Leibniz Supercomputing Centre

I/O Considerations | June 2022 | Patrick Böhl

Outline

- Unix-like Filesystems
- Parallel Filesystems: Spectrum Scale
- HPC I/O Systems and I/O Patterns
- Short introduction to HDF5
- Hyperslabs in HDF5
- I/O Profiling: Darshan

Unix-like Filesystems

- On your laptop you likely use something like Btrfs, Ext4,…
- Most filesystems divide the available disk space in two regions:
  - Inode (index node) region
  - Data region
- Each file is assigned an Inode which contains the metadata of a file:
  - Name of the file
  - Size of the file
  - UID (User ID; Ownership) and GID (Group ID)
  - Creation Date
  - Pointers to the data blocks of the file
  - Some more things… we see in a second

Unix-like Filesystems

- The data region is split into “blocks”.
- A block is the largest contiguous amount of disk space that can be allocated to a file.
- Files larger than one block are stored in multiple blocks (not necessarily contiguous) → see later for parallel filesystems.
- Files smaller than one block occupy a full block → see below.
- The block size determines the maximum size of a read request or write request that a file system sends to the I/O device driver.
- It is also the largest amount of data that can be transferred in a single I/O operation (IOP).

Unix-like Filesystems

- You can check the meta-data of a file and also the block size with “stat” (here for my Laptop):

```
~> touch myfile # create an empty file
~> stat myfile
  File: myfile
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
  Device: 801h/2049d   Inode: 12324216    Links: 1
  Access: (0664/-rw-rw-r--)  Uid: ( 1000/ patrick)   Gid: ( 1000/  patrick)
  Access: 2022-06-22 12:24:26.132397644 +0200
  Modify: 2022-06-22 12:24:26.132397644 +0200
  Change: 2022-06-22 12:24:26.132397644 +0200
  Birth: -
```

- Block size of 4kiB is pretty common.

Unix-like Filesystems

Note: “stat” returns blocks of 512 bytes
instead of the block size of the filesystem.

Actual block size of the file system;
so here 8 blocks are one I/O block.

```
# create a non-empty file
~> echo "foo" > myfile

~> stat myfile
   File: myfile
   Size: 4               Blocks: 8          IO Block: 4096   regular file
   Device: 801h/2049d      Inode: 12324218    Links: 1
   Access: (0664/-rw-rw-r--)  Uid: ( 1000/ patrick)   Gid: ( 1000/ patrick)
   Access: 2022-06-22 12:25:32.069346984 +0200
   Modify: 2022-06-22 12:25:32.069346984 +0200
   Change: 2022-06-22 12:25:32.069346984 +0200
   Birth: -
```

Unix-like Filesystems

```
# check size and disk usage of a file:
~> stat --format="File is %s bytes and uses %B*%b bytes on disk" myfile
File is 4 bytes and uses 512*8 bytes on disk # 512*8bytes = 4096 bytes = 1 block size

# create a 6kb file
~> dd if=/dev/zero of=myfile2 bs=6k count=1
1+0 records in
1+0 records out
6144 bytes (6,1 kB, 6,0 KiB) copied, 0,000349934 s, 17,6 MB/s

~> stat --format="File is %s bytes and uses %B*%b bytes on disk" myfile2
File is 6144 bytes and uses 512*16 bytes on disk # = 8192 bytes = 2 block sizes
```

Unix-like Filesystems

- Theoretical maximum number of inodes: 2^{32} for 32-bit and 2^{64} for 64-bit filesystems.

- Practical limit: capacity of hard drive/block size of the filesystem # no more space left.

- Use of many inodes (i.e. many small files) is generally NOT a good thing:
  - Especially on parallel filesystems with large blocksizes (4MB-16MB) you lose a lot of performance.
  - On a notebook (likely with SSD) this does not matter too much.
  - But this can become a real nightmare on parallel filesystems with “spinning disks”.
  - We saw users with more than 25 millions of files → Disaster.
  - Most users are not aware that what works on their Laptop does not automatically scale to HPC systems.

Unix-like Filesystems

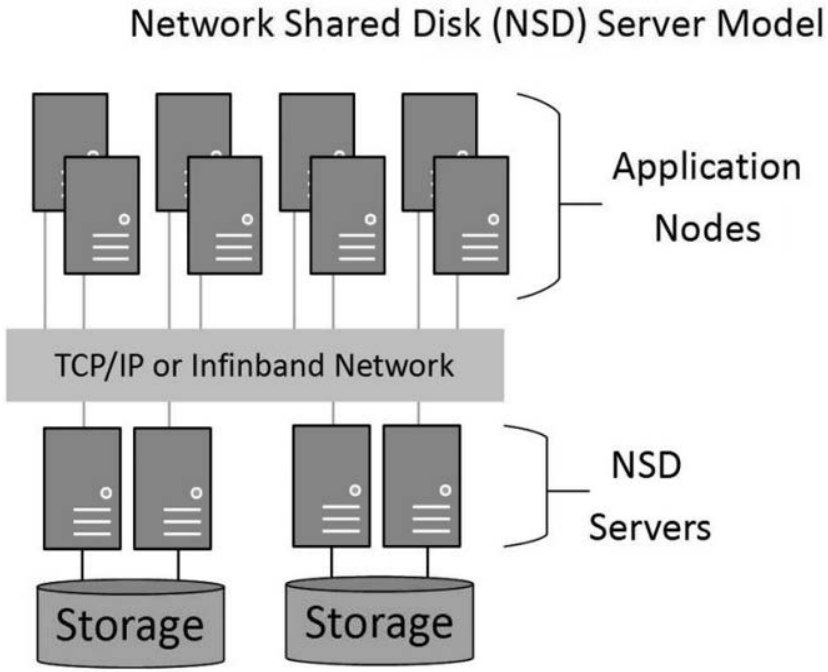
General recommendations:

- There are several (also not so hard to use) alternatives available:

  - High-level I/O libraries like HDF5 (see in a few minutes) and NetCDF (not covered in this lecture).
  - If really necessary, one can also put the small files in a tar-ball (e.g. with the mpifileutils) and on the compute nodes extract the tar-ball into a RAM disk, process the data and afterwards put the data again in a tar-ball).
  - File based databases like LMDB, LevelDB, *Petastorm or* WebDataset etc. (Disclaimer: no personal experience yet, so not covered in this lecture. But recommended by other colleagues who suffer from having users with even 600Mio. files).

Parallel Filesystems: IBM Spectrum Scale (GPFS)

- Here at LRZ we use IBM Spectrum Scale, formerly known as GPFS (General Parallel File System) on both the Linux Cluster and SuperMUC-NG. Other parallel filesystems: Lustre, BeeGFS,…, also Object Storages like DAOS.
- Storage clusters at LRZ consists only of spinning disks (no SSDs etc.).
- Spectrum Scale is a cluster file system that provides concurrent access to filesystems/files.
- Enables high performance access to this common set of data.



Picture Source, High Performance Parallel I/O Book, Chapter 9. Editors Prabhat, Quincey Koziol. October 2014.

Parallel Filesystems: Spectrum Scale

- Main characteristics of Spectrum Scale:
  - Scalability: Large files are divided into equal-sized blocks and the consecutive blocks are placed on different disks in a round-robin fashion. → No contigous files on a single disk anymore. All data and meta-data is distributed across all resources, so-called „wide striping“.
  - Client-side Caching: Cache is kept in a dedicated and pinned area of each node called the pagepool. The cache is managed with both read-ahead techniques and write-behind techniques.
  - Cache coherence and protocol: Uses the distributed locking to synchronize the access to data and metadata on a shared disk.
  - Metadata management: It uses inodes and indirect blocks to record file attributes and data block addresses.

Parallel Filesystems: Spectrum Scale

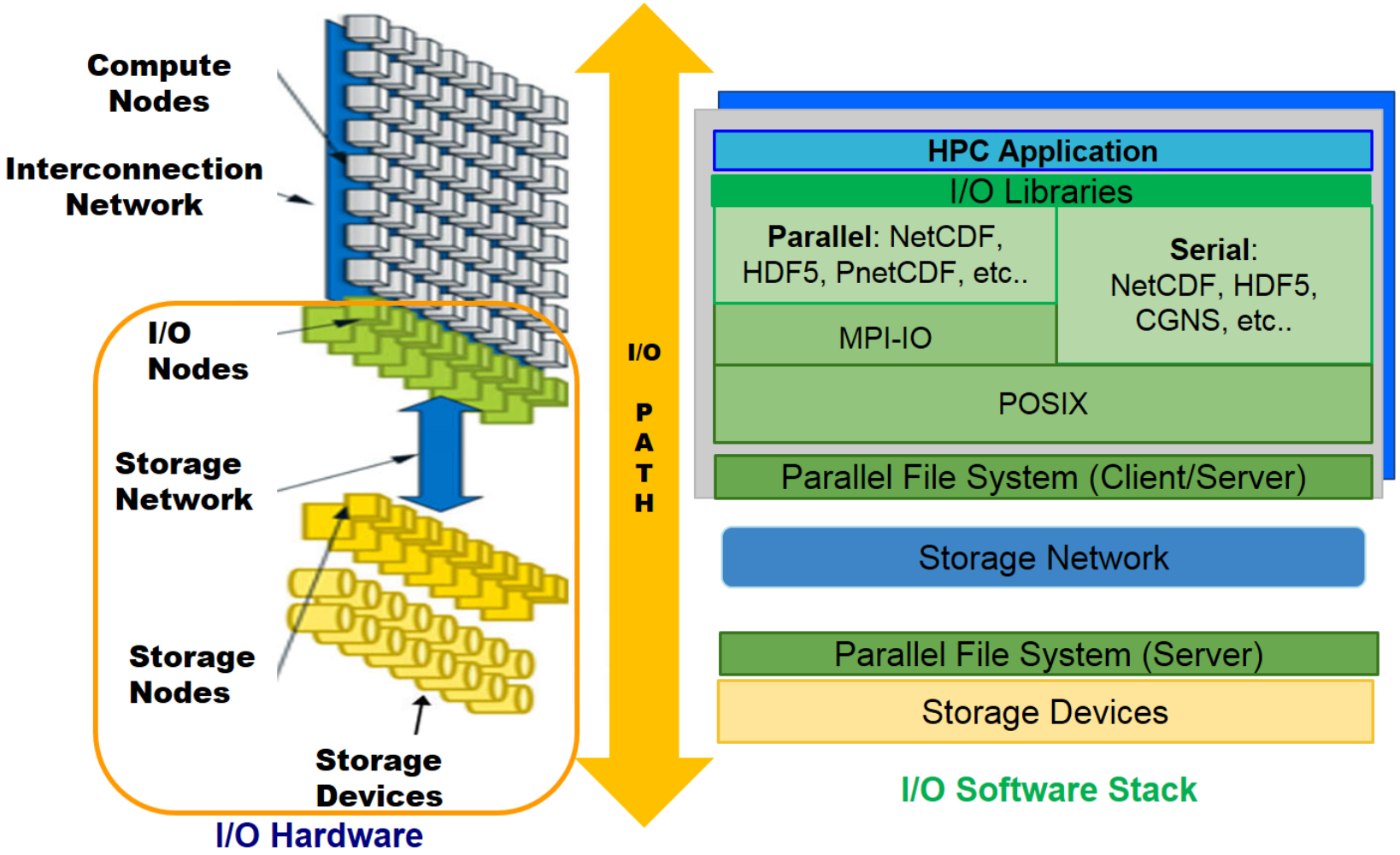
- The block size is rather large: 8MiB on the Linux SCRATCH and 16MiB on WORK/SCRATCH on SuperMUC-NG.
- But: Not every small file occupies 8/16MiB. Each block consists of an integer number of subblocks.
- Subblocks (or fragments) are the smallest amount of contiguous disk space that can be allocated to a file.
- Files smaller than one block occupy as many subblocks as required to store the data.
- Files larger than one block are stored in several full blocks plus the required number of subblocks of the last block holding the data.
- One can check the block and subblock size with “mmlsfs”, here for WORK on SuperMUC-NG:

Parallel Filesystems: Spectrum Scale

```
~> /usr/lpp/mmfs/bin/mmlsfs work
[…]
-f         […]
           65536    Minimum fragment (subblock) size in bytes (other pools)
-B         […]
           16777216 Block size (other pools)
[…]
```

- Each block consists here of 256 subblocks, so each file occupies at least 64kiB of disk space → Ratio controls the balance between performance (i.e. large subblocks) and the treatment of small files (i.e. small subblocks).
- Larger block sizes have better performance with spinning disks, but it becomes more likely to cause problems if different tasks try to write to the same block which can cause severe performance drops.

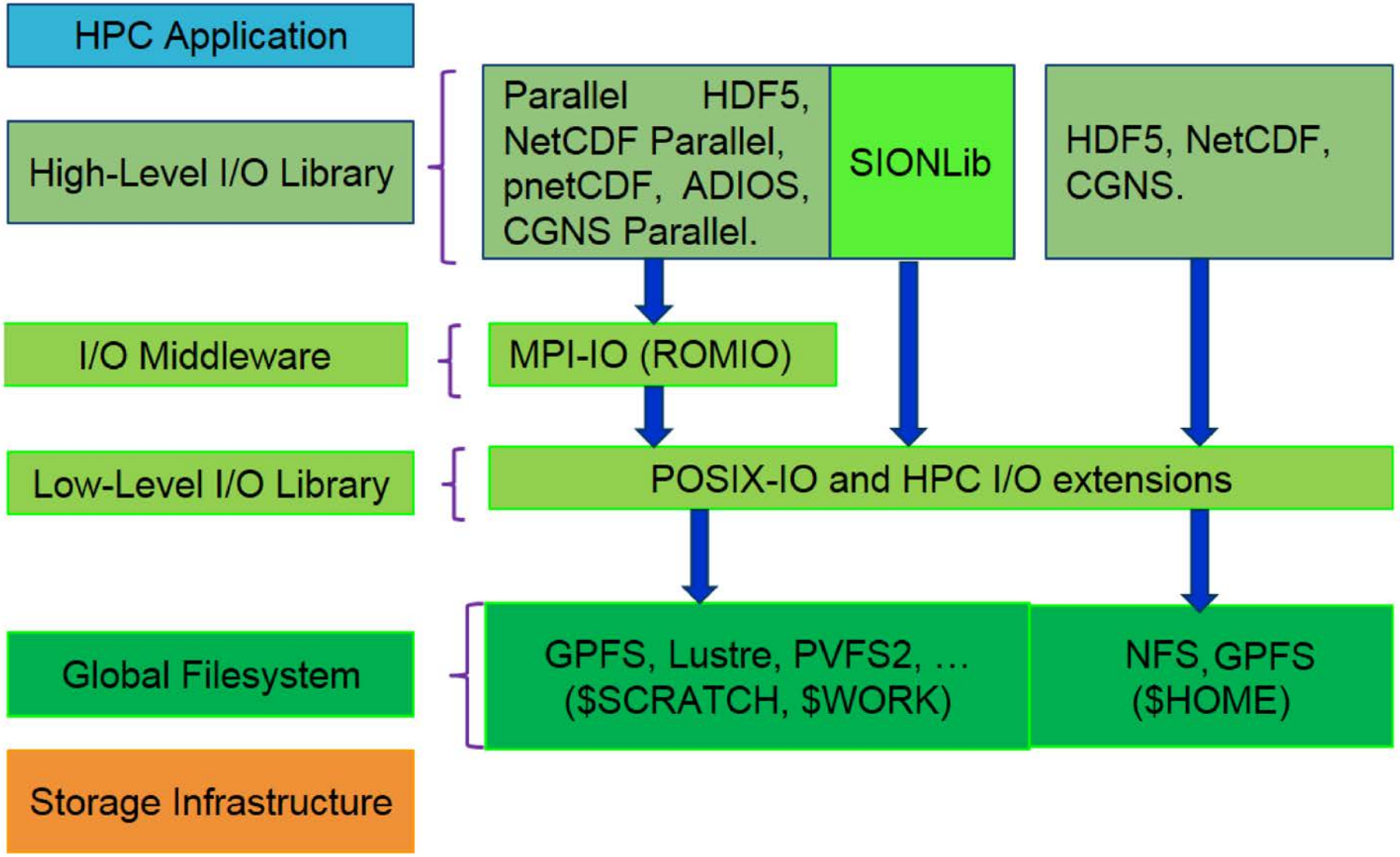
HPC I/O system



HPC I/O system

- **POSIX**: (**Portable Operating System Interface)** is a set of standard operating system interfaces based on the Unix operating system. → I/O layer present on all unix-like systems.
- The portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.
- **MPI-IO:** provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files.
- Many applications make use of higher-level I/O libraries such as the Hierarchical Data Form (HDF), the Network Common Data Format (NetCDF).

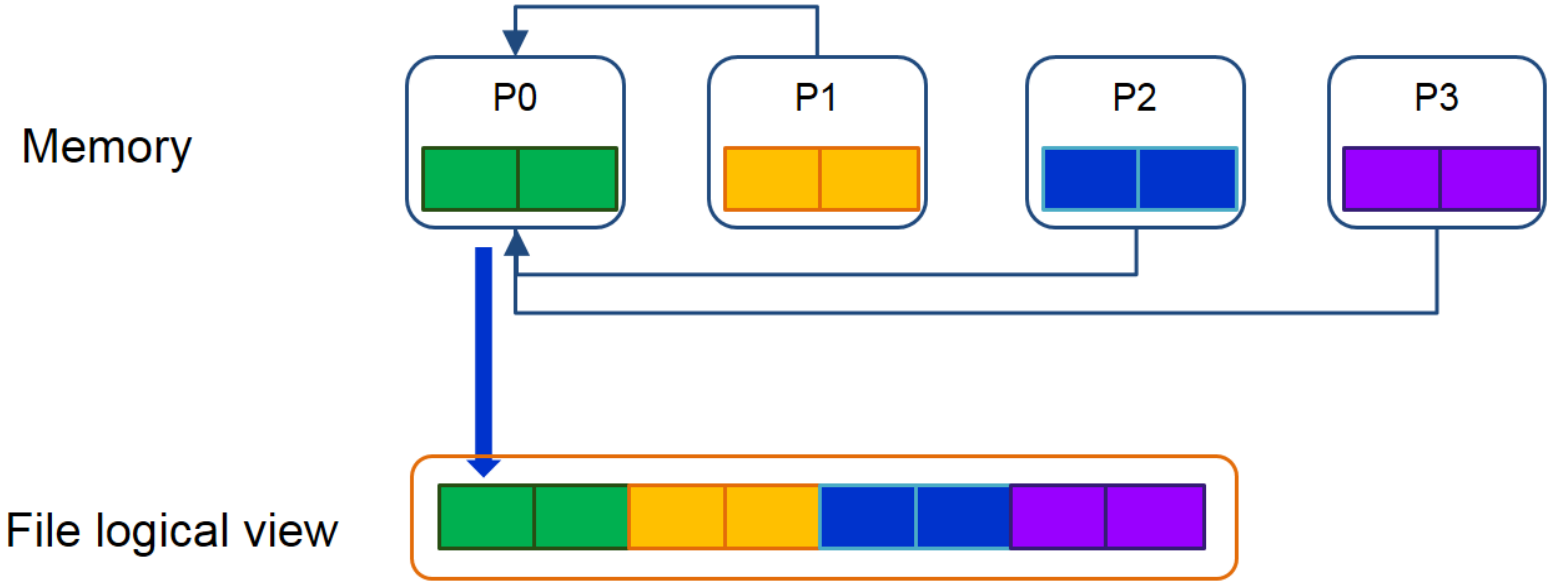
HPC I/O software stack



HPC I/O patterns

Parallel applications perform I/O in serial and parallel.

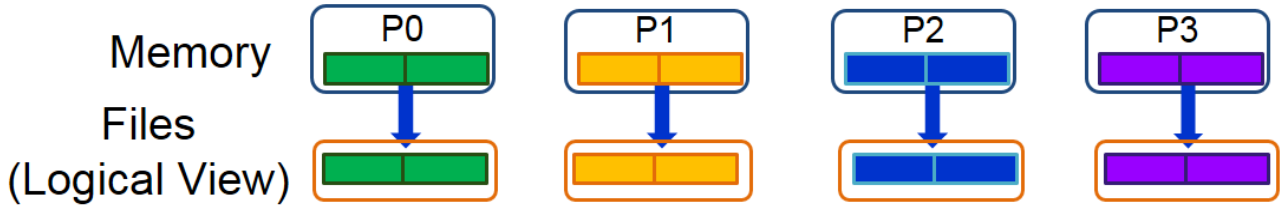
Serial I/O: A single process gets all information to be written via communication from all other processes.



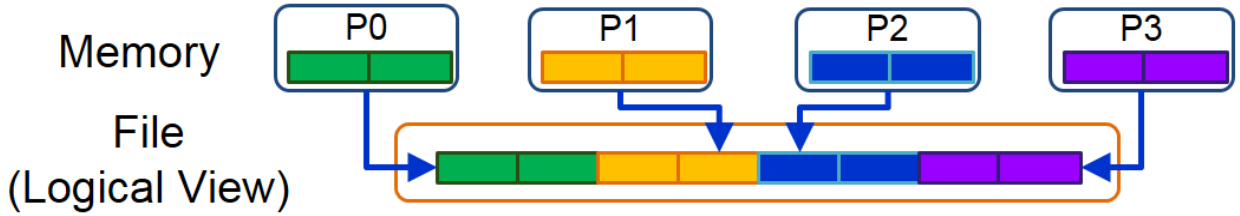
HPC I/O patterns

`Parallel I/O:` Several processes perform I/O within a single file or multiple files.

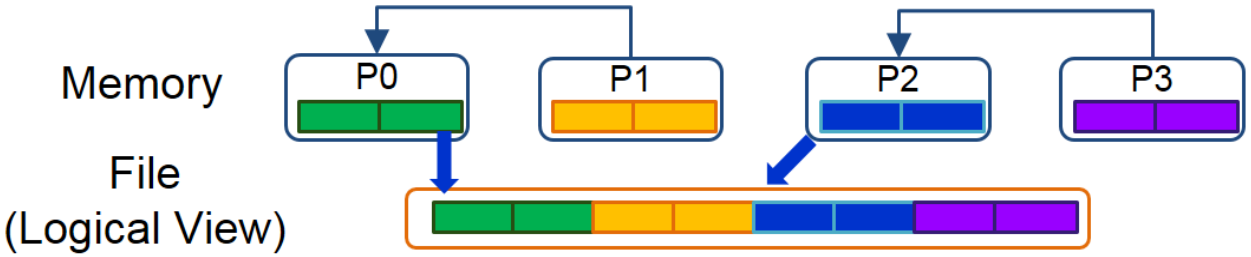
1 file per process (UNIQUE access type)



A Single shared File (SHARED access type)



Single file shared for "N" processes



HPC I/O patterns

- **1 file per process (UNIQUE access type)**
  Limited by file system and it does not scale for large count of processes. Number of files creates bottleneck with metadata operations and a number of simultaneous disk accesses can create contention for file system resources.
- **A Single shared File (SHARED access type)**
  Data layout within the shared file must be defined appropriately to avoid the contention due to concurrent accesses.
- **Single file shared for ”N” processes**
  The number of shared files increases and it decreases the number of processes per file. In this way, it is possible reduce the metadata operations and the concurrent accesses to shared files.

Main messages:

- Avoid unnecessary I/O. For example: switch off debug output for production runs.
- Perform I/O in few and large chunks. In parallel file systems, the chunk size should be multiple of the block size of the file system.
- Avoid unnecessary/large-scale open/close statements.
- Think carefully what filesystem to use. Parallel file systems may have bad scaling for metadata operations, but provide high/scalable bandwidth.
- You want to have effective I/O for long simulations → Checkpointing is important and you want it to be fast.

- Avoid explicit flushes of data to disk, except when needed for consistency reasons.
- Use existing specialized I/O libraries.

Main messages:

- No one forces you to use many small files and you should really avoid this.
- If you need to transfer the binary files between different architectures, consider the little vs. big-endian byte order (can pretty much all be avoided using e.g. HDF5). Limitations may apply on file sizes and data types.
- For parallel programs, the strategy “single file per process” could provide highest throughput, but usually this needs post-processing. And one ends up again with many single files for large simulations.
- HDF-Group: “Friends don’t let friends use file-per-process!” [1]

  Pros: No post-processing. Possible to change the number of processes when reading a checkpoint.
- There is no general “rule” how to do proper I/O → Experiments are necessary.

[1] https://www.hdfgroup.org/wp-content/uploads/2020/06/2020-06-26-Parallel-HDF5-Performance-Tuning.pdf

HDF5: Hierarchical Data Format

- Designed to store and organize large amounts of data.
- HDF5 has no file size limitation and is able to manage files as big as the largest allowed by the operating system.
- Supports more than one unlimited dimension in a data type (no need to know the extend in advance).
- HDF5 provides support for C, Fortran, Java, C++, Python…
- Can be compiled to provide parallel support using the MPI library.
- Contents of an HDF file can be accessed using the POSIX-like syntax `/path/to/mydata`. → Kind of a filesystem in a file → way to get rid of many small files.
- Supports inline compression of datasets using GNUzip, Szip and also external compression.
- NetCDF4 and later is based on HDF5.

HDF5: Hierarchical Data Format

HDF5 is designed at three levels:

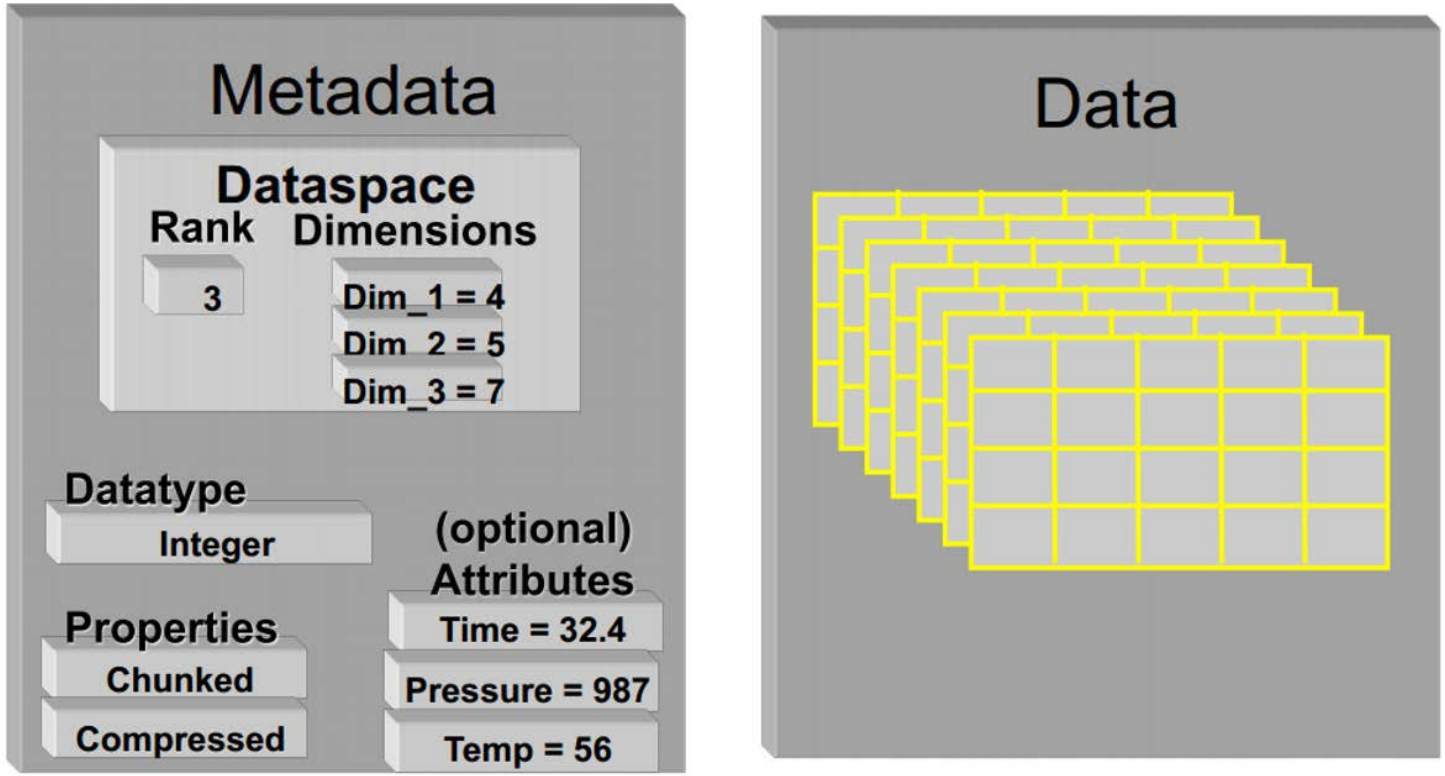
- A Data Model:
  - consists of abstract classes, such as files, datasets, groups, datatypes and dataspaces.
  - Developers use them to construct a model of their higher-level concepts.
- A Software Library:
  - designed to provide applications with an object-oriented programming interface.
  - a powerful, flexible and high performance interface.
- A file format:
  - provides portable, backward and forward compatible, and extensible instantiation of the HDF5 data model.

HDF5: Hierarchical Data Format

- HDF5 files are organized in a hierarchical structure, with two primary structures: groups and datasets.
  - HDF5 group: a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.
  - HDF5 dataset: a multidimensional array of data elements, together with supporting metadata.
- The primary classes in the HDF5 data model are:
  - File
  - Dataset
  - Group
  - Link
  - Attribute

HDF5: Hierarchical Data Format

- HDF5 datasets organize and contain data elements.
- HDF5 datatype describes individual data elements.
- HDF5 dataspace describes the logical layout of the data elements.



Source: http://press3.mcs.anl.gov/computingschool/files/2014/01/QKHDF5-Intro-v2.pdf

HDF5 interface conventions

- **H5** general purpose library functions
- **H5A** annotations: attribute access and manipulation routines
- **H5D** dataset access and manipulation routines
- **H5E** error handling routines
- **H5F** file access routines
- **H5G** group creation and operation routines
- **H5I** identifier routines
- **H5L** link routines
- **H5O** object routines
- **H5P** object property list manipulation routines
- **H5R** reference routines
- **H5S** dataspace definition and access routines
- **H5T** datatype creation and manipulation routines
- **H5Z** compression routine(s)

HDF5 API

- The API looks (and actually is) really is extensive: more than 300 functions.
- But do not worry to much:
  - For the beginning, only a few functions are needed to start with HDF5.
  - If you need more advanced features, there is quite a good documentation to learn from.
- Basic workflow:
  - An object is opened or created.
  - The object is accessed, possibly many times.
  - The object is closed.

- Properties of objects are ***optionally*** defined, like:
  - Access or creation properties (e.g. open an existing or overwrite an existing file; parallel file access…).

HDF5 API

Abstact simple call scheme:

H5Fcreate (H5Fopen)	create (open) File
H5Screate_simple/H5Screate	create Dataspace
H5Dcreate (H5Dopen)	create (open) Dataset
H5Dread, H5Dwrite	access Dataset
H5Dclose	close Dataset
H5Sclose	close Dataspace
H5Fclose	close File

HDF5 API

If you want to see a detailed example have a look at `hdf5_examples/h5_write.c` . It is taken from the source tarball of HDF5 but with some added explanations and the Endian conversion was removed for clarity.

You can compile and run it:

```
~> module load hdf5 # already MPI parallel version
```

HDF5 comes with some convenient compiler wrappers:

```
~> h5pcc –o h5_write h5_write.c
~> ./h5_write
```

And then you can look at the created HDF5 file, e.g. with:

```
~> h5dump SDS.h5
```

Hyperslabs

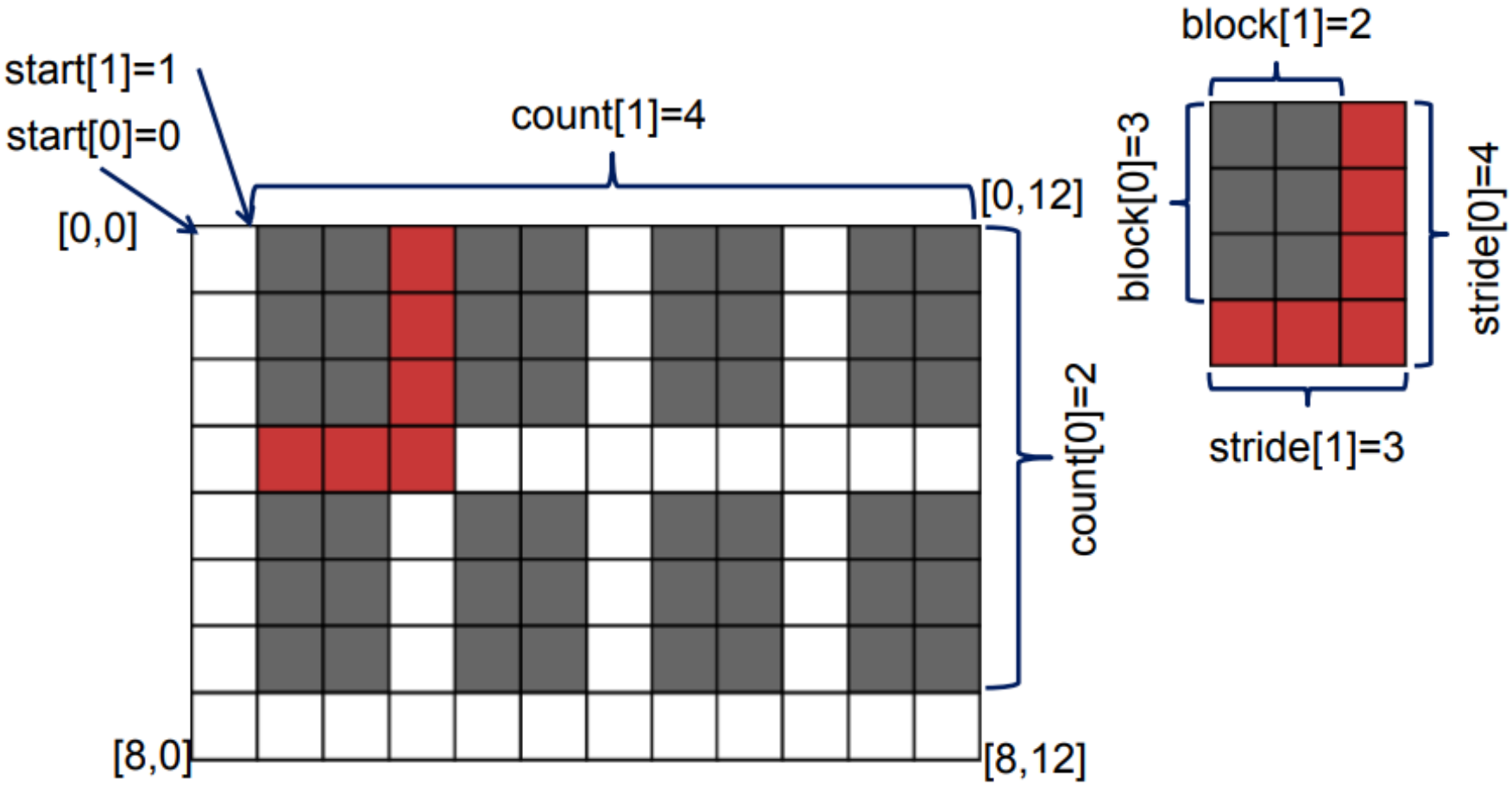
- HDF5 allows reading or writing to a portion of a dataset by use of a ***hyperslab*** selection.
- Can be a logically contiguous collection or a regular pattern of points or blocks.
- Hyperslabs are described by four parameters:
  - start: (or offset): starting location.
  - stride: separation blocks to be selected.
  - count: number of blocks to be selected.
  - block: size of block to be selected from dataspace.

- The dimensions of these four parameters correspond to dimensions of the underlying dataspace.

Hyperslabs Example 1

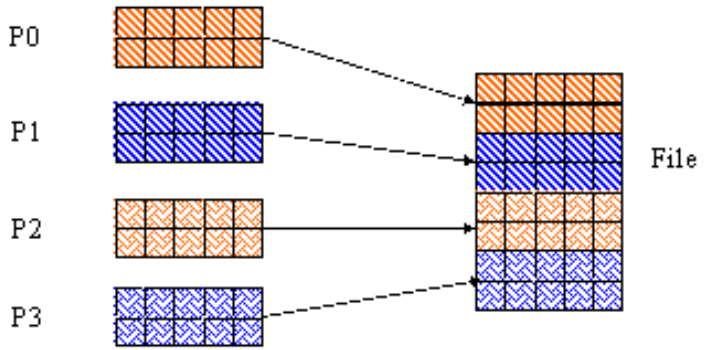
start: starting location
count: number of blocks to be selected

stride: separation blocks to be selected
block: size of block to be selected



Hyperslabs Example 2

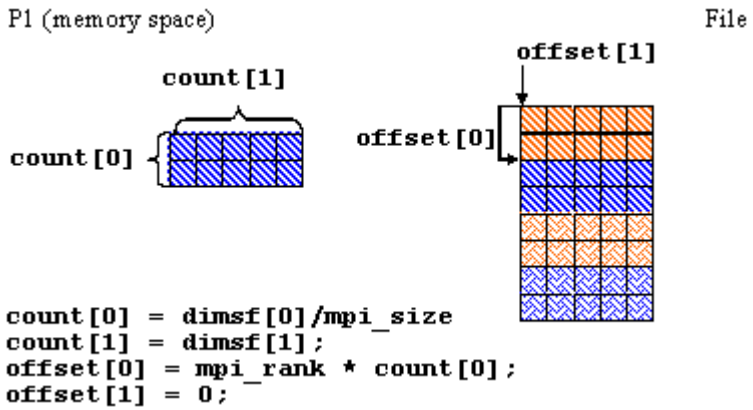
- Now we use hyperslabs to write a distributed dataset to a single file.
- The dimension of the dataset is 8x5.
- Each process writes 1 or 2 or 4 o 8 complete rows to a file.
- Here four process writes 2 row into a file:




```
dimsf[0] = 8  # number of rows in the full dataset
dimsf[1] = 5  # number of columns
count [0] = dimsf [0] / mpi_size
count [1] = dimsf [1]
# stride and count is not needed here
```

Source: https://portal.hdfgroup.org/display/HDF5/Writing+by+Contiguous+Hyperslab+in+PHDF5

Hyperslabs Example 2



The offset for ***each*** rank is then

```
offset[0] = mpi_rank * count[0];
offset[1] = 0;
# stride and block are not needed here
```

The principle calling signature to tell HDF5 about the hyperslab is here:

```
H5Sselect_hyperslab(dataspace_in_file, H5S_SELECT_SET, *start, *stride, *count, *block)
```

Where we have to set `*stride` and `*block` to `NULL`:

```
H5Sselect_hyperslab(dataspace_in_file, H5S_SELECT_SET, *start, NULL, *count, NULL)
```

Now `dataspace_in_file` knows where the local data should be written to.

Source: https://portal.hdfgroup.org/display/HDF5/Writing+by+Contiguous+Hyperslab+in+PHDF5

HDF5 Example 2

The full example code can be found in `hdf5_examples/Hyperslab_by_row.c`
Compilation is completely similar:

```
~> h5pcc –o Hyperslab_by_row Hyperslab_by_row.c
~> mpiexec –n NUM_PROCESSES ./Hyperslab_by_row
```

You can check again the output for different number of processes with h5dump.

I/O Profiling: Darshan

- Darshan is a lightweight, scalable I/O characterization tool that transparently captures I/O access pattern information from production applications.
- Darshan provides I/O profile for C and Fortran calls including: POSIX and MPI-IO (including support for HDF5).
- Darshan does not provide information about the I/O activity along the runtime, only afterwards.
- It uses a LD_PRELOAD mechanism to wrap the I/O calls.
- Easy to use:

```
~> cd $SCRATCH  # you should have learned this by now!
~> module load darshan-runtime
~> export DARSHAN_LOGHINTS="" # needed for Intel-MPI
~> mpiexec -n 4 -env LD_PRELOAD=$DARSHAN_LIBDIR/libdarshan.so \
   ./Hyperslab_by_row
```

I/O Profiling: Darshan

- At LRZ the Darshan logfiles are stored in `$SCRATCH/.darshan-logs`
- You can generate a pdf job summary using darshan-utils:

```
~> module load darshan-utils
~> cd $SCRATCH/.darshan-logs
~> for i in *.darshan; do darshan-job-summary.pl $i ; done
```

In the following example logfile I adjusted the values in `Hyperslab_by_row.c` to

```
#define NX     1024*1024
#define NY     500
```

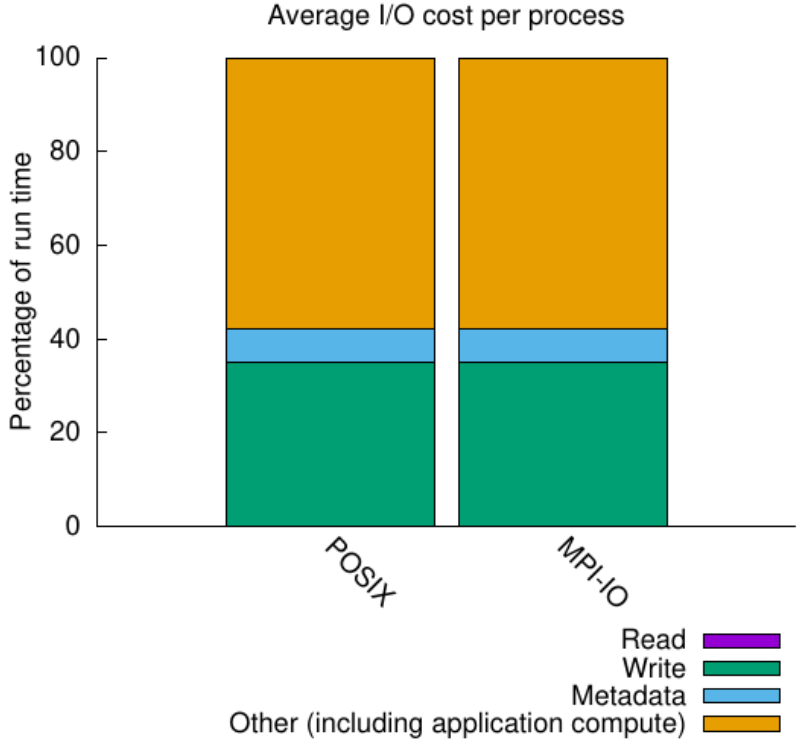
So overall 500*1024*1024*4bytes=2GiB are written, where 4bytes is the size of an integer.

I/O Profiling: Darshan

jobid: 28029	uid: 4022018	nprocs: 1	runtime: 2 seconds

I/O performance *estimate* (at the MPI-IO layer): transferred 2000.0 MiB at 2362.17 MiB/s

It shows you how much time your program spends doing I/O:

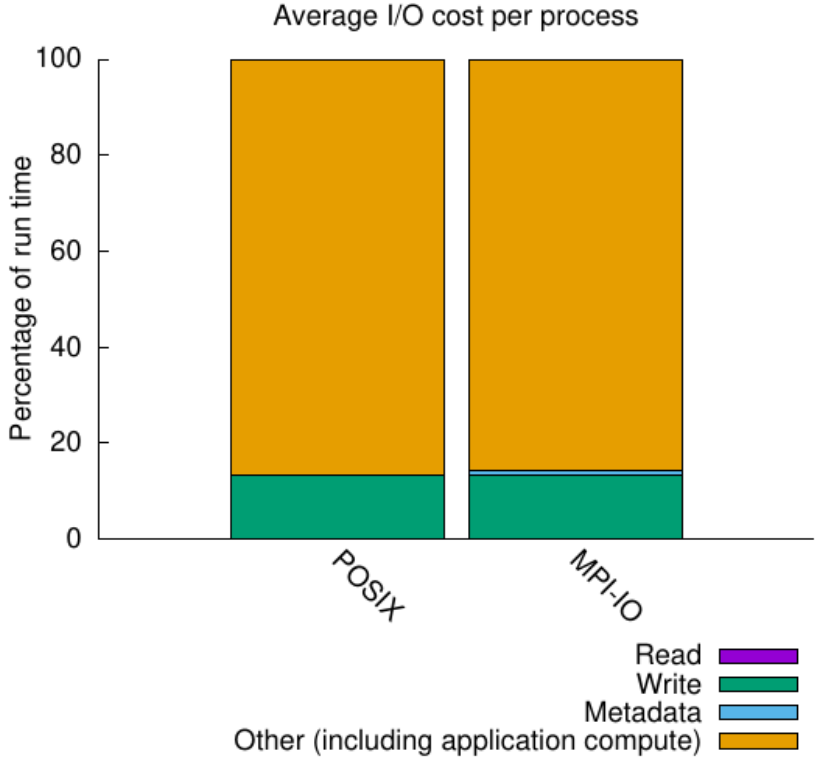


Rule of thumb: more than 10% percent of runtime for I/O indicate an I/O bottleneck. Ooops.

I/O Profiling: Darshan

jobid: 28464	uid: 4022018	nprocs: 16	runtime: 1 seconds

I/O performance *estimate* (at the MPI-IO layer): transferred 2000.0 MiB at 7955.04 MiB/s



Looks much better. For more information please refer to the Darshan Documentation or also Best Practice Guide – Parallel I/O.