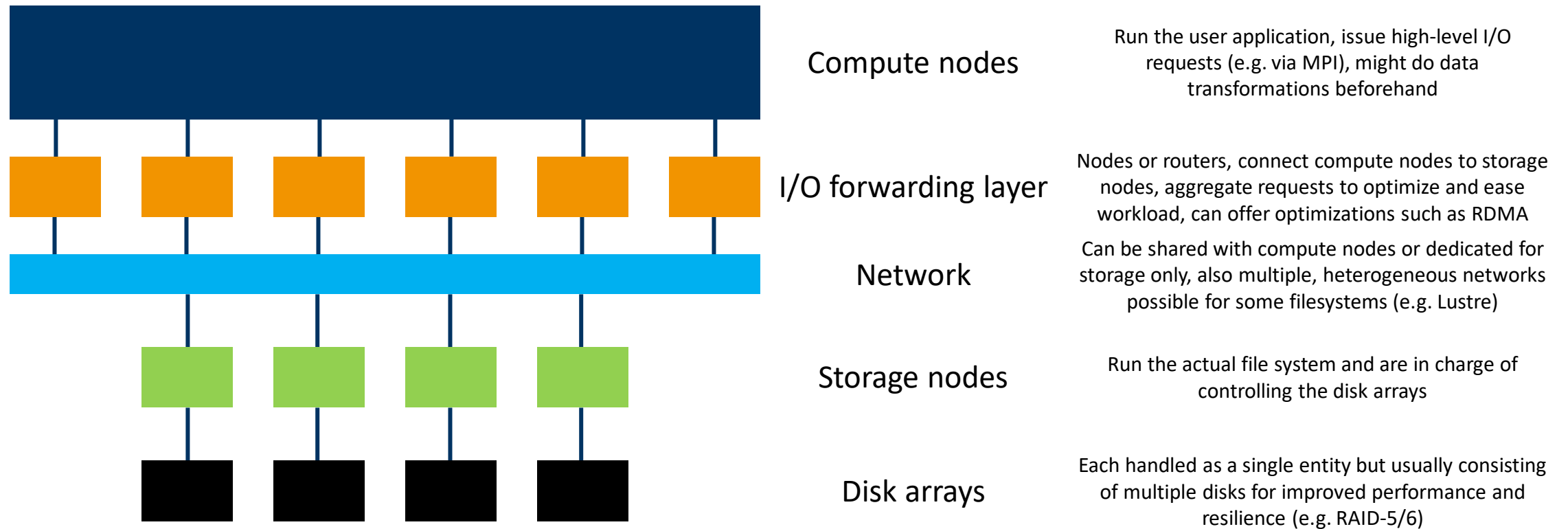




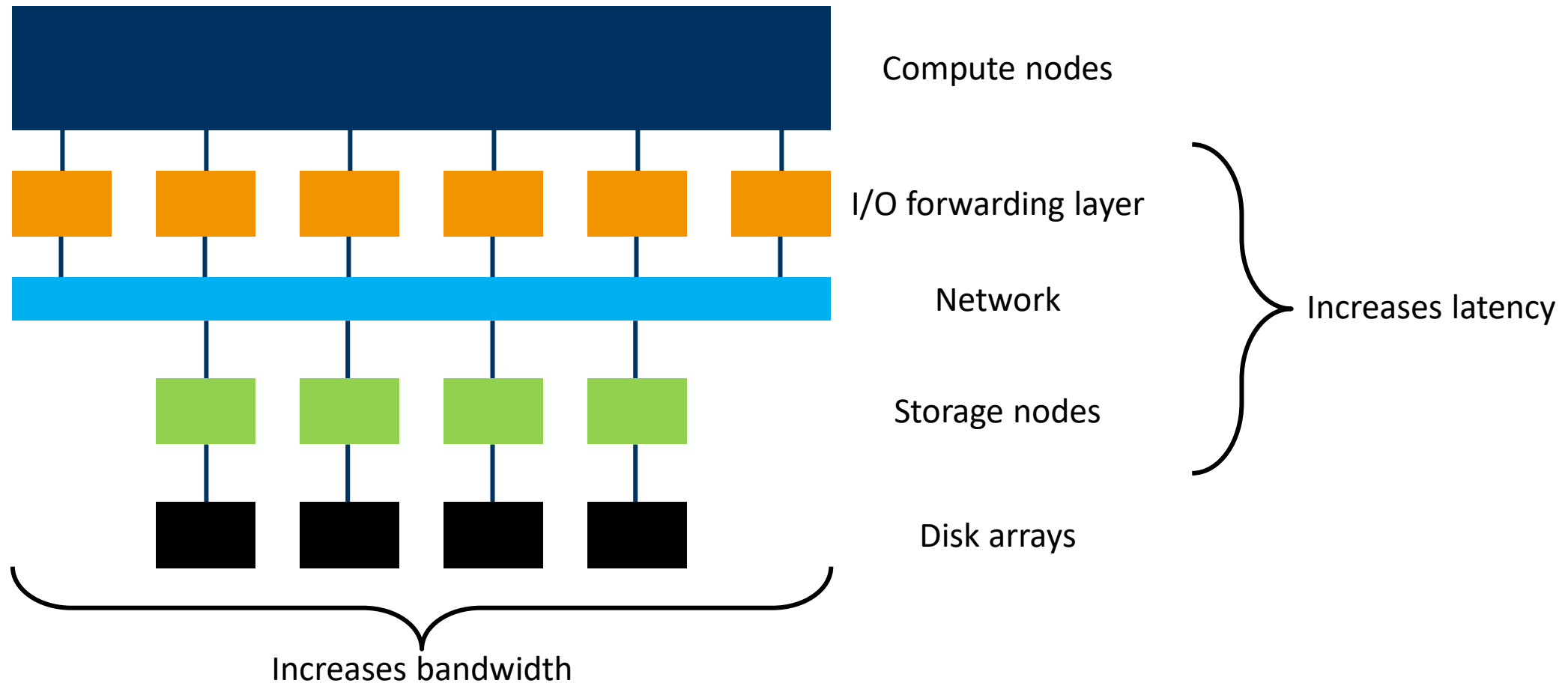
Parallel Filesystems and I/O Concepts



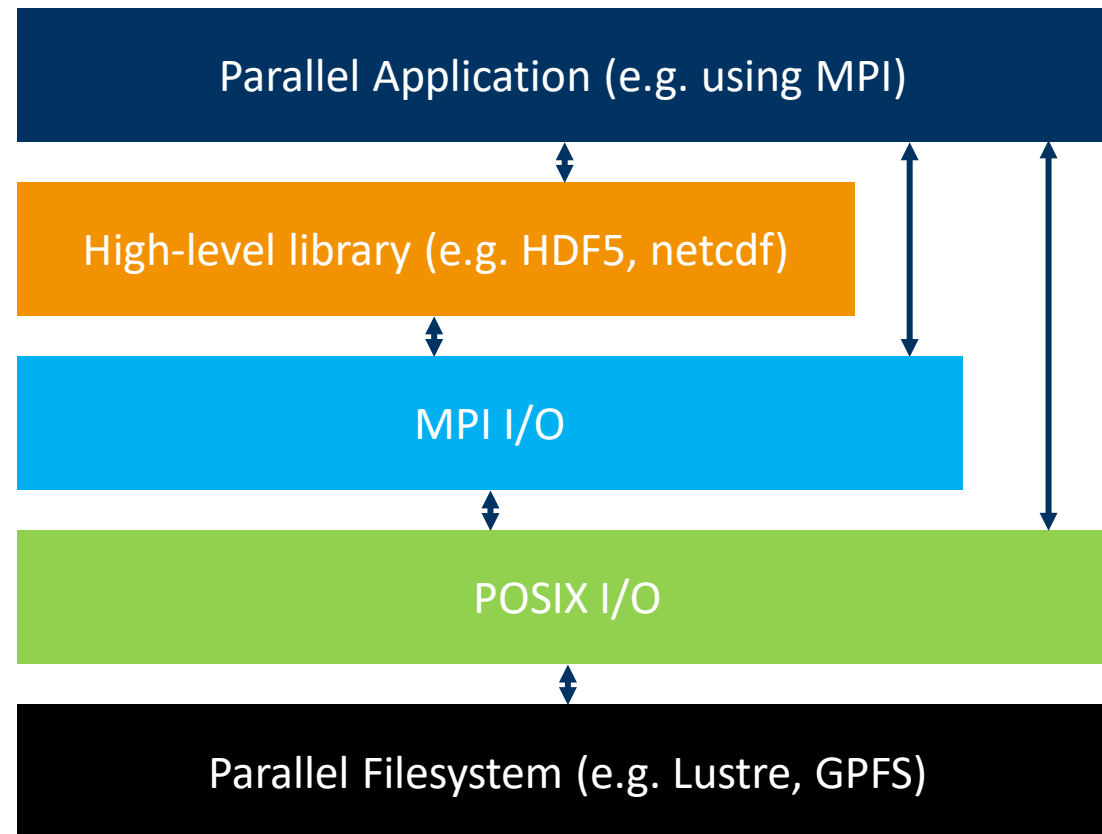
General hardware stack



General hardware stack



General software stack



Parallel filesystems

- ▶ Provide explicit support for parallel I/O in the application (or library)
- ▶ Lots of additional features
 - ▶ Load balancing
 - ▶ Caching
 - ▶ Request aggregation
 - ▶ Other performance optimizations
- ▶ Many big players, including
 - ▶ Lustre
 - ▶ DAOS
 - ▶ GPFS (IBM Spectrum Scale)
 - ▶ BeeGFS
 - ▶ OrangeFS
- ▶ Try to adhere to POSIX requirements but are usually not fully compliant
 - ▶ Only need to support enough to not break most applications
 - ▶ Often unsupported: atomic move/rename, file locks, unlinking open files, symbolic or hard links, etc.

Lustre

- ▶ Many large-scale HPC systems use Lustre
 - ▶ Basically all EuroHPC JU systems
 - ▶ More than half of the first 100 TOP500 systems
 - ▶ Easily achieves TB/s bandwidths if properly configured and provisioned
 - ▶ GPLv2
- ▶ Uses two main concepts of storage
 - ▶ Object storage: stores the actual files
 - ▶ Meta data: stores filenames, permissions and which OST holds the file(s)

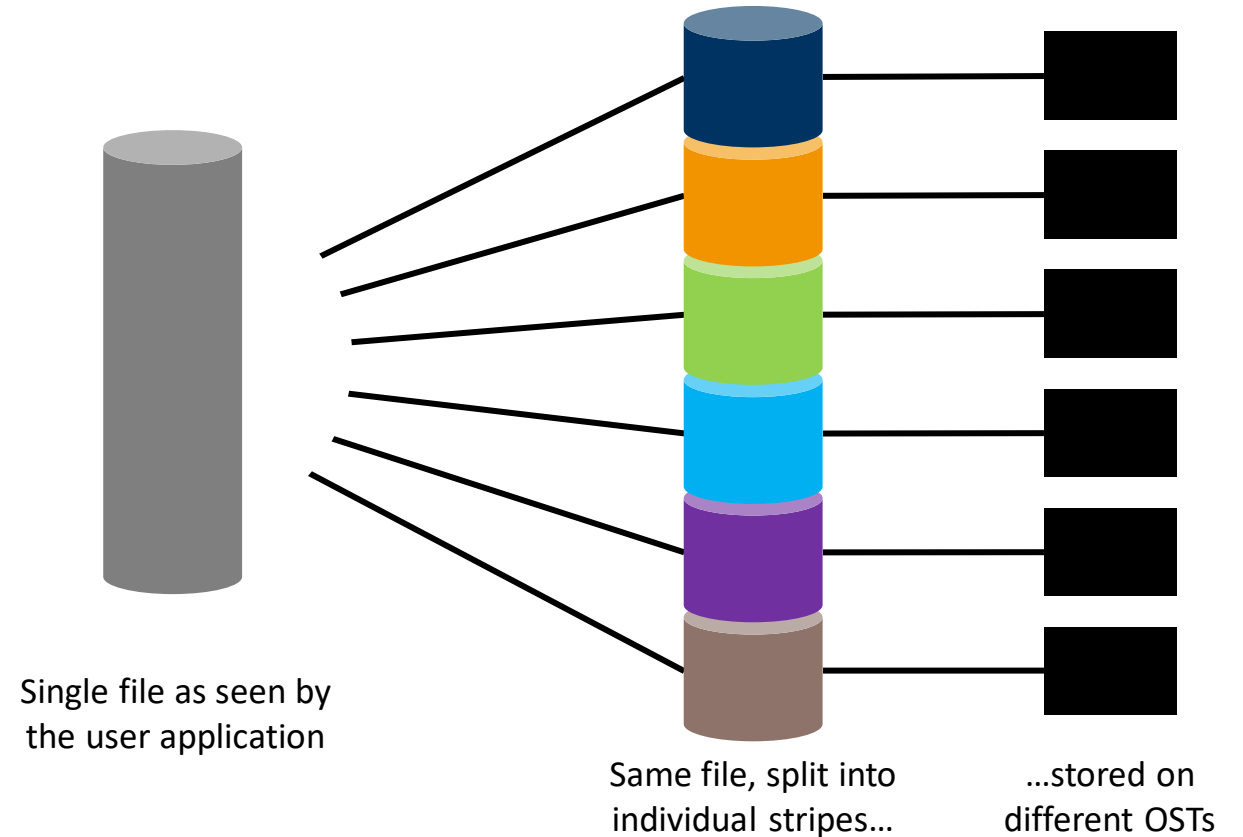
- ▶ Uses ext4 or ZFS as backend storage
 - ▶ Invisible to user
 - ▶ Enables user-independent optimizations such as compression, deduplication or copy-on-write
 - ▶ But no support for resilience!

	Server	Target (disk arrays)
Meta data	MDS	MDT
Object storage (=user data)	OSS	OST

Lustre terminology

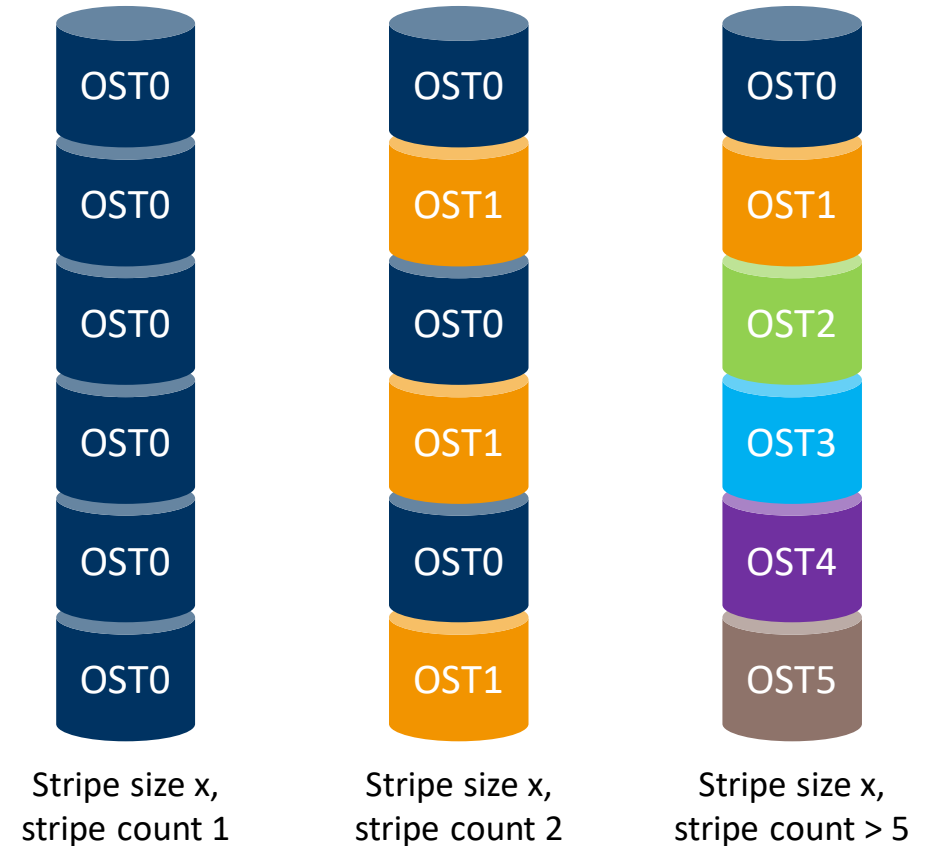
Lustre striping

- ▶ Performance gain derived through “striping”
 - ▶ File is split by the file system in e.g. 2 MB stripes
 - ▶ Size can be controlled at runtime, per file/path (usually between 1 and 32 MB)
- ▶ Enables parallel file access without requiring application support
 - ▶ Careful: no resilience in Lustre, losing an OST means losing all files on it
 - ▶ Resilience must be handled within OST (e.g. RAID-5/6), also: backups



Lustre stripe size and stripe count

- ▶ **Stripe Size**
 - ▶ The length of a single stripe
 - ▶ A single stripe is always located contiguously on an OST
- ▶ **Stripe Count**
 - ▶ The number of OSTs to use per file
- ▶ Note: most parallel filesystems use such an approach, with different terminology (GPFS: “blocks”) and slight differences in semantics

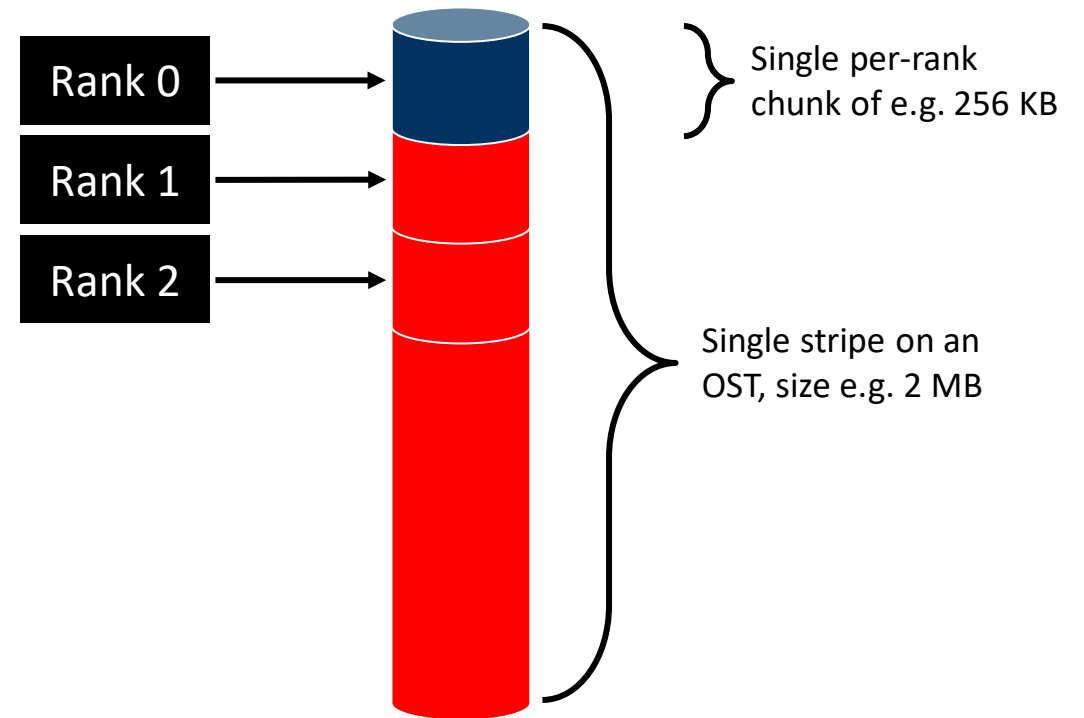


Common Lustre tools

- ▶ `lfs osts <path>`
 - ▶ List name and status of OSTs for given file path
- ▶ `lfs getstripe <path>`
 - ▶ Get striping information
- ▶ `lfs setstripe <args> <path>`
 - ▶ Set striping information
- ▶ `lfs df`
 - ▶ Get disk usage statistics

Performance pitfall: false sharing

- ▶ Similar to false sharing in multi-threaded programs
 - ▶ Cache line ~ stripe
 - ▶ Also: lock contention
- ▶ Rank 0 issues 256 KB write request to Lustre
 - ▶ “No Problem! Let me lock the file on OST X, read 2 MB, update 256 KB of it, write the 2 MB back to disk, and release the lock”
- ▶ Rank 1 issues 256 KB write request to Lustre
 - ▶ “No Problem! Let me lock the file on OST X, read 2 MB, update 256 KB of it, write the 2 MB back to disk, and release the lock”
- ▶ ...
- ▶ Stripe-aligned I/O access is vital for good performance!
 - ▶ User-controlled (domain decomposition and/or I/O request granularity)!



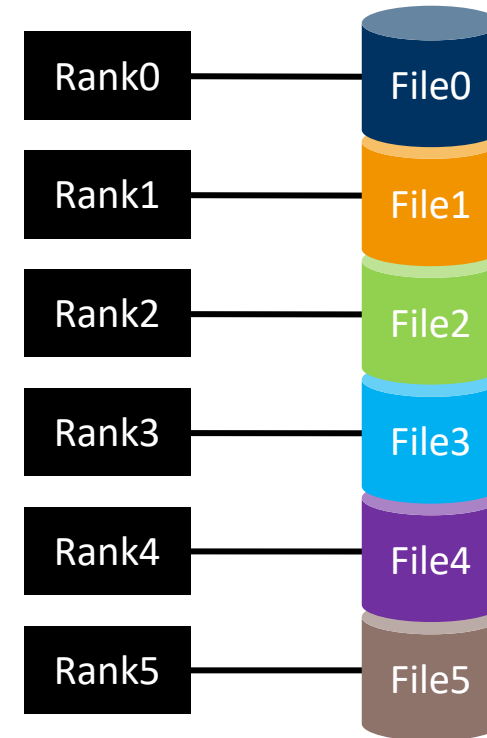
File access patterns: file-per-rank

► Advantages

- Simple to implement
- No coordination between processes needed
- No false sharing of stripes/blocks

► Disadvantages

- Number of files quickly becomes unmanageable
- Files often need to be merged to create a canonical dataset for post-processing
- Meta data handling bottleneck (e.g. some systems only have one MDS for all users)



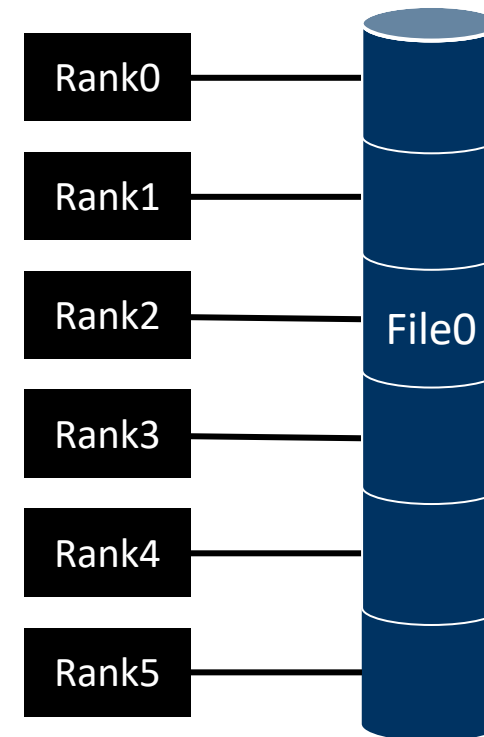
File access patterns: file-per-app

► Advantages

- Number of files is independent of number of processes
- Files can be in canonical representation (no post-processing)

► Disadvantages

- Uncoordinated client requests might induce performance penalties
- File layout may induce false sharing of file system blocks
- Implementation overhead



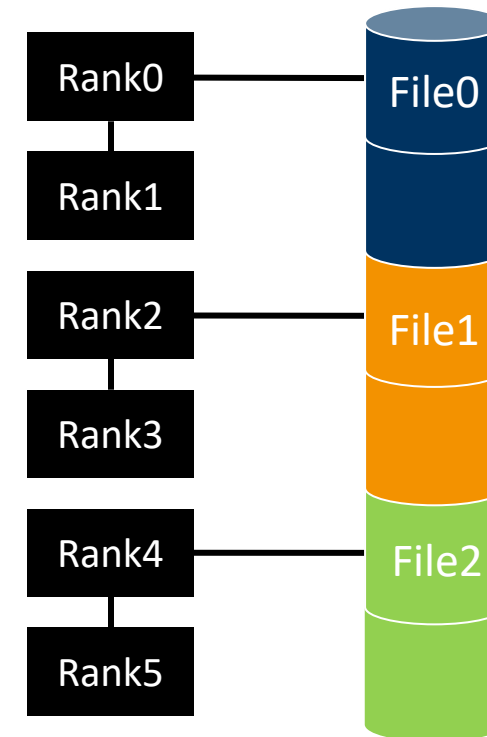
File access patterns: file-per-subset

► Advantages

- ▶ Number of files only depends on e.g. number of nodes
- ▶ Possibly reduces strain on I/O subsystem

► Disadvantages

- ▶ Requires application-side tuning for optimal results
- ▶ Postprocessing required
- ▶ Additional implementation overhead



UIBK I/O infrastructure

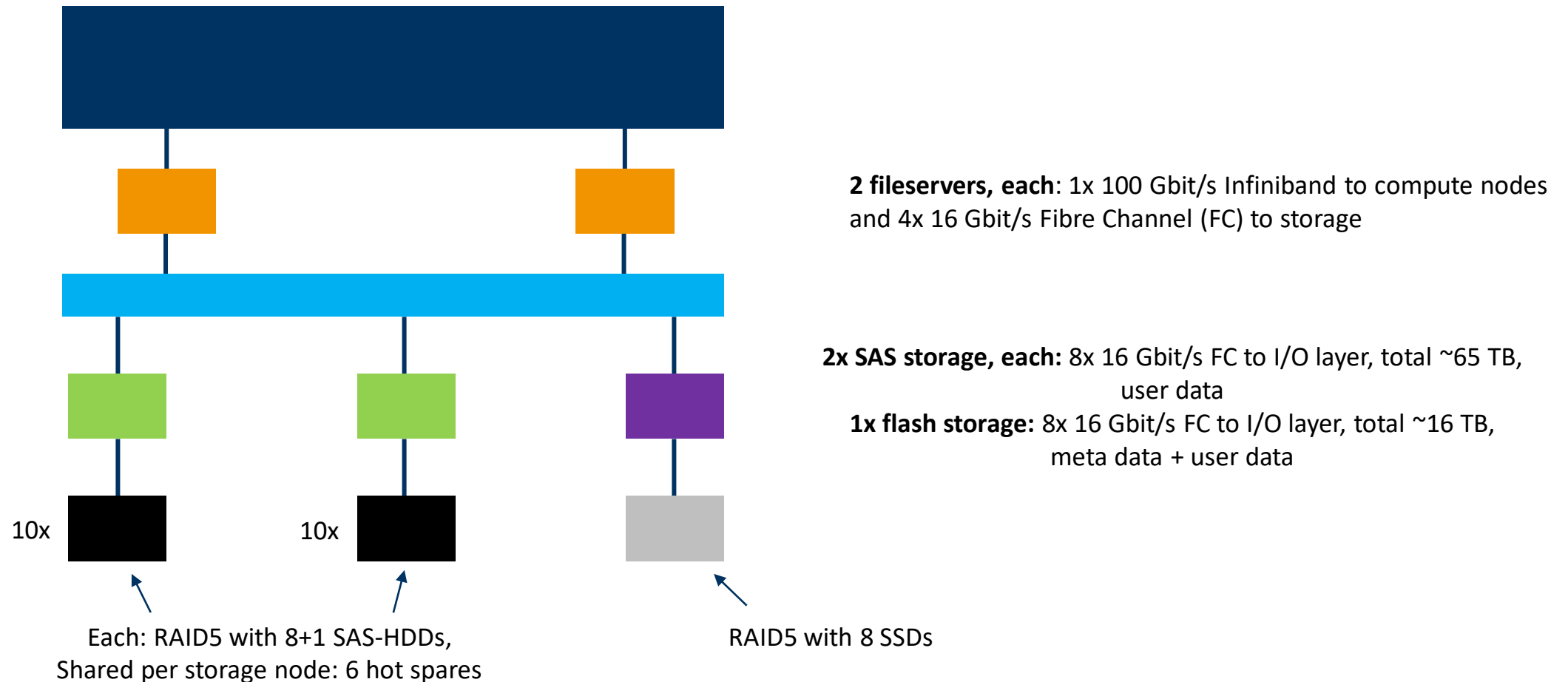
- ▶ LCC2 (\$SCRATCH)

- ▶ 1 Gbit/s NFS to general UIBK storage, no parallel file system ☹️

- ▶ LEO3e + LEO4

- ▶ GPFS
- ▶ HDDs for user data
- ▶ Flash storage for meta data (LEO4: and user data)
- ▶ RAID5 disk arrays for performance and resilience

Local example: LEO4 @ UIBK (GPFS)



IO500

- ▶ Similar to TOP500 but for storage performance

- ▶ <https://io500.org/>

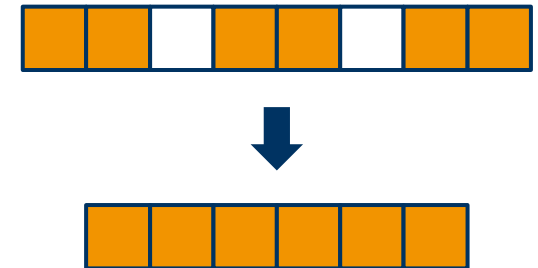
#	INFORMATION								IO500	
	BOF	INSTITUTION	SYSTEM	STORAGE VENDOR	FILE SYSTEM TYPE	CLIENT NODES	TOTAL CLIENT PROC.	SCORE	BW ↑ (GIB/S)	MD (KIOP/S)
1	SC22	Argonne National Laboratory	Aurora Storage	Intel	DAOS	260	27,040	20,694.50	6,048.69	70,802.51
2	ISC21	Pengcheng Laboratory	Pengcheng Cloudbrain-II on Atlas 900	Pengcheng	MadFS	512	36,864	36,850.40	3,421.62	396,872.82
3	SC22	Sugon Cloud Storage Laboratory	ParaStor	Sugon	ParaStor	10	2,560	8,726.42	718.11	106,042.93
4	SC20	JCAHPC	Oakforest-PACS	DDN	IME	2,048	4,096	253.57	697.20	92.22
5	ISC20	Korea Institute of Science and Technology Information (KISTI)	NURION	DDN	IME	2,048	2,048	282.45	515.59	154.74



MPI I/O

Characteristics

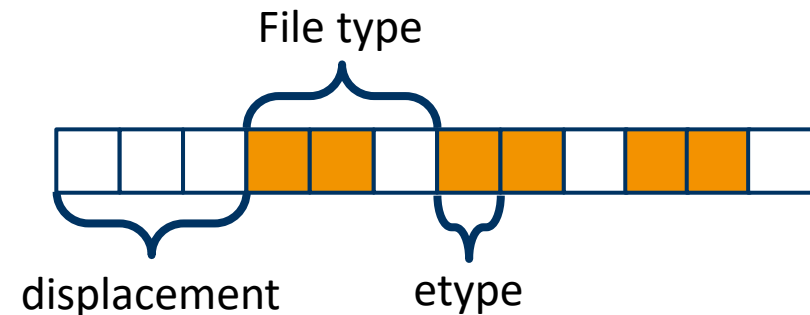
- ▶ Provides “high-level” interface for
 - ▶ Data partitioning
 - ▶ Asynchronous I/O operation
 - ▶ Strided read & write access
- ▶ Interfaces with derived datatypes
 - ▶ Reading and writing files similar to exchanging messages
 - ▶ Allows MPI-idiomatic message transfer and file I/O
 - ▶ Allows to aggregate strided file accesses
 - ▶ Remember MPI vector derived datatype
 - ▶ Storage media like contiguous accesses!



Basics

- ▶ Very similar to POSIX (and most other) I/O
 - ▶ `fopen()` == `MPI_File_open()`
 - ▶ `fwrite()` == `MPI_File_write()` (and a lot of others)
 - ▶ `fclose()` == `MPI_File_close()`
- ▶ Features could be implemented manually with POSIX, but complicated
 - ▶ Parallel access, buffering, offsets, flushes, etc.

- ▶ Basic building blocks & terminology
 - ▶ `etype`: type of individual elements
 - ▶ `filetype`: which portion of file is visible to current rank
 - ▶ `displacement`: number of bytes to be skipped from the start of the file
 - ▶ `view`: tuple of (`etype`, `filetype`, `displ.`)



Example

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    MPI_File fh;
    MPI_File_open(MPI_COMM_WORLD, "my_file", MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);

    MPI_Offset offset = ...; // e.g. compute a rank-dependent file offset
    char* buf = ... // data to be written
    int size = ...; // size of data to be written
    MPI_File_write_at(fh, offset, buf, size, MPI_CHAR, MPI_STATUS_IGNORE);

    MPI_File_close(&fh);
    free(buf);
    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Opening and closing files

- ▶ `MPI_File_open(comm, filename, amode, info, fh)`
 - ▶ Is a collective call
 - ▶ Filename is implementation-dependent but must reference same file on all ranks
 - ▶ Allows specifying access modes and info hints
 - ▶ Internally, MPI maintains individual and shared file pointers after opening a file
 - ▶ subsequent operations decide which of those are used and/or updated
- ▶ `MPI_File_close(fh)`
 - ▶ Is a collective call
 - ▶ Will delete file if open mode included `MPI_DELETE_ON_CLOSE`
 - ▶ Useful for temporary files

File open access modes

- ▶ `MPI_MODE_RDONLY`: read-only
 - ▶ `MPI_MODE_WRONLY`: write-only
 - ▶ `MPI_MODE_RDWR`: read and write
 - ▶ `MPI_MODE_CREATE`: create if file does not exist
 - ▶ `MPI_MODE_EXCL`: fail if file already exists
 - ▶ `MPI_MODE_DELETE_ON_CLOSE`: delete on close
 - ▶ `MPI_MODE_UNIQUE_OPEN`: file not opened concurrently (even outside MPI; removes need for locks)
 - ▶ `MPI_MODE_SEQUENTIAL`: file is only accessed sequentially: mandatory for sequential stream files (pipes, tapes, ...); mutually exclusive with `MPI_MODE_RDWR`
 - ▶ `MPI_MODE_APPEND`: all file pointers set to end of file
-
- ▶ Must be the same for all ranks participating in the `MPI_File_open()` call

File info: hints for MPI

- ▶ Used in `MPI_File_open()`, `MPI_File_set_view()`, or `MPI_File_set_info()`
 - ▶ collective buffering
 - ▶ `collective_buffering`: specifies whether the application may benefit from coalescing small I/O requests in the I/O forwarding layer to form larger blocks for the storage layer
 - ▶ `cb_block_size`: data access in chunks of this size
 - ▶ `cb_buffer_size`: on each node, usually a multiple of stripe/block size
 - ▶ `cb_nodes`: number of nodes to be used for collective buffering
 - ▶ disk striping (only relevant in `MPI_FILE_OPEN()`)
 - ▶ `striping_factor`: number of I/O devices used for striping
 - ▶ `striping_unit`: length of a chunk on a device (in bytes)
- ▶ Use `MPI_INFO_NULL` if no hints

Writing and reading data

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	MPI_File_read_at()	MPI_File_read_at_all()
		MPI_File_write_at()	MPI_File_write_at_all()
	nonblocking	MPI_File_iread_at()	MPI_File_iread_at_all()
		MPI_File_iwrite_at()	MPI_File_iwrite_at_all()
individual file pointers	blocking	MPI_File_read()	MPI_File_read_all()
		MPI_File_write()	MPI_File_write_all()
	nonblocking	MPI_File_iread()	MPI_File_iread_all()
		MPI_File_iwrite()	MPI_File_iwrite_all()
shared file pointer	blocking	MPI_File_read_shared()	MPI_File_read_ordered()
		MPI_File_write_shared()	MPI_File_write_ordered()
	nonblocking	MPI_File_iread_shared()	
		MPI_File_iwrite_shared()	

File pointer semantics

- ▶ **Explicit offsets**
 - ▶ Most basic I/O, like POSIX
 - ▶ read/write location is always computed using start of file + offset, no file pointers are changed
- ▶ **Individual file pointers**
 - ▶ Allows use of derived data types
 - ▶ File pointer is advanced when used in an operation
 - ▶ Each operation on a file pointer only affects this rank's individual file pointer
- ▶ **Shared file pointers**
 - ▶ Same as individual FP, but single, shared file pointer is advanced
 - ▶ Non-collective: concurrent calls by multiple ranks are serialized in nondeterministic order
 - ▶ Collective: concurrent calls by multiple ranks are serialized in-order

File Views

- ▶ Provide a visible and accessible set of data from an open file
- ▶ `MPI_File_set_view(fh, disp, etype, filetype, datarep, info)`
 - ▶ collective operation
 - ▶ Allows to change the rank's view of the data, even repeatedly
 - ▶ local and shared file pointers are reset to zero
 - ▶ `datarep` argument is a string that specifies the format in which data is written to a file: `native`, `internal`, `external32`, or `user-defined`
 - ▶ same `etype` extent and same `datarep` on all processes

General guidelines

- ▶ Lustre

- ▶ Keep stripe count above 1 but below total number of OSTs (allows Lustre to avoid slow OSTs)

- ▶ MPI I/O

- ▶ Explicit offsets, non-collective: similar to independent POSIX calls, (often) bad performance
- ▶ Explicit offsets, collective: exploits parallelism in filesystem, better performance
- ▶ Individual file pointers, non-collective: exploits request aggregation due to derived data types, better performance
- ▶ Individual file pointers, collective: best of both worlds, best performance

- ▶ A ton of tuning involved, dependent on: filesystem semantics; number of storage and I/O forwarding nodes; size, bandwidth and latency of disk arrays; buffer sizes; network contention; application characteristics; etc.

Error Handling

- ▶ File handles have their own error handler
 - ▶ Default is `MPI_ERRORS_RETURN` (c.f. message passing: `MPI_ERRORS_FATAL`)
- ▶ Changing the default:
 - ▶ `MPI_File_set_errhandler(MPI_FILE_NULL, MPI_ERRORS_FATAL);`
- ▶ Note: MPI behavior is undefined after first erroneous MPI call
 - ▶ But a high quality implementation will support I/O error handling facilities



High-level I/O libraries



HDF5 / NetCDF4

- ▶ Hierarchical Data Format / Network Common Data Form
- ▶ Are both libraries for efficiently writing and reading scientific data
 - ▶ MPI is quite low-level, doesn't provide any file structure
- ▶ Are somewhat compatible
 - ▶ NetCDF4 uses a subset of HDF5 file structure + some additions
- ▶ Provide API and tools for reading and manipulating files
 - ▶ HDF5: C, C++, Fortran 90, Python, Java, Command line (and probably others)
 - ▶ E.g. `h5diff`

HDF5 Structure

- ▶ Provides a unix-like path structure
 - ▶ E.g. /Group1/ Group5/DataSet9
- ▶ Terminology:
 - ▶ File: Container for storing data
 - ▶ Group: Structure that can contain HDF5 objects (datasets, attributes)
 - ▶ Attribute: Describe datasets (e.g. experiment parameters, file versions, etc.)
 - ▶ Dataspace: Describes shape of dataset (e.g. dimensionality)
 - ▶ Dataset: Multi-dimensional arrays of data

HDF5 File Operations

H5Fcreate(), H5Fopen()

File level

H5Screate_simple()

Dataspace level

H5Dcreate(), H5Dopen()

Dataset level

H5Dread(), H5Dwrite()

H5Dclose()

H5Sclose()

H5Fclose()

HDF5 Features

- ▶ **Parallelism**

- ▶ Supports collective or non-collective MPI I/O, controlled by user

- ▶ **Compression**

- ▶ Data can be automatically compressed in-transit

- ▶ **Hyperslabs**

- ▶ Select a subset of data corresponding to e.g. a certain dimension
- ▶ Facilitates extracting e.g. a 2D plane from 3D data

Summary

- ▶ Parallel I/O is very complex
- ▶ Getting high performance means knowing the platform and its settings
 - ▶ Stripe/block sizes, counts, individual disk array performance, etc.
- ▶ MPI I/O offers standardized I/O interface
 - ▶ Higher-level libraries exist, use them if suitable!

Sources

- ▶ Rolf Rabenseifner, HLRS
- ▶ Richard J Zamora, ANL
- ▶ David Henty, EPCC
- ▶ ...